Documentation PostgreSQL 9.0.18

The PostgreSQL Global Development Group

Documentation PostgreSQL 9.0.18

The PostgreSQL Global Development Group
Copyright © 1996-2014 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL™ is Copyright (c) 1996-2014 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95™ is Copyright (c) 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN « AS-IS » BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Préface xx
1. Définition de PostgreSQL™ xx
2. Bref historique de PostgreSQL™ xx
3. Conventions xxii
4. Pour plus d'informations xxii
5. Lignes de conduite pour les rapports de bogues xxii
I. Tutoriel 1
1. Démarrage 2
1.1. Installation 2
1.2. Concepts architecturaux de base 2
1.3. Création d'une base de données 2
1.4. Accéder à une base 4
2. Le langage SQL 5
2.1. Introduction 5
2.2. Concepts 5
2.3. Créer une nouvelle table 5
2.4. Remplir une table avec des lignes 6
2.5. Interroger une table 6
2.6. Jointures entre les tables 8
2.7. Fonctions d'agrégat 9
2.8. Mises à jour 10
2.9. Suppressions 11
3. Fonctionnalités avancées 12
3.1. Introduction 12
3.2. Vues 12
3.3. Clés étrangères 12
3.4. Transactions 13
3.5. Fonctions de fenêtrage 14
3.6. Héritage .....
3.7. Conclusion 18
II. Langage SQL 19
4. Syntaxe SQL 20
4.1. Structure lexicale 20
4.2. Expressions de valeurs 27
4.3. Fonctions appelantes 35
5. Définition des données 37
5.1. Notions fondamentales sur les tables 37
5.2. Valeurs par défaut 38
5.3. Contraintes 38
5.4. Colonnes système 44
5.5. Modification des tables 45
5.6. Droits 47
5.7. Schémas 47
5.8. L'héritage .....
5.9. Partitionnement 53
5.10. Autres objets de la base de données 59
5.11. Gestion des dépendances 59
6. Manipulation de données 61
6.1. Insérer des données 61
6.2. Actualiser les données 62
6.3. Supprimer des données 62
7. Requêtes 64
7.1. Aperçu 64
7.2. Expressions de table 64
7.3. Listes de sélection 72
7.4. Combiner des requêtes 73
7.5. Tri des lignes 73
7.6. `LIMIT` et `OFFSET` 74
7.7. Listes `VALUES` 75
7.8. Requêtes `WITH` (*Common Table Expressions*) 75
8. Types de données 79
8.1. Types numériques 80

8.2. Types monétaires ... 83
8.3. Types caractère ... 83
8.4. Types de données binaires ... 85
8.5. Types date/heure ... 87
8.6. Type booléen ... 95
8.7. Types énumération ... 96
8.8. Types géométriques ... 97
8.9. Types adresses réseau ... 99
8.10. Type chaîne de bits ... 100
8.11. Types de recherche plein texte ... 101
8.12. Type UUID ... 103
8.13. Type XML ... 103
8.14. Tableaux ... 105
8.15. Types composites ... 111
8.16. Types identifiant d'objet ... 114
8.17. Pseudo-Types ... 115
9. Fonctions et opérateurs ... 117
9.1. Opérateurs logiques ... 117
9.2. Opérateurs de comparaison ... 117
9.3. Fonctions et opérateurs mathématiques ... 119
9.4. Fonctions et opérateurs de chaînes ... 121
9.5. Fonctions et opérateurs de chaînes binaires ... 129
9.6. Fonctions et opérateurs sur les chaînes de bits ... 131
9.7. Correspondance de motif ... 131
9.8. Fonctions de formatage des types de données ... 143
9.9. Fonctions et opérateurs sur date/heure ... 148
9.10. Fonctions de support enum ... 157
9.11. Fonctions et opérateurs géométriques ... 158
9.12. Fonctions et opérateurs sur les adresses réseau ... 161
9.13. Fonctions et opérateurs de la recherche plein texte ... 163
9.14. Fonctions XML ... 166
9.15. Fonctions de manipulation de séquences ... 173
9.16. Expressions conditionnelles ... 175
9.17. Fonctions et opérateurs de tableaux ... 177
9.18. Fonctions d'agrégat ... 178
9.19. Fonctions Window ... 182
9.20. Expressions de sous-requêtes ... 183
9.21. Comparaisons de lignes et de tableaux ... 186
9.22. Fonctions retournant des ensembles ... 188
9.23. Fonctions d'informations système ... 190
9.24. Fonctions d'administration système ... 197
9.25. Fonctions trigger ... 204
10. Conversion de types ... 206
10.1. Aperçu ... 206
10.2. Opérateurs ... 207
10.3. Fonctions ... 209
10.4. Stockage de valeurs ... 211
10.5. Constructions `UNION`, `CASE` et constructions relatives ... 212
11. Index ... 214
11.1. Introduction ... 214
11.2. Types d'index ... 214
11.3. Index multicolonnes ... 216
11.4. Index et `ORDER BY` ... 216
11.5. Combiner des index multiples ... 217
11.6. Index d'unicité ... 217
11.7. Index d'expressions ... 218
11.8. Index partiels ... 218
11.9. Classes et familles d'opérateurs ... 220
11.10. Examiner l'utilisation des index ... 221
12. Recherche plein texte ... 223
12.1. Introduction ... 223
12.2. Tables et index ... 225
12.3. Contrôler la recherche plein texte ... 227

12.4. Fonctionnalités supplémentaires ... 232
12.5. Analyseurs ... 236
12.6. Dictionnaires ... 238
12.7. Exemple de configuration ... 244
12.8. Tester et déboguer la recherche plein texte ... 245
12.9. Types d'index GiST et GIN ... 249
12.10. Support de psql ... 250
12.11. Limites ... 252
12.12. Migration à partir d'une recherche plein texte antérieure à 8.3 ... 252
13. Contrôle d'accès simultané ... 254
13.1. Introduction ... 254
13.2. Isolation des transactions ... 254
13.3. Verrouillage explicite ... 257
13.4. Vérification de cohérence des données au niveau de l'application ... 260
13.5. Verrouillage et index ... 261
14. Conseils sur les performances ... 262
14.1. Utiliser **EXPLAIN** ... 262
14.2. Statistiques utilisées par le planificateur ... 266
14.3. Contrôler le planificateur avec des clauses `JOIN` explicites ... 267
14.4. Remplir une base de données ... 268
14.5. Configuration avec une perte acceptée ... 271
III. Administration du serveur ... 272
15. Procédure d'installation de PostgreSQL™ du code source ... 273
15.1. Version courte ... 273
15.2. Prérequis ... 273
15.3. Obtenir les sources ... 274
15.4. Mise à jour ... 275
15.5. Procédure d'installation ... 276
15.6. Initialisation post-installation ... 283
15.7. Démarrer ... 284
15.8. Et maintenant ? ... 285
15.9. Plateformes supportées ... 285
15.10. Notes spécifiques à des plateformes ... 286
16. Installation à partir du code source sur Windows™ ... 294
16.1. Construire avec Visual C++™ ou le Platform SDK™ ... 294
16.2. Construire libpq avec Visual C++™ ou Borland C++™ ... 297
17. Configuration du serveur et mise en place ... 299
17.1. Compte utilisateur PostgreSQL™ ... 299
17.2. Créer un groupe de base de données ... 299
17.3. Lancer le serveur de bases de données ... 300
17.4. Gérer les ressources du noyau ... 302
17.5. Arrêter le serveur ... 308
17.6. Empêcher l'usurpation de serveur ... 309
17.7. Options de chiffrement ... 309
17.8. Connexions tcp/ip sécurisées avec ssl ... 310
17.9. Connexions tcp/ip sécurisées avec des tunnels ssh tunnels ... 312
18. Configuration du serveur ... 313
18.1. Paramètres de configuration ... 313
18.2. Emplacement des fichiers ... 314
18.3. Connexions et authentification ... 315
18.4. Consommation des ressources ... 318
18.5. Write Ahead Log ... 321
18.6. Planification des requêtes ... 326
18.7. Remonter et tracer les erreurs ... 330
18.8. Statistiques d'exécution ... 336
18.9. Nettoyage (**vacuum**) automatique ... 337
18.10. Valeurs par défaut des connexions client ... 339
18.11. Gestion des verrous ... 343
18.12. Compatibilité de version et de plateforme ... 343
18.13. Options préconfigurées ... 345
18.14. Options personnalisées ... 346
18.15. Options pour les développeurs ... 346
18.16. Options courtes ... 348

19. Authentification du client ... 350
19.1. Le fichier pg_hba.conf ... 350
19.2. Correspondances d'utilisateurs ... 354
19.3. Méthodes d'authentification ... 355
19.4. Problèmes d'authentification ... 360
20. Rôles et droits de la base de données ... 362
20.1. Rôles de la base de données ... 362
20.2. Attributs des rôles ... 362
20.3. Droits ... 363
20.4. Appartenance d'un rôle ... 364
20.5. Sécurité des fonctions et déclencheurs (triggers) ... 365
21. Administration des bases de données ... 366
21.1. Aperçu ... 366
21.2. Création d'une base de données ... 366
21.3. Bases de données modèles ... 367
21.4. Configuration d'une base de données ... 368
21.5. Détruire une base de données ... 368
21.6. Tablespaces ... 368
22. Localisation ... 370
22.1. Support des locales ... 370
22.2. Support des jeux de caractères ... 372
23. Planifier les tâches de maintenance ... 377
23.1. Nettoyages réguliers ... 377
23.2. Ré-indexation régulière ... 382
23.3. Maintenance du fichier de traces ... 382
24. Sauvegardes et restaurations ... 384
24.1. Sauvegarde SQL ... 384
24.2. Sauvegarde de niveau système de fichiers ... 386
24.3. Archivage continu et récupération d'un instantané (PITR) ... 387
24.4. Migration entre versions ... 394
25. Haute disponibilité, répartition de charge et réplication ... 397
25.1. Comparaison de différentes solutions ... 397
25.2. Serveurs de Standby par transfert de journaux ... 400
25.3. Failover (bascule) ... 403
25.4. Méthode alternative pour le log shipping ... 403
25.5. Hot Standby ... 405
26. Configuration de la récupération ... 411
26.1. Paramètres de récupération de l'archive ... 411
26.2. Paramètres de cible de récupération ... 411
26.3. Paramètres de serveur de Standby ... 412
27. Surveiller l'activité de la base de données ... 413
27.1. Outils Unix standard ... 413
27.2. Le récupérateur de statistiques ... 413
27.3. Visualiser les verrous ... 419
27.4. Traces dynamiques ... 420
28. Surveiller l'utilisation des disques ... 428
28.1. Déterminer l'utilisation des disques ... 428
28.2. Panne pour disque saturé ... 429
29. Fiabilité et journaux de transaction ... 430
29.1. Fiabilité ... 430
29.2. Write-Ahead Logging (WAL) ... 431
29.3. Validation asynchrone (Asynchronous Commit) ... 431
29.4. Configuration des journaux de transaction ... 432
29.5. Vue interne des journaux de transaction ... 434
30. Tests de régression ... 436
30.1. Lancer les tests ... 436
30.2. Évaluation des tests ... 437
30.3. Fichiers de comparaison de variants ... 439
30.4. Examen de la couverture du test ... 439
IV. Interfaces client ... 441
31. libpq - Bibliothèque C ... 442
31.1. Fonctions de contrôle de connexion à la base de données ... 442
31.2. Fonctions de statut de connexion ... 447

31.3. Fonctions de commandes d'exécution ... 450
31.4. Traitement des commandes asynchrones ... 461
31.5. Annuler des requêtes en cours d'exécution ... 464
31.6. Interface à chemin rapide ... 464
31.7. Notification asynchrone ... 465
31.8. Fonctions associées avec la commande **COPY** ... 466
31.9. Fonctions de contrôle ... 469
31.10. Fonctions diverses ... 470
31.11. Traitement des messages ... 472
31.12. Système d'événements ... 473
31.13. Variables d'environnement ... 478
31.14. Fichier de mots de passe ... 479
31.15. Fichier des connexions de service ... 480
31.16. Recherches LDAP des paramètres de connexion ... 480
31.17. Support de SSL ... 481
31.18. Comportement des programmes threadés ... 484
31.19. Construire des applications avec libpq ... 484
31.20. Exemples de programmes ... 485
32. Objets larges ... 493
32.1. Introduction ... 493
32.2. Fonctionnalités d'implémentation ... 493
32.3. Interfaces client ... 493
32.4. Fonctions du côté serveur ... 495
32.5. Programme d'exemple ... 496
33. ECPG - SQL embarqué dans du C ... 501
33.1. Concept ... 501
33.2. Se connecter au serveur de bases de données ... 501
33.3. Fermer une connexion ... 502
33.4. Exécuter des commandes SQL ... 502
33.5. Choisir une connexion ... 503
33.6. Utiliser des variables hôtes ... 503
33.7. SQL dynamique ... 506
33.8. Bibliothèque pgtypes ... 507
33.9. Utiliser les aires de descripteur ... 517
33.10. Mode de compatibilité Informix™ ... 520
33.11. Gestion des erreurs ... 532
33.12. Directives du préprocesseur ... 535
33.13. Traiter les programmes en SQL embarqué ... 537
33.14. Fonctions de bibliothèque ... 537
33.15. Fonctionnement interne ... 538
34. Schéma d'information ... 540
34.1. Le schéma ... 540
34.2. Types de données ... 540
34.3. `information_schema_catalog_name` ... 540
34.4. `administrable_role_authorizations` ... 540
34.5. `applicable_roles` ... 541
34.6. `attributes` ... 541
34.7. `check_constraint_routine_usage` ... 543
34.8. `check_constraints` ... 543
34.9. `column_domain_usage` ... 544
34.10. `column_privileges` ... 544
34.11. `column_udt_usage` ... 544
34.12. `columns` ... 545
34.13. `constraint_column_usage` ... 547
34.14. `constraint_table_usage` ... 548
34.15. `data_type_privileges` ... 548
34.16. `domain_constraints` ... 549
34.17. `domain_udt_usage` ... 549
34.18. `domains` ... 549
34.19. `element_types` ... 551
34.20. `enabled_roles` ... 553
34.21. `foreign_data_wrapper_options` ... 553
34.22. `foreign_data_wrappers` ... 553

34.23. foreign_server_options ... 554
34.24. foreign_servers ... 554
34.25. key_column_usage ... 554
34.26. parameters ... 555
34.27. referential_constraints ... 556
34.28. role_column_grants ... 557
34.29. role_routine_grants ... 557
34.30. role_table_grants ... 558
34.31. role_usage_grants ... 558
34.32. routine_privileges ... 559
34.33. routines ... 559
34.34. schemata ... 563
34.35. sequences ... 563
34.36. sql_features ... 564
34.37. sql_implementation_info ... 564
34.38. sql_languages ... 565
34.39. sql_packages ... 565
34.40. sql_parts ... 565
34.41. sql_sizing ... 566
34.42. sql_sizing_profiles ... 566
34.43. table_constraints ... 566
34.44. table_privileges ... 567
34.45. tables ... 567
34.46. triggered_update_columns ... 568
34.47. triggers ... 568
34.48. usage_privileges ... 569
34.49. user_mapping_options ... 570
34.50. user_mappings ... 570
34.51. view_column_usage ... 571
34.52. view_routine_usage ... 571
34.53. view_table_usage ... 572
34.54. views ... 572
V. Programmation serveur ... 573
35. Étendre SQL ... 574
35.1. L'extensibilité ... 574
35.2. Le système des types de PostgreSQL™ ... 574
35.3. Fonctions utilisateur ... 575
35.4. Fonctions en langage de requêtes (SQL) ... 576
35.5. Surcharge des fonctions ... 585
35.6. Catégories de volatilité des fonctions ... 586
35.7. Fonctions en langage de procédures ... 587
35.8. Fonctions internes ... 587
35.9. Fonctions en langage C ... 587
35.10. Agrégats utilisateur ... 607
35.11. Types utilisateur ... 608
35.12. Opérateurs définis par l'utilisateur ... 611
35.13. Informations sur l'optimisation d'un opérateur ... 611
35.14. Interfacer des extensions d'index ... 615
35.15. Utiliser le langage C++ ... 623
36. Déclencheurs (triggers) ... 624
36.1. Aperçu du comportement des déclencheurs ... 624
36.2. Visibilité des modifications des données ... 625
36.3. Écrire des fonctions déclencheurs en C ... 626
36.4. Un exemple complet de trigger ... 627
37. Système de règles ... 631
37.1. Arbre de requêtes ... 631
37.2. Vues et système de règles ... 632
37.3. Règles sur **insert**, **update** et **delete** ... 637
37.4. Règles et droits ... 645
37.5. Règles et statut de commande ... 647
37.6. Règles contre déclencheurs ... 647
38. Langages de procédures ... 650
38.1. Installation des langages de procédures ... 650

39. PL/pgSQL - Langage de procédures SQL ... 652
39.1. Aperçu ... 652
39.2. Structure de PL/pgSQL ... 653
39.3. Déclarations ... 654
39.4. Expressions ... 658
39.5. Instructions de base ... 658
39.6. Structures de contrôle ... 663
39.7. Curseurs ... 671
39.8. Erreurs et messages ... 676
39.9. Procédures trigger ... 677
39.10. Les dessous de PL/pgSQL ... 681
39.11. Astuces pour développer en PL/pgSQL ... 684
39.12. Portage d'Oracle™ PL/SQL ... 686
40. PL/Tcl - Langage de procédures Tcl ... 694
40.1. Aperçu ... 694
40.2. Fonctions et arguments PL/Tcl ... 694
40.3. Valeurs des données avec PL/Tcl ... 695
40.4. Données globales avec PL/Tcl ... 695
40.5. Accès à la base de données depuis PL/Tcl ... 696
40.6. Procédures pour déclencheurs en PL/Tcl ... 697
40.7. Les modules et la commande `unknown` ... 698
40.8. Noms de procédure Tcl ... 699
41. PL/Perl - Langage de procédures Perl ... 700
41.1. Fonctions et arguments PL/Perl ... 700
41.2. Valeurs en PL/Perl ... 702
41.3. Fonction incluses ... 703
41.4. Valeurs globales dans PL/Perl ... 706
41.5. Niveaux de confiance de PL/Perl ... 707
41.6. Déclencheurs PL/Perl ... 708
41.7. PL/Perl sous le capot ... 709
42. PL/Python - Langage de procédures Python ... 711
42.1. Python 2 et Python 3 ... 711
42.2. Fonctions PL/Python ... 712
42.3. Valeur des données avec PL/Python ... 713
42.4. Sharing Data ... 716
42.5. Blocs de code anonymes ... 717
42.6. Fonctions de déclencheurs ... 717
42.7. Accès à la base de données ... 717
42.8. Fonctions outils ... 718
42.9. Variables d'environnement ... 718
43. Interface de programmation serveur ... 720
43.1. Fonctions d'interface ... 720
43.2. Fonctions de support d'interface ... 748
43.3. Gestion de la mémoire ... 756
43.4. Visibilité des modifications de données ... 765
43.5. Exemples ... 765
VI. Référence ... 768
I. Commandes SQL ... 769
ABORT ... 770
ALTER AGGREGATE ... 771
ALTER CONVERSION ... 772
ALTER DATABASE ... 773
ALTER DEFAULT PRIVILEGES ... 775
ALTER DOMAIN ... 777
ALTER FOREIGN DATA WRAPPER ... 779
ALTER FUNCTION ... 780
ALTER GROUP ... 782
ALTER INDEX ... 783
ALTER LANGUAGE ... 785
ALTER LARGE OBJECT ... 786
ALTER OPERATOR ... 787
ALTER OPERATOR CLASS ... 788
ALTER OPERATOR FAMILY ... 789

ALTER ROLE 792
ALTER SCHEMA 795
ALTER SEQUENCE 796
ALTER SERVER 798
ALTER TABLE 799
ALTER TABLESPACE 806
ALTER TEXT SEARCH CONFIGURATION 807
ALTER TEXT SEARCH DICTIONARY 809
ALTER TEXT SEARCH PARSER 811
ALTER TEXT SEARCH TEMPLATE 812
ALTER TRIGGER 813
ALTER TYPE 814
ALTER USER 815
ALTER USER MAPPING 816
ALTER VIEW 817
ANALYZE 818
BEGIN 820
CHECKPOINT 822
CLOSE 823
CLUSTER 824
COMMENT 826
COMMIT 829
COMMIT PREPARED 830
COPY 831
CREATE AGGREGATE 838
CREATE CAST 841
CREATE CONSTRAINT TRIGGER 844
CREATE CONVERSION 845
CREATE DATABASE 847
CREATE DOMAIN 849
CREATE FOREIGN DATA WRAPPER 851
CREATE FUNCTION 853
CREATE GROUP 859
CREATE INDEX 860
CREATE LANGUAGE 865
CREATE OPERATOR 867
CREATE OPERATOR CLASS 869
CREATE OPERATOR FAMILY 871
CREATE ROLE 872
CREATE RULE 875
CREATE SCHEMA 877
CREATE SEQUENCE 879
CREATE SERVER 882
CREATE TABLE 884
CREATE TABLE AS 894
CREATE TABLESPACE 896
CREATE TEXT SEARCH CONFIGURATION 897
CREATE TEXT SEARCH DICTIONARY 898
CREATE TEXT SEARCH PARSER 899
CREATE TEXT SEARCH TEMPLATE 900
CREATE TRIGGER 901
CREATE TYPE 904
CREATE USER 910
CREATE USER MAPPING 911
CREATE VIEW 912
DEALLOCATE 914
DECLARE 915
DELETE 918
DISCARD 920
DO 921
DROP AGGREGATE 922
DROP CAST 923
DROP CONVERSION 924

DROP DATABASE 925
DROP DOMAIN 926
DROP FOREIGN DATA WRAPPER 927
DROP FUNCTION 928
DROP GROUP 929
DROP INDEX 930
DROP LANGUAGE 931
DROP OPERATOR 932
DROP OPERATOR CLASS 933
DROP OPERATOR FAMILY 934
DROP OWNED 935
DROP ROLE 936
DROP RULE 937
DROP SCHEMA 938
DROP SEQUENCE 939
DROP SERVER 940
DROP TABLE 941
DROP TABLESPACE 942
DROP TEXT SEARCH CONFIGURATION 943
DROP TEXT SEARCH DICTIONARY 944
DROP TEXT SEARCH PARSER 945
DROP TEXT SEARCH TEMPLATE 946
DROP TRIGGER 947
DROP TYPE 948
DROP USER 949
DROP USER MAPPING 950
DROP VIEW 951
END 952
EXECUTE 953
EXPLAIN 954
FETCH 958
GRANT 961
INSERT 966
LISTEN 968
LOAD 969
LOCK 970
MOVE 972
NOTIFY 973
PREPARE 975
PREPARE TRANSACTION 977
REASSIGN OWNED 979
REINDEX 980
RELEASE SAVEPOINT 982
RESET 983
REVOKE 984
ROLLBACK 987
ROLLBACK PREPARED 988
ROLLBACK TO SAVEPOINT 989
SAVEPOINT 991
SELECT 992
SELECT INTO 1005
SET 1007
SET CONSTRAINTS 1010
SET ROLE 1011
SET SESSION AUTHORIZATION 1013
SET TRANSACTION 1014
SHOW 1016
START TRANSACTION 1018
TRUNCATE 1019
UNLISTEN 1021
UPDATE 1022
VACUUM 1025
VALUES 1028

II. Applications client de PostgreSQL 1030
clusterdb 1031
createdb 1033
createlang 1035
createuser 1037
dropdb 1040
droplang 1042
dropuser 1044
ecpg 1046
pg_config 1048
pg_dump 1050
pg_dumpall 1056
pg_restore 1060
psql 1065
reindexdb 1085
vacuumdb 1087
III. Applications relatives au serveur PostgreSQL 1089
initdb 1090
pg_controldata 1093
pg_ctl 1094
pg_resetxlog 1098
postgres 1100
postmaster 1105
VII. Internes 1106
44. Présentation des mécanismes internes de PostgreSQL 1107
44.1. Chemin d'une requête 1107
44.2. Établissement des connexions 1107
44.3. Étape d'analyse 1108
44.4. Système de règles de PostgreSQL™ 1108
44.5. Planificateur/Optimiseur 1109
44.6. Exécuteur 1110
45. Catalogues système 1111
45.1. Aperçu 1111
45.2. pg_aggregate 1112
45.3. pg_am 1112
45.4. pg_amop 1113
45.5. pg_amproc 1114
45.6. pg_attrdef 1114
45.7. pg_attribute 1115
45.8. pg_authid 1116
45.9. pg_auth_members 1117
45.10. pg_cast 1117
45.11. pg_class 1118
45.12. pg_constraint 1120
45.13. pg_conversion 1121
45.14. pg_database 1121
45.15. pg_db_role_setting 1122
45.16. pg_default_acl 1123
45.17. pg_depend 1123
45.18. pg_description 1124
45.19. pg_enum 1124
45.20. pg_foreign_data_wrapper 1125
45.21. pg_foreign_server 1125
45.22. pg_index 1126
45.23. pg_inherits .....
45.24. pg_language 1127
45.25. pg_largeobject 1128
45.26. pg_largeobject_metadata 1128
45.27. pg_namespace 1128
45.28. pg_opclass 1129
45.29. pg_operator 1129
45.30. pg_opfamily 1130
45.31. pg_pltemplate 1130

45.32. pg_proc ... 1131
45.33. pg_rewrite ... 1133
45.34. pg_shdepend ... 1133
45.35. pg_shdescription ... 1134
45.36. pg_statistic ... 1134
45.37. pg_tablespace ... 1135
45.38. pg_trigger ... 1136
45.39. pg_ts_config ... 1137
45.40. pg_ts_config_map ... 1137
45.41. pg_ts_dict ... 1138
45.42. pg_ts_parser ... 1138
45.43. pg_ts_template ... 1138
45.44. pg_type ... 1139
45.45. pg_user_mapping ... 1142
45.46. Vues système ... 1142
45.47. pg_cursors ... 1143
45.48. pg_group ... 1144
45.49. pg_indexes ... 1144
45.50. pg_locks ... 1144
45.51. pg_prepared_statements ... 1146
45.52. pg_prepared_xacts ... 1147
45.53. pg_roles ... 1147
45.54. pg_rules ... 1148
45.55. pg_settings ... 1148
45.56. pg_shadow ... 1149
45.57. pg_stats ... 1149
45.58. pg_tables ... 1150
45.59. pg_timezone_abbrevs ... 1151
45.60. pg_timezone_names ... 1151
45.61. pg_user ... 1151
45.62. pg_user_mappings ... 1152
45.63. pg_views ... 1152
46. Protocole client/serveur ... 1153
46.1. Aperçu ... 1153
46.2. Flux de messages ... 1154
46.3. Types de données des message ... 1162
46.4. Protocole de réplication en continu ... 1162
46.5. Formats de message ... 1163
46.6. Champs des messages d'erreur et d'avertissement ... 1172
46.7. Résumé des modifications depuis le protocole 2.0 ... 1173
47. Conventions de codage pour PostgreSQL ... 1175
47.1. Formatage ... 1175
47.2. Reporter les erreurs dans le serveur ... 1175
47.3. Guide de style des messages d'erreurs ... 1177
48. Support natif des langues ... 1181
48.1. Pour le traducteur ... 1181
48.2. Pour le développeur ... 1183
49. Écrire un gestionnaire de langage procédural ... 1186
50. Optimiseur génétique de requêtes (*Genetic Query Optimizer*) ... 1189
50.1. Gérer les requêtes, un problème d'optimisation complexe ... 1189
50.2. Algorithmes génétiques ... 1189
50.3. Optimisation génétique des requêtes (GEQO) dans PostgreSQL ... 1190
50.4. Lectures supplémentaires ... 1191
51. Définition de l'interface des méthodes d'accès aux index ... 1192
51.1. Entrées du catalogue pour les index ... 1192
51.2. Fonctions de la méthode d'accès aux index ... 1193
51.3. Parcours d'index ... 1195
51.4. Considérations sur le verrouillage d'index ... 1196
51.5. Vérification de l'unicité de l'index ... 1197
51.6. Fonctions d'estimation des coûts d'index ... 1198
52. Index GiST ... 1201
52.1. Introduction ... 1201
52.2. Extensibilité ... 1201

52.3. Implantation ... 1201
52.4. Exemples ... 1206
52.5. Récupération après un arrêt brutal ... 1207
53. Index GIN ... 1208
53.1. Introduction ... 1208
53.2. Extensibilité ... 1208
53.3. Implantation ... 1209
53.4. Conseils et astuces GIN ... 1209
53.5. Limitations ... 1210
53.6. Exemples ... 1210
54. Stockage physique de la base de données ... 1212
54.1. Emplacement des fichiers de la base de données ... 1212
54.2. TOAST ... 1213
54.3. Carte des espaces libres ... 1214
54.4. Carte de visibilité ... 1215
54.5. Emplacement des pages de la base de données ... 1215
55. Interface du moteur, BKI ... 1218
55.1. Format des fichiers BKI ... 1218
55.2. Commandes BKI ... 1218
55.3. Structure du fichier BKI de « bootstrap » ... 1219
55.4. Exemple ... 1219
56. Comment le planificateur utilise les statistiques ... 1220
56.1. Exemples d'estimation des lignes ... 1220
VIII. Annexes ... 1225
A. Codes d'erreurs de PostgreSQL™ ... 1226
B. Support de date/heure ... 1233
B.1. Interprétation des Date/Heure saisies ... 1233
B.2. Mots clés Date/Heure ... 1234
B.3. Fichiers de configuration date/heure ... 1235
B.4. Histoire des unités ... 1235
C. Mots-clé SQL ... 1237
D. Conformité SQL ... 1257
D.1. Fonctionnalités supportées ... 1258
D.2. Fonctionnalités non supportées ... 1266
E. Notes de version ... 1275
E.1. Release 9.0.18 ... 1275
E.2. Release 9.0.17 ... 1277
E.3. Release 9.0.16 ... 1277
E.4. Release 9.0.15 ... 1280
E.5. Release 9.0.14 ... 1281
E.6. Release 9.0.13 ... 1283
E.7. Release 9.0.12 ... 1284
E.8. Release 9.0.11 ... 1285
E.9. Release 9.0.10 ... 1287
E.10. Release 9.0.9 ... 1288
E.11. Release 9.0.8 ... 1290
E.12. Release 9.0.7 ... 1291
E.13. Release 9.0.6 ... 1294
E.14. Release 9.0.5 ... 1296
E.15. Release 9.0.4 ... 1299
E.16. Release 9.0.3 ... 1301
E.17. Release 9.0.2 ... 1302
E.18. Release 9.0.1 ... 1304
E.19. Release 9.0 ... 1305
E.20. Release 8.4.22 ... 1320
E.21. Release 8.4.21 ... 1322
E.22. Release 8.4.20 ... 1323
E.23. Release 8.4.19 ... 1325
E.24. Release 8.4.18 ... 1326
E.25. Release 8.4.17 ... 1327
E.26. Release 8.4.16 ... 1328
E.27. Release 8.4.15 ... 1329
E.28. Release 8.4.14 ... 1331

E.29. Release 8.4.13 1331
E.30. Release 8.4.12 1333
E.31. Release 8.4.11 1334
E.32. Release 8.4.10 1336
E.33. Release 8.4.9 1338
E.34. Release 8.4.8 1340
E.35. Release 8.4.7 1341
E.36. Release 8.4.6 1342
E.37. Release 8.4.5 1344
E.38. Release 8.4.4 1346
E.39. Release 8.4.3 1348
E.40. Release 8.4.2 1350
E.41. Release 8.4.1 1353
E.42. Release 8.4 1354
E.43. Release 8.3.23 1369
E.44. Release 8.3.22 1370
E.45. Release 8.3.21 1372
E.46. Release 8.3.20 1372
E.47. Release 8.3.19 1374
E.48. Release 8.3.18 1375
E.49. Release 8.3.17 1376
E.50. Release 8.3.16 1378
E.51. Release 8.3.15 1380
E.52. Release 8.3.14 1380
E.53. Release 8.3.13 1381
E.54. Release 8.3.12 1383
E.55. Release 8.3.11 1384
E.56. Release 8.3.10 1386
E.57. Release 8.3.9 1387
E.58. Release 8.3.8 1389
E.59. Release 8.3.7 1390
E.60. Release 8.3.6 1392
E.61. Release 8.3.5 1393
E.62. Release 8.3.4 1394
E.63. Release 8.3.3 1396
E.64. Release 8.3.2 1396
E.65. Release 8.3.1 1398
E.66. Release 8.3 1400
E.67. Release 8.2.23 1412
E.68. Release 8.2.22 1413
E.69. Release 8.2.21 1415
E.70. Release 8.2.20 1416
E.71. Release 8.2.19 1416
E.72. Release 8.2.18 1418
E.73. Release 8.2.17 1419
E.74. Release 8.2.16 1420
E.75. Release 8.2.15 1422
E.76. Release 8.2.14 1423
E.77. Release 8.2.13 1424
E.78. Release 8.2.12 1425
E.79. Release 8.2.11 1426
E.80. Release 8.2.10 1427
E.81. Release 8.2.9 1428
E.82. Release 8.2.8 1428
E.83. Release 8.2.7 1429
E.84. Release 8.2.6 1431
E.85. Release 8.2.5 1432
E.86. Release 8.2.4 1433
E.87. Release 8.2.3 1434
E.88. Release 8.2.2 1434
E.89. Release 8.2.1 1435
E.90. Release 8.2 1436
E.91. Release 8.1.23 1447

E.92. Release 8.1.22 1448
E.93. Release 8.1.21 1449
E.94. Release 8.1.20 1450
E.95. Release 8.1.19 1451
E.96. Release 8.1.18 1452
E.97. Release 8.1.17 1453
E.98. Release 8.1.16 1454
E.99. Release 8.1.15 1454
E.100. Release 8.1.14 1455
E.101. Release 8.1.13 1456
E.102. Release 8.1.12 1457
E.103. Release 8.1.11 1458
E.104. Release 8.1.10 1459
E.105. Release 8.1.9 1460
E.106. Release 8.1.8 1460
E.107. Release 8.1.7 1461
E.108. Release 8.1.6 1461
E.109. Release 8.1.5 1462
E.110. Release 8.1.4 1463
E.111. Release 8.1.3 1464
E.112. Release 8.1.2 1465
E.113. Release 8.1.1 1466
E.114. Release 8.1 1467
E.115. Release 8.0.26 1478
E.116. Release 8.0.25 1479
E.117. Release 8.0.24 1480
E.118. Release 8.0.23 1481
E.119. Release 8.0.22 1482
E.120. Release 8.0.21 1482
E.121. Release 8.0.20 1483
E.122. Release 8.0.19 1483
E.123. Release 8.0.18 1484
E.124. Release 8.0.17 1485
E.125. Release 8.0.16 1485
E.126. Release 8.0.15 1486
E.127. Release 8.0.14 1488
E.128. Release 8.0.13 1488
E.129. Release 8.0.12 1489
E.130. Release 8.0.11 1489
E.131. Release 8.0.10 1489
E.132. Release 8.0.9 1490
E.133. Release 8.0.8 1491
E.134. Release 8.0.7 1492
E.135. Release 8.0.6 1493
E.136. Release 8.0.5 1493
E.137. Release 8.0.4 1494
E.138. Release 8.0.3 1495
E.139. Release 8.0.2 1496
E.140. Release 8.0.1 1498
E.141. Release 8.0 1498
E.142. Release 7.4.30 1510
E.143. Release 7.4.29 1510
E.144. Release 7.4.28 1511
E.145. Release 7.4.27 1512
E.146. Release 7.4.26 1513
E.147. Release 7.4.25 1513
E.148. Release 7.4.24 1514
E.149. Release 7.4.23 1514
E.150. Release 7.4.22 1515
E.151. Release 7.4.21 1515
E.152. Release 7.4.20 1516
E.153. Release 7.4.19 1517
E.154. Release 7.4.18 1518

E.155. Release 7.4.17 1518
E.156. Release 7.4.16 1519
E.157. Release 7.4.15 1519
E.158. Release 7.4.14 1519
E.159. Release 7.4.13 1520
E.160. Release 7.4.12 1521
E.161. Release 7.4.11 1521
E.162. Release 7.4.10 1522
E.163. Release 7.4.9 1523
E.164. Release 7.4.8 1523
E.165. Release 7.4.7 1525
E.166. Release 7.4.6 1526
E.167. Release 7.4.5 1527
E.168. Release 7.4.4 1527
E.169. Release 7.4.3 1527
E.170. Release 7.4.2 1528
E.171. Release 7.4.1 1530
E.172. Release 7.4 1531
E.173. Release 7.3.21 1543
E.174. Release 7.3.20 1543
E.175. Release 7.3.19 1544
E.176. Release 7.3.18 1544
E.177. Release 7.3.17 1544
E.178. Release 7.3.16 1545
E.179. Release 7.3.15 1545
E.180. Release 7.3.14 1546
E.181. Release 7.3.13 1547
E.182. Release 7.3.12 1547
E.183. Release 7.3.11 1548
E.184. Release 7.3.10 1548
E.185. Release 7.3.9 1549
E.186. Release 7.3.8 1550
E.187. Release 7.3.7 1550
E.188. Release 7.3.6 1551
E.189. Release 7.3.5 1551
E.190. Release 7.3.4 1552
E.191. Release 7.3.3 1553
E.192. Release 7.3.2 1555
E.193. Release 7.3.1 1556
E.194. Release 7.3 1556
E.195. Release 7.2.8 1566
E.196. Release 7.2.7 1566
E.197. Release 7.2.6 1567
E.198. Release 7.2.5 1567
E.199. Release 7.2.4 1568
E.200. Release 7.2.3 1568
E.201. Release 7.2.2 1569
E.202. Release 7.2.1 1569
E.203. Release 7.2 1570
E.204. Release 7.1.3 1578
E.205. Release 7.1.2 1579
E.206. Release 7.1.1 1579
E.207. Release 7.1 1579
E.208. Release 7.0.3 1583
E.209. Release 7.0.2 1584
E.210. Release 7.0.1 1584
E.211. Release 7.0 1584
E.212. Release 6.5.3 1590
E.213. Release 6.5.2 1590
E.214. Release 6.5.1 1591
E.215. Release 6.5 1591
E.216. Release 6.4.2 1595
E.217. Release 6.4.1 1595

E.218. Release 6.4 ... 1596
E.219. Release 6.3.2 ... 1600
E.220. Release 6.3.1 ... 1600
E.221. Release 6.3 ... 1601
E.222. Release 6.2.1 ... 1604
E.223. Release 6.2 ... 1605
E.224. Release 6.1.1 ... 1607
E.225. Release 6.1 ... 1608
E.226. Release 6.0 ... 1610
E.227. Release 1.09 ... 1611
E.228. Release 1.02 ... 1612
E.229. Release 1.01 ... 1613
E.230. Release 1.0 ... 1615
E.231. Postgres95™ Release 0.03 ... 1616
E.232. Postgres95™ Release 0.02 ... 1617
E.233. Postgres95™ Release 0.01 ... 1618
E.234. Timing Results ... 1618
F. Modules supplémentaires fournis ... 1620
F.1. adminpack ... 1620
F.2. auto_explain ... 1621
F.3. btree_gin ... 1622
F.4. btree_gist ... 1623
F.5. chkpass ... 1623
F.6. citext ... 1624
F.7. cube ... 1625
F.8. dblink ... 1628
F.9. dict_int ... 1654
F.10. dict_xsyn ... 1654
F.11. earthdistance ... 1655
F.12. fuzzystrmatch ... 1657
F.13. hstore ... 1659
F.14. intagg ... 1664
F.15. intarray ... 1665
F.16. isn ... 1667
F.17. lo ... 1670
F.18. ltree ... 1671
F.19. oid2name ... 1677
F.20. pageinspect ... 1680
F.21. passwordcheck ... 1681
F.22. pg_archivecleanup ... 1682
F.23. pgbench ... 1683
F.24. pg_buffercache ... 1688
F.25. pgcrypto ... 1689
F.26. pg_freespacemap ... 1699
F.27. pgrowlocks ... 1700
F.28. pg_standby ... 1701
F.29. pg_stat_statements ... 1704
F.30. pgstattuple ... 1706
F.31. pg_trgm ... 1708
F.32. pg_upgrade ... 1710
F.33. seg ... 1714
F.34. spi ... 1717
F.35. sslinfo ... 1718
F.36. tablefunc ... 1720
F.37. test_parser ... 1728
F.38. tsearch2 ... 1729
F.39. unaccent ... 1730
F.40. uuid-ossp ... 1731
F.41. vacuumlo ... 1733
F.42. xml2 ... 1734
G. Projets externes ... 1738
G.1. Interfaces client ... 1738
G.2. Langages procéduraux ... 1738

G.3. Extensions 1739
H. Dépôt du code source 1740
H.1. Récupérer les sources via Git™ 1740
I. Documentation 1741
I.1. DocBook 1741
I.2. Ensemble d'outils 1741
I.3. Construire la documentation 1744
I.4. Écriture de la documentation 1747
I.5. Guide des styles 1748
J. Acronymes 1750
K. Traduction française 1754
Bibliographie 1756

Préface

Cet ouvrage représente l'adaptation française de la documentation officielle de PostgreSQL™. Celle-ci a été rédigée par les développeurs de PostgreSQL™ et quelques volontaires en parallèle du développement du logiciel. Elle décrit toutes les fonctionnalités officiellement supportées par la dernière version de PostgreSQL™.

Afin de faciliter l'accès aux informations qu'il contient, cet ouvrage est organisé en plusieurs parties. Chaque partie est destinée à une classe précise d'utilisateurs ou à des utilisateurs de niveaux d'expertise différents :

- la Partie I, « Tutoriel » est une introduction informelle destinée aux nouveaux utilisateurs ;
- la Partie II, « Langage SQL » présente l'environnement du langage de requêtes SQL, notamment les types de données, les fonctions et les optimisations utilisateurs. Tout utilisateur de PostgreSQL™ devrait la lire ;
- la Partie III, « Administration du serveur », destinée aux administrateurs PostgreSQL™, décrit l'installation et l'administration du serveur ;
- la Partie IV, « Interfaces client » décrit les interfaces de programmation ;
- la Partie V, « Programmation serveur », destinée aux utilisateurs expérimentés, présente les éléments d'extension du serveur, notamment les types de données et les fonctions utilisateurs ;
- la Partie VI, « Référence » contient la documentation de référence de SQL et des programmes client et serveur. Cette partie est utilisée comme référence par les autres parties ;
- la Partie VII, « Internes » contient diverses informations utiles aux développeurs de PostgreSQL™.

1. Définition de PostgreSQL™

PostgreSQL™ est un système de gestion de bases de données relationnelles objet (ORDBMS) fondé sur *POSTGRES, Version 4.2*™. Ce dernier a été développé à l'université de Californie au département des sciences informatiques de Berkeley. POSTGRES est à l'origine de nombreux concepts qui ne seront rendus disponibles au sein de systèmes de gestion de bases de données commerciaux que bien plus tard.

PostgreSQL™ est un descendant libre du code original de Berkeley. Il supporte une grande partie du standard SQL tout en offrant de nombreuses fonctionnalités modernes :

- requêtes complexes ;
- clés étrangères ;
- triggers ;
- vues ;
- intégrité transactionnelle ;
- contrôle des versions concurrentes (MVCC, acronyme de « MultiVersion Concurrency Control »).

De plus, PostgreSQL™ peut être étendu par l'utilisateur de multiples façons, en ajoutant, par exemple :

- de nouveaux types de données ;
- de nouvelles fonctions ;
- de nouveaux opérateurs ;
- de nouvelles fonctions d'agrégat ;
- de nouvelles méthodes d'indexage ;
- de nouveaux langages de procédure.

Et grâce à sa licence libérale, PostgreSQL™ peut être utilisé, modifié et distribué librement, quel que soit le but visé, qu'il soit privé, commercial ou académique.

2. Bref historique de PostgreSQL™

Le système de bases de données relationnel objet PostgreSQL™ est issu de POSTGRES™, programme écrit à l'université de Californie à Berkeley. Après plus d'une vingtaine d'années de développement, PostgreSQL™ annonce être devenu la base de données libre de référence.

2.1. Le projet POSTGRES™ à Berkeley

Le projet POSTGRES™, mené par le professeur Michael Stonebraker, était sponsorisé par le DARPA (acronyme de *Defense Ad-*

vanced Research Projects Agency), l'ARO (acronyme de *Army Research Office*), la NSF (acronyme de *National Science Foundation*) et ESL, Inc. Le développement de POSTGRES™ a débuté en 1986. Les concepts initiaux du système ont été présentés dans Stonebraker and Rowe, 1986 et la définition du modèle de données initial apparut dans Rowe and Stonebraker, 1987. Le système de règles fût décrit dans Stonebraker, Hanson, Hong, 1987, l'architecture du gestionnaire de stockage dans Stonebraker, 1987.

Depuis, plusieurs versions majeures de POSTGRES™ ont vu le jour. La première « démo » devint opérationnelle en 1987 et fut présentée en 1988 lors de la conférence ACM-SIGMOD. La version 1, décrite dans Stonebraker, Rowe, Hirohama, 1990, fut livrée à quelques utilisateurs externes en juin 1989. Suite à la critique du premier mécanisme de règles (Stonebraker et al, 1989), celui-ci fut réécrit (Stonebraker et al, ACM, 1990) pour la version 2, présentée en juin 1990. La version 3 apparut en 1991. Elle apporta le support de plusieurs gestionnaires de stockage, un exécuteur de requêtes amélioré et une réécriture du gestionnaire de règles. La plupart des versions qui suivirent, jusqu'à Postgres95™ (voir plus loin), portèrent sur la portabilité et la fiabilité.

POSTGRES™ fût utilisé dans plusieurs applications, en recherche et en production. On peut citer, par exemple : un système d'analyse de données financières, un programme de suivi des performances d'un moteur à réaction, une base de données de suivi d'astéroïdes, une base de données médicale et plusieurs systèmes d'informations géographiques. POSTGRES™ a aussi été utilisé comme support de formation dans plusieurs universités. Illustra Information Technologies (devenu *Informix*™, maintenant détenu par *IBM*) a repris le code et l'a commercialisé. Fin 1992, POSTGRES™ est devenu le gestionnaire de données principal du *projet de calcul scientifique Sequoia 2000*.

La taille de la communauté d'utilisateurs doubla quasiment au cours de l'année 1993. De manière évidente, la maintenance du prototype et le support prenaient un temps considérable, temps qui aurait dû être employé à la recherche en bases de données. Dans un souci de réduction du travail de support, le projet POSTGRES™ de Berkeley se termina officiellement avec la version 4.2.

2.2. Postgres95™

En 1994, Andrew Yu et Jolly Chen ajoutèrent un interpréteur de langage SQL à POSTGRES™. Sous le nouveau nom de Postgres95™, le projet fut publié sur le Web comme descendant libre (OpenSource) du code source initial de POSTGRES™, version Berkeley.

Le code de Postgres95™ était écrit en pur C ANSI et réduit de 25%. De nombreux changements internes améliorèrent les performances et la maintenabilité. Les versions 1.0.x de Postgres95™ passèrent le Wisconsin Benchmark avec des performances meilleures de 30 à 50% par rapport à POSTGRES™, version 4.2. À part les correctifs de bogues, les principales améliorations furent les suivantes :

- le langage PostQUEL est remplacé par SQL (implanté sur le serveur) ; les requêtes imbriquées n'ont pas été supportées avant PostgreSQL™ (voir plus loin) mais elles pouvaient être imitées dans Postgres95™ à l'aide de fonctions SQL utilisateur ; les agrégats furent reprogrammés, la clause GROUP BY ajoutée ;
- un nouveau programme, psql, qui utilise GNU Readline, permet l'exécution interactive de requêtes SQL ; c'est la fin du programme monitor ;
- une nouvelle bibliothèque cliente, `libpgtcl`, supporte les programmes écrits en Tcl ; un shell exemple, **pgtclsh**, fournit de nouvelles commandes Tcl pour interfacer des programmes Tcl avec Postgres95™ ;
- l'interface de gestion des « Large Objects » est réécrite ; jusque-là, le seul mécanisme de stockage de ces objets passait par le système de fichiers Inversion (« Inversion file system ») ; ce système est abandonné ;
- le système de règles d'instance est supprimé ; les règles sont toujours disponibles en tant que règles de réécriture ;
- un bref tutoriel présentant les possibilités du SQL ainsi que celles spécifiques à Postgres95™ est distribué avec les sources ;
- la version GNU de make est utilisée pour la construction à la place de la version BSD ; Postgres95™ peut également être compilé avec un GCC™ sans correctif (l'alignement des doubles est corrigé).

2.3. PostgreSQL™

En 1996, le nom « Postgres95 » commence à mal vieillir. Le nom choisi, PostgreSQL™, souligne le lien entre POSTGRES™ et les versions suivantes qui intégrent le SQL. En parallèle, la version est numérotée 6.0 pour reprendre la numérotation du projet POSTGRES™ de Berkeley.

Beaucoup de personnes font référence à PostgreSQL™ par « Postgres » (il est rare que le nom soit écrit en capitales) par tradition ou parce que c'est plus simple à prononcer. Cet usage est accepté comme alias ou pseudo.

Lors du développement de Postgres95™, l'effort était axé sur l'identification et la compréhension des problèmes dans le code. Avec PostgreSQL™, l'accent est mis sur les nouvelles fonctionnalités, sans pour autant abandonner les autres domaines.

L'historique de PostgreSQL™ à partir de ce moment est disponible dans l'Annexe E, Notes de version.

3. Conventions

Cet ouvrage utilise les conventions typographiques suivantes pour marquer certaines portions du texte : les nouveaux termes, phrases en langue étrangère et autres passages importants sont tous affichés en *italique*. Tout ce qui représente une entrée ou une sortie à l'écran, et en particulier le code, les commandes et les sorties d'écran, est affiché à l'aide d'une police à espacement fixe (`exemple`). À l'intérieur de tels passages, l'italique (*`exemple`*) indique des contenants ; une valeur particulière doit être insérée à la place de ce contenant. Parfois, des bouts de code de programme sont affichés en gras (`exemple`). Cela permet de souligner les ajouts et modifications par rapport à l'exemple qui précéde.

Les conventions suivantes sont utilisées dans le synopsis d'une commande : les crochets (`[` et `]`) indiquent des parties optionnelles. (Dans le synopsis d'une commande Tcl, des points d'interrogation (`?`) sont utilisés, comme c'est habituellement le cas en Tcl.) Les accolades (`{` et `}`) et les barres verticales (`|`) indiquent un choix entre plusieurs options. Les points de suspension (`...`) signifient que l'élément précédent peut être répété.

Lorsque cela améliore la clarté, les commandes SQL sont précédées d'une invite `=>`, tandis que les commandes shell le sont par `$`. Dans le cadre général, les invites ne sont pas indiquées.

Un *administrateur* est généralement une personne en charge de l'installation et de la bonne marche du serveur. Un *utilisateur* est une personne qui utilise ou veut utiliser une partie quelconque du système PostgreSQL™. Ces termes ne doivent pas être pris trop à la lettre ; cet ouvrage n'a pas d'avis figé sur les procédures d'administration système.

4. Pour plus d'informations

En dehors de la documentation, il existe d'autres ressources concernant PostgreSQL™ :

Wiki
: Le *wiki* de PostgreSQL™ contient la *FAQ* (liste des questions fréquemment posées), la liste *TODO* et des informations détaillées sur de nombreux autres thèmes.

Site web
: Le *site web* de PostgreSQL™ contient des détails sur la dernière version, et bien d'autres informations pour rendre un travail ou un investissement personnel avec PostgreSQL™ plus productif.

Listes de discussion
: Les listes de discussion constituent un bon endroit pour trouver des réponses à ses questions, pour partager ses expériences avec celles d'autres utilisateurs et pour contacter les développeurs. La consultation du site web de PostgreSQL™ fournit tous les détails.

Soi-même !
: PostgreSQL™ est un projet OpenSource. En tant que tel, le support dépend de la communauté des utilisateurs. Lorsque l'on débute avec PostgreSQL™, on est tributaire de l'aide des autres, soit au travers de la documentation soit par les listes de discussion. Il est important de faire partager à son tour ses connaissances par la lecture des listes de discussion et les réponses aux questions. Lorsque quelque chose est découvert qui ne figurait pas dans la documentation, pourquoi ne pas en faire profiter les autres ? De même lors d'ajout de fonctionnalités au code.

5. Lignes de conduite pour les rapports de bogues

Lorsque vous trouvez un bogue dans PostgreSQL™, nous voulons en entendre parler. Vos rapports de bogues jouent un rôle important pour rendre PostgreSQL™ plus fiable car même avec la plus grande attention, nous ne pouvons pas garantir que chaque partie de PostgreSQL™ fonctionnera sur toutes les plates-formes et dans toutes les circonstances.

Les suggestions suivantes ont pour but de vous former à la saisie d'un rapport de bogue qui pourra ensuite être gérée de façon efficace. Il n'est pas requis de les suivre mais ce serait à l'avantage de tous.

Nous ne pouvons pas promettre de corriger tous les bogues immédiatement. Si le bogue est évident, critique ou affecte un grand nombre d'utilisateurs, il y a de grandes chances pour que quelqu'un s'en charge. Il se peut que nous vous demandions d'utiliser une version plus récente pour vérifier si le bogue est toujours présent. Ou nous pourrions décider que le bogue ne peut être corrigé avant qu'une réécriture massive, que nous avions planifiée, ne soit faite. Ou peut-être est-ce trop difficile et que des choses plus importantes nous attendent. Si vous avez besoin d'aide immédiatement, envisagez l'obtention d'un contrat de support commercial.

5.1. Identifier les bogues

Avant de rapporter un bogue, merci de lire et re-lire la documentation pour vérifier que vous pouvez réellement faire ce que vous essayez de faire. Si ce n'est pas clair, rapportez-le aussi ; c'est un bogue dans la documentation. S'il s'avère que le programme fait différemment de ce qu'indique la documentation, c'est un bogue. Ceci peut inclure les circonstances suivantes, sans s'y limiter :

- Un programme se terminant avec un signal fatal ou un message d'erreur du système d'exploitation qui indiquerait un problème avec le programme. (Un contre-exemple pourrait être le message « disk full », disque plein, car vous devez le régler vous-même.)
- Un programme produit une mauvaise sortie pour une entrée donnée.
- Un programme refuse d'accepter une entrée valide (c'est-à-dire telle que définie dans la documentation).
- Un programme accepte une entrée invalide sans information ou message d'erreur. Mais gardez en tête que votre idée d'entrée invalide pourrait être notre idée d'une extension ou d'une compatibilité avec les pratiques traditionnelles.
- PostgreSQL™ échoue à la compilation, à la construction ou à l'installation suivant les instructions des plateformes supportées.

Ici, « programme » fait référence à un exécutable, pas au moteur du serveur.

Une lenteur ou une absorption des ressources n'est pas nécessairement un bogue. Lisez la documentation ou demandez sur une des listes de discussion pour de l'aide concernant l'optimisation de vos applications. Ne pas se conformer au standard SQL n'est pas nécessairement un bogue sauf si une telle conformité est indiquée explicitement.

Avant de continuer, vérifiez sur la liste des choses à faire ainsi que dans la FAQ pour voir si votre bogue n'est pas déjà connu. Si vous n'arrivez pas à décoder les informations sur la liste des choses à faire, écrivez un rapport. Le minimum que nous puissions faire est de rendre cette liste plus claire.

5.2. Que rapporter ?

Le point le plus important à se rappeler avec les rapports de bogues est de donner tous les faits et seulement les faits. Ne spéculez pas sur ce que vous pensez qui ne va pas, sur ce qu'« il semble faire » ou sur quelle partie le programme a une erreur. Si vous n'êtes pas familier avec l'implémentation, vous vous tromperez probablement et vous ne nous aiderez pas. Et même si vous avez raison, des explications complètes sont un bon supplément mais elles ne doivent pas se substituer aux faits. Si nous pensons corriger le bogue, nous devons toujours le reproduire nous-même. Rapporter les faits stricts est relativement simple (vous pouvez probablement copier/coller à partir de l'écran) mais, trop souvent, des détails importants sont oubliés parce que quelqu'un a pensé qu'ils n'avaient pas d'importance ou que le rapport serait compris.

Les éléments suivants devraient être fournis avec chaque rapport de bogue :

- La séquence exacte des étapes nécessaires pour reproduire le problème *à partir du lancement du programme*. Ceci devrait se suffire ; il n'est pas suffisant d'envoyer une simple instruction **SELECT** sans les commandes **CREATE TABLE** et **INSERT** qui ont précédé, si la sortie devrait dépendre des données contenues dans les tables. Nous n'avons pas le temps de comprendre le schéma de votre base de données. Si nous sommes supposés créer nos propres données, nous allons probablement ne pas voir le problème.

  Le meilleur format pour un test suite à un problème relatif à SQL est un fichier qui peut être lancé via l'interface psql et qui montrera le problème. (Assurez-vous de ne rien avoir dans votre fichier de lancement `~/.psqlrc`.) Un moyen facile pour créer ce fichier est d'utiliser pg_dump pour récupérer les déclarations des tables ainsi que les données nécessaires pour mettre en place la scène. Il ne reste plus qu'à ajouter la requête posant problème. Vous êtes encouragé à minimiser la taille de votre exemple mais ce n'est pas une obligation. Si le bogue est reproductible, nous le trouverons de toute façon.

  Si votre application utilise une autre interface client, telle que PHP, alors essayez d'isoler le problème aux requêtes erronées. Nous n'allons certainement pas mettre en place un serveur web pour reproduire votre problème. Dans tous les cas, rappelez-vous d'apporter les fichiers d'entrée exacts ; n'essayez pas de deviner que le problème se pose pour les « gros fichiers » ou pour les « bases de données de moyenne taille », etc. car cette information est trop inexacte, subjective pour être utile.
- La sortie que vous obtenez. Merci de ne pas dire que cela « ne fonctionne pas » ou s'est « arrêté brutalement ». S'il existe un message d'erreur, montrez-le même si vous ne le comprenez pas. Si le programme se termine avec une erreur du système d'exploitation, dites-le. Même si le résultat de votre test est un arrêt brutal du programme ou un autre souci évident, il pourrait ne pas survenir sur notre plateforme. Le plus simple est de copier directement la sortie du terminal, si possible.

Note

Si vous rapportez un message d'erreur, merci d'obtenir la forme la plus verbeuse de ce message. Avec psql, exécutez `\set VERBOSITY verbose` avant tout. Si vous récupérez le message des traces du serveur, initialisez la variable d'exécution log_error_verbosity avec `verbose` pour que tous les détails soient tracés.

Note

Dans le cas d'erreurs fatales, le message d'erreur rapporté par le client pourrait ne pas contenir toutes les infor-

mations disponibles. Jetez aussi un œil aux traces du serveur de la base de données. Si vous ne conservez pas les traces de votre serveur, c'est le bon moment pour commencer à le faire.

- Il est très important de préciser ce que vous attendez en sortie. Si vous écrivez uniquement « Cette commande m'a donné cette réponse. » ou « Ce n'est pas ce que j'attendais. », nous pourrions le lancer nous-même, analyser la sortie et penser que tout est correct car cela correspond exactement à ce que nous attendions. Nous ne devrions pas avoir à passer du temps pour décoder la sémantique exacte de vos commandes. Tout spécialement, ne vous contentez pas de dire que « Ce n'est pas ce que SQL spécifie/Oracle fait. » Rechercher le comportement correct à partir de SQL n'est pas amusant et nous ne connaissons pas le comportement de tous les autres serveurs de base de données relationnels. (Si votre problème est un arrêt brutal du serveur, vous pouvez évidemment omettre cet élément.)
- Toutes les options en ligne de commande ainsi que les autres options de lancement incluant les variables d'environnement ou les fichiers de configuration que vous avez modifié. Encore une fois, soyez exact. Si vous utilisez une distribution pré-packagée qui lance le serveur au démarrage, vous devriez essayer de retrouver ce que cette distribution fait.
- Tout ce que vous avez fait de différent à partir des instructions d'installation.
- La version de PostgreSQL™. Vous pouvez lancer la commande `SELECT version();` pour trouver la version du serveur sur lequel vous êtes connecté. La plupart des exécutables disposent aussi d'une option `--version` ; `postgres --version` et `psql --version` devraient au moins fonctionner. Si la fonction ou les options n'existent pas, alors votre version est bien trop ancienne et vous devez mettre à jour. Si vous avez lancé une version préparée sous forme de paquets, tel que les RPM, dites-le en incluant la sous-version que le paquet pourrait avoir. Si vous êtes sur une version Git, mentionnez-le en indiquant le hachage du commit.

  Si votre version est antérieure à la 9.0.18, nous allons certainement vous demander de mettre à jour. Beaucoup de corrections de bogues et d'améliorations sont apportées dans chaque nouvelle version, donc il est bien possible qu'un bogue rencontré dans une ancienne version de PostgreSQL™ soit déjà corrigé. Nous ne fournissons qu'un support limité pour les sites utilisant d'anciennes versions de PostgreSQL™ ; si vous avez besoin de plus de support que ce que nous fournissons, considérez l'acquisition d'un contrat de support commercial.

- Informations sur la plate-forme. Ceci inclut le nom du noyau et sa version, bibliothèque C, processeur, mémoires et ainsi de suite. Dans la plupart des cas, il est suffisant de préciser le vendeur et la version mais ne supposez pas que tout le monde sait ce que « Debian » contient ou que tout le monde utilise des i386. Si vous avez des problèmes à l'installation, des informations sur l'ensemble des outils de votre machine (compilateurs, **make**, etc.) sont aussi nécessaires.

N'ayez pas peur si votre rapport de bogue devient assez long. C'est un fait. Il est préférable de rapporter tous les faits la première fois plutôt que nous ayons à vous tirer les vers du nez. D'un autre côté, si vos fichiers d'entrée sont trop gros, il est préférable de demander si quelqu'un souhaite s'y plonger. Voici un *article* qui relève quelques autres conseils sur les rapports de bogues.

Ne passez pas tout votre temps à vous demander quelles modifications apporter pour que le problème s'en aille. Ceci ne nous aidera probablement pas à le résoudre. S'il arrive que le bogue ne peut pas être corrigé immédiatement, vous aurez toujours l'opportunité de chercher ceci et de partager vos trouvailles. De même, encore une fois, ne perdez pas votre temps à deviner pourquoi le bogue existe. Nous le trouverons assez rapidement.

Lors de la rédaction d'un rapport de bogue, merci de choisir une terminologie qui ne laisse pas place aux confusions. Le paquet logiciel en totalité est appelé « PostgreSQL », quelquefois « Postgres » en court. Si vous parlez spécifiquement du serveur, mentionnez-le mais ne dites pas seulement « PostgreSQL a planté ». Un arrêt brutal d'un seul processus serveur est assez différent de l'arrêt brutal du « postgres » père ; merci de ne pas dire que « le serveur a planté » lorsque vous voulez dire qu'un seul processus s'est arrêté, ni vice versa. De plus, les programmes clients tels que l'interface interactive « psql » sont complètement séparés du moteur. Essayez d'être précis sur la provenance du problème : client ou serveur.

5.3. Où rapporter des bogues ?

En général, envoyez vos rapports de bogue à la liste de discussion des rapports de bogue (`<pgsql-bogues@postgresql.org>`). Nous vous demandons d'utiliser un sujet descriptif pour votre courrier électronique, par exemple une partie du message d'erreur.

Une autre méthode consiste à remplir le formulaire web disponible sur le *site web* du projet. Saisir un rapport de bogue de cette façon fait que celui-ci est envoyé à la liste de discussion `<pgsql-bogues@postgresql.org>`.

Si votre rapport de bogue a des implications sur la sécurité et que vous préfèreriez qu'il ne soit pas immédiatement visible dans les archives publiques, ne l'envoyez pas sur `pgsql-bugs`. Les problèmes de sécurité peuvent être rapportés de façon privé sur `<security@postgresql.org>`.

N'envoyez pas de rapports de bogue aux listes de discussion des utilisateurs, comme `<pgsql-sql@postgresql.org>` ou `<pgsql-general@postgresql.org>`. Ces listes de discussion servent à répondre aux questions des utilisateurs et les abon-

nés ne souhaitent pas recevoir de rapports de bogues. Plus important, ils ont peu de chance de les corriger.

De même, n'envoyez *pas* vos rapports de bogue à la liste de discussion des développeurs `<pgsql-hackers@postgresql.org>`. Cette liste sert aux discussions concernant le développement de PostgreSQL™ et il serait bon de conserver les rapports de bogue séparément. Nous pourrions choisir de discuter de votre rapport de bogue sur `pgsql-hackers` si le problème nécessite que plus de personnes s'en occupent.

Si vous avez un problème avec la documentation, le meilleur endroit pour le rapporter est la liste de discussion pour la documentation `<pgsql-docs@postgresql.org>`. Soyez précis sur la partie de la documentation qui vous déplaît.

Si votre bogue concerne un problème de portabilité sur une plate-forme non supportée, envoyez un courrier électronique à `<pgsql-hackers@postgresql.org>`, pour que nous puissions travailler sur le portage de PostgreSQL™ sur votre plate-forme.

Note

Dû, malheureusement, au grand nombre de pourriels (spam), toutes les adresses de courrier électronique ci-dessus appartiennent à des listes de discussion fermées. Autrement dit, vous devez être abonné pour être autorisé à y envoyer un courrier. Néanmoins, vous n'avez pas besoin de vous abonner pour utiliser le formulaire web de rapports de bogue. Si vous souhaitez envoyer des courriers mais ne pas recevoir le trafic de la liste, vous pouvez vous abonner et configurer l'option `nomail`. Pour plus d'informations, envoyez un courrier à `<majordomo@postgresql.org>` avec le seul mot `help` dans le corps du message.

Partie I. Tutoriel

Bienvenue dans le tutoriel de PostgreSQL™. Les chapitres suivants présentent une courte introduction à PostgreSQL™, aux concepts des bases de données relationnelles et au langage SQL à ceux qui débutent dans l'un de ces domaines. Seules sont nécessaires des connaissances générales sur l'utilisation des ordinateurs. Aucune expérience particulière d'Unix ou de programmation n'est requise. Ce tutoriel a surtout pour but de faire acquérir une expérience pratique des aspects importants du système PostgreSQL™. Il n'est ni exhaustif ni complet, mais introductif.

À la suite de ce tutoriel, la lecture de la Partie II, « Langage SQL » permettra d'acquérir une connaissance plus complète du langage SQL, celle de la Partie IV, « Interfaces client » des informations sur le développement d'applications. La configuration et la gestion sont détaillées dans la Partie III, « Administration du serveur ».

Chapitre 1. Démarrage

1.1. Installation

Avant de pouvoir utiliser PostgreSQL™, vous devez l'installer. Il est possible que PostgreSQL™ soit déjà installé dans votre environnement, soit parce qu'il est inclus dans votre distribution, soit parce que votre administrateur système s'en est chargé. Dans ce cas, vous devriez obtenir les informations nécessaires pour accéder à PostgreSQL™ dans la documentation de votre distribution ou de la part de votre administrateur.

Si vous n'êtes pas sûr que PostgreSQL™ soit déjà disponible ou que vous puissiez l'utiliser pour vos tests, vous avez la possibilité de l'installer vous-même. Le faire n'est pas difficile et peut être un bon exercice. PostgreSQL™ peut être installé par n'importe quel utilisateur sans droit particulier. Aucun accès administrateur (`root`) n'est requis.

Si vous installez PostgreSQL™ vous-même, référez-vous au Chapitre 15, Procédure d'installation de PostgreSQL™ du code source, pour les instructions sur l'installation, puis revenez à ce guide quand l'installation est terminée. Nous vous conseillons de suivre attentivement la section sur la configuration des variables d'environnement appropriées.

Si votre administrateur n'a pas fait une installation par défaut, vous pouvez avoir à effectuer un paramétrage supplémentaire. Par exemple, si le serveur de bases de données est une machine distante, vous aurez besoin de configurer la variable d'environnement `PGHOST` avec le nom du serveur de bases de données. Il sera aussi peut-être nécessaire de configurer la variable d'environnement `PGPORT`. La démarche est la suivante : si vous essayez de démarrer un programme et qu'il se plaint de ne pas pouvoir se connecter à la base de données, vous devez consulter votre administrateur ou, si c'est vous, la documentation pour être sûr que votre environnement est correctement paramétré. Si vous n'avez pas compris le paragraphe précédent, lisez donc la prochaine section.

1.2. Concepts architecturaux de base

Avant de continuer, vous devez connaître les bases de l'architecture système de PostgreSQL™. Comprendre comment les parties de PostgreSQL™ interagissent entre elles rendra ce chapitre un peu plus clair.

Dans le jargon des bases de données, PostgreSQL™ utilise un modèle client/serveur. Une session PostgreSQL™ est le résultat de la coopération des processus (programmes) suivants :

- Un processus serveur, qui gère les fichiers de la base de données, accepte les connexions à la base de la part des applications clientes et effectue sur la base les actions des clients. Le programme serveur est appelé `postgres`.
- L'application cliente (l'application de l'utilisateur), qui veut effectuer des opérations sur la base de données. Les applications clientes peuvent être de nature très différentes : un client peut être un outil texte, une application graphique, un serveur web qui accède à la base de données pour afficher des pages web ou un outil spécialisé dans la maintenance de bases de données. Certaines applications clientes sont fournies avec PostgreSQL™ ; la plupart sont développées par les utilisateurs.

Comme souvent avec les applications client/serveur, le client et le serveur peuvent être sur des hôtes différents. Dans ce cas, ils communiquent à travers une connexion réseau TCP/IP. Vous devez garder cela à l'esprit car les fichiers qui sont accessibles sur la machine cliente peuvent ne pas l'être (ou l'être seulement en utilisant des noms de fichiers différents) sur la machine exécutant le serveur de bases de données.

Le serveur PostgreSQL™ peut traiter de multiples connexions simultanées depuis les clients. Dans ce but, il démarre un nouveau processus pour chaque connexion. À ce moment, le client et le nouveau processus serveur communiquent sans intervention de la part du processus `postgres` original. Ainsi, le processus serveur maître s'exécute toujours, attendant de nouvelles connexions clientes, tandis que le client et les processus serveurs associés vont et viennent (bien sûr, tout ceci est invisible pour l'utilisateur ; nous le mentionnons ici seulement par exhaustivité).

1.3. Création d'une base de données

Le premier test pour voir si vous pouvez accéder au serveur de bases de données consiste à essayer de créer une base. Un serveur PostgreSQL™ peut gérer plusieurs bases de données. Généralement, une base de données distincte est utilisée pour chaque projet ou pour chaque utilisateur.

Il est possible que votre administrateur ait déjà créé une base pour vous. Il devrait vous avoir dit son nom. Dans ce cas, vous pouvez omettre cette étape et aller directement à la prochaine section.

Pour créer une nouvelle base, nommée `ma_base` dans cet exemple, utilisez la commande suivante :

```
$ createdb ma_base
```

Si cette commande ne fournit aucune réponse, cette étape est réussie et vous pouvez sauter le reste de cette section.

Si vous voyez un message similaire à :

```
createdb: command not found
```

alors PostgreSQL™ n'a pas été installé correctement. Soit il n'a pas été installé du tout, soit le chemin système n'a pas été configuré pour l'inclure. Essayez d'appeler la commande avec le chemin absolu :

```
$ /usr/local/pgsql/bin/createdb ma_base
```

Le chemin sur votre serveur peut être différent. Contactez votre administrateur ou vérifiez dans les instructions d'installation pour corriger la commande.

Voici une autre réponse possible :

```
createdb: could not connect to database postgres: could not connect to server: No such
file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Cela signifie que le serveur n'était pas démarré, ou qu'il n'était pas démarré là où **createdb** l'attendait. Une fois encore, vérifiez les instructions d'installation ou consultez votre administrateur.

Voici encore une autre réponse possible :

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

mais avec votre propre nom de connexion mentionné à la place de joe. Ceci survient si l'administrateur n'a pas créé de compte utilisateur PostgreSQL™ pour vous (les comptes utilisateurs PostgreSQL™ sont distincts de ceux du système d'exploitation). Si vous êtes l'administrateur, la lecture du Chapitre 20, Rôles et droits de la base de données vous expliquera comment créer de tels comptes. Vous aurez besoin de prendre l'identité de l'utilisateur du système d'exploitation sous lequel PostgreSQL™ a été installé (généralement `postgres`) pour créer le compte du premier utilisateur. Cela pourrait aussi signifier que vous avez un nom d'utilisateur PostgreSQL™ qui est différent de celui de votre compte utilisateur du système d'exploitation. Dans ce cas, vous avez besoin d'utiliser l'option `-U` ou de configurer la variable d'environnement `PGUSER` pour spécifier votre nom d'utilisateur PostgreSQL™.

Si vous n'avez pas les droits requis pour créer une base, vous verrez le message suivant :

```
createdb: database creation failed: ERROR:  permission denied to create database
```

Tous les utilisateurs n'ont pas l'autorisation de créer de nouvelles bases de données. Si PostgreSQL™ refuse de créer des bases pour vous, alors il faut que l'administrateur vous accorde ce droit. Consultez votre administrateur si cela arrive. Si vous avez installé vous-même PostgreSQL™, alors vous devez ouvrir une session sous le compte utilisateur que vous avez utilisé pour démarrer le serveur. [1]

Vous pouvez aussi créer des bases de données avec d'autres noms. PostgreSQL™ vous permet de créer un nombre quelconque de bases sur un site donné. Le nom des bases doit avoir comme premier caractère un caractère alphabétique et est limité à 63 octets de longueur. Un choix pratique est de créer une base avec le même nom que votre nom d'utilisateur courant. Beaucoup d'outils utilisent ce nom comme nom par défaut pour la base : cela permet de gagner du temps en saisie. Pour créer cette base, tapez simplement :

```
$ createdb
```

Si vous ne voulez plus utiliser votre base, vous pouvez la supprimer. Par exemple, si vous êtes le propriétaire (créateur) de la base `ma_base`, vous pouvez la détruire en utilisant la commande suivante :

```
$ dropdb ma_base
```

(Pour cette commande, le nom de la base n'est pas par défaut le nom du compte utilisateur. Vous devez toujours en spécifier un.) Cette action supprime physiquement tous les fichiers associés avec la base de données et elle ne peut pas être annulée, donc cela doit se faire avec beaucoup de prévoyance.

createdb(1) et dropdb(1) apportent beaucoup plus d'informations sur **createdb** et **dropdb**.

[1] Quelques explications : les noms d'utilisateurs de PostgreSQL™ sont différents des comptes utilisateurs du système d'exploitation. Quand vous vous connectez à une base de données, vous pouvez choisir le nom d'utilisateur PostgreSQL™ que vous utilisez. Si vous ne spécifiez rien, cela sera par défaut le même nom que votre compte système courant. En fait, il existe toujours un compte utilisateur PostgreSQL™ qui a le même nom que l'utilisateur du système d'exploitation qui a démarré le serveur, et cet utilisateur a toujours le droit de créer des bases. Au lieu de vous connecter au système en tant que cet utilisateur, vous pouvez spécifier partout l'option `-U` pour sélectionner un nom d'utilisateur PostgreSQL™ sous lequel vous connecter.

1.4. Accéder à une base

Une fois que vous avez créé la base, vous pouvez y accéder :

- Démarrez le programme en ligne de commande de PostgreSQL™, appelé *psql*, qui vous permet de saisir, d'éditer et d'exécuter de manière interactive des commandes SQL.
- Utilisez un outil existant avec une interface graphique comme pgAdmin ou une suite bureautique avec un support ODBC ou JDBC pour créer et manipuler une base. Ces possibilités ne sont pas couvertes dans ce tutoriel.
- Écrivez une application personnalisée en utilisant un des nombreux langages disponibles. Ces possibilités sont davantage examinées dans la Partie IV, « Interfaces client ».

Vous aurez probablement besoin de lancer **psql** pour essayer les exemples de ce tutoriel. Pour cela, saisissez la commande suivante :

```
$ psql ma_base
```

Si vous n'indiquez pas le nom de la base, alors **psql** utilisera par défaut le nom de votre compte utilisateur. Vous avez déjà découvert ce principe dans la section précédente en utilisant **createdb**.

Dans **psql**, vous serez accueilli avec le message suivant :

```
psql (9.0.18)
Type "help" for help.

ma_base=>
```

La dernière ligne peut aussi être :

```
ma_base=#
```

Cela veut dire que vous êtes le super-utilisateur de la base de données, ce qui est souvent le cas si vous avez installé PostgreSQL™ vous-même. Être super-utilisateur ou administrateur signifie que vous n'êtes pas sujet aux contrôles d'accès. Concernant ce tutoriel, cela n'a pas d'importance.

Si vous rencontrez des problèmes en exécutant **psql**, alors retournez à la section précédente. Les diagnostiques de **psql** et de **createdb** sont semblables. Si le dernier fonctionnait, alors le premier devrait fonctionner également.

La dernière ligne affichée par **psql** est l'invite. Cela indique que **psql** est à l'écoute et que vous pouvez saisir des requêtes SQL dans l'espace de travail maintenu par **psql**. Essayez ces commandes :

```
ma_base=> SELECT version();
                                 version
 ----------------------------------------------------------------------
 PostgreSQL 9.0.18 on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

ma_base=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

ma_base=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

Le programme **psql** dispose d'un certain nombre de commandes internes qui ne sont pas des commandes SQL. Elles commencent avec le caractère antislash (une barre oblique inverse, « \ »). Par exemple, vous pouvez obtenir de l'aide sur la syntaxe de nombreuses commandes SQL de PostgreSQL™ en exécutant :

```
ma_base=> \h
```

Pour sortir de **psql**, saisissez :

```
ma_base=> \q
```

et **psql** se terminera et vous ramènera à votre shell. Pour plus de commandes internes, saisissez `\?` à l'invite de **psql**. Les possibilités complètes de **psql** sont documentées dans psql(1). Si PostgreSQL™ est installé correctement, vous pouvez aussi exécuter `man psql` à l'invite du shell système pour voir sa documentation. Dans ce tutoriel, nous ne verrons pas ces caractéristiques explicitement mais vous pouvez les utiliser vous-même quand cela vous est utile.

Chapitre 2. Le langage SQL

2.1. Introduction

Ce chapitre fournit un panorama sur la façon d'utiliser SQL pour exécuter des opérations simples. Ce tutoriel est seulement prévu pour vous donner une introduction et n'est, en aucun cas, un tutoriel complet sur SQL. De nombreux livres ont été écrits sur SQL, incluant melt93 et date97. Certaines caractéristiques du langage de PostgreSQL™ sont des extensions de la norme.

Dans les exemples qui suivent, nous supposons que vous avez créé une base de données appelée `ma_base`, comme cela a été décrit dans le chapitre précédent et que vous avez été capable de lancer psql.

Les exemples dans ce manuel peuvent aussi être trouvés dans le répertoire `src/tutorial/` de la distribution source de PostgreSQL™. (Les distributions binaires de PostgreSQL™ pourraient ne pas proposer ces fichiers.) Pour utiliser ces fichiers, commencez par changer de répertoire et lancez make :

```
$ cd ..../src/tutorial
$ make
```

Ceci crée les scripts et compile les fichiers C contenant des fonctions et types définis par l'utilisateur. Puis, pour lancer le tutoriel, faites ce qui suit :

```
$ cd ..../tutorial
$ psql -s ma_base
...

ma_base=> \i basics.sql
```

La commande `\i` de **psql** lit les commandes depuis le fichier spécifié. L'option `-s` vous place dans un mode pas à pas qui fait une pause avant d'envoyer chaque instruction au serveur. Les commandes utilisées dans cette section sont dans le fichier `basics.sql`.

2.2. Concepts

PostgreSQL™ est un *système de gestion de bases de données relationnelles* (SGBDR). Cela signifie que c'est un système pour gérer des données stockées dans des *relations*. Relation est essentiellement un terme mathématique pour *table*. La notion de stockage de données dans des tables est si commune aujourd'hui que cela peut sembler en soi évident mais il y a de nombreuses autres manières d'organiser des bases de données. Les fichiers et répertoires dans les systèmes d'exploitation de type Unix forment un exemple de base de données hiérarchique. Un développement plus moderne est une base de données orientée objets.

Chaque table est un ensemble de *lignes*. Chaque ligne d'une table donnée a le même ensemble de *colonnes* et chaque colonne est d'un type de données particulier. Tandis que les colonnes ont un ordre fixé dans chaque ligne, il est important de se rappeler que SQL ne garantit, d'aucune façon, l'ordre des lignes à l'intérieur de la table (bien qu'elles puissent être explicitement triées pour l'affichage).

Les tables sont groupées dans des bases de données et un ensemble de bases gérées par une instance unique du serveur PostgreSQL™ constitue une *instance* de bases (*cluster* en anglais).

2.3. Créer une nouvelle table

Vous pouvez créer une nouvelle table en spécifiant le nom de la table, suivi du nom de toutes les colonnes et de leur type :

```
CREATE TABLE temps (
    ville           varchar(80),
    t_basse         int,            -- température basse
    t_haute         int,            -- température haute
    prcp            real,           -- précipitation
    date            date
);
```

Vous pouvez saisir cela dans **psql** avec les sauts de lignes. **psql** reconnaîtra que la commande n'est pas terminée jusqu'à arriver à un point-virgule.

Les espaces blancs (c'est-à-dire les espaces, les tabulations et les retours à la ligne) peuvent être librement utilisés dans les commandes SQL. Cela signifie que vous pouvez saisir la commande ci-dessus alignée différemment ou même sur une seule ligne. Deux tirets (« `--` ») introduisent des commentaires. Ce qui les suit est ignoré jusqu'à la fin de la ligne. SQL est insensible à la casse pour les mots-clés et les identifiants excepté quand les identifiants sont entre double guillemets pour préserver leur casse (non fait ci-dessus).

varchar(80) spécifie un type de données pouvant contenir une chaîne de caractères arbitraires de 80 caractères au maximum. int est le type entier normal. real est un type pour les nombres décimaux en simple précision. date devrait s'expliquer de lui-même (oui, la colonne de type date est aussi nommée `date` ; cela peut être commode ou porter à confusion, à vous de choisir).

PostgreSQL™ prend en charge les types SQL standards int, smallint, real, double precision, char(*N*), varchar(*N*), date, time, timestamp et interval ainsi que d'autres types d'utilité générale et un riche ensemble de types géométriques. PostgreSQL™ peut être personnalisé avec un nombre arbitraire de types de données définis par l'utilisateur. En conséquence, les noms des types ne sont pas des mots-clé dans la syntaxe sauf lorsqu'il est requis de supporter des cas particuliers dans la norme SQL.

Le second exemple stockera des villes et leur emplacement géographique associé :

```
CREATE TABLE villes (
    nom            varchar(80),
    emplacement    point
);
```

Le type point est un exemple d'un type de données spécifique à PostgreSQL™.

Pour finir, vous devez savoir que si vous n'avez plus besoin d'une table ou que vous voulez la recréer différemment, vous pouvez la supprimer en utilisant la commande suivante :

```
DROP TABLE nom_table;
```

2.4. Remplir une table avec des lignes

L'instruction **INSERT** est utilisée pour remplir une table avec des lignes :

```
INSERT INTO temps VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Notez que tous les types utilisent des formats d'entrées plutôt évident. Les constantes qui ne sont pas des valeurs numériques simples doivent être habituellement entourées par des guillemets simples (`'`) comme dans l'exemple. Le type date est en réalité tout à fait flexible dans ce qu'il accepte mais, pour ce tutoriel, nous collerons au format non ambigu montré ici.

Le type point demande une paire de coordonnées en entrée comme cela est montré ici :

```
INSERT INTO villes VALUES ('San Francisco', '(-194.0, 53.0)');
```

La syntaxe utilisée jusqu'à maintenant nécessite de se rappeler l'ordre des colonnes. Une syntaxe alternative vous autorise à lister les colonnes explicitement :

```
INSERT INTO temps (ville, t_basse, t_haute, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Vous pouvez lister les colonnes dans un ordre différent si vous le souhaitez ou même omettre certaines colonnes ; par exemple, si la précipitation est inconnue :

```
INSERT INTO temps (date, ville, t_haute, t_basse)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

De nombreux développeurs considèrent que le listage explicite des colonnes est un meilleur style que de compter sur l'ordre implicite.

Merci d'exécuter toutes les commandes vues ci-dessus de façon à avoir des données sur lesquelles travailler dans les prochaines sections.

Vous auriez pu aussi utiliser **COPY** pour charger de grandes quantités de données depuis des fichiers texte. C'est habituellement plus rapide car la commande **COPY** est optimisée pour cet emploi mais elle est moins flexible que **INSERT**. Par exemple :

```
COPY temps FROM '/home/utilisateur/temps.txt';
```

où le nom du fichier source doit être disponible sur la machine qui exécute le processus serveur, et non pas sur le client, car le processus serveur lit le fichier directement. Vous avez plus d'informations sur la commande **COPY** dans COPY(7).

2.5. Interroger une table

Pour retrouver les données d'une table, elle est *interrogée*. Une instruction SQL **SELECT** est utilisée pour faire cela. L'instruction est divisée en liste de sélection (la partie qui liste les colonnes à retourner), une liste de tables (la partie qui liste les tables à partir desquelles les données seront retrouvées) et une qualification optionnelle (la partie qui spécifie les restrictions). Par exemple, pour retrouver toutes les lignes de la table temps, saisissez :

```
SELECT * FROM temps;
```

Ici, * est un raccourci pour « toutes les colonnes ». [1] Donc, le même résultat pourrait être obtenu avec :

```
SELECT ville, t_basse, t_haute, prcp, date FROM temps;
```

Le résultat devrait être ceci :

```
     ville     | t_basse | t_haute | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |   50    | 0.25 | 1994-11-27
 San Francisco |      43 |   57    |    0 | 1994-11-29
 Hayward       |      37 |   54    |      | 1994-11-29
(3 rows)
```

Vous pouvez écrire des expressions, pas seulement des références à de simples colonnes, dans la liste de sélection. Par exemple, vous pouvez faire :

```
SELECT ville, (t_haute+t_basse)/2 AS temp_moy, date FROM temps;
```

Cela devrait donner :

```
     ville     | temp_moy |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Notez comment la clause AS est utilisée pour renommer la sortie d'une colonne (cette clause AS est optionnelle).

Une requête peut être « qualifiée » en ajoutant une clause WHERE qui spécifie les lignes souhaitées. La clause WHERE contient une expression booléenne et seules les lignes pour lesquelles l'expression booléenne est vraie sont renvoyées. Les opérateurs booléens habituels (AND, OR et NOT) sont autorisés dans la qualification. Par exemple, ce qui suit recherche le temps à San Francisco les jours pluvieux :

```
SELECT * FROM temps
    WHERE ville = 'San Francisco' AND prcp > 0.0;
```

Résultat :

```
     ville     | t_basse | t_haute | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

Vous pouvez demander à ce que les résultats d'une requête soient renvoyés dans un ordre trié :

```
SELECT * FROM temps
ORDER BY ville;
```

```
 ville        | t_basse | t_haute | prcp |    date
--------------+---------+---------+------+------------
Hayward       |      37 |      54 |      | 1994-11-29
San Francisco |      43 |      57 |    0 | 1994-11-29
San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

Dans cet exemple, l'ordre de tri n'est pas spécifié complètement, donc vous pouvez obtenir les lignes San Francisco dans n'importe quel ordre. Mais, vous auriez toujours obtenu les résultats affichés ci-dessus si vous aviez fait :

```
SELECT * FROM temps
ORDER BY ville, t_basse;
```

Vous pouvez demander que les lignes dupliquées soient supprimées du résultat d'une requête :

```
SELECT DISTINCT ville
    FROM temps;
```

```
     ville
---------------
 Hayward
 San Francisco
(2 rows)
```

[1] Alors que SELECT * est utile pour des requêtes rapides, c'est généralement considéré comme un mauvais style dans un code en production car l'ajout d'une colonne dans la table changerait les résultats.

De nouveau, l'ordre des lignes résultats pourrait varier. Vous pouvez vous assurer des résultats cohérents en utilisant `DISTINCT` et `ORDER BY` ensemble : [2]

```
SELECT DISTINCT ville
FROM temps
ORDER BY ville;
```

2.6. Jointures entre les tables

Jusqu'ici, nos requêtes avaient seulement consulté une table à la fois. Les requêtes peuvent accéder à plusieurs tables en même temps ou accéder à la même table de façon à ce que plusieurs lignes de la table soient traitées en même temps. Une requête qui consulte plusieurs lignes de la même ou de différentes tables en même temps est appelée requête de *jointure*. Comme exemple, supposez que vous souhaitez comparer la colonne *`ville`* de chaque ligne de la table temps avec la colonne *`nom`* de toutes les lignes de la table villes et que vous choisissez les paires de lignes où ces valeurs correspondent.

Note

Ceci est uniquement un modèle conceptuel. La jointure est habituellement exécutée d'une manière plus efficace que la comparaison de chaque paire de lignes mais c'est invisible pour l'utilisateur.

Ceci sera accompli avec la requête suivante :

```
SELECT *
    FROM temps, villes
    WHERE ville = nom;
```

```
     ville     | t_basse | t_haute | prcp |    date    |     nom       | emplacement
---------------+---------+---------+------+------------+---------------+-------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Deux remarques à propos du résultat :

- Il n'y a pas de lignes pour la ville de Hayward dans le résultat. C'est parce qu'il n'y a aucune entrée correspondante dans la table villes pour Hayward, donc la jointure ignore les lignes n'ayant pas de correspondance avec la table temps. Nous verrons rapidement comment cela peut être résolu.
- Il y a deux colonnes contenant le nom des villes. C'est correct car les listes des colonnes des tables temps et villes sont concaténées. En pratique, ceci est indésirable, vous voudrez probablement lister les colonnes explicitement plutôt que d'utiliser `*` :

  ```
  SELECT ville, t_basse, t_haute, prcp, date, emplacement
      FROM temps, villes
      WHERE ville = nom;
  ```

Exercice : Essayez de déterminer la sémantique de cette requête quand la clause `WHERE` est omise.

Puisque toutes les colonnes ont un nom différent, l'analyseur a automatiquement trouvé à quelle table elles appartiennent. Si des noms de colonnes sont communs entre les deux tables, vous aurez besoin de *qualifier* les noms des colonnes pour préciser celles dont vous parlez. Par exemple :

```
SELECT temps.ville, temps.t_basse, temps.t_haute,
       temps.prcp, temps.date, villes.emplacement
    FROM temps, villes
    WHERE villes.nom = temps.ville;
```

La qualification des noms de colonnes dans une requête de jointure est fréquemment considérée comme une bonne pratique. Cela évite l'échec de la requête si un nom de colonne dupliqué est ajouté plus tard dans une des tables.

Les requêtes de jointure vues jusqu'ici peuvent aussi être écrites sous une autre forme :

```
SELECT *
    FROM temps INNER JOIN villes ON (temps.ville = villes.nom);
```

[2] Dans certains systèmes de bases de données, ceci incluant les anciennes versions de PostgreSQL™, l'implémentation de `DISTINCT` ordonne automatiquement les lignes. Du coup, `ORDER BY` n'est pas nécessaire. Mais, ceci n'est pas requis par le standard SQL et PostgreSQL™ ne vous garantit pas actuellement que `DISTINCT` ordonne les lignes.

Cette syntaxe n'est pas aussi couramment utilisée que les précédentes mais nous la montrons ici pour vous aider à comprendre les sujets suivants.

Maintenant, nous allons essayer de comprendre comment nous pouvons avoir les entrées de Hayward. Nous voulons que la requête parcourt la table temps et que, pour chaque ligne, elle trouve la (ou les) ligne(s) de villes correspondante(s). Si aucune ligne correspondante n'est trouvée, nous voulons que les valeurs des colonnes de la table villes soient remplacées par des « valeurs vides ». Ce genre de requêtes est appelé *jointure externe* (outer join). (Les jointures que nous avons vus jusqu'ici sont des jointures internes -- inner joins). La commande ressemble à cela :

```
SELECT *
    FROM temps LEFT OUTER JOIN villes ON (temps.ville = villes.nom);

     ville     | t_basse | t_haute | prcp |    date    |     nom       | emplacement
---------------+---------+---------+------+------------+---------------+-------------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

Cette requête est appelée une *jointure externe à gauche* (left outer join) parce que la table mentionnée à la gauche de l'opérateur de jointure aura au moins une fois ses lignes dans le résultat tandis que la table sur la droite aura seulement les lignes qui correspondent à des lignes de la table de gauche. Lors de l'affichage d'une ligne de la table de gauche pour laquelle il n'y a pas de correspondance dans la table de droite, des valeurs vides (appelées NULL) sont utilisées pour les colonnes de la table de droite.

Exercice : Il existe aussi des jointures externes à droite et des jointures externes complètes. Essayez de trouver ce qu'elles font.

Nous pouvons également joindre une table avec elle-même. Ceci est appelé une *jointure réflexive*. Comme exemple, supposons que nous voulons trouver toutes les entrées de temps qui sont dans un intervalle de température d'autres entrées de temps. Nous avons donc besoin de comparer les colonnes `t_basse` et `t_haute` de chaque ligne de temps aux colonnes `t_basse` et `t_haute` de toutes les autres lignes de temps. Nous pouvons faire cela avec la requête suivante :

```
SELECT T1.ville, T1.t_basse AS bas, T1.t_haute AS haut,
    T2.ville, T2.t_basse AS bas, T2.t_haute AS haus
    FROM temps T1, temps T2
    WHERE T1.t_basse < T2.t_basse
    AND T1.t_haute > T2.t_haute;

     ville      | bas | haut |     ville     | bas | haut
----------------+-----+------+---------------+-----+------
  San Francisco |  43 |   57 | San Francisco |  46 |   50
  Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Dans cet exemple, nous avons renommé la table temps en `T1` et en `T2` pour être capable de distinguer respectivement le côté gauche et droit de la jointure. Vous pouvez aussi utiliser ce genre d'alias dans d'autres requêtes pour économiser de la frappe, c'est-à-dire :

```
SELECT *
    FROM temps t, villes v
    WHERE t.ville = v.nom;
```

Vous rencontrerez ce genre d'abréviation assez fréquemment.

2.7. Fonctions d'agrégat

Comme la plupart des autres produits de bases de données relationnelles, PostgreSQL™ supporte les *fonctions d'agrégat*. Une fonction d'agrégat calcule un seul résultat à partir de plusieurs lignes en entrée. Par exemple, il y a des agrégats pour calculer le nombre (`count`), la somme (`sum`), la moyenne (`avg`), le maximum (`max`) et le minimum (`min`) d'un ensemble de lignes.

Comme exemple, nous pouvons trouver la température la plus haute parmi les températures basses avec :

```
SELECT max(t_basse) FROM temps;
```

```
 max
-----
  46
(1 row)
```

Si nous voulons connaître dans quelle ville (ou villes) ces lectures se sont produites, nous pouvons essayer :

```
SELECT ville FROM temps WHERE t_basse = max(t_basse);
```

```
FAUX
```

mais cela ne marchera pas puisque l'agrégat `max` ne peut pas être utilisé dans une clause `WHERE` (cette restriction existe parce que la clause `WHERE` détermine les lignes qui seront traitées par l'agrégat ; donc les lignes doivent être évaluées avant que les fonctions d'agrégat ne calculent leur résultat). Cependant, comme cela est souvent le cas, la requête peut être répétée pour arriver au résultat attendu, ici en utilisant une *sous-requête* :

```
SELECT ville FROM temps
    WHERE t_basse = (SELECT max(t_basse) FROM temps);
```

```
     ville
---------------
 San Francisco
(1 row)
```

Ceci est correct car la sous-requête est un calcul indépendant qui traite son propre agrégat séparément à partir de ce qui se passe dans la requête externe.

Les agrégats sont également très utiles s'ils sont combinés avec les clauses `GROUP BY`. Par exemple, nous pouvons obtenir la température la plus haute parmi les températures basses observées dans chaque ville avec :

```
SELECT ville, max(t_basse)
    FROM temps
    GROUP BY ville;
```

```
     ville     | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

ce qui nous donne une ligne par ville dans le résultat. Chaque résultat d'agrégat est calculé avec les lignes de la table correspondant à la ville. Nous pouvons filtrer ces lignes groupées en utilisant `HAVING` :

```
SELECT ville, max(t_basse)
    FROM temps
    GROUP BY ville
    HAVING max(t_basse) < 40;
```

```
  ville  | max
---------+-----
 Hayward |  37
(1 row)
```

ce qui nous donne le même résultat uniquement pour les villes qui ont toutes leurs valeurs de *`t_basse`* en-dessous de 40. Pour finir, si nous nous préoccupons seulement des villes dont le nom commence par « `S` », nous pouvons faire :

```
SELECT ville, max(t_basse)
    FROM temps
    WHERE ville LIKE 'S%'
    GROUP BY ville
    HAVING max(t_basse) < 40;
```

1 L'opérateur `LIKE` fait la correspondance avec un motif ; cela est expliqué dans la Section 9.7, « Correspondance de motif ».

1 Il est important de comprendre l'interaction entre les agrégats et les clauses SQL `WHERE` et `HAVING`. La différence fondamentale entre `WHERE` et `HAVING` est que `WHERE` sélectionne les lignes en entrée avant que les groupes et les agrégats ne soient traités (donc, cette clause contrôle les lignes qui se retrouvent dans le calcul de l'agrégat) tandis que `HAVING` sélectionne les lignes groupées après que les groupes et les agrégats aient été traités. Donc, la clause `WHERE` ne doit pas contenir de fonctions d'agrégat ; cela n'a aucun sens d'essayer d'utiliser un agrégat pour déterminer les lignes en entrée des agrégats. D'un autre côté, la clause `HAVING` contient toujours des fonctions d'agrégat (pour être précis, vous êtes autorisés à écrire une clause `HAVING` qui n'utilise pas d'agrégats mais c'est rarement utilisé. La même condition pourra être utilisée plus efficacement par un `WHERE`).

Dans l'exemple précédent, nous pouvons appliquer la restriction sur le nom de la ville dans la clause `WHERE` puisque cela ne nécessite aucun agrégat. C'est plus efficace que d'ajouter la restriction dans `HAVING` parce que nous évitons le groupement et les calculs d'agrégat pour toutes les lignes qui ont échoué lors du contrôle fait par `WHERE`.

2.8. Mises à jour

Vous pouvez mettre à jour une ligne existante en utilisant la commande **UPDATE**. Supposez que vous découvrez que les températures sont toutes excédentes de deux degrés après le 28 novembre. Vous pouvez corriger les données de la façon suivante :

```
UPDATE temps
    SET t_haute = t_haute - 2,  t_basse = t_basse - 2
    WHERE date > '1994-11-28';
```

Regardez le nouvel état des données :

```
SELECT * FROM temps;

     ville      | t_basse | t_haute | prcp |    date
----------------+---------+---------+------+------------
 San Francisco  |      46 |      50 | 0.25 | 1994-11-27
 San Francisco  |      41 |      55 |    0 | 1994-11-29
 Hayward        |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Suppressions

Les lignes peuvent être supprimées de la table avec la commande **DELETE**. Supposez que vous n'êtes plus intéressé par le temps de Hayward. Vous pouvez faire ce qui suit pour supprimer ses lignes de la table :

```
DELETE FROM temps WHERE ville = 'Hayward';
```

Toutes les entrées de temps pour Hayward sont supprimées.

```
SELECT * FROM temps;
```

```
     ville     | t_basse | t_haute | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

Faire très attention aux instructions de la forme

```
DELETE FROM nom_table;
```

Sans une qualification, **DELETE** supprimera *toutes* les lignes de la table donnée, la laissant vide. Le système le fera sans demander de confirmation !

Chapitre 3. Fonctionnalités avancées

3.1. Introduction

Le chapitre précédent couvre les bases de l'utilisation de SQL pour le stockage et l'accès aux données avec PostgreSQL™. Il est temps d'aborder quelques fonctionnalités avancées du SQL qui simplifient la gestion et empêchent la perte ou la corruption des données. Quelques extensions de PostgreSQL™ sont également abordées.

Ce chapitre fait occasionnellement référence aux exemples disponibles dans le Chapitre 2, Le langage SQL pour les modifier ou les améliorer. Il est donc préférable d'avoir lu ce chapitre. Quelques exemples de ce chapitre sont également disponibles dans `advanced.sql` situé dans le répertoire du tutoriel. De plus, ce fichier contient quelques données à charger pour utiliser l'exemple. Cela n'est pas repris ici (on peut se référer à la Section 2.1, « Introduction » pour savoir comment utiliser ce fichier).

3.2. Vues

Se référer aux requêtes de la Section 2.6, « Jointures entre les tables ». Si la liste des enregistrements du temps et des villes est d'un intérêt particulier pour l'application considérée mais qu'il devient contraignant de saisir la requête à chaque utilisation, il est possible de créer une *vue* avec la requête. De ce fait, la requête est nommée et il peut y être fait référence de la même façon qu'il est fait référence à une table :

```
CREATE VIEW ma_vue AS
    SELECT ville, t_basse, t_haute, prcp, date, emplacement
        FROM temps, villes
        WHERE ville = nom;

SELECT * FROM ma_vue;
```

L'utilisation des vues est un aspect clé d'une bonne conception des bases de données SQL. Les vues permettent d'encapsuler les détails de la structure des tables. Celle-ci peut alors changer avec l'évolution de l'application, tandis que l'interface reste constante.

Les vues peuvent être utilisées dans quasiment toutes les situations où une vraie table est utilisable. De plus, il n'est pas inhabituel de construire des vues reposant sur d'autres vues.

3.3. Clés étrangères

Soient les tables `temps` et `villes` définies dans le Chapitre 2, Le langage SQL. Il s'agit maintenant de s'assurer que personne n'insère de ligne dans la table `temps` qui ne corresponde à une entrée dans la table `villes`. On appelle cela maintenir l'*intégrité référentielle* des données. Dans les systèmes de bases de données simplistes, lorsqu'au moins c'est possible, cela est parfois obtenu par la vérification préalable de l'existence d'un enregistrement correspondant dans la table `villes`, puis par l'insertion, ou l'interdiction, du nouvel enregistrement dans `temps`. Puisque cette approche, peu pratique, présente un certain nombre d'inconvénients, PostgreSQL™ peut se charger du maintien de l'*intégrité référentielle*.

La nouvelle déclaration des tables ressemble alors à ceci :

```
CREATE TABLE villes (
        ville       varchar(80) primary key,
        emplacement point
);

CREATE TABLE temps (
        ville      varchar(80) references villes,
        t_haute int,
        t_basse int,
        prcp       real,
        date       date
);
```

Lors d'une tentative d'insertion d'enregistrement non valide :

```
INSERT INTO temps VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR:  insert or update on table "temps" violates foreign key constraint
"temps_ville_fkey"
DETAIL : Key (ville)=(a) is not present in table "villes".
```

Le comportement des clés étrangères peut être adapté très finement à une application particulière. Ce tutoriel ne va pas plus loin que cet exemple simple. De plus amples informations sont accessibles dans le Chapitre 5, Définition des données. Une utilisation efficace des clés étrangères améliore la qualité des applications accédant aux bases de données. Il est donc fortement conseillé d'apprendre à les utiliser.

3.4. Transactions

Les *transactions* sont un concept fondamental de tous les systèmes de bases de données. Une transaction assemble plusieurs étapes en une seule opération tout-ou-rien. Les états intermédiaires entre les étapes ne sont pas visibles par les transactions concurrentes. De plus, si un échec survient qui empêche le succès de la transaction, alors aucune des étapes n'affecte la base de données.

Si l'on considère, par exemple, la base de données d'une banque qui contient le solde de différents comptes clients et le solde total des dépôts par branches et que l'on veuille enregistrer un virement de 100 euros du compte d'Alice vers celui de Bob, les commandes SQL peuvent ressembler à cela (après simplification) :

```
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom = 'Alice');
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom = 'Bob');
```

Ce ne sont pas les détails des commandes qui importent ici ; le point important est la nécessité de plusieurs mises à jour séparées pour accomplir cette opération assez simple. Les employés de la banque veulent être assurés que, soit toutes les commandes sont effectuées, soit aucune ne l'est. Il n'est pas envisageable que, suite à une erreur du système, Bob reçoive 100 euros qui n'ont pas été débités du compte d'Alice. De la même façon, Alice ne restera pas longtemps une cliente fidèle si elle est débitée du montant sans que celui-ci ne soit crédité sur le compte de Bob. Il est important de garantir que si quelque chose se passe mal, aucune des étapes déjà exécutées n'est prise en compte. Le regroupement des mises à jour au sein d'une *transaction* apporte cette garantie. Une transaction est dite *atomique* : du point de vue des autres transactions, elle passe complètement ou pas du tout.

Il est également nécessaire de garantir qu'une fois la transaction terminée et validée par la base de données, les transactions sont enregistrées définitivement et ne peuvent être perdues, même si une panne survient peu après. Ainsi, si un retrait d'argent est effectué par Bob, il ne faut absolument pas que le débit de son compte disparaisse suite à une panne survenant juste après son départ de la banque. Une base de données transactionnelle garantit que toutes les mises à jour faites lors d'une transaction sont stockées de manière persistante (c'est-à-dire sur disque) avant que la transaction ne soit déclarée validée.

Une autre propriété importante des bases de données transactionnelles est en relation étroite avec la notion de mises à jour atomiques : quand plusieurs transactions sont lancées en parallèle, aucune d'entre elles ne doit être capable de voir les modifications incomplètes effectuées par les autres. Ainsi, si une transaction calcule le total de toutes les branches, inclure le débit de la branche d'Alice sans le crédit de la branche de Bob, ou vice-versa, est une véritable erreur. Les transactions doivent donc être tout-ou-rien, non seulement pour leur effet persistant sur la base de données, mais aussi pour leur visibilité au moment de leur exécution. Les mises à jour faites jusque-là par une transaction ouverte sont invisibles aux autres transactions jusqu'à la fin de celle-ci. À ce moment, toutes les mises à jours deviennent simultanément visibles.

Sous PostgreSQL™, une transaction est déclarée en entourant les commandes SQL de la transaction par les commandes **BEGIN** et **COMMIT**. La transaction bancaire ressemble alors à ceci :

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
-- etc etc
COMMIT;
```

Si, au cours de la transaction, il est décidé de ne pas valider (peut-être la banque s'aperçoit-elle que la balance d'Alice passe en négatif), la commande **ROLLBACK** peut être utilisée à la place de **COMMIT**. Toutes les mises à jour réalisées jusque-là sont alors annulées.

En fait, PostgreSQL™ traite chaque instruction SQL comme si elle était exécutée dans une transaction. En l'absence de commande **BEGIN** explicite, chaque instruction individuelle se trouve implicitement entourée d'un **BEGIN** et (en cas de succès) d'un **COMMIT**. Un groupe d'instructions entourées par **BEGIN** et **COMMIT** est parfois appelé *bloc transactionnel*.

Note

Quelques bibliothèques clientes lancent les commandes **BEGIN** et **COMMIT** automatiquement. L'utilisateur béné-

ficie alors des effets des blocs transactionnels sans les demander. Vérifiez la documentation de l'interface que vous utilisez.

Il est possible d'augmenter la granularité du contrôle des instructions au sein d'une transaction en utilisant des *points de retournement* (*savepoint*). Ceux-ci permettent d'annuler des parties de la transaction tout en validant le reste. Après avoir défini un point de retournement à l'aide de **SAVEPOINT**, les instructions exécutées depuis ce point peuvent, au besoin, être annulées avec **ROLLBACK TO**. Toutes les modifications de la base de données effectuées par la transaction entre le moment où le point de retournement a été défini et celui où l'annulation est demandée sont annulées mais les modifications antérieures à ce point sont conservées.

Le retour à un point de retournement ne l'annule pas. Il reste défini et peut donc être utilisé plusieurs fois. À l'inverse, lorsqu'il n'est plus nécessaire de revenir à un point de retournement particulier, il peut être relâché, ce qui permet de libérer des ressources systèmes. Il faut savoir toutefois que relâcher un point de retournement ou y revenir relâche tous les points de retournement qui ont été définis après.

Tout ceci survient à l'intérieur du bloc de transaction, et n'est donc pas visible par les autres sessions en cours sur la base de données. Si le bloc est validé, et à ce moment-là seulement, toutes les actions validées deviennent immédiatement visibles par les autres sessions, tandis que les actions annulées ne le seront jamais.

Reconsidérant la base de données de la banque, on peut supposer vouloir débiter le compte d'Alice de $100.00, somme à créditer sur le compte de Bob, mais considérer plus tard que c'est le compte de Wally qu'il convient de créditer. À l'aide des points de retournement, cela peut se dérouler ainsi :

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
SAVEPOINT mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
-- oups ... oublions ça et créditons le compte de Wally
ROLLBACK TO mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Wally';
COMMIT;
```

Cet exemple est bien sûr très simplifié mais de nombreux contrôles sont réalisables au sein d'un bloc de transaction grâce à l'utilisation des points de retournement. Qui plus est, **ROLLBACK TO** est le seul moyen de regagner le contrôle d'un bloc de transaction placé dans un état d'annulation par le système du fait d'une erreur. C'est plus rapide que de tout annuler pour tout recommencer.

3.5. Fonctions de fenêtrage

Une *fonction de fenêtrage* effectue un calcul sur un jeu d'enregistrements liés d'une certaine façon à l'enregistrement courant. On peut les rapprocher des calculs réalisables par une fonction d'agrégat mais, contrairement à une fonction d'agrégat, l'utilisation d'une fonction de fenêtrage (de fenêtrage) n'entraîne pas le regroupement des enregistrements traités en un seul. Chaque enregistrement garde son identité propre. En coulisse, la fonction de fenêtrage est capable d'accéder à d'autres enregistrements que l'enregistrement courant du résultat de la requête.

Voici un exemple permettant de comparer le salaire d'un employé avec le salaire moyen de sa division :

```
SELECT nomdep, noemp, salaire, avg(salaire) OVER (PARTITION BY nomdep) FROM salaireemp;
```

```
  nomdep   | noemp | salaire |          avg
-----------+-------+---------+-----------------------
 develop   |    11 |    5200 | 5020.0000000000000000
 develop   |     7 |    4200 | 5020.0000000000000000
 develop   |     9 |    4500 | 5020.0000000000000000
 develop   |     8 |    6000 | 5020.0000000000000000
 develop   |    10 |    5200 | 5020.0000000000000000
 personnel |     5 |    3500 | 3700.0000000000000000
 personnel |     2 |    3900 | 3700.0000000000000000
 ventes    |     3 |    4800 | 4866.6666666666666667
 ventes    |     1 |    5000 | 4866.6666666666666667
 ventes    |     4 |    4800 | 4866.6666666666666667
(10 rows)
```

Les trois premières colonnes viennent directement de la table salaireemp, et il y a une ligne de sortie pour chaque ligne de la table.

La quatrième colonne représente une moyenne calculée sur tous les enregistrements de la table qui ont la même valeur de *nomdep* que la ligne courante. (Il s'agit effectivement de la même fonction que la fonction d'agrégat classique `avg`, mais la clause `OVER` entraîne son exécution en tant que fonction de fenêtrage et son calcul sur le jeu approprié d'enregistrements.)

Un appel à une fonction de fenêtrage contient toujours une clause `OVER` qui suit immédiatement le nom et les arguments de la fonction. C'est ce qui permet de la distinguer syntaxiquement d'une fonction simple ou d'une fonction d'agrégat. La clause `OVER` détermine précisément comment les lignes de la requête sont éclatées pour être traitées par la fonction de fenêtrage. La liste `PARTITION BY` contenue dans la clause `OVER` spécifie la répartition des enregistrements en groupes, ou partitions, qui partagent les mêmes valeurs pour la (les) expression(s) contenue(s) dans la clause `PARTITION BY`. Pour chaque enregistrement, la fonction de fenêtrage est calculée sur les enregistrements qui se retrouvent dans la même partition que l'enregistrement courant.

Bien que la fonction `avg` produise le même résultat quel que soit l'ordre dans lequel elle traite les enregistrements de la partition, ceci n'est pas généralisable à toutes les fonctions de fenêtrage. Lorsque cela s'avère nécessaire, l'ordre peut être contrôlé à l'aide d'une clause `ORDER BY` à l'intérieur de la clause `OVER`. Par exemple :

```
SELECT nomdep, noemp, salaire, rank() OVER (PARTITION BY nomdep ORDER BY salaire DESC)
FROM salaireemp;
```

```
  nomdep   | noemp | salaire| rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 ventes    |     1 |   5000 |    1
 ventes    |     4 |   4800 |    2
 ventes    |     3 |   4800 |    2
(10 rows)
```

On remarque que la fonction `rank` produit un rang numérique dans la partition de l'enregistrement pour chaque valeur différente de l'`ORDER BY`, dans l'ordre défini par la clause `ORDER BY`. `rank` n'a pas besoin de paramètre explicite, puisque son comportement est entièrement déterminé par la clause `OVER`.

Les lignes prises en compte par une fonction de fenêtrage sont celles de la table virtuelle produite par la clause `FROM` de la requête filtrée par ses clauses `WHERE`, `GROUP BY` et `HAVING`, s'il y en a. Par exemple, une ligne rejetée parce qu'elle ne satisfait pas à la condition `WHERE` n'est vue par aucune fonction de fenêtrage. Une requête peut contenir plusieurs de ces fonctions de fenêtrage qui découpent les données de façons différentes, par le biais de clauses `OVER` différentes, mais elles travaillent toutes sur le même jeu d'enregistrements, défini par cette table virtuelle.

`ORDER BY` peut être omis lorsque l'ordre des enregistrements est sans importance. Il est aussi possible d'omettre `PARTITION BY`, auquel cas il n'y a qu'une seule partition, contenant tous les enregistrements.

Il y a un autre concept important associé aux fonctions de fenêtrage : pour chaque enregistrement, il existe un jeu d'enregistrements dans sa partition appelé son *window frame* (cadre de fenêtre). Beaucoup de fonctions de fenêtrage, mais pas toutes, travaillent uniquement sur les enregistrements du *window frame*, plutôt que sur l'ensemble de la partition. Par défaut, si on a précisé une clause `ORDER BY`, la *window frame* contient tous les enregistrements du début de la partition jusqu'à l'enregistrement courant, ainsi que tous les enregistrements suivants qui sont égaux à l'enregistrement courant au sens de la clause `ORDER BY`. Quand `ORDER BY` est omis, la *window frame* par défaut contient tous les enregistrements de la partition. [1] Voici un exemple utilisant `sum` :

```
SELECT salaire, sum(salaire) OVER () FROM salaireemp;
```

```
 salaire|  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
```

[1] Il existe des options pour définir la *window frame* autrement, mais ce tutoriel ne les présente pas. Voir la Section 4.2.8, « Appels de fonction de fenêtrage » pour les détails.

```
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Dans l'exemple ci-dessus, puisqu'il n'y a pas d'`ORDER BY` dans la clause `OVER`, la *window frame* est égale à la partition ; en d'autres termes, chaque somme est calculée sur toute la table, ce qui fait qu'on a le même résultat pour chaque ligne du résultat. Mais si on ajoute une clause `ORDER BY`, on a un résultat très différent :

```
SELECT salaire, sum(salaire) OVER (ORDER BY salaire) FROM salaireemp;
```

```
 salaire|  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Ici, sum est calculé à partir du premier salaire (c'est-à-dire le plus bas) jusqu'au salaire courant, en incluant tous les doublons du salaire courant (remarquez les valeurs pour les salaires identiques).

Les fonctions window ne sont autorisées que dans la liste `SELECT` et la clause `ORDER BY` de la requête. Elles sont interdites ailleurs, comme par exemple dans les clauses `GROUP BY`,`HAVING` et `WHERE`. La raison en est qu'elles sont exécutées après le traitement de ces clauses. Par ailleurs, les fonctions de fenêtrage s'exécutent après les fonctions d'agrégat classiques. Cela signifie qu'il est permis d'inclure une fonction d'agrégat dans les arguments d'une fonction de fenêtrage, mais pas l'inverse.

S'il y a besoin de filtrer ou de grouper les enregistrements après le calcul des fonctions de fenêtrage, une sous-requête peut être utilisée. Par exemple :

```
SELECT nomdep, noemp, salaire, date_embauche
FROM
  (SELECT nomdep, noemp, salaire, date_embauche,
          rank() OVER (PARTITION BY nomdep ORDER BY salaire DESC, noemp) AS pos
     FROM salaireemp
  ) AS ss
WHERE pos < 3;
```

La requête ci-dessus n'affiche que les enregistrements de la requête interne ayant un rang inférieur à 3.

Quand une requête met en jeu plusieurs fonctions de fenêtrage, il est possible d'écrire chacune avec une clause `OVER` différente, mais cela entraîne des duplications de code et augmente les risques d'erreurs si on souhaite le même comportement pour plusieurs fonctions de fenêtrage. À la place, chaque comportement de fenêtrage peut être associé à un nom dans une clause `WINDOW` et ensuite être référencé dans `OVER`. Par exemple :

```
SELECT sum(salaire) OVER w, avg(salaire) OVER w
  FROM salaireemp
  WINDOW w AS (PARTITION BY nomdep ORDER BY salaire DESC);
```

Plus de détails sur les fonctions de fenêtrage sont disponibles dans la Section 4.2.8, « Appels de fonction de fenêtrage », la Section 9.19, « Fonctions Window », la Section 7.2.4, « Traitement de fonctions Window » et la page de référence SELECT(7).

3.6. Héritage

L'héritage est un concept issu des bases de données orientées objet. Il ouvre de nouvelles possibilités intéressantes en conception de bases de données.

Soient deux tables : une table `villes` et une table `capitales`. Les capitales étant également des villes, il est intéressant d'avoir la possibilité d'afficher implicitement les capitales lorsque les villes sont listées. Un utilisateur particulièrement brillant peut écrire

ceci

```
CREATE TABLE capitales (
  nom        text,
  population real,
  altitude   int,    -- (en pied)
  etat       char(2)
);

CREATE TABLE non_capitales (
  nom        text,
  population real,
  altitude   int     -- (en pied)
);

CREATE VIEW villes AS
  SELECT nom, population, altitude FROM capitales
    UNION
  SELECT nom, population, altitude FROM non_capitales;
```

Cela fonctionne bien pour les requêtes, mais la mise à jour d'une même donnée sur plusieurs lignes devient vite un horrible casse-tête.

Une meilleure solution peut être :

```
CREATE TABLE villes (
  nom        text,
  population real,
  altitude   int     -- (en pied)
);

CREATE TABLE capitales (
  etat      char(2)
) INHERITS (villes);
```

Dans ce cas, une ligne de `capitales` *hérite* de toutes les colonnes (*`nom`*, *`population`* et *`altitude`*) de son *parent*, `villes`. Le type de la colonne *`nom`* est text, un type natif de PostgreSQL™ pour les chaînes de caractères à longueur variable. Les capitales d'état ont une colonne supplémentaire, *`etat`*, qui affiche l'état dont elles sont la capitale. Sous PostgreSQL™, une table peut hériter de zéro à plusieurs autres tables.

La requête qui suit fournit un exemple d'extraction des noms de toutes les villes, en incluant les capitales des états, situées à une altitude de plus de 500 pieds :

```
SELECT nom, altitude
  FROM villes
  WHERE altitude > 500;
```

ce qui renvoie :

```
   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

À l'inverse, la requête qui suit récupère toutes les villes qui ne sont pas des capitales et qui sont situées à une altitude d'au moins 500 pieds :

```
SELECT nom, altitude
    FROM ONLY villes
    WHERE altitude > 500;
```

```
   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Ici, `ONLY` avant `villes` indique que la requête ne doit être exécutée que sur la table `villes`, et non pas sur les tables en dessous de `villes` dans la hiérarchie des héritages. La plupart des commandes déjà évoquées -- **SELECT**, **UPDATE** et **DELETE** -

- supportent cette notation (`ONLY`).

> **Note**
>
> Bien que l'héritage soit fréquemment utile, il n'a pas été intégré avec les contraintes d'unicité et les clés étrangères, ce qui limite son utilité. Voir la Section 5.8, « L'héritage » pour plus de détails.

3.7. Conclusion

PostgreSQL™ dispose d'autres fonctionnalités non décrites dans ce tutoriel d'introduction orienté vers les nouveaux utilisateurs de SQL. Ces fonctionnalités sont discutées plus en détails dans le reste de ce livre.

Si une introduction plus approfondie est nécessaire, le lecteur peut visiter le *site web* de PostgreSQL qui fournit des liens vers d'autres ressources.

Partie II. Langage SQL

Cette partie présente l'utilisation du langage SQL au sein de PostgreSQL™. La syntaxe générale de SQL y est expliquée, ainsi que la création des structures de stockage des données, le peuplement de la base et son interrogation. La partie centrale liste les types de données et les fonctions disponibles ainsi que leur utilisation dans les requêtes SQL. Le reste traite de l'optimisation de la base de données en vue d'obtenir des performances idéales.

L'information dans cette partie est présentée pour qu'un utilisateur novice puisse la suivre du début à la fin et obtenir ainsi une compréhension complète des sujets sans avoir à effectuer de fréquents sauts entre les chapitres. Les chapitres sont indépendants. Un utilisateur plus expérimenté pourra, donc, ne consulter que les chapitres l'intéressant. L'information est présentée dans un style narratif par unité thématique. Les lecteurs qui cherchent une description complète d'une commande particulière peuvent se référer à la Partie VI, « Référence ».

Pour profiter pleinement de cette partie, il est nécessaire de savoir se connecter à une base PostgreSQL™ et d'y exécuter des commandes SQL. Les lecteurs qui ne sont pas familiers avec ces prérequis sont encouragés à lire préalablement la Partie I, « Tutoriel ».

Les commandes SQL sont généralement saisies à partir du terminal interactif de PostgreSQL™, psql. D'autres programmes possédant des fonctionnalités similaires peuvent également être utilisés.

Chapitre 4. Syntaxe SQL

Ce chapitre décrit la syntaxe de SQL. Il donne les fondements pour comprendre les chapitres suivants qui iront plus en détail sur la façon dont les commandes SQL sont appliquées pour définir et modifier des données.

Nous avertissons aussi nos utilisateurs, déjà familiers avec le SQL, qu'ils doivent lire ce chapitre très attentivement car il existe plusieurs règles et concepts implémentés différemment suivant les bases de données SQL ou spécifiques à PostgreSQL™.

4.1. Structure lexicale

Une entrée SQL consiste en une séquence de *commandes*. Une commande est composée d'une séquence de *jetons*, terminés par un point-virgule (« ; »). La fin du flux en entrée termine aussi une commande. Les jetons valides dépendent de la syntaxe particulière de la commande.

Un jeton peut être un *mot clé*, un *identificateur*, un *identificateur entre guillemets*, une *constante* ou un symbole de caractère spécial. Les jetons sont normalement séparés par des espaces blancs (espace, tabulation, nouvelle ligne) mais n'ont pas besoin de l'être s'il n'y a pas d'ambiguïté (ce qui est seulement le cas si un caractère spécial est adjacent à des jetons d'autres types).

Par exemple, ce qui suit est (syntaxiquement) valide pour une entrée SQL :

```
SELECT * FROM MA_TABLE;
UPDATE MA_TABLE SET A = 5;
INSERT INTO MA_TABLE VALUES (3, 'salut ici');
```

C'est une séquence de trois commandes, une par ligne (bien que cela ne soit pas requis ; plusieurs commandes peuvent se trouver sur une même ligne et une commande peut se répartir sur plusieurs lignes).

De plus, des *commentaires* peuvent se trouver dans l'entrée SQL. Ce ne sont pas des jetons, ils sont réellement équivalents à un espace blanc.

La syntaxe SQL n'est pas très cohérente en ce qui concerne les jetons identificateurs des commandes et lesquels sont des opérandes ou des paramètres. Les premiers jetons sont généralement le nom de la commande. Dans l'exemple ci-dessus, nous parlons d'une commande « SELECT », d'une commande « UPDATE » et d'une commande « INSERT ». Mais en fait, la commande **UPDATE** requiert toujours un jeton SET apparaissant à une certaine position, et cette variante particulière de **INSERT** requiert aussi un VALUES pour être complète. Les règles précises de syntaxe pour chaque commande sont décrites dans la Partie VI, « Référence ».

4.1.1. identificateurs et mots clés

Les jetons tels que SELECT, UPDATE ou VALUES dans l'exemple ci-dessus sont des exemples de *mots clés*, c'est-à-dire des mots qui ont une signification dans le langage SQL. Les jetons MA_TABLE et A sont des exemples d'*identificateurs*. Ils identifient des noms de tables, colonnes ou d'autres objets de la base de données suivant la commande qui a été utilisée. Du coup, ils sont quelques fois simplement nommés des « noms ». Les mots clés et les identificateurs ont la même structure lexicale, signifiant que quelqu'un ne peut pas savoir si un jeton est un identificateur ou un mot clé sans connaître le langage. Une liste complète des mots clé est disponible dans l'Annexe C, Mots-clé SQL.

Les identificateurs et les mots clés SQL doivent commencer avec une lettre (`a-z`, mais aussi des lettres de marques diacritiques différentes et des lettres non latines) ou un tiret bas (_). Les caractères suivants dans un identificateur ou dans un mot clé peuvent être des lettres, des tirets-bas, des chiffres (`0-9`) ou des signes dollar (`$`). Notez que les signes dollar ne sont pas autorisés en tant qu'identificateur d'après le standard SQL, donc leur utilisation pourrait rendre les applications moins portables. Le standard SQL ne définira pas un mot clé contenant des chiffres ou commençant ou finissant par un tiret bas, donc les identificateurs de cette forme sont sûr de ne pas entrer en conflit avec les futures extensions du standard.

Le système utilise au plus NAMEDATALEN-1 octets d'un identificateur ; les noms longs peuvent être écrits dans des commandes mais ils seront tronqués. Par défaut, NAMEDATALEN vaut 64. Du coup, la taille maximum de l'identificateur est de 63 octets. Si cette limite est problématique, elle peut être élevée en modifiant NAMEDATALEN dans `src/include/pg_config_manual.h`.

Les mots clés et les identificateurs sans guillemets doubles sont insensibles à la casse. Du coup :

```
UPDATE MA_TABLE SET A = 5;
```

peut aussi s'écrire de cette façon :

```
uPDaTE ma_TabLE SeT a = 5;
```

Une convention couramment utilisée revient à écrire les mots clés en majuscule et les noms en minuscule, c'est-à-dire :

```
UPDATE ma_table SET a = 5;
```

Voici un deuxième type d'identificateur : l'*identificateur délimité* ou l'*identificateur entre guillemets*. Il est formé en englobant une séquence arbitraire de caractères entre des guillemets doubles (`"`). Un identificateur délimité est toujours un identificateur, jamais un mot clé. Donc, `"select"` pourrait être utilisé pour faire référence à une colonne ou à une table nommée « select », alors qu'un `select` sans guillemets sera pris pour un mot clé et du coup, pourrait provoquer une erreur d'analyse lorsqu'il est utilisé alors qu'un nom de table ou de colonne est attendu. L'exemple peut être écrit avec des identificateurs entre guillemets comme ceci :

```
UPDATE "ma_table" SET "a" = 5;
```

Les identificateurs entre guillemets peuvent contenir tout caractère autre que celui de code 0. (Pour inclure un guillemet double, écrivez deux guillemets doubles.) Ceci permet la construction de noms de tables et de colonnes qui ne seraient pas possible autrement, comme des noms contenant des espaces ou des arobases. La limitation de la longueur s'applique toujours.

Une variante des identifiants entre guillemets permet d'inclure des caractères Unicode échappés en les identifiant par leur code. Cette variante commence par `U&` (U en majuscule ou minuscule suivi par un « et ») immédiatement suivi par un guillemet double d'ouverture, sans espace entre eux. Par exemple `U&"foo"`. (Notez que c'est source d'ambiguïté avec l'opérateur `&`. Utilisez les espaces autour de l'opérateur pouréviter ce problème.) À l'intérieur des guillemets, les caractères Unicode peuvent être indiqués dans une forme échappée en écrivant un antislash suivi par le code hexadécimal sur quatre chiffres ou, autre possibilité, un antislash suivi du signe plus suivi d'un code hexadécimal sur six chiffres. Par exemple, l'identifiant `"data"` peut être écrit ainsi :

```
U&"d\0061t\+000061"
```

L'exemple suivant, moins trivial, écrit le mot russe « slon » (éléphant) en lettres cyrilliques :

```
U&"\0441\043B\043E\043D"
```

Si un caractère d'échappement autre que l'antislash est désiré, il peut être indiqué en utilisant la clause `UESCAPE` après la chaîne. Par exemple :

```
U&"d!0061t!+000061" UESCAPE '!'
```

La chaîne d'échappement peut être tout caractère simple autre qu'un chiffre hexadécimal, le signe plus, un guillemet simple ou double, ou un espace blanc. Notez que le caractère d'échappement est écrit entre guillemets simples, pas entre guillemets doubles.

Pour inclure le caractère d'échappement dans l'identificateur sans interprétation, écrivez-le deux fois.

La syntaxe d'échappement Unicode fonctionne seulement quand l'encodage serveur est `UTF8`. Quand d'autres encodages clients sont utilisés, seuls les codes dans l'échelle ASCII (jusqu'à `\007F`) peuvent être utilisés. La forme sur quatre chiffres et la forme sur six chiffres peuvent être utilisées pour indiquer des paires UTF-16 composant ainsi des caractères comprenant des points de code plus grands que U+FFFF (et ce, bien que la disponibilité de la forme sur six chiffres ne le nécessite pas techniquement). (Quand des paires de substitution sont utilisée et que l'encodage du serveur est `UTF8`, elles sont tout d'abord combinées en un point code seul qui est ensuite encodé en UTF-8.)

Mettre un identificateur entre guillemets le rend sensible à la casse alors que les noms sans guillemets sont toujours convertis en minuscules. Par exemple, les identificateurs `FOO`, `foo` et `"foo"` sont considérés identiques par PostgreSQL™ mais `"Foo"` et `"FOO"` sont différents des trois autres et entre eux. La mise en minuscule des noms sans guillemets avec PostgreSQL™ n'est pas compatible avec le standard SQL qui indique que les noms sans guillemets devraient être mis en majuscule. Du coup, `foo` devrait être équivalent à `"FOO"` et non pas à `"foo"` en respect avec le standard. Si vous voulez écrire des applications portables, nous vous conseillons de toujours mettre entre guillemets un nom particulier ou de ne jamais le mettre.

4.1.2. Constantes

Il existe trois *types implicites de constantes* dans PostgreSQL™ : les chaînes, les chaînes de bits et les nombres. Les constantes peuvent aussi être spécifiées avec des types explicites, ce qui peut activer des représentations plus précises et gérées plus efficacement par le système. Les constantes implicites sont décrites ci-dessous ; ces constantes sont discutées dans les sous-sections suivantes.

4.1.2.1. Constantes de chaînes

Une constante de type chaîne en SQL est une séquence arbitraire de caractères entourée par des guillemets simples (`'`), par exemple `'Ceci est une chaîne'`. Pour inclure un guillemet simple dans une chaîne constante, saisissez deux guillemets simples adjacents, par exemple `'Le cheval d''Anne'`. Notez que ce n'est *pas* identique à un guillemet double (`"`).

Deux constantes de type chaîne séparées par un espace blanc *avec au moins une nouvelle ligne* sont concaténées et traitées réellement comme si la chaîne avait été écrite dans une constante. Par exemple :

```
SELECT 'foo'
'bar';
```

est équivalent à :

```
SELECT 'foobar';
```

mais :

```
SELECT 'foo'      'bar';
```

n'a pas une syntaxe valide (ce comportement légèrement bizarre est spécifié par le standard SQL ; PostgreSQL™ suit le standard).

4.1.2.2. Constantes chaîne avec des échappements de style C

PostgreSQL™ accepte aussi les constantes de chaîne utilisant des échappements qui sont une extension au standard SQL. Une constante de type chaîne d'échappement est indiquée en écrivant la lettre `E` (en majuscule ou minuscule) juste avant le guillemet d'ouverture, par exemple `E'foo'`. (Pour continuer une constante de ce type sur plusieurs lignes, écrire `E` seulement avant le premier guillemet d'ouverture.) À l'intérieur d'une chaîne d'échappement, un caractère antislash (`\`) est géré comme une séquence d'*échappement avec antislash* du langage C. La combinaison d'antislash et du (ou des) caractère(s) suivant représente une valeur spéciale, comme indiqué dans le Tableau 4.1, « Séquences d'échappements avec antislash ».

Tableau 4.1. Séquences d'échappements avec antislash

Séquence d'échappement avec antislash	Interprétation
`\b`	suppression
`\f`	retour en début de ligne
`\n`	saut de ligne
`\r`	saut de ligne
`\t`	tabulation
`\`*o*, `\`*oo*, `\`*ooo* (*o* = 0 - 7)	valeur octale
`\x`*h*, `\x`*hh* (*h* = 0 - 9, A - F)	valeur hexadécimale
`\u`*xxxx*, `\U`*xxxxxxxx* (*x* = 0 - 9, A - F)	caractère Unicode hexadécimal sur 16 ou 32 bits

Tout autre caractère suivi d'un antislash est pris littéralement. Du coup, pour inclure un caractère antislash, écrivez deux antislashs (`\\`). De plus, un guillemet simple peut être inclus dans une chaîne d'échappement en écrivant `\'`, en plus de la façon normale `''`.

Il est de votre responsabilité que les séquences d'octets que vous créez, tout spécialement lorsque vous utilisez les échappements octaux et hexadécimaux, soient des caractères valides dans l'encodage du jeu de caractères du serveur. Quand l'encodage est UTF-8, alors les échappements Unicode ou l'autre syntaxe d'échappement Unicode, expliqués dans la Section 4.1.2.3, « Constantes de chaînes avec des échappements Unicode », devraient être utilisés. (L'alternative serait de réaliser l'encodage UTF-8 manuellement et d'écrire les octets, ce qui serait très lourd.)

La syntaxe d'échappement Unicode fonctionne complètement mais seulement quand l'encodage du serveur est justement `UTF8`. Lorsque d'autres encodages serveur sont utilisés, seuls les points de code dans l'échelle ASCII (jusqu'à `\u007F`) peuvent être utilisés. La forme sur quatre chiffres et la forme sur six chiffres peuvent être utilisées pour indiquer des paires UTF-16 composant ainsi des caractères comprenant des points de code plus grands que U+FFFF et ce, bien que la disponibilité de la forme sur six chiffres ne le nécessite pas techniquement. (Quand des paires de substitution sont utilisées et que l'encodage du serveur est `UTF8`, elles sont tout d'abord combinées en un point code seul qui est ensuite encodé en UTF-8.)

Attention

Si le paramètre de configuration standard_conforming_strings est désactivé (`off`), alors PostgreSQL™ reconnaît les échappements antislashs dans les constantes traditionnelles de type chaînes et celles échappées. Ceci est nécessaire pour la compatibilité ascendante avec le comportement historique pour lequel les échappements à base d'antislash étaient toujours reconnus. Bien que `standard_conforming_strings` est désactivée par défaut, cette valeur par défaut vaudra `on` dans une prochaine version pour améliorer le respect des standards. Du coup, la migration des applications vers un meilleur espect des standards est fortement conseillé. Si vous avez besoin d'utiliser un échappement antislash pour représenter un caractère spécial, écrivez la constante avec un `E` pour vous assurer qu'elle sera bien gérée dans les prochaines versions.

En plus de `standard_conforming_strings`, les paramètres de configuration escape_string_warning et

backslash_quote imposent le traitement des antislashs dans les constantes de type chaîne.

Le caractère de code zéro ne peut pas être placé dans une constante de type chaîne.

4.1.2.3. Constantes de chaînes avec des échappements Unicode

PostgreSQL™ supporte aussi un autre type de syntaxe d'échappement pour les chaînes qui permettent d'indiquer des caractères Unicode arbitraires par code. Une constante de chaîne d'échappement Unicode commence avec `U&` (U en majuscule ou minuscule suivi par un « et commercial ») immédiatement suivi par un guillemet double d'ouverture, sans espace entre eux. Par exemple `U&"foo"`. (Notez que c'est source d'ambiguïté avec l'opérateur `&`. Utilisez les espaces autour de l'opérateur pour éviter ce problème.) À l'intérieur des guillemets, les caractères Unicode peuvent être indiqués dans une forme échappée en écrivant un antislash suivi par le code hexadécimal sur quatre chiffres ou, autre possibilité, un antislash suivi du signe plus suivi d'un code hexadécimal sur six chiffres. Par exemple, l'identificateur `'data'` peut être écrit ainsi :

```
U&'d\0061t\+000061'
```

L'exemple suivant, moins trivial, écrit le mot russe « slon » (éléphant) en lettres cyrilliques :

```
U&'\0441\043B\043E\043D'
```

Si un caractère d'échappement autre que l'antislash est souhaité, il peut être indiqué en utilisant la clause `UESCAPE` après la chaîne. Par exemple :

```
U&'d!0061t!+000061' UESCAPE '!'
```

Le caractère d'échappement peut être tout caractère simple autre qu'un chiffre hexadécimal, le signe plus, un guillement simple ou double, ou un espace blanc.

La syntaxe d'échappement Unicode fonctionne seulement quand l'encodage du serveur est `UTF8`. Quand d'autres encodages de serveur sont utilisés, seuls les codes dans l'échelle ASCII (jusqu'à `\007F`) peuvent être utilisés. La forme sur quatre chiffres et la forme sur six chiffres peuvent être utilisées pour indiquer des paires de substitution UTF-16 composant ainsi des caractères comprenant des points de code plus grands que U+FFFF (et ce, bien que la disponibilité de la forme sur six chiffres ne le nécessite pas techniquement). (Quand des paires de substitution sont utilisées avec un encodage serveur `UTF8`, elles sont tout d'abord combinées en un seul point de code, qui est ensuite encodé en UTF-8.)

De plus, la syntaxe d'échappement de l'Unicode pour les constantes de chaînes fonctionne seulement quand le paramètre de configuration standard_conforming_strings est activé. Dans le cas contraire, cette syntaxe est confuse pour les clients qui analysent les instructions SQL au point que cela pourrait amener des injections SQL et des problèmes de sécurité similaires. Si le paramètre est désactivé, cette syntaxe sera rejetée avec un message d'erreur.

Pour inclure le caractère d'échappement littéralement dans la chaîne, écrivez-le deux fois.

4.1.2.4. Constantes de chaînes avec guillemet dollar

Alors que la syntaxe standard pour la spécification des constantes de chaînes est généralement agréable, elle peut être difficile à comprendre quand la chaîne désirée contient un grand nombre de guillemets ou d'antislashs car chacun d'entre eux doit être doublé. Pour permettre la saisie de requêtes plus lisibles dans de telles situations, PostgreSQL™ fournit une autre façon, appelée « guillemet dollar », pour écrire des constantes de chaînes. Une constante de chaîne avec guillemet dollar consiste en un signe dollar (`$`), une « balise » optionnelle de zéro ou plus de caractères, un autre signe dollar, une séquence arbitraire de caractères qui constitue le contenu de la chaîne, un signe dollar, la même balise et un signe dollar. Par exemple, voici deux façons de spécifier la chaîne « Le cheval d'Anne » en utilisant les guillemets dollar :

```
$$Le cheval d'Anne$$
$UneBalise$Le cheval d'Anne$UneBalise$
```

Notez qu'à l'intérieur de la chaîne avec guillemet dollar, les guillemets simples peuvent être utilisés sans devoir être échappés. En fait, aucun caractère à l'intérieur d'une chaîne avec guillemet dollar n'a besoin d'être échappé : le contenu est toujours écrit littéralement. Les antislashs ne sont pas spéciaux, pas plus que les signes dollar, sauf s'ils font partie d'une séquence correspondant à la balise ouvrante.

Il est possible d'imbriquer les constantes de chaînes avec guillemets dollar en utilisant différentes balises pour chaque niveau d'imbrication. Ceci est habituellement utilisé lors de l'écriture de définition de fonctions. Par exemple :

```
$fonction$
BEGIN
  RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
```

```
END;
$fonction$
```

Dans cet exemple, la séquence `$q$[\t\r\n\v\\]$q$` représente une chaîne constante avec guillemet dollar `[\t\r\n\v\\]`, qui sera reconnue quand le corps de la fonction est exécuté par PostgreSQL™. Mais comme la séquence ne correspond pas au délimiteur `$fonction$`, il s'agit juste de quelques caractères à l'intérieur de la constante pour ce qu'en sait la chaîne externe.

La balise d'une chaîne avec guillemets dollar, si elle existe, suit les mêmes règles qu'un identificateur sans guillemets, sauf qu'il ne peut pas contenir de signes dollar. Les balises sont sensibles à la casse, du coup `$balise$Contenu de la chaîne$balise$` est correct mais `$BALISE$Contenu de la chaîne$balise$` ne l'est pas.

Une chaîne avec guillemets dollar suivant un mot clé ou un identificateur doit en être séparé par un espace blanc ; sinon, le délimiteur du guillemet dollar serait pris comme faisant parti de l'identificateur précédent.

Le guillemet dollar ne fait pas partie du standard SQL mais c'est un moyen bien plus agréable pour écrire des chaînes constantes que d'utiliser la syntaxe des guillemets simples, bien que compatible avec le standard. Elle est particulièrement utile pour représenter des constantes de type chaîne à l'intérieur d'autres constantes, comme cela est souvent le cas avec les définitions de fonctions. Avec la syntaxe des guillemets simples, chaque antislash dans l'exemple précédent devrait avoir été écrit avec quatre antislashs, ce qui sera réduit à deux antislashs dans l'analyse de la constante originale, puis à un lorsque la constante interne est analysée de nouveau lors de l'exécution de la fonction.

4.1.2.5. Constantes de chaînes de bits

Les constantes de chaînes de bits ressemblent aux constantes de chaînes standards avec un `B` (majuscule ou minuscule) juste avant le guillemet du début (sans espace blanc), c'est-à-dire `B'1001'`. Les seuls caractères autorisés dans les constantes de type chaîne de bits sont `0` et `1`.

Les constantes de chaînes de bits peuvent aussi être spécifiées en notation hexadécimale en utilisant un `X` avant (minuscule ou majuscule), c'est-à-dire `X'1FF'`. Cette notation est équivalente à une constante de chaîne de bits avec quatre chiffres binaires pour chaque chiffre hexadécimal.

Les deux formes de constantes de chaînes de bits peuvent être continuées sur plusieurs lignes de la même façon que les constantes de chaînes habituelles. Le guillemet dollar ne peut pas être utilisé dans une constante de chaîne de bits.

4.1.2.6. Constantes numériques

Les constantes numériques sont acceptées dans ces formes générales :

```
chiffres
chiffres.[chiffres][e[+-]chiffres]
[chiffres].chiffres[e[+-]chiffres]
chiffresE[+-]chiffres
```

où *`chiffres`* est un ou plusieurs chiffres décimaux (de 0 à 9). Au moins un chiffre doit être avant ou après le point décimal, s'il est utilisé. Au moins un chiffre doit suivre l'indicateur d'exponentiel (`e`), s'il est présent. Il ne peut pas y avoir d'espaces ou d'autres caractères imbriqués dans la constante. Notez que tout signe plus ou moins en avant n'est pas considéré comme faisant part de la constante ; il est un opérateur appliqué à la constante.

Voici quelques exemples de constantes numériques valides :

```
42
3.5
4.
.001
5e2
1.925e-3
```

Une constante numérique ne contenant ni un point décimal ni un exposant est tout d'abord présumée du type integer si sa valeur est contenue dans le type integer (32 bits) ; sinon, il est présumé de type bigint si sa valeur entre dans un type bigint (64 bits) ; sinon, il est pris pour un type numeric. Les constantes contenant des poins décimaux et/ou des exposants sont toujours présumées de type numeric.

Le type de données affecté initialement à une constante numérique est seulement un point de départ pour les algorithmes de résolution de types. Dans la plupart des cas, la constante sera automatiquement convertie dans le type le plus approprié suivant le contexte. Si nécessaire, vous pouvez forcer l'interprétation d'une valeur numérique sur un type de données spécifique en la convertissant. Par exemple, vous pouvez forcer une valeur numérique à être traitée comme un type real (float4) en écrivant :

```
REAL '1.23'  -- style chaîne
```

```
1.23::REAL   -- style PostgreSQL (historique)
```

Ce sont en fait des cas spéciaux des notations de conversion générales discutées après.

4.1.2.7. Constantes d'autres types

Une constante de type arbitraire peut être saisie en utilisant une des notations suivantes :

```
type 'chaîne'
'chaîne'::type
CAST ( 'chaîne' AS type )
```

Le texte de la chaîne constante est passé dans la routine de conversion pour le type appelé *type*. Le résultat est une constante du type indiqué. La conversion explicite de type peut être omise s'il n'y a pas d'ambiguïté sur le type de la constante (par exemple, lorsqu'elle est affectée directement à une colonne de la table), auquel cas elle est convertie automatiquement.

La constante chaîne peut être écrite en utilisant soit la notation SQL standard soit les guillemets dollar.

Il est aussi possible de spécifier une conversion de type en utilisant une syntaxe style fonction :

```
nom_type ( 'chaîne' )
```

mais tous les noms de type ne peuvent pas être utilisés ainsi ; voir la Section 4.2.9, « Conversions de type » pour plus de détails.

Les syntaxes `::`, `CAST()` et d'appels de fonctions sont aussi utilisables pour spécifier les conversions de type à l'exécution d'expressions arbitraires, comme discuté dans la Section 4.2.9, « Conversions de type ». Pour éviter une ambiguïté syntaxique, la syntaxe *type* `'chaîne'` peut seulement être utilisée pour spécifier le type d'une constante. Une autre restriction sur la syntaxe *type* `'chaîne'` est qu'il ne fonctionne pas pour les types de tableau ; utilisez `::` ou `CAST()` pour spécifier le type d'une constante de type tableau.

La syntaxe de `CAST()` est conforme au standard SQL. La syntaxe *type* `'chaine'` est une généralisation du standard : SQL spécifie cette syntaxe uniquement pour quelques types de données mais PostgreSQL™ l'autorise pour tous les types. La syntaxe `::` est un usage historique dans PostgreSQL™, comme l'est la syntaxe d'appel de fonction.

4.1.3. Opérateurs

Un nom d'opérateur est une séquence d'au plus NAMEDATALEN-1 (63 par défaut) caractères provenant de la liste suivante :

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

Néanmoins, il existe quelques restrictions sur les noms d'opérateurs :

- `--` et `/*` ne peuvent pas apparaître quelque part dans un nom d'opérateur car ils seront pris pour le début d'un commentaire.
- Un nom d'opérateur à plusieurs caractères ne peut pas finir avec `+` ou `-`, sauf si le nom contient aussi un de ces caractères :

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  Par exemple, `@-` est un nom d'opérateur autorisé mais `*-` ne l'est pas. Cette restriction permet à PostgreSQL™ d'analyser des requêtes compatibles avec SQL sans requérir des espaces entre les jetons.

Lors d'un travail avec des noms d'opérateurs ne faisant pas partie du standard SQL, vous aurez habituellement besoin de séparer les opérateurs adjacents avec des espaces pour éviter toute ambiguïté. Par exemple, si vous avez défini un opérateur unaire gauche nommé `@`, vous ne pouvez pas écrire `X*@Y` ; vous devez écrire `X* @Y` pour vous assurer que PostgreSQL™ le lit comme deux noms d'opérateurs, et non pas comme un seul.

4.1.4. Caractères spéciaux

Quelques caractères non alphanumériques ont une signification spéciale, différente de celui d'un opérateur. Les détails sur leur utilisation sont disponibles à l'endroit où l'élément de syntaxe respectif est décrit. Cette section existe seulement pour avertir de leur existence et pour résumer le but de ces caractères.

- Un signe dollar ($) suivi de chiffres est utilisé pour représenter un paramètre de position dans le corps de la définition d'une fonction ou d'une instruction préparée. Dans d'autres contextes, le signe dollar pourrait faire partie d'un identificateur ou d'une constante de type chaîne utilisant le dollar comme guillemet.
- Les parenthèses (`()`) ont leur signification habituelle pour grouper leurs expressions et renforcer la précédence. Dans certains cas, les parenthèses sont requises car faisant partie de la syntaxe d'une commande SQL particulière.
- Les crochets (`[]`) sont utilisés pour sélectionner les éléments d'un tableau. Voir la Section 8.14, « Tableaux » pour plus

d'informations sur les tableaux.

- Les virgules (`,`) sont utilisées dans quelques constructions syntaxiques pour séparer les éléments d'une liste.
- Le point-virgule (`;`) termine une commande SQL. Il ne peut pas apparaître quelque part dans une commande, sauf à l'intérieur d'une constante de type chaîne ou d'un identificateur entre guillemets.
- Le caractère deux points (`:`) est utilisé pour sélectionner des « morceaux » de tableaux (voir la Section 8.14, « Tableaux »). Dans certains dialectes SQL (tel que le SQL embarqué), il est utilisé pour préfixer les noms de variables.
- L'astérisque (`*`) est utilisé dans certains contextes pour indiquer tous les champs de la ligne d'une table ou d'une valeur composite. Elle a aussi une signification spéciale lorsqu'elle est utilisée comme argument d'une fonction d'agrégat. Cela signifie que l'agrégat ne requiert par de paramètre explicite.
- Le point (`.`) est utilisé dans les constantes numériques et pour séparer les noms de schéma, table et colonne.

4.1.5. Commentaires

Un commentaire est une séquence de caractères commençant avec deux tirets et s'étendant jusqu'à la fin de la ligne, par exemple :

```
-- Ceci est un commentaire standard en SQL
```

Autrement, les blocs de commentaires style C peuvent être utilisés :

```
/* commentaires multilignes
 * et imbriqués: /* bloc de commentaire imbriqué */
 */
```

où le commentaire commence avec `/*` et s'étend jusqu'à l'occurrence de `*/`. Ces blocs de commentaires s'imbriquent, comme spécifié dans le standard SQL mais pas comme dans le langage C. De ce fait, vous pouvez commenter des blocs importants de code pouvant contenir des blocs de commentaires déjà existants.

Un commentaire est supprimé du flux en entrée avant une analyse plus poussée de la syntaxe et est remplacé par un espace blanc.

4.1.6. Précédence lexicale

Le Tableau 4.2, « Précédence des opérateurs (en ordre décroissant) » affiche la précédence et l'associativité des opérateurs dans PostgreSQL™. La plupart des opérateurs ont la même précédence et sont associatifs par la gauche. La précédence et l'associativité des opérateurs sont codées en dur dans l'analyseur. Ceci pourrait conduire à un comportement non intuitif ; par exemple, les opérateurs booléens < et > ont une précédence différente des opérateurs booléens <= et >=. De même, vous aurez quelque fois besoin d'ajouter des parenthèses lors de l'utilisation de combinaisons d'opérateurs binaires et unaires. Par exemple :

```
SELECT 5 ! - 6;
```

sera analysé comme :

```
SELECT 5 ! (- 6);
```

parce que l'analyseur n'a aucune idée, jusqu'à ce qu'il ne soit trop tard, que ! est défini comme un opérateur suffixe, et non pas préfixe. Pour obtenir le comportement désiré dans ce cas, vous devez écrire :

```
SELECT (5 !) - 6;
```

C'est le prix à payer pour l'extensibilité.

Tableau 4.2. Précédence des opérateurs (en ordre décroissant)

Opérateur/Élément	Associativité	Description
.	gauche	séparateur de noms de table et de colonne
::	gauche	conversion de type, style PostgreSQL™
[ ]	gauche	sélection d'un élément d'un tableau
-	droite	négation unaire
^	gauche	exposant
* / %	gauche	multiplication, division, modulo
+ -	gauche	addition, soustraction
IS		`IS TRUE`, `IS FALSE`, `IS UNKNOWN`, `IS NULL`

Opérateur/Élément	Associativité	Description
ISNULL		test pour NULL
NOTNULL		test pour non NULL
(autres)	gauche	tout autre opérateur natif ou défini par l'utilisateur
IN		appartenance à un ensemble
BETWEEN		compris entre
OVERLAPS		surcharge un intervalle de temps
LIKE ILIKE SIMILAR		correspondance de motifs de chaînes
< >		inférieur, supérieur à
=	droite	égalité, affectation
NOT	droite	négation logique
AND	gauche	conjonction logique
OR	gauche	disjonction logique

Notez que les règles de précédence des opérateurs s'appliquent aussi aux opérateurs définis par l'utilisateur qui ont le même nom que les opérateurs internes mentionnés ici. Par exemple, si vous définissez un opérateur « + » pour un type de données personnalisé, il aura la même précédence que l'opérateur interne « + », peu importe ce que fait le votre.

Lorsqu'un nom d'opérateur qualifié par un schéma est utilisé dans la syntaxe `OPERATOR`, comme par exemple dans :

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

la construction `OPERATOR` est prise pour avoir la précédence par défaut affichée dans le Tableau 4.2, « Précédence des opérateurs (en ordre décroissant) » pour les opérateurs « autres ». Ceci est vrai quelque soit le nom spécifique de l'opérateur apparaissant à l'intérieur de `OPERATOR()`.

4.2. Expressions de valeurs

Les expressions de valeurs sont utilisées dans une grande variété de contextes, tels que dans la liste cible d'une commande **SELECT**, dans les nouvelles valeurs de colonnes d'une commande **INSERT** ou **UPDATE**, ou dans les conditions de recherche d'un certain nombre de commandes. Le résultat d'une expression de valeurs est quelquefois appelé *scalaire*, pour le distinguer du résultat d'une expression de table (qui est une table). Les expressions de valeurs sont aussi appelées des *expressions scalaires* (voire même simplement des *expressions*). La syntaxe d'expression permet le calcul des valeurs à partir de morceaux primitifs en utilisant les opérations arithmétiques, logiques, d'ensemble et autres.

Une expression de valeur peut être :

- une constante ou une valeur constante ;
- une référence de colonne ;
- une référence de la position d'un paramètre, dans le corps d'une définition de fonction ou d'instruction préparée ;
- une expression indicée ;
- une expression de sélection de champs ;
- un appel d'opérateur ;
- un appel de fonction ;
- une expression d'agrégat ;
- un appel de fonction de fenêtrage ;
- une conversion de type ;
- Une sous-requête scalaire.
- un constructeur de tableau ;
- un constructeur de ligne ;
- toute expression de valeur entre parenthèses, utile pour grouper des sous-expressions et surcharger la précédence.

En plus de cette liste, il existe un certain nombre de constructions pouvant être classées comme une expression mais ne suivant aucune règle de syntaxe générale. Elles ont généralement la sémantique d'une fonction ou d'un opérateur et sont expliquées au Chapitre 9, Fonctions et opérateurs. Un exemple est la clause `IS NULL`.

Nous avons déjà discuté des constantes dans la Section 4.1.2, « Constantes ». Les sections suivantes discutent des options restantes.

4.2.1. Références de colonnes

Une colonne peut être référencée avec la forme :

```
correlation.nom_colonne
```

`correlation` est le nom d'une table (parfois qualifié par son nom de schéma) ou un alias d'une table définie au moyen de la clause `FROM`. Le nom de corrélation et le point de séparation peuvent être omis si le nom de colonne est unique dans les tables utilisées par la requête courante (voir aussi le Chapitre 7, Requêtes).

4.2.2. Paramètres de position

Un paramètre de position est utilisé pour indiquer une valeur fournie en externe par une instruction SQL. Les paramètres sont utilisés dans des définitions de fonction SQL et dans les requêtes préparées. Quelques bibliothèques clients supportent aussi la spécification de valeurs de données séparément de la chaîne de commande SQL, auquel cas les paramètres sont utilisés pour référencer les valeurs de données en dehors. Le format d'une référence de paramètre est :

```
$numéro
```

Par exemple, considérez la définition d'une fonction : `dept` :

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE nom = $1 $$
    LANGUAGE SQL;
```

Dans cet exemple, `$1` référence la valeur du premier argument de la fonction à chaque appel de cette commande.

4.2.3. Indices

Si une expression récupère une valeur de type tableau, alors un élément spécifique du tableau peut être extrait en écrivant :

```
expression[indice]
```

Des éléments adjacents (un « morceau de tableau ») peuvent être extraits en écrivant :

```
expression[indice_bas:indice_haut]
```

Les crochets `[ ]` doivent apparaître réellement. Chaque *`indice`* est lui-même une expression, devant contenir une valeur entière.

En général, l'*`expression`* de type tableau doit être entre parenthèses mais ces dernières peuvent être omises lorsque l'expression utilisée comme indice est seulement une référence de colonne ou un paramètre de position. De plus, les indices multiples peuvent être concaténés lorsque le tableau original est multi-dimensionnel. Par exemple :

```
ma_table.colonnetableau[4]
ma_table.colonnes_deux_d[17][34]
$1[10:42]
(fonctiontableau(a,b))[42]
```

Dans ce dernier exemple, les parenthèses sont requises. Voir la Section 8.14, « Tableaux » pour plus d'informations sur les tableaux.

4.2.4. Sélection de champs

Si une expression récupère une valeur de type composite (type row), alors un champ spécifique de la ligne est extrait en écrivant :

```
expression.nom_champ
```

En général, l'*`expression`* de ligne doit être entre parenthèses mais les parenthèses peuvent être omises lorsque l'expression à partir de laquelle se fait la sélection est seulement une référence de table ou un paramètre de position. Par exemple :

```
ma_table.macolonne
$1.unecolonne
```

```
(fonctionligne(a,b)).col3
```

En fait, une référence de colonne qualifiée est un cas spécial de syntaxe de sélection de champ. Un cas spécial important revient à extraire un champ de la colonne de type composite d'une table :

```
(colcomposite).unchamp
(matable.colcomposite).unchamp
```

Les parenthèses sont requises ici pour montrer que *colcomposite* est un nom de colonne, et non pas un nom de table, ou que matable est un nom de table, par un nom de schéma dans le deuxième cas.

4.2.5. Appels d'opérateurs

Il existe trois syntaxes possibles pour l'appel d'un opérateur :

expression opérateur expression (opérateur binaire préfixe)
opérateur expression (opérateur unaire préfixe)
expression opérateur (opérateur unaire suffixe)
où le jeton *opérateur* suit les règles de syntaxe de la Section 4.1.3, « Opérateurs », ou est un des mots clés AND, OR et NOT, ou est un nom d'opérateur qualifié de la forme

```
OPERATOR(schema.nom_operateur)
```

Quel opérateur particulier existe et est-il unaire ou binaire dépend des opérateurs définis par le système ou l'utilisateur. Le Chapitre 9, Fonctions et opérateurs décrit les opérateurs internes.

4.2.6. Appels de fonctions

La syntaxe pour un appel de fonction est le nom d'une fonction (qualifié ou non du nom du schéma) suivi par sa liste d'arguments entre parenthèses :

```
nom_fonction([expression [,expression ...]] )
```

Par exemple, ce qui suit calcule la racine carré de 2 :

```
sqrt(2)
```

La liste des fonctions intégrées se trouve dans le Chapitre 9, Fonctions et opérateurs. D'autres fonctions pourraient être ajoutées par l'utilisateur.

En option, les arguments peuvent avoir leur nom attaché. Voir la Section 4.3, « Fonctions appelantes » pour les détails.

4.2.7. Expressions d'agrégat

Une *expression d'agrégat* représente l'application d'une fonction d'agrégat à travers les lignes sélectionnées par une requête. Une fonction d'agrégat réduit les nombres entrés en une seule valeur de sortie, comme la somme ou la moyenne des valeurs en entrée. La syntaxe d'une expression d'agrégat est une des suivantes :

```
nom_agregat (expression [ , ... ] [ clause_order_by ] )
nom_agregat (ALL expression [ , ... ] [ clause_order_by ] )
nom_agregat (DISTINCT expression [ , ... ] [ clause_order_by ] )
nom_agregat ( * )
```

où *nom_agregat* est un agrégat précédemment défini (parfois qualifié d'un nom de schéma), *expression* est toute expression de valeur qui ne contient pas lui-même une expression d'agrégat ou un appel à une fonction de fenêtrage. *order_by_clause* est une clause `ORDER BY` optionnelle comme décrite ci-dessous.

La première forme d'expression d'agrégat appelle l'agrégat une fois pour chaque ligne en entrée. La seconde forme est identique à la première car `ALL` est une clause active par défaut. La troisième forme fait appel à l'agrégat une fois pour chaque valeur distincte de l'expression (ou ensemble distinct de valeurs, pour des expressions multiples) trouvée dans les lignes en entrée. La dernière forme appelle l'agrégat une fois pour chaque ligne en entrée ; comme aucune valeur particulière en entrée n'est spécifiée, c'est généralement utile pour la fonction d'agrégat `count(*)`.

La plupart des fonctions d'agrégats ignorent les entrées NULL, pour que les lignes qui renvoient une ou plusieurs expressions NULL soient disqualifiées. Ceci peut être considéré vrai pour tous les agrégats internes sauf indication contraire.

Par exemple, `count(*)` trouve le nombre total de lignes en entrée alors que `count(f1)` récupère le nombre de lignes en entrée pour lesquelles `f1` n'est pas NULL. En effet, la fonction `count` ignore les valeurs NULL mais `count(distinct f1)` retrouve le nombre de valeurs distinctes non NULL de `f1`.

D'habitude, les lignes en entrée sont passées à la fonction d'agrégat dans un ordre non spécifié. Dans la plupart des cas, cela n'a pas d'importance. Par exemple, `min` donne le même résultat quelque soit l'ordre dans lequel il reçoit les données. Néanmoins, certaines fonctions d'agrégat (tels que `array_agg` et `string_agg`) donnent un résultat dépendant de l'ordre des lignes en entrée. Lors de l'utilisation de ce type d'agrégat, la clause *`clause_order_by`* peut être utilisée pour préciser l'ordre de tri désiré. La clause *`clause_order_by`* a la même syntaxe que la clause `ORDER BY` d'une requête, qui est décrite dans la Section 7.5, « Tri des lignes », sauf que ses expressions sont toujours des expressions simples et ne peuvent pas être des noms de colonne en sortie ou des numéros. Par exemple :

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Lors de l'utilisation de fonctions d'agrégat à plusieurs arguments, la clause `ORDER BY` arrive après tous les arguments de l'agrégat. Par exemple, il faut écrire ceci :

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

et non pas ceci :

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

Ce dernier exemple est syntaxiquement correct mais il concerne un appel à une fonction d'agrégat à un seul argument avec deux clés pour le `ORDER BY` (le deuxième étant inutile car il est constant).

Si `DISTINCT` est indiqué en plus de la clause *`clause_order_by`*, alors toutes les expressions de l'`ORDER BY` doivent correspondre aux arguments de l'agrégat ; autrement dit, vous ne pouvez pas trier sur une expression qui n'est pas inclus dans la liste `DISTINCT`.

> **Note**
>
> La possibilité de spécifier à la fois `DISTINCT` et `ORDER BY` dans une fonction d'agrégat est une extension de PostgreSQL™.

Les fonctions d'agrégat prédéfinies sont décrites dans la Section 9.18, « Fonctions d'agrégat ». D'autres fonctions d'agrégat pourraient être ajoutées par l'utilisateur.

Une expression d'agrégat peut seulement apparaître dans la liste de résultats ou dans la clause `HAVING` d'une commande **SELECT**. Elle est interdite dans d'autres clauses, tels que `WHERE`, parce que ces clauses sont logiquement évaluées avant que les résultats des agrégats ne soient calculés.

Lorsqu'une expression d'agrégat apparaît dans une sous-requête (voir la Section 4.2.10, « Sous-requêtes scalaires » et la Section 9.20, « Expressions de sous-requêtes »), l'agrégat est normalement évalué sur les lignes de la sous-requête. Cependant, une exception survient si les arguments de l'agrégat contiennent seulement des niveaux externes de variables : ensuite, l'agrégat appartient au niveau externe le plus proche et est évalué sur les lignes de cette requête. L'expression de l'agrégat est une référence externe pour la sous-requête dans laquelle il apparaît et agit comme une constante sur toute évaluation de cette requête. La restriction apparaissant seulement dans la liste de résultat ou dans la clause `HAVING` s'applique avec respect du niveau de requête auquel appartient l'agrégat.

4.2.8. Appels de fonction de fenêtrage

Un *appel de fonction de fenêtrage* représente l'application d'une fonction de type agrégat sur une portion des lignes sélectionnées par une requête. Contrairement aux appels de fonction d'agrégat standard, ce n'est pas lié au groupement des lignes sélectionnées en une seule ligne résultat -- chaque ligne reste séparée dans les résultats. Néanmoins, la fonction de fenêtrage est capable de parcourir toutes les lignes qui font partie du groupe de la ligne courante d'après la spécification du groupe (liste `PARTITION BY`) de l'appel de la fonction de fenêtrage. La syntaxe d'un appel de fonction de fenêtrage est une des suivantes :

```
nom_fonction ([expression [, expression ... ]]) OVER nom_window
nom_fonction ([expression [, expression ... ]]) OVER ( définition_window )
nom_fonction ( * ) OVER nom_window
nom_fonction ( * ) OVER ( définition_window )
```

où *`définition_fenêtrage`* a comme syntaxe :

```
[ nom_fenêtrage_existante ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING opérateur ] [ NULLS { FIRST | LAST } ] [,
...] ]
[ clause_portée ]
```

et la clause *clause_portée* optionnelle fait partie de :

```
[ RANGE | ROWS ] début_portée
[ RANGE | ROWS ] BETWEEN début_portée AND fin_portée
```

avec *début_portée* et *fin_portée* pouvant faire partie de

```
UNBOUNDED PRECEDING
valeur PRECEDING
CURRENT ROW
valeur FOLLOWING
UNBOUNDED FOLLOWING
```

Ici, *expression* représente toute expression de valeur qui ne contient pas elle-mêle d'appel à des fonctions window. Les listes PARTITION BY et ORDER BY ont essentiellement la même syntaxe et la même sémantique que les clauses GROUP BY et ORDER BY de la requête complète, sauf que leur expressions sont toujours seulement des expressions et ne peuvent pas être des noms ou des numéros de colonnes en sortie. *nom_window* est une référence à la spécification d'une window nommée, définie dans la clause WINDOW de la requête. Autrement, une *définition_window* complète peut être donnée dans des parenthèses, en utilisant la même syntaxe que pour la défintion d'une fenêtre nommée dans la clause WINDOW ; voir la page de référence SELECT(7) pour les détails. Il est préférable de préciser que OVER nom_fenêtre n'est pas strictement équivalent à OVER (nom_fenêtre) ; ce dernier signifie la copie et la modification de la définition de la fenêtre, et sera rejeté si la spécification de la fenêtre référencée inclut une clause de portée.

La clause *clause_frame* indique l'ensemble de lignes constituant le *frame window*, pour les fonctions window qui agissent sur le frame et non pas sur la partition entière. Si *fin_frame* est omis, sa valeur par défaut est CURRENT ROW. Les restrictions sont les suivantes : *début_frame* ne peut pas valoir UNBOUNDED FOLLOWING, *fin_frame* ne peut pas valoir UNBOUNDED PRECEDING et *fin_frame* ne peut pas apparaître avant *début_frame* -- par exemple, RANGE BETWEEN CURRENT ROW AND *valeur* PRECEDING n'est pas autorisé. L'option de frame par défaut est RANGE UNBOUNDED PRECEDING, qui est identique à RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ; cela configure la frame à toutes les lignes à partir du début de la partition jusqu'à la ligne actuelle du prochain dans l'ordre du ORDER BY (ce qui signifie toutes les lignes si la clause ORDER BY est absente). En général, UNBOUNDED PRECEDING signifie que la frame commence avec la première ligne de la partition et, de façon similaire, UNBOUNDED FOLLOWING signifie que la frame se termine avec la dernière ligne de la partition (quel que soit le mode, RANGE ou ROWS. Dans le mode ROWS, CURRENT ROW signifie que la frame commence ou finit avec la ligne actuelle ; mais dans le mode RANGE, cela signifie que la frame commence ou finit avec le premier ou le prochain élément à partir de la ligne actuel dans l'ordre indiqué par la clause ORDER BY. Les cas *valeur* PRECEDING et *valeur* FOLLOWING sont actuellement seulement autorisés dans le mode ROWS. Ils indiquent que la frame commence ou finit avec la ligne qui se trouve à ce nombre de lignes avant ou après la ligne actuelle. *valeur* doit être une expression entière ne contenant aucune variable, fonction d'agrégat ou fonction de fenêtrage. La valeur ne doit pas être NULL ou négative. Par contre, elle peut valoir zéro, ce qui a pour effet de sélectionner la ligne actuelle.

Les fonctions de fenêtrage internes sont décrites dans la Tableau 9.44, « Fonctions Window généralistes ». D'autres fonctions de fenêtrage peuvent être ajoutées par l'utilisateur. De plus, toute fonction d'agrégat interne ou définie par l'utilisateur peut être utilisée comme fonction de fenêtrage.

Les syntaxes utilisant * sont utilisées pour appeler des fonctions d'agrégats sans paramètres en tant que fonctions de fenêtrage. Par exemple : count(*) OVER (PARTITION BY x ORDER BY y). * n'est habituellement pas utilisé pour les fonctions de fenêtrage qui ne sont pas des agrégats. Les fonctions de fenêtrage agrégats, contrairement aux fonctions d'agrégats normales, n'autorisent pas l'utilisation de DISTINCT ou ORDER BY dans la liste des arguments de la fonction.

Les appels de fonctions de fenêtrage sont autorisés seulement dans la liste SELECT et dans la clause ORDER BY de la requête.

Il existe plus d'informations sur les fonctions de fenêtrages dans la Section 3.5, « Fonctions de fenêtrage », dans la Section 9.19, « Fonctions Window » et dans la Section 7.2.4, « Traitement de fonctions Window ».

4.2.9. Conversions de type

Une conversion de type spécifie une conversion à partir d'un type de données vers un autre. PostgreSQL™ accepte deux syntaxes équivalentes pour les conversions de type :

```
CAST ( expression AS type )
expression::type
```

La syntaxe CAST est conforme à SQL ; la syntaxe avec :: est historique dans PostgreSQL™.

Lorsqu'une conversion est appliquée à une expression de valeur pour un type connu, il représente une conversion de type à l'exécution. Cette conversion réussira seulement si une opération convenable de conversion de type a été définie. Notez que ceci est subtilement différent de l'utilisation de conversion avec des constantes, comme indiqué dans la Section 4.1.2.7, « Constantes

d'autres types ». Une conversion appliquée à une chaîne constante représente l'affectation initiale d'un type pour une valeur constante, et donc cela réussira pour tout type (si le contenu de la chaîne constante est une syntaxe acceptée en entrée pour le type de donnée).

Une conversion de type explicite pourrait être habituellement omise s'il n'y a pas d'ambiguïté sur le type qu'une expression de valeur pourrait produire (par exemple, lorsqu'elle est affectée à une colonne de table) ; le système appliquera automatiquement une conversion de type dans de tels cas. Néanmoins, la conversion automatique est réalisée seulement pour les conversions marquées « OK pour application implicite » dans les catalogues système. D'autres conversions peuvent être appelées avec la syntaxe de conversion explicite. Cette restriction a pour but d'empêcher l'exécution silencieuse de conversions surprenantes.

Il est aussi possible de spécifier une conversion de type en utilisant une syntaxe de type fonction :

```
nom_type ( expression )
```

Néanmoins, ceci fonctionne seulement pour les types dont les noms sont aussi valides en tant que noms de fonctions. Par exemple, `double precision` ne peut pas être utilisé de cette façon mais son équivalent `float8` le peut. De même, les noms `interval`, `time` et `timestamp` peuvent seulement être utilisés de cette façon s'ils sont entre des guillemets doubles à cause des conflits de syntaxe. Du coup, l'utilisation de la syntaxe de conversion du style fonction amène à des incohérences et devrait probablement être évitée.

Note

La syntaxe par fonction est en fait seulement un appel de fonction. Quand un des deux standards de syntaxe de conversion est utilisé pour faire une conversion à l'exécution, elle appellera en interne une fonction enregistrée pour réaliser la conversion. Par convention, ces fonctions de conversion ont le même nom que leur type de sortie et, du coup, la syntaxe par fonction n'est rien de plus qu'un appel direct à la fonction de conversion sous-jacente. Évidemment, une application portable ne devrait pas s'y fier. Pour plus d'informations, voir la page de manuel de CREATE CAST(7).

4.2.10. Sous-requêtes scalaires

Une sous-requête scalaire est une requête **SELECT** ordinaire entre parenthèses renvoyant exactement une ligne avec une colonne (voir le Chapitre 7, Requêtes pour plus d'informations sur l'écriture des requêtes). La requête **SELECT** est exécutée et la seule valeur renvoyée est utilisée dans l'expression de valeur englobante. C'est une erreur d'utiliser une requête qui renvoie plus d'une ligne ou plus d'une colonne comme requête scalaire. Mais si, lors d'une exécution particulière, la sous-requête ne renvoie pas de lignes, alors il n'y a pas d'erreur ; le résultat scalaire est supposé NULL. La sous-requête peut référencer des variables de la requête englobante, qui agiront comme des constantes durant toute évaluation de la sous-requête. Voir aussi la Section 9.20, « Expressions de sous-requêtes » pour d'autres expressions impliquant des sous-requêtes.

Par exemple, ce qui suit trouve la ville disposant de la population la plus importante dans chaque état :

```
SELECT nom, (SELECT max(pop) FROM villes WHERE villes.etat = etat.nom)
    FROM etats;
```

4.2.11. Constructeurs de tableaux

Un constructeur de tableau est une expression qui construit une valeur de tableau à partir de la valeur de ses membres. Un constructeur de tableau simple utilise le mot clé `ARRAY`, un crochet ouvrant `[`, une liste d'expressions (séparées par des virgules) pour les valeurs des éléments du tableau et finalement un crochet fermant `]`. Par exemple :

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

Par défaut, le type d'élément du tableau est le type commun des expressions des membres, déterminé en utilisant les mêmes règles que pour les constructions `UNION` ou `CASE` (voir la Section 10.5, « Constructions `UNION`, `CASE` et constructions relatives »). Vous pouvez surcharger ceci en convertissant explicitement le constructeur de tableau vers le type désiré. Par exemple :

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

Ceci a le même effet que la conversion de chaque expression vers le type d'élément du tableau individuellement. Pour plus d'informations sur les conversions, voir la Section 4.2.9, « Conversions de type ».

Les valeurs de tableaux multidimensionnels peuvent être construits par des constructeurs de tableaux imbriqués. Pour les constructeurs internes, le mot-clé ARRAY peut être omis. Par exemple, ces expressions produisent le même résultat :

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Comme les tableaux multidimensionnels doivent être rectangulaires, les constructeurs internes du même niveau doivent produire des sous-tableaux de dimensions identiques. Toute conversion appliquée au constructeur ARRAY externe se propage automatiquement à tous les constructeurs internes.

Les éléments d'un constructeur de tableau multidimensionnel peuvent être tout ce qui récupère un tableau du bon type, pas seulement une construction d'un tableau imbriqué. Par exemple :

```
CREATE TABLE tab(f1 int[], f2 int[]);

INSERT INTO tab VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM tab;
                      array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

Vous pouvez construire un tableau vide mais, comme il est impossible d'avoir un tableau sans type, vous devez convertir explicitement votre tableau vide dans le type désiré. Par exemple :

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

Il est aussi possible de construire un tableau à partir des résultats d'une sous-requête. Avec cette forme, le constructeur de tableau est écrit avec le mot clé ARRAY suivi par une sous-requête entre parenthèses (et non pas des crochets). Par exemple :

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                          ?column?
-------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
(1 row)
```

La sous-requête doit renvoyer une seule colonne. Le tableau à une dimension résultant aura un élément pour chaque ligne dans le résultat de la sous-requête, avec un type élément correspondant à celui de la colonne en sortie de la sous-requête.

Les indices d'un tableau construit avec ARRAY commencent toujours à un. Pour plus d'informations sur les tableaux, voir la Section 8.14, « Tableaux ».

4.2.12. Constructeurs de lignes

Un constructeur de ligne est une expression qui construit une valeur de ligne (aussi appelée une valeur composite) à partir des valeurs de ses membres. Un constructeur de ligne consiste en un mot clé ROW, une parenthèse gauche, zéro ou une ou plus d'une expression (séparées par des virgules) pour les valeurs des champs de la ligne, et enfin une parenthèse droite. Par exemple :

```
SELECT ROW(1,2.5,'ceci est un test');
```

Le mot clé ROW est optionnel lorsqu'il y a plus d'une expression dans la liste.

Un constructeur de ligne peut inclure la syntaxe *valeurligne*.*, qui sera étendue en une liste d'éléments de la valeur ligne, ce qui est le comportement habituel de la syntaxe .* utilisée au niveau haut d'une liste **SELECT**. Par exemple, si la table t a les colonnes f1 et f2, ces deux requêtes sont identiques :

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Note**
>
> Avant PostgreSQL™ 8.2, la syntaxe `.*` n'était pas étendue. De ce fait, `ROW(t.*, 42)` créait une ligne à deux champs dont le premier était une autre valeur de ligne. Le nouveau comportement est généralement plus utile. Si vous avez besoin de l'ancien comportement de valeurs de ligne imbriquées, écrivez la valeur de ligne interne sans `.*`, par exemple `ROW(t, 42)`.

Par défaut, la valeur créée par une expression `ROW` est d'un type d'enregistrement anonyme. Si nécessaire, il peut être converti en un type composite nommé -- soit le type de ligne d'une table soit un type composite créé avec **CREATE TYPE AS**. Une conversion explicite pourrait être nécessaire pour éviter toute ambiguïté. Par exemple :

```
CREATE TABLE ma_table(f1 int, f2 float, f3 text);

CREATE FUNCTION recup_f1(ma_table) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Aucune conversion nécessaire parce que seul un recup_f1() existe
SELECT recup_f1(ROW(1,2.5,'ceci est un test'));
 recup_f1
----------
1
(1 row)

CREATE TYPE mon_typeligne AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION recup_f1(mon_typeligne) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Maintenant, nous avons besoin d'une conversion
-- pour indiquer la fonction à appeler
SELECT recup_f1(ROW(1,2.5,'ceci est un test'));
ERROR:  function recup_f1(record) is not unique

SELECT recup_f1(ROW(1,2.5,'ceci est un test')::ma_table);
 getf1
-------
 1
(1 row)

SELECT recup_f1(CAST(ROW(11,'ceci est un test',2.5) AS mon_typeligne));
 getf1
-------
 11
(1 row)
```

Les constructeurs de lignes peuvent être utilisés pour construire des valeurs composites à stocker dans une colonne de table de type composite ou pour être passé à une fonction qui accepte un paramètre composite. De plus, il est possible de comparer deux valeurs de lignes ou pour tester une ligne avec `IS NULL` ou `IS NOT NULL`, par exemple

```
SELECT ROW(1,2.5,'ceci est un test') = ROW(1, 3, 'pas le même');

SELECT ROW(table.*) IS NULL FROM table; -- détecte toutes les lignes non NULL
```

Pour plus de détails, voir la Section 9.21, « Comparaisons de lignes et de tableaux ». Les constructeurs de lignes peuvent aussi être utilisés en relation avec des sous-requêtes, comme discuté dans la Section 9.20, « Expressions de sous-requêtes ».

4.2.13. Règles d'évaluation des expressions

L'ordre d'évaluation des sous-expressions n'est pas défini. En particulier, les entrées d'un opérateur ou d'une fonction ne sont pas obligatoirement évaluées de la gauche vers la droite ou dans un autre ordre fixé.

De plus, si le résultat d'une expression peut être déterminé par l'évaluation de certaines parties de celle-ci, alors d'autres sous-expressions devraient ne pas être évaluées du tout. Par exemple, si vous écrivez :

```
SELECT true OR une_fonction();
```

alors `une_fonction()` pourrait (probablement) ne pas être appelée du tout. Pareil dans le cas suivant :

```
SELECT une_fonction() OR true;
```

Notez que ceci n'est pas identique au « court-circuitage » de gauche à droite des opérateurs booléens utilisé par certains langages de programmation.

En conséquence, il est déconseillé d'utiliser des fonctions ayant des effets de bord dans une partie des expressions complexes. Il est particulièrement dangereux de se fier aux effets de bord ou à l'ordre d'évaluation dans les clauses `WHERE` et `HAVING` car ces clauses sont reproduites de nombreuses fois lors du développement du plan d'exécution. Les expressions booléennes (combinaisons `AND`/`OR`/`NOT`) dans ces clauses pourraient être réorganisées d'une autre façon autorisée dans l'algèbre booléenne.

Quand il est essentiel de forcer l'ordre d'évaluation, une construction `CASE` (voir la Section 9.16, « Expressions conditionnelles ») peut être utilisée. Voici un exemple qui ne garantie pas qu'une division par zéro ne soit faite dans une clause `WHERE` :

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Mais ceci est sûr :

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

Une construction `CASE` utilisée de cette façon déjouera les tentatives d'optimisation, donc cela ne sera à faire que si c'est nécessaire (dans cet exemple particulier, il serait sans doute mieux de contourner le problème en écrivant `y > 1.5*x`).

4.3. Fonctions appelantes

PostgreSQL™ permet aux fonctions qui ont des paramètres nommés d'être appelées en utilisant soit la notation par *position* soit la notation par *nom*. La notation par nom est particulièrement utile pour les fonctions qui ont un grand nombre de paramètres car elle rend l'association entre paramètre et argument plus explicite et fiable. Dans la notation par position, un appel de fonction précise les valeurs en argument dans le même ordre que ce qui a été défini à la création de la fonction. Dans la notation nommée, les arguments sont précisés par leur nom et peuvent du coup être intégrés dans n'importe quel ordre.

Quel que soit la notation, les paramètres qui ont des valeurs par défaut dans leur déclaration n'ont pas besoin d'être précisés dans l'appel. Ceci est particulièrement utile dans la notation nommée car toute combinaison de paramètre peut être omise alors que dans la notation par position, les paramètres peuvent seulement être omis de la droite vers la gauche.

PostgreSQL™ supporte aussi la notation *mixée*. Elle combine la notation par position avec la notation par nom. Dans ce cas, les paramètres de position sont écrits en premier, les paramètres nommés apparaissent après.

Les exemples suivants illustrent l'utilisation des trois notations, en utilisant la définition de fonction suivante :

```
CREATE FUNCTION assemble_min_ou_maj(a text, b text, majuscule boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

La fonction `assemble_min_ou_maj` a deux paramètres obligatoires, `a` et `b`. Il existe en plus un paramètre optionnel, `majuscule`, qui vaut par défaut `false`. Les arguments `a` et `b` seront concaténés et forcés soit en majuscule soit en minuscule suivant la valeur du paramètre `majuscule`. Les détails restant ne sont pas importants ici (voir le Chapitre 35, Étendre SQL pour plus d'informations).

4.3.1. En utilisant la notation par position

La notation par position est le mécanisme traditionnel pour passer des arguments aux fonctions avec PostgreSQL™. En voici un exemple :

```
SELECT assemble_min_ou_maj('Hello', 'World', true);
 assemble_min_ou_maj
---------------------
 HELLO WORLD
(1 row)
```

Tous les arguments sont indiqués dans l'ordre. Le résultat est en majuscule car l'argument `majuscule` est indiqué à `true`. Voici un autre exemple :

```
SELECT assemble_min_ou_maj('Hello', 'World');
 assemble_min_ou_maj
-----------------------
 hello world
(1 row)
```

Ici, le paramètre `majuscule` est omis, donc il récupère la valeur par défaut, soit `false`, ce qui a pour résultat une sortie en minuscule. Dans la notation par position, les arguments peuvent être omis de la droite à la gauche à partir du moment où ils ont des valeurs par défaut.

4.3.2. En utilisant la notation par nom

Dans la notation par nom, chaque nom d'argument est précisé en utilisant `:=` pour le séparer de l'expression de la valeur de l'argument. Par exemple :

```
SELECT assemble_min_ou_maj(a := 'Hello', b := 'World');
 assemble_min_ou_maj
-----------------------
 hello world
(1 row)
```

Encore une fois, l'argument `majuscule` a été omis, donc il dispose de sa valeur par défaut, `false`, implicitement. Un avantage à utiliser la notation par nom est que les arguments peuvent être saisis dans n'importe quel ordre. Par exemple :

```
SELECT assemble_min_ou_maj(a := 'Hello', b := 'World', majuscule := true);
 assemble_min_ou_maj
-----------------------
 HELLO WORLD
(1 row)

SELECT assemble_min_ou_maj(a := 'Hello', majuscule := true, b := 'World');
 assemble_min_ou_maj
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. En utilisant la notation mixée

La notation mixée combine les notations par position et par nom. Néanmoins, comme cela a déjà été expliqué, les arguments par nom ne peuvent pas précéder les arguments par position. Par exemple :

```
SELECT assemble_min_ou_maj('Hello', 'World', majuscule := true);
 assemble_min_ou_maj
-----------------------
 HELLO WORLD
(1 row)
```

Dans la requête ci-dessus, les arguments `a` et `b` sont précisés par leur position alors que `majuscule` est indiqué par son nom. Dans cet exemple, cela n'apporte pas grand-chose, sauf pour une documentation de la fonction. Avec une fonction plus complexe, comprenant de nombreux paramètres avec des valeurs par défaut, les notations par nom et mixées améliorent l'écriture des appels de fonction et permettent de réduire les risques d'erreurs.

Note

Les notations par appel nommé ou mixe ne peuvent pas être utilisé lors de l'appel d'une fonction d'agrégat (mais elles fonctionnent quand une fonction d'agrégat est utilisée en tant que fonction de fenêtrage).

Chapitre 5. Définition des données

Ce chapitre couvre la création des structures de données amenées à contenir les données. Dans une base relationnelle, les données brutes sont stockées dans des tables. De ce fait, une grande partie de ce chapitre est consacrée à l'explication de la création et de la modification des tables et aux fonctionnalités disponibles pour contrôler les données stockées dans les tables. L'organisation des tables dans des schémas et l'attribution de privilèges sur les tables sont ensuite décrits. Pour finir, d'autres fonctionnalités, telles que l'héritage, les vues, les fonctions et les déclencheurs sont passées en revue.

5.1. Notions fondamentales sur les tables

Une table dans une base relationnelle ressemble beaucoup à un tableau sur papier : elle est constituée de lignes et de colonnes. Le nombre et l'ordre des colonnes sont fixes et chaque colonne a un nom. Le nombre de lignes est variable -- il représente le nombre de données stockées à un instant donné. Le SQL n'apporte aucune garantie sur l'ordre des lignes dans une table. Quand une table est lue, les lignes apparaissent dans un ordre non spécifié, sauf si un tri est demandé explicitement. Tout cela est expliqué dans le Chapitre 7, Requêtes. De plus, le SQL n'attribue pas d'identifiant unique aux lignes. Il est donc possible d'avoir plusieurs lignes identiques au sein d'une table. C'est une conséquence du modèle mathématique sur lequel repose le SQL, même si cela n'est habituellement pas souhaitable. Il est expliqué plus bas dans ce chapitre comment traiter ce problème.

Chaque colonne a un type de données. Ce type limite l'ensemble de valeurs qu'il est possible d'attribuer à une colonne. Il attribue également une sémantique aux données stockées dans la colonne pour permettre les calculs sur celles-ci. Par exemple, une colonne déclarée dans un type numérique n'accepte pas les chaînes textuelles ; les données stockées dans une telle colonne peuvent être utilisées dans des calculs mathématiques. Par opposition, une colonne déclarée de type chaîne de caractères accepte pratiquement n'importe quel type de donnée mais ne se prête pas aux calculs mathématiques. D'autres types d'opérations, telle la concaténation de chaînes, sont cependant disponibles.

PostgreSQL™ inclut un ensemble conséquent de types de données intégrés pour s'adapter à diverses applications. Les utilisateurs peuvent aussi définir leurs propres types de données.

La plupart des types de données intégrés ont des noms et des sémantiques évidents. C'est pourquoi leur explication détaillée est reportée au Chapitre 8, Types de données.

Parmi les types les plus utilisés, on trouve integer pour les entiers, numeric pour les éventuelles fractions, text pour les chaînes de caractères, date pour les dates, time pour les heures et timestamp pour les valeurs qui contiennent à la fois une date et une heure.

Pour créer une table, on utilise la commande bien nommée CREATE TABLE(7). Dans cette commande, il est nécessaire d'indiquer, au minimum, le nom de la table, les noms des colonnes et le type de données de chacune d'elles. Par exemple :

```
CREATE TABLE ma_premiere_table (
    premiere_colonne text,
    deuxieme_colonne integer
);
```

Cela crée une table nommée `ma_premiere_table` avec deux colonnes. La première colonne, nommée `premiere_colonne`, est de type text ; la seconde colonne, nommée `deuxieme_colonne`, est de type integer. Les noms des table et colonnes se conforment à la syntaxe des identifiants expliquée dans la Section 4.1.1, « identificateurs et mots clés ». Les noms des types sont souvent aussi des identifiants mais il existe des exceptions. Le séparateur de la liste des colonnes est la virgule. La liste doit être entre parenthèses.

L'exemple qui précède est à l'évidence extrêmement simpliste. On donne habituellement aux tables et aux colonnes des noms qui indiquent les données stockées. L'exemple ci-dessous est un peu plus réaliste :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric
);
```

(Le type numeric peut stocker des fractions telles que les montants.)

Astuce

Quand de nombreuses tables liées sont créées, il est préférable de définir un motif cohérent pour le nommage des tables et des colonnes. On a ainsi la possibilité d'utiliser le pluriel ou le singulier des noms, chacune ayant ses fidèles et ses détracteurs.

Le nombre de colonnes d'un table est limité. En fonction du type de colonnes, il oscille entre 250 et 1600. Définir une table avec un nombre de colonnes proche de cette limite est, cependant, très inhabituel et doit conduire à se poser des questions quant à la conception du modèle.

Lorsqu'une table n'est plus utile, elle peut être supprimée à l'aide de la commande DROP TABLE(7). Par exemple :

```
DROP TABLE ma_premiere_table;
DROP TABLE produits;
```

Tenter de supprimer une table qui n'existe pas lève une erreur. Il est, néanmoins, habituel dans les fichiers de scripts SQL d'essayer de supprimer chaque table avant de la créer. Les messages d'erreur sont alors ignorés afin que le script fonctionne que la table existe ou non. (La variante `DROP TABLE IF EXISTS` peut aussi être utilisée pour éviter les messages d'erreur mais elle ne fait pas partie du standard SQL.)

Pour la procéduure de modification d'une table qui existe déjà, voir la Section 5.5, « Modification des tables » plus loin dans ce chapitre.

Les outils précédemment décrits permettent de créer des tables fonctionnelles. Le reste de ce chapitre est consacré à l'ajout de fonctionnalités à la définition de tables pour garantir l'intégrité des données, la sécurité ou l'ergonomie. Le lecteur impatient d'insérer des données dans ses tables peut sauter au Chapitre 6, Manipulation de données et lire le reste de ce chapitre plus tard.

5.2. Valeurs par défaut

Une valeur par défaut peut être attribuée à une colonne. Quand une nouvelle ligne est créée et qu'aucune valeur n'est indiquée pour certaines de ses colonnes, celles-ci sont remplies avec leurs valeurs par défaut respectives. Une commande de manipulation de données peut aussi demander explicitement que la valeur d'une colonne soit positionnée à la valeur par défaut, sans qu'il lui soit nécessaire de connaître cette valeur (les détails concernant les commandes de manipulation de données sont donnés dans le Chapitre 6, Manipulation de données).

Si aucune valeur par défaut n'est déclarée explicitement, la valeur par défaut est la valeur NULL. Cela a un sens dans la mesure où l'on peut considérer que la valeur NULL représente des données inconnues.

Dans la définition d'une table, les valeurs par défaut sont listées après le type de données de la colonne. Par exemple:

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric DEFAULT 9.99
);
```

La valeur par défaut peut être une expression, alors évaluée à l'insertion de cette valeur (*pas* à la création de la table). Un exemple commun est la colonne de type timestamp dont la valeur par défaut est `now()`. Elle se voit ainsi attribuée l'heure d'insertion. Un autre exemple est la génération d'un « numéro de série » pour chaque ligne. Dans PostgreSQL™, cela s'obtient habituellement par quelque chose comme

```
CREATE TABLE produits (
    no_produit integer DEFAULT nextval('produits_no_produit_seq'),
    ...
);
```

où la fonction `nextval()` fournit des valeurs successives à partir d'un *objet séquence* (voir la Section 9.15, « Fonctions de manipulation de séquences »). Cet arrangement est suffisamment commun pour qu'il ait son propre raccourci :

```
CREATE TABLE produits (
    no_produit SERIAL,
    ...
);
```

Le raccourci `SERIAL` est discuté plus tard dans la Section 8.1.4, « Types seriés ».

5.3. Contraintes

Les types de données sont un moyen de restreindre la nature des données qui peuvent être stockées dans une table. Pour beaucoup d'applications, toutefois, la contrainte fournie par ce biais est trop grossière. Par exemple, une colonne qui contient le prix d'un produit ne doit accepter que des valeurs positives. Mais il n'existe pas de type de données standard qui n'accepte que des valeurs positives. Un autre problème peut provenir de la volonté de contraindre les données d'une colonne par rapport aux autres colonnes ou lignes. Par exemple, dans une table contenant des informations de produit, il ne peut y avoir qu'une ligne par numéro de produit.

Pour cela, SQL permet de définir des contraintes sur les colonnes et les tables. Les contraintes donnent autant de contrôle sur les données des tables qu'un utilisateur peut le souhaiter. Si un utilisateur tente de stocker des données dans une colonne en violation d'une contrainte, une erreur est levée. Cela s'applique même si la valeur vient de la définition de la valeur par défaut.

5.3.1. Contraintes de vérification

La contrainte de vérification est la contrainte la plus générique qui soit. Elle permet d'indiquer que la valeur d'une colonne particulière doit satisfaire une expression booléenne (valeur de vérité). Par exemple, pour obliger les prix des produits à être positifs, on peut utiliser :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0)
);
```

La définition de contrainte vient après le type de données, comme pour les définitions de valeur par défaut. Les valeurs par défaut et les contraintes peuvent être données dans n'importe quel ordre. Une contrainte de vérification s'utilise avec le mot clé `CHECK` suivi d'une expression entre parenthèses. L'expression de la contrainte implique habituellement la colonne à laquelle elle s'applique, la contrainte n'ayant dans le cas contraire que peu de sens.

la contrainte peut prendre un nom distinct. Cela clarifie les messages d'erreur et permet de faire référence à la contrainte lorsqu'elle doit être modifiée. La syntaxe est :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CONSTRAINT prix_positif CHECK (prix > 0)
);
```

Pour indiquer une contrainte nommée, on utilise le mot-clé `CONSTRAINT` suivi d'un identifiant et de la définition de la contrainte (si aucun nom n'est précisé, le système en choisit un).

Une contrainte de vérification peut aussi faire référence à plusieurs colonnes. Dans le cas d'un produit, on peut vouloir stocker le prix normal et un prix réduit en s'assurant que le prix réduit soit bien inférieur au prix normal.

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0),
    prix_promotion numeric CHECK (prix_promotion > 0),
    CHECK (prix > prix_promotion)
);
```

Si les deux premières contraintes n'offrent pas de nouveauté, la troisième utilise une nouvelle syntaxe. Elle n'est pas attachée à une colonne particulière mais apparaît comme un élément distinct dans la liste des colonnes. Les définitions de colonnes et ces définitions de contraintes peuvent être définies dans un ordre quelconque.

Les deux premières contraintes sont appelées contraintes de colonne tandis que la troisième est appelée contrainte de table parce qu'elle est écrite séparément d'une définition de colonne particulière. Les contraintes de colonne peuvent être écrites comme des contraintes de table, mais l'inverse n'est pas forcément possible puisqu'une contrainte de colonne est supposée ne faire référence qu'à la colonne à laquelle elle est attachée (PostgreSQL™ ne vérifie pas cette règle mais il est préférable de la suivre pour s'assurer que les définitions de tables fonctionnent avec d'autres systèmes de bases de données). L'exemple ci-dessus peut aussi s'écrire :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric,
    CHECK (prix > 0),
    prix_promotion numeric,
    CHECK (prix_promotion > 0),
    CHECK (prix > prix_promotion)
);
```

ou même :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0),
```

```
    prix_promotion numeric,
    CHECK (prix_promotion > 0 AND prix > prix_promotion)
);
```

C'est une question de goût.

Les contraintes de table peuvent être nommées, tout comme les contraintes de colonne :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric,
    CHECK (prix > 0),
    prix_promotion numeric,
    CHECK (prix_promotion > 0),
    CONSTRAINT promo_valide CHECK (prix > prix_promotion)
);
```

Une contrainte de vérification est satisfaite si l'expression est évaluée vraie ou NULL. Puisque la plupart des expressions sont évaluées NULL si l'une des opérandes est nulle, elles n'interdisent pas les valeurs NULL dans les colonnes contraintes. Pour s'assurer qu'une colonne ne contient pas de valeurs NULL, la contrainte NOT NULL décrite dans la section suivante peut être utilisée.

5.3.2. Contraintes de non nullité (NOT NULL)

Une contrainte NOT NULL indique simplement qu'une colonne ne peut pas prendre la valeur NULL. Par exemple :

```
CREATE TABLE produits (
    no_produit integer NOT NULL,
    nom text NOT NULL,
    prix numeric
);
```

Une contrainte NOT NULL est toujours écrite comme une contrainte de colonne. Elle est fonctionnellement équivalente à la création d'une contrainte de vérification `CHECK (nom_colonne IS NOT NULL)`. Toutefois, dans PostgreSQL™, il est plus efficace de créer explicitement une contrainte NOT NULL. L'inconvénient est que les contraintes de non-nullité ainsi créées ne peuvent pas être explicitement nommées.

Une colonne peut évidemment avoir plusieurs contraintes. Il suffit d'écrire les contraintes les unes après les autres :

```
CREATE TABLE produits (
    no_produit integer NOT NULL,
    nom text NOT NULL,
    prix numeric NOT NULL CHECK (prix > 0)
);
```

L'ordre n'a aucune importance. Il ne détermine pas l'ordre de vérification des contraintes.

La contrainte `NOT NULL` a un contraire ; la contrainte `NULL`. Elle ne signifie pas que la colonne doit être NULL, ce qui est assurément inutile, mais sélectionne le comportement par défaut, à savoir que la colonne peut être NULL. La contrainte `NULL` n'est pas présente dans le standard SQL et ne doit pas être utilisée dans des applications portables (elle n'a été ajoutée dans PostgreSQL™ que pour assurer la compatibilité avec d'autres bases de données). Certains utilisateurs l'apprécient néanmoins car elle permet de basculer aisément d'une contrainte à l'autre dans un fichier de script. On peut, par exemple, commencer avec :

```
CREATE TABLE produits (
    no_produit integer NULL,
    nom text NULL,
    prix numeric NULL
);
```

puis insérer le mot-clé `NOT` en fonction des besoins.

Astuce

Dans la plupart des bases de données, il est préférable que la majorité des colonnes soient marquées NOT NULL.

5.3.3. Contraintes d'unicité

Les contraintes d'unicité garantissent l'unicité des données contenues dans une colonne ou un groupe de colonnes par rapport à toutes les lignes de la table. La syntaxe est :

```
CREATE TABLE produits (
    no_produit integer UNIQUE,
    nom text,
    prix numeric
);
```

lorsque la contrainte est écrite comme contrainte de colonne et :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric,
    UNIQUE (no_produit)
);
```

lorsqu'elle est écrite comme contrainte de table.

Lorsqu'une contrainte d'unicité fait référence à un groupe de colonnes, celles-ci sont listées séparées par des virgules :

```
CREATE TABLE exemple (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

Cela précise que la combinaison de valeurs dans les colonnes indiquées est unique sur toute la table. Sur une colonne prise isolément ce n'est pas nécessairement le cas (et habituellement cela ne l'est pas).

Une contrainte d'unicité peut être nommée, de la façon habituelle :

```
CREATE TABLE produits (
    no_produit integer CONSTRAINT doit_etre_different UNIQUE,
    nom text,
    prix numeric
);
```

L'ajout d'une contrainte unique créera automatiquement un index B-tree unique sur la colonne ou le groupe de colonnes utilisé dans la contrainte.

En général, une contrainte d'unicité est violée lorsque plus d'une ligne de la table possèdent des valeurs identiques sur toutes les colonnes de la contrainte. En revanche, deux valeurs NULL ne sont pas considérées égales. Cela signifie qu'il est possible de stocker des lignes dupliquées contenant une valeur NULL dans au moins une des colonnes contraintes. Ce comportement est conforme au standard SQL, mais d'autres bases SQL n'appliquent pas cette règle. Il est donc préférable d'être prudent lors du développement d'applications portables.

5.3.4. Clés primaires

Techniquement, une contrainte de clé primaire n'est que la combinaison d'une contrainte d'unicité et d'une contrainte NOT NULL. Les définitions de table suivantes acceptent de ce fait les mêmes données :

```
CREATE TABLE produits (
        no_produit integer UNIQUE NOT NULL,
        nom text,
        prix numeric
    );
```

```
CREATE TABLE produits (
        no_produit integer PRIMARY KEY,
        nom text,
        prix numeric
    );
```

Les clés primaires peuvent également contraindre plusieurs colonnes ; la syntaxe est semblable aux contraintes d'unicité :

```
CREATE TABLE exemple (
        a integer,
        b integer,
        c integer,
        PRIMARY KEY (a, c)
    );
```

Une clé primaire indique qu'une colonne ou un groupe de colonnes peut être utilisé(e) comme identifiant unique des lignes de la table. (C'est une conséquence directe de la définition d'une clé primaire. Une contrainte d'unicité ne suffit pas à fournir un identifiant unique car elle n'exclut pas les valeurs NULL). Ceci est utile à la fois pour des raisons documentaires et pour les applications clientes. Par exemple, une application graphique qui permet de modifier les valeurs de lignes a probablement besoin de connaître la clé primaire d'une table pour pouvoir identifier les lignes de manière unique.

L'ajout d'une clé primaire créera automatiquement un index B-tree unique sur la colonne ou le groupe de colonnes utilisé dans la clé primaire.

Une table a, au plus, une clé primaire. (Le nombre de contraintes UNIQUE NOT NULL, qui assurent la même fonction, n'est pas limité, mais une seule peut être identifiée comme clé primaire.) La théorie des bases de données relationnelles impose que chaque table ait une clé primaire. Cette règle n'est pas forcée par PostgreSQL™, mais il est préférable de la respecter.

5.3.5. Clés étrangères

Une contrainte de clé étrangère stipule que les valeurs d'une colonne (ou d'un groupe de colonnes) doivent correspondre aux valeurs qui apparaissent dans les lignes d'une autre table. On dit que cela maintient l'*intégrité référentielle* entre les deux tables.

Soit la table de produits, déjà utilisée plusieurs fois :

```
CREATE TABLE produits (
        no_produit integer PRIMARY KEY,
        nom text,
        prix numeric
    );
```

Soit également une table qui stocke les commandes de ces produits. Il est intéressant de s'assurer que la table des commandes ne contient que des commandes de produits qui existent réellement. Pour cela, une contrainte de clé étrangère est définie dans la table des commandes qui référence la table produit :

```
CREATE TABLE commandes (
        id_commande integer PRIMARY KEY,
        no_produit integer REFERENCES produits (no_produit),
        quantite integer
    );
```

Il est désormais impossible de créer des commandes pour lesquelles `no_produit` n'apparaît pas dans la table produits.

Dans cette situation, on dit que la table des commandes est la table *qui référence* et la table des produits est la table *référencée*. De la même façon, il y a des colonnes qui référencent et des colonnes référencées.

La commande précédente peut être raccourcie en

```
CREATE TABLE commandes (
        id_commande integer PRIMARY KEY,
        no_produit integer REFERENCES produits,
        quantite integer
    );
```

parce qu'en l'absence de liste de colonnes, la clé primaire de la table de référence est utilisée comme colonne de référence.

Une clé étrangère peut aussi contraindre et référencer un groupe de colonnes. Comme cela a déjà été évoqué, il faut alors l'écrire sous forme d'une contrainte de table. Exemple de syntaxe :

```
CREATE TABLE t1 (
      a integer PRIMARY KEY,
      b integer,
      c integer,
      FOREIGN KEY (b, c) REFERENCES autre_table (c1, c2)
    );
```

Le nombre et le type des colonnes contraintes doivent correspondre au nombre et au type des colonnes référencées.

Une contrainte de clé étrangère peut être nommée de la façon habituelle.

Une table peut contenir plusieurs contraintes de clé étrangère. Les relation n-n entre tables sont implantées ainsi. Soient des tables qui contiennent des produits et des commandes, avec la possibilité d'autoriser une commande à contenir plusieurs produits (ce que la structure ci-dessus ne permet pas). On peut pour cela utiliser la structure de table suivante :

```
CREATE TABLE produits (
        no_produit integer PRIMARY KEY,
        nom text,
        prix numeric
```

```
    );

    CREATE TABLE commandes (
        id_commande integer PRIMARY KEY,
        adresse_de_livraison text,
        ...
    );

    CREATE TABLE commande_produits (
        no_produit integer REFERENCES produits,
        id_commande integer REFERENCES commandes,
        quantite integer,
        PRIMARY KEY (no_produit, id_commande)
    );
```

La clé primaire de la dernière table recouvre les clés étrangères.

Les clés étrangères interdisent désormais la création de commandes qui ne soient pas liées à un produit. Qu'arrive-t-il si un produit est supprimé alors qu'une commande y fait référence ? SQL permet aussi de le gérer. Intuitivement, plusieurs options existent :

- interdire d'effacer un produit référencé ;
- effacer aussi les commandes ;
- autre chose ?

Pour illustrer ce cas, la politique suivante est implantée sur l'exemple de relations n-n évoqué plus haut :

- quand quelqu'un veut retirer un produit qui est encore référencé par une commande (au travers de `commande_produits`), on l'interdit ;
- si quelqu'un supprime une commande, les éléments de la commande sont aussi supprimés.

```
CREATE TABLE produits (
        no_produit integer PRIMARY KEY,
        nom text,
        prix numeric
    );

    CREATE TABLE commandes (
        id_commande integer PRIMARY KEY,
        adresse_de_livraison text,
        ...
    );

    CREATE TABLE commande_produits (
        no_produit integer REFERENCES produits ON DELETE RESTRICT,
        id_commande integer REFERENCES commandes ON DELETE CASCADE,
        quantite integer,
        PRIMARY KEY (no_produit, id_commande)
    );
```

Restreindre les suppressions et les cascader sont les deux options les plus communes. `RESTRICT` empêche la suppression d'une ligne référencée. `NO ACTION` impose la levée d'une erreur si des lignes référençant existent lors de la vérification de la contrainte. Il s'agit du comportement par défaut en l'absence de précision. La différence entre `RESTRICT` et `NO ACTION` est l'autorisation par `NO ACTION` du report de la vérification à la fin de la transaction, ce que `RESTRICT` ne permet pas. `CASCADE` indique que, lors de la suppression d'une ligne référencée, les lignes la référençant doivent être automatiquement supprimées. Il existe deux autres options : `SET NULL` et `SET DEFAULT`. Celles-ci imposent que les colonnes qui référencent soient réinitialisées à NULL ou à leur valeur par défaut, respectivement, lors de la suppression d'une ligne référencée. Elles ne dispensent pas pour autant d'observer les contraintes. Par exemple, si une action précise `SET DEFAULT` mais que la valeur par défaut ne satisfait pas la clé étrangère, l'opération échoue.

À l'instar de `ON DELETE`, existe `ON UPDATE`, évoqué lorsqu'une colonne référencée est modifiée (actualisée). Les actions possibles sont les mêmes.

Comme la suppression d'une ligne de la table référencée ou la mise à jour d'une colonne référencée nécessitera un parcours de la table référée pour trouver les lignes correspondant à l'ancienne valeur, il est souvent intéressant d'indexer les colonnes référencées. Comme cela n'est pas toujours nécessaire et qu'il y a du choix sur la façon d'indexer, l'ajout d'une contrainte de clé étrangère ne crée pas automatiquement un index sur les colonnes référencées.

Le Chapitre 6, Manipulation de données contient de plus amples informations sur l'actualisation et la suppression de données.

Une clé étrangère peut faire référence à des colonnes qui constituent une clé primaire ou forment une contrainte d'unicité. Si la clé étrangère référence une contrainte d'unicité, des possibilités supplémentaires sont offertes concernant la correspondance des valeurs NULL. Celles-ci sont expliquées dans la documentation de référence de CREATE TABLE(7).

5.3.6. Contraintes d'exclusion

Les contraintes d'exclusion vous assurent que si deux lignes sont comparées sur les colonnes ou expressions spécifiées en utilisant les opérateurs indiqués, au moins une de ces comparaisons d'opérateurs reverra false ou NULL. La syntaxe est :

```
CREATE TABLE cercles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Voir aussi **CREATE TABLE ... CONSTRAINT ... EXCLUDE** pour plus de détails.

L'ajout d'une contrainte d'exclusion créera automatiquement un index du type spécifié dans la déclaration de la contrainte.

5.4. Colonnes système

Chaque table contient plusieurs *colonnes système* implicitement définies par le système. De ce fait, leurs noms ne peuvent pas être utilisés comme noms de colonnes utilisateur (ces restrictions sont distinctes de celles sur l'utlisation de mot-clés ; mettre le nom entre guillemets ne permet pas d'échapper à cette règle). Il n'est pas vraiment utile de se préoccuper de ces colonnes, mais au minimum de savoir qu'elles existent.

oid
: L'identifiant objet (*object ID*) d'une ligne. Cette colonne n'est présente que si la table a été créée en précisant `WITH OIDS` ou si la variable de configuration default_with_oids était activée à ce moment-là. Cette colonne est de type oid (même nom que la colonne) ; voir la Section 8.16, « Types identifiant d'objet » pour obtenir plus d'informations sur ce type.

tableoid
: L' OID de la table contenant la ligne. Cette colonne est particulièrement utile pour les requêtes qui utilisent des hiérarchies d'héritage (voir Section 5.8, « L'héritage »). Il est, en effet, difficile, en son absence, de savoir de quelle table provient une ligne. *tableoid* peut être joint à la colonne *oid* de pg_class pour obtenir le nom de la table.

xmin
: L'identifiant (ID de transaction) de la transaction qui a inséré cette version de la ligne. (Une version de ligne est un état individuel de la ligne ; toute mise à jour d'une ligne crée une nouvelle version de ligne pour la même ligne logique.)

cmin
: L'identifiant de commande (à partir de zéro) au sein de la transaction d'insertion.

xmax
: L'identifiant (ID de transaction) de la transaction de suppression, ou zéro pour une version de ligne non effacée. Il est possible que la colonne ne soit pas nulle pour une version de ligne visible ; cela indique habituellement que la transaction de suppression n'a pas été effectuée, ou qu'une tentative de suppression a été annulée.

cmax
: L'identifiant de commande au sein de la transaction de suppression, ou zéro.

ctid
: La localisation physique de la version de ligne au sein de sa table. Bien que le *ctid* puisse être utilisé pour trouver la version de ligne très rapidement, le *ctid* d'une ligne change si la ligne est actualisée ou déplacée par un **VACUUM FULL**. *ctid* est donc inutilisable comme identifiant de ligne sur le long terme. Il est préférable d'utiliser l'OID, ou, mieux encore, un numéro de série utilisateur, pour identifier les lignes logiques.

Les OID sont des nombres de 32 bits et sont attribués à partir d'un compteur unique sur le cluster. Dans une base de données volumineuse ou agée, il est possible que le compteur boucle. Il est de ce fait peu pertinent de considérer que les OID puissent être uniques ; pour identifier les lignes d'une table, il est fortement recommandé d'utiliser un générateur de séquence. Néanmoins, les OID peuvent également être utilisés sous réserve que quelques précautions soient prises :

- une contrainte d'unicité doit être ajoutée sur la colonne OID de chaque table dont l'OID est utilisé pour identifier les lignes.

Dans ce cas (ou dans celui d'un index d'unicité), le système n'engendre pas d'OID qui puisse correspondre à celui d'une ligne déjà présente. Cela n'est évidemment possible que si la table contient moins de 2^{32} (4 milliards) lignes ; en pratique, la taille de la table a tout intérêt à être bien plus petite que ça, dans un souci de performance ;

- l'unicité inter-tables des OID ne doit jamais être envisagée ; pour obtenir un identifiant unique sur l'ensemble de la base, il faut utiliser la combinaison du *`tableoid`* et de l'OID de ligne ;
- les tables en question doivent être créées avec l'option `WITH OIDS`. Depuis PostgreSQL™ 8.1, `WITHOUT OIDS` est l'option par défaut.

Les identifiants de transaction sont aussi des nombres de 32 bits. Dans une base de données agée, il est possible que les ID de transaction bouclent. Cela n'est pas un problème fatal avec des procédures de maintenance appropriées ; voir le Chapitre 23, Planifier les tâches de maintenance pour les détails. Il est, en revanche, imprudent de considérer l'unicité des ID de transaction sur le long terme (plus d'un milliard de transactions).

Les identifiants de commande sont aussi des nombres de 32 bits. Cela crée une limite dure de 2^{32} (4 milliards) commandes SQL au sein d'une unique transaction. En pratique, cette limite n'est pas un problème -- la limite est sur le nombre de commandes SQL, pas sur le nombre de lignes traitées. De plus, à partir de PostgreSQL™ 8.3, seules les commandes qui modifient réellement le contenu de la base de données consomment un identifiant de commande.

5.5. Modification des tables

Lorsqu'une table est créée et qu'une erreur a été commise ou que les besoins de l'application changent, il est alors possible de la supprimer et de la récréer. Cela n'est toutefois pas pratique si la table contient déjà des données ou qu'elle est référencée par d'autres objets de la base de données (une contrainte de clé étrangère, par exemple). C'est pourquoi PostgreSQL™ offre une série de commandes permettant de modifier une table existante. Cela n'a rien à voir avec la modification des données contenues dans la table ; il ne s'agit ici, que de modifier la définition, ou structure, de la table.

Il est possible

- d'ajouter des colonnes ;
- de supprimer des colonnes ;
- d'ajouter des contraintes ;
- de supprimer des contraintes ;
- de modifier des valeurs par défaut ;
- de modifier les types de données des colonnes ;
- de renommer des colonnes ;
- de renommer des tables.

Toutes ces actions sont réalisées à l'aide de la commande ALTER TABLE(7), dont la page de référence est bien plus détaillée.

5.5.1. Ajouter une colonne

La commande d'ajout d'une colonne ressemble à :

```
ALTER TABLE produits ADD COLUMN description text;
```

La nouvelle colonne est initialement remplie avec la valeur par défaut précisée (NULL en l'absence de clause `DEFAULT`).

Des contraintes de colonne peuvent être définies dans la même commande, à l'aide de la syntaxe habituelle :

```
ALTER TABLE produits ADD COLUMN description text CHECK (description <> '');
```

En fait, toutes les options applicables à la description d'une colonne dans **CREATE TABLE** peuvent être utilisées ici. Il ne faut toutefois pas oublier que la valeur par défaut doit satisfaire les contraintes données. Dans le cas contraire, `ADD` échoue. Il est aussi possible d'ajouter les contraintes ultérieurement (voir ci-dessous) après avoir rempli la nouvelle colonne correctement.

> **Astuce**
>
> Ajouter une colonne avec une valeur par défaut nécessite la mise à jour de chaque ligne de la table pour stocker la valeur de la nouvelle colonne. Cependant, si aucune valeur par défaut n'est précisée, PostgreSQL™ peut éviter la mise à jour physique. Il est, de ce fait, préférable, si la colonne doit être remplie en majorité avec des valeurs différentes de la valeur par défaut, d'ajouter la colonne sans valeur par défaut, d'insérer les bonnes valeurs avec une commande **UPDATE** puis d'ajouter la valeur par défaut désirée comme décrit ci-dessus.

5.5.2. Retirer une colonne

La commande de suppression d'une colonne ressemble à celle-ci :

```
ALTER TABLE produits DROP COLUMN description;
```

Toute donnée dans cette colonne disparaît. Les contraintes de table impliquant la colonne sont également supprimées. Néanmoins, si la colonne est référencée par une contrainte de clé étrangère d'une autre table, PostgreSQL™ ne supprime pas silencieusement cette contrainte. La suppression de tout ce qui dépend de la colonne peut être autorisée en ajoutant `CASCADE` :

```
ALTER TABLE produits DROP COLUMN description CASCADE;
```

Voir la Section 5.11, « Gestion des dépendances » pour une description du mécanisme général.

5.5.3. Ajouter une contrainte

Pour ajouter une contrainte, la syntaxe de contrainte de table est utilisée. Par exemple :

```
ALTER TABLE produits ADD CHECK (nom <> '');
ALTER TABLE produits ADD CONSTRAINT autre_nom UNIQUE (no_produit);
ALTER TABLE produits ADD FOREIGN KEY (id_groupe_produit) REFERENCES groupes_produits;
```

Pour ajouter une contrainte NOT NULL, qui ne peut pas être écrite sous forme d'une contrainte de table, la syntaxe suivante est utilisée :

```
ALTER TABLE produits ALTER COLUMN no_produit SET NOT NULL;
```

La contrainte étant immédiatement vérifiée, les données de la table doivent satisfaire la contrainte avant qu'elle ne soit ajoutée.

5.5.4. Supprimer une contrainte

Pour supprimer une contrainte, il faut connaître son nom. Si elle a été explicitement nommé, il n'y a aucune difficulté. Dans le cas contraire, le système a engendré et attribué un nom qu'il faut découvrir. La commande `\d table` de psql peut être utile ici ; d'autres interfaces offrent aussi la possibilité d'examiner les détails de table. La commande est :

```
ALTER TABLE produits DROP CONSTRAINT un_nom;
```

(Dans le cas d'un nom de contrainte engendré, comme `$2`, il est nécessaire de l'entourer de guillemets doubles pour en faire un identifiant valable.)

Comme pour la suppression d'une colonne, `CASCADE` peut être ajouté pour supprimer une contrainte dont dépendent d'autres objets. Une contrainte de clé étrangère, par exemple, dépend d'une contrainte de clé primaire ou d'unicité sur la(les) colonne(s) référencée(s).

Cela fonctionne de la même manière pour tous les types de contrainte, à l'exception des contraintes NOT NULL. Pour supprimer une contrainte NOT NULL, on écrit :

```
ALTER TABLE produits ALTER COLUMN no_produit DROP NOT NULL;
```

(Les contraintes NOT NULL n'ont pas de noms.)

5.5.5. Modifier la valeur par défaut d'une colonne

La commande de définition d'une nouvelle valeur par défaut de colonne ressemble à celle-ci :

```
ALTER TABLE produits ALTER COLUMN prix SET DEFAULT 7.77;
```

Cela n'affecte pas les lignes existantes de la table, mais uniquement la valeur par défaut pour les futures commandes **INSERT**.

Pour retirer toute valeur par défaut, on écrit :

```
ALTER TABLE produits ALTER COLUMN prix DROP DEFAULT;
```

C'est équivalent à mettre la valeur par défaut à NULL. En conséquence, il n'y a pas d'erreur à retirer une valeur par défaut qui n'a pas été définie car NULL est la valeur par défaut implicite.

5.5.6. Modifier le type de données d'une colonne

La commande de conversion du type de données d'une colonne ressemble à celle-ci :

```
ALTER TABLE produits ALTER COLUMN prix TYPE numeric(10,2);
```

Elle ne peut réussir que si chaque valeur de la colonne peut être convertie dans le nouveau type par une conversion implicite. Si une conversion plus complexe est nécessaire, une clause `USING` peut être ajoutée qui indique comment calculer les nouvelles valeurs à partir des anciennes.

PostgreSQL™ tente de convertir la valeur par défaut de la colonne le cas échéant, ainsi que toute contrainte impliquant la colonne. Mais ces conversions peuvent échouer ou produire des résultats surprenants. Il est souvent préférable de supprimer les contraintes de la colonne avant d'en modifier le type, puis d'ajouter ensuite les contraintes convenablement modifiées.

5.5.7. Renommer une colonne

Pour renommer une colonne :

```
ALTER TABLE produits RENAME COLUMN no_produit TO numero_produit;
```

5.5.8. Renommer une table

Pour renommer une table :

```
ALTER TABLE produits RENAME TO elements;
```

5.6. Droits

Tout utilisateur qui crée un objet dans la base de données en devient le propriétaire. Par défaut, seul le propriétaire peut l'utiliser. Pour que d'autres utilisateurs puissent l'utiliser, des *privilèges* doivent être accordés (les superutilisateurs ont, toutefois, accès à tout objet, toujours).

Il existe un certain nombre de droits différents : `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE` et `USAGE`. Les droits applicables à un objet particulier varient selon le type d'objet (table, fonction...). La page de référence GRANT(7) fournit une information complète sur les différents types de droits gérés par PostgreSQL™. La section et les chapitres suivants présentent l'utilisation de ces droits.

Le droit de modifier ou de détruire un objet est le privilège du seul propriétaire.

Note

Pour modifier le propriétaire d'une table, d'un index, d'une séquence ou d'une vue, on utilise la commande ALTER TABLE(7). Il existe des commandes `ALTER` correspondantes pour les autres types d'objets.

La commande **GRANT** est utilisée pour accorder des privilèges (on dit aussi « granter un privilège »). Par exemple, si `joe` est un utilisateur et `comptes` une table, le privilège d'actualiser la table `comptes` peut être accordé à `joe` avec :

```
GRANT UPDATE ON comptes TO joe;
```

Écrire `ALL` à la place d'un droit spécifique accorde tous les droits applicables à ce type d'objet.

Le nom d'« utilisateur » spécial `PUBLIC` peut être utilisé pour donner un privilège à tous les utilisateurs du système. De plus, les rôles de type « group » peuvent être configurés pour aider à la gestion des droits quand il y a beaucoup d'utilisateurs dans une base -- pour les détails, voir Chapitre 20, Rôles et droits de la base de données.

Pour révoquer un privilège, on utilise la commande bien-nommée **REVOKE**, comme dans l'exemple ci-dessous :

```
REVOKE ALL ON comptes FROM PUBLIC;
```

Les privilèges spéciaux du propriétaire de l'objet (c'est-à-dire, le droit d'exécuter **DROP**, **GRANT**, **REVOKE**, etc.) appartiennent toujours implicitement au propriétaire. Il ne peuvent être ni accordés ni révoqués. Mais le propriétaire de l'objet peut choisir de révoquer ses propres droits ordinaires pour, par exemple, mettre une table en lecture seule pour lui-même et pour les autres.

Habituellement, seul le propriétaire de l'objet (ou un superutilisateur) peut granter ou révoquer les droits sur un objet. Néanmoins, il est possible de donner un privilège « avec possibilité de transmission » (*« with grant option »*), qui donne à celui qui le reçoit la permission de le donner à d'autres. Si cette option est ensuite révoquée, alors tous ceux qui ont reçu ce privilège par cet utilisateur (directement ou indirectement via la chaîne des dons) perdent ce privilège. Pour les détails, voir les pages de références GRANT(7) et REVOKE(7).

5.7. Schémas

Un cluster de bases de données PostgreSQL™ contient une ou plusieurs base(s) nommée(s). Si les utilisateurs et groupes d'utilisateurs sont partagés sur l'ensemble du cluster, aucune autre donnée n'est partagée. Toute connexion cliente au serveur ne peut accéder qu'aux données d'une seule base, celle indiquée dans la requête de connexion.

> **Note**
>
> Les utilisateurs d'un cluster n'ont pas obligatoirement le droit d'accéder à toutes les bases du cluster. Le partage des noms d'utilisateur signifie qu'il ne peut pas y avoir plusieurs utilisateurs nommés `joe`, par exemple, dans deux bases du même cluster ; mais le système peut être configuré pour n'autoriser `joe` à accéder qu'à certaines bases.

Une base de données contient un ou plusieurs *schéma*(s) nommé(s) qui, eux, contiennent des tables. Les schémas contiennent aussi d'autres types d'objets nommés (types de données, fonctions et opérateurs, par exemple). Le même nom d'objet peut être utilisé dans différents schémas sans conflit ; par exemple, `schema1` et `mon_schema` peuvent tous les deux contenir une table nommée `ma_table`. À la différence des bases de données, les schémas ne sont pas séparés de manière rigide : un utilisateur peut accéder aux objets de n'importe quel schéma de la base de données à laquelle il est connecté, sous réserve qu'il en ait le droit.

Il existe plusieurs raisons d'utiliser les schémas :

- autoriser de nombreux utilisateurs à utiliser une base de données sans interférer avec les autres ;
- organiser les objets de la base de données en groupes logiques afin de faciliter leur gestion ;
- les applications tiers peuvent être placées dans des schémas séparés pour éviter les collisions avec les noms d'autres objets.

Les schémas sont comparables aux répertoires du système d'exploitation, à ceci près qu'ils ne peuvent pas être imbriqués.

5.7.1. Créer un schéma

Pour créer un schéma, on utilise la commande CREATE SCHEMA(7). Le nom du schéma est libre. Par exemple :

```
CREATE SCHEMA mon_schema;
```

Pour créer les objets d'un schéma ou y accéder, on écrit un *nom qualifié* constitué du nom du schéma et du nom de la table séparés par un point :

```
schema.table
```

Cela fonctionne partout où un nom de table est attendu, ce qui inclut les commandes de modification de la table et les commandes d'accès aux données discutées dans les chapitres suivants. (Pour des raisons de simplification, seules les tables sont évoquées, mais les mêmes principes s'appliquent aux autres objets nommés, comme les types et les fonctions.)

La syntaxe encore plus générale

```
base.schema.table
```

peut aussi être utilisée, mais à l'heure actuelle, cette syntaxe n'existe que pour des raisons de conformité avec le standard SQL. Si un nom de base de données est précisé, ce doit être celui de la base à laquelle l'utilisateur est connecté.

Pour créer une table dans le nouveau schéma, on utilise :

```
CREATE TABLE mon_schema.ma_table (
 ...
);
```

Pour effacer un schéma vide (tous les objets qu'il contient ont été supprimés), on utilise :

```
DROP SCHEMA mon_schema;
```

Pour effacer un schéma et les objets qu'il contient, on utilise :

```
DROP SCHEMA mon_schema CASCADE;
```

La Section 5.11, « Gestion des dépendances » décrit le mécanisme général sous-jacent.

Il n'est pas rare de vouloir créer un schéma dont un autre utilisateur est propriétaire (puisque c'est l'une des méthodes de restriction de l'activité des utilisateurs à des *namespaces* pré-définis). La syntaxe en est :

```
CREATE SCHEMA nom_schema AUTHORIZATION nom_utilisateur;
```

Le nom du schéma peut être omis, auquel cas le nom de l'utilisateur est utilisé. Voir la Section 5.7.6, « Utilisation » pour en connaître l'utilité.

Les noms de schéma commençant par `pg_` sont réservés pour les besoins du système et ne peuvent être créés par les utilisateurs.

5.7.2. Le schéma public

Dans les sections précédentes, les tables sont créées sans qu'un nom de schéma soit indiqué. Par défaut, ces tables (et les autres objets) sont automatiquement placées dans un schéma nommé « public ». Toute nouvelle base de données contient un tel schéma. Les instructions suivantes sont donc équivalentes :

```
CREATE TABLE produits ( ... );
```

et :

```
CREATE TABLE public.produits ( ... );
```

5.7.3. Chemin de parcours des schémas

Non seulement l'écriture de noms qualifiés est contraignante, mais il est, de toute façon, préférable de ne pas fixer un nom de schéma dans les applications. De ce fait, les tables sont souvent appelées par des *noms non-qualifiés*, soit le seul nom de la table. Le système détermine la table appelée en suivant un *chemin de recherche*, liste de schémas dans lesquels chercher. La première table correspondante est considérée comme la table voulue. S'il n'y a pas de correspondance, une erreur est remontée, quand bien même il existerait des tables dont le nom correspond dans d'autres schémas de la base.

Le premier schéma du chemin de recherche est appelé schéma courant. En plus d'être le premier schéma parcouru, il est aussi le schéma dans lequel les nouvelles tables sont créées si la commande **CREATE TABLE** ne précise pas de nom de schéma.

Le chemin de recherche courant est affiché à l'aide de la commande :

```
SHOW search_path;
```

Dans la configuration par défaut, ceci renvoie :

```
 search_path
--------------
 "$user",public
```

Le premier élément précise qu'un schéma de même nom que l'utilisateur courant est recherché. En l'absence d'un tel schéma, l'entrée est ignorée. Le deuxième élément renvoie au schéma public précédemment évoqué.

C'est, par défaut, dans le premier schéma du chemin de recherche qui existe que sont créés les nouveaux objets. C'est la raison pour laquelle les objets sont créés, par défaut, dans le schéma public. Lorsqu'il est fait référence à un objet, dans tout autre contexte, sans qualification par un schéma (modification de table, modification de données ou requêtes), le chemin de recherche est traversé jusqu'à ce qu'un objet correspondant soit trouvé. C'est pourquoi, dans la configuration par défaut, tout accès non qualifié ne peut que se référer au schéma public.

Pour ajouter un schéma au chemin, on écrit :

```
SET search_path TO mon_schema,public;
```

(`$user` est omis à ce niveau car il n'est pas immédiatement nécessaire.) Il est alors possible d'accéder à la table sans qu'elle soit qualifiée par un schéma :

```
DROP TABLE ma_table;
```

Puisque `mon_schema` est le premier élément du chemin, les nouveaux objets sont, par défaut, créés dans ce schéma.

On peut aussi écrire :

```
SET search_path TO mon_schema;
```

Dans ce cas, le schéma public n'est plus accessible sans qualification explicite. Hormis le fait qu'il existe par défaut, le schéma public n'a rien de spécial. Il peut même être effacé.

On peut également se référer à la Section 9.23, « Fonctions d'informations système » qui détaille les autres façons de manipuler le chemin de recherche des schémas.

Le chemin de recherche fonctionne de la même façon pour les noms de type de données, les noms de fonction et les noms d'opérateur que pour les noms de table. Les noms des types de données et des fonctions peuvent être qualifiés de la même façon que les noms de table. S'il est nécessaire d'écrire un nom d'opérateur qualifié dans une expression, il y a une condition spéciale. Il faut écrire :

```
OPERATOR(schéma.opérateur)
```

Cela afin d'éviter toute ambiguïté syntaxique. Par exemple :

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

En pratique, il est préférable de s'en remettre au chemin de recherche pour les opérateurs, afin de ne pas avoir à écrire quelque chose d'aussi étrange.

5.7.4. Schémas et privilèges

Par défaut, les utilisateurs ne peuvent pas accéder aux objets présents dans les schémas qui ne leur appartiennent pas. Pour le permettre, le propriétaire du schéma doit donner le droit `USAGE` sur le schéma. Pour autoriser les utilisateurs à manipuler les objets d'un schéma, des privilèges supplémentaires doivent éventuellement être accordés, en fonction de l'objet.

Un utilisateur peut aussi être autorisé à créer des objets dans le schéma d'un d'autre. Pour cela, le privilège `CREATE` sur le schéma doit être accordé. Par défaut, tout le monde bénéficie des droits `CREATE` et `USAGE` sur le schéma `public`. Cela permet à tous les utilisateurs qui peuvent se connecter à une base de données de créer des objets dans son schéma `public`. Si cela ne doit pas être le cas, ce privilège peut être révoqué :

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Le premier « public » est le schéma, le second « public » signifie « tout utilisateur ». Dans le premier cas, c'est un identifiant, dans le second, un mot clé, d'où la casse différente. (Se reporter aux règles de la Section 4.1.1, « identificateurs et mots clés ».)

5.7.5. Le schéma du catalogue système

En plus du schéma `public` et de ceux créés par les utilisateurs, chaque base de données contient un schéma `pg_catalog`. Celui-ci contient les tables systèmes et tous les types de données, fonctions et opérateurs intégrés. `pg_catalog` est toujours dans le chemin de recherche. S'il n'est pas nommé explicitement dans le chemin, il est parcouru implicitement *avant* le parcours des schémas du chemin. Cela garantit que les noms internes sont toujours accessibles. En revanche, `pg_catalog` peut être explicitement placé à la fin si les noms utilisateur doivent surcharger les noms internes.

Dans les versions de PostgreSQL™ antérieures à la 7.3, les noms de table commençant par `pg_` étaient réservés. Cela n'est plus vrai : une telle table peut être créée dans n'importe quel schéma qui n'est pas un schéma système. En revanche, il est préférable de continuer à éviter d'utiliser de tels noms pour se prémunir d'éventuels conflits si une version ultérieure devait définir une table système qui porte le même nom que la table créée. (Le chemin de recherche par défaut implique qu'une référence non qualifiée à cette table pointe sur la table système). Les tables systèmes continueront de suivre la convention qui leur impose des noms préfixés par `pg_`. Il n'y a donc pas de conflit possible avec des noms de table utilisateur non qualifiés, sous réserve que les utilisateurs évitent le préfixe `pg_`.

5.7.6. Utilisation

Les schémas peuvent être utilisés de différentes façons pour organiser les données. Certaines d'entre elles, recommandées, sont facilement supportés par la configuration par défaut :

- si aucun schéma n'est créé, alors tous les utilisateurs ont implicitement accès au schéma public. Cela permet de simuler une situation dans laquelle les schémas ne sont pas disponibles. Cette situation est essentiellement recommandée lorsqu'il n'y a qu'un utilisateur, ou un très petit nombre d'utilisateurs qui coopèrent au sein d'une base de données. Cette configuration permet aussi d'opérer une transition en douceur depuis un monde où les schémas sont inconnus ;
- pour chaque utilisateur, un schéma, de nom identique à celui de l'utilisateur, peut être créé. Le chemin de recherche par défaut commence par `$user`, soit le nom de l'utilisateur. Si tous les utilisateurs disposent d'un schéma distinct, ils accèdent, par défaut, à leur propre schéma. Dans cette configuration, il est possible de révoquer l'accès au schéma public (voire de supprimer ce schéma) pour confiner les utilisateurs dans leur propre schéma ;
- l'installation d'applications partagées (tables utilisables par tout le monde, fonctionnalités supplémentaires fournies par des applications tiers, etc) peut se faire dans des schémas distincts. Il faut alors accorder des privilèges appropriés pour permettre aux autres utilisateurs d'y accéder. Les utilisateurs peuvent alors se référer à ces objets additionnels en qualifiant leur nom du nom de schéma ou ajouter les schémas supplémentaires dans leur chemin de recherche, au choix.

5.7.7. Portabilité

Dans le standard SQL, la notion d'objets d'un même schéma appartenant à des utilisateurs différents n'existe pas. De plus, certaines implantations ne permettent pas de créer des schémas de nom différent de celui de leur propriétaire. En fait, les concepts de schéma et d'utilisateur sont presque équivalents dans un système de base de données qui n'implante que le support basique des schémas tel que spécifié dans le standard. De ce fait, beaucoup d'utilisateurs considèrent les noms qualifiés comme correspondant en réalité à *`utilisateur.table`*. C'est comme cela que PostgreSQL™ se comporte si un schéma utilisateur est créé pour chaque utilisateur.

Le concept de schéma `public` n'existe pas non plus dans le standard SQL. Pour plus de conformité au standard, le schéma `public` ne devrait pas être utilisé (voire être supprimé).

Certains systèmes de bases de données n'implantent pas du tout les schémas, ou fournissent le support de *namespace* en autorisant (peut-être de façon limitée) l'accès inter-bases de données. Dans ce cas, la portabilité maximale est obtenue en n'utilisant pas les

schémas.

5.8. L'héritage

PostgreSQL™ implante l'héritage des tables, qui peut s'avérer très utile pour les concepteurs de bases de données. (SQL:1999 et les versions suivantes définissent une fonctionnalité d'héritage de type qui diffère par de nombreux aspects des fonctionnalités décrites ici.)

Soit l'exemple d'un modèle de données de villes. Chaque état comporte plusieurs villes mais une seule capitale. Pour récupérer rapidement la ville capitale d'un état donné, on peut créer deux tables, une pour les capitales et une pour les villes qui ne sont pas des capitales. Mais, que se passe-t'il dans le cas où toutes les données d'une ville doivent être récupérées, qu'elle soit une capitale ou non ? L'héritage peut aider à résoudre ce problème. La table capitales est définie pour hériter de villes :

```
CREATE TABLE villes (
    nom             text,
    population      float,
    altitude        int     -- (en pied)
);

CREATE TABLE capitales (
    etat           char(2)
) INHERITS (villes);
```

Dans ce cas, la table capitales *hérite* de toutes les colonnes de sa table parent, villes. Les capitales ont aussi une colonne supplémentaire, `etat`, qui indique l'état dont elles sont capitales.

Dans PostgreSQL™, une table peut hériter de zéro à plusieurs autres tables et une requête faire référence aux lignes d'une table ou à celles d'une table et de ses descendantes. Ce dernier comportement est celui par défaut.

Par exemple, la requête suivante retourne les noms et altitudes de toutes les villes, y compris les capitales, situées à une altitude supérieure à 500 pieds :

```
SELECT nom, altitude
    FROM villes
    WHERE altitude > 500;
```

Avec les données du tutoriel de PostgreSQL™ (voir Section 2.1, « Introduction »), ceci renvoie :

```
   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

D'un autre côté, la requête suivante retourne les noms et altitudes de toutes les villes, qui ne sont pas des capitales, situées à une altitude supérieure à 500 pieds :

```
SELECT nom, altitude
    FROM ONLY villes
    WHERE altitude > 500;

   nom     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

Le mot clé `ONLY` indique que la requête s'applique uniquement aux villes, et non pas à toutes les tables en-dessous de villes dans la hiérarchie de l'héritage. Un grand nombre des commandes déjà évoquées -- **SELECT**, **UPDATE** et **DELETE** -- supportent le mot clé `ONLY`.

Vous pouvez aussi écrire le nom de la table avec une `*` à la fin pour indiquer spécifiquement que les tables filles sont incluses :

```
SELECT name, altitude
    FROM cities*
    WHERE altitude > 500;
```

Indiquer `*` n'est pas nécessaire car ce comportement est le comportement par défaut (sauf si vous avez modifié la configuration du paramètre sql_inheritance). Néanmoins, écrire `*` pourrait être utile pour insister sur le fait que des tables supplémentaires seront parcourues.

Dans certains cas, il peut être intéressant de savoir de quelle table provient une ligne donnée. Une colonne système appelée `TA-`

BLEOID présente dans chaque table donne la table d'origine :

```
SELECT v.tableoid, v.nom, v.altitude
FROM villes v
WHERE v.altitude > 500;
```

qui renvoie :

```
 tableoid |   nom     | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

(Reproduire cet exemple conduit probablement à des OID numériques différents). Une jointure avec pg_class, permet d'obtenir les noms réels des tables :

```
SELECT p.relname, v.nom, v.altitude
FROM villes v, pg_class p
WHERE v.altitude > 500 AND v.tableoid = p.oid;
```

ce qui retourne :

```
 relname   |   nom     | altitude
-----------+-----------+----------
 villes    | Las Vegas |     2174
 villes    | Mariposa  |     1953
 capitales | Madison   |      845
```

L'héritage ne propage pas automatiquement les données des commandes **INSERT** ou **COPY** aux autres tables de la hiérarchie de l'héritage. Dans l'exemple considéré, l'instruction **INSERT** suivante échoue :

```
INSERT INTO villes (nom, population, altitude, etat)
VALUES ('New York', NULL, NULL, 'NY');
```

On pourrait espérer que les données soient magiquement routées vers la table capitales mais ce n'est pas le cas : **INSERT** insère toujours dans la table indiquée. Dans certains cas, il est possible de rediriger l'insertion en utilisant une règle (voir Chapitre 37, Système de règles). Néanmoins, cela n'est d'aucune aide dans le cas ci-dessus car la table villes ne contient pas la colonne *etat*. La commande est donc rejetée avant que la règle ne soit appliquée.

Toutes les contraintes de vérification et toutes les contraintes NOT NULL sur une table parent sont automatiquement héritées par les tables enfants. Les autres types de contraintes (unicité, clé primaire, clé étrangère) ne sont pas hérités.

Une table peut hériter de plusieurs tables, auquel cas elle possède l'union des colonnes définies par les tables mèress. Toute colonne déclarée dans la définition de la table enfant est ajoutée à cette dernière. Si le même nom de colonne apparaît dans plusieurs tables mères, ou à la fois dans une table mère et dans la définition de la table enfant, alors ces colonnes sont « assemblées » pour qu'il n'en existe qu'une dans la table enfant. Pour être assemblées, les colonnes doivent avoir le même type de données, sinon une erreur est levée. La colonne assemblée hérite de toutes les contraintes de vérification en provenance de chaque définition de colonnes dont elle provient, et est marquée NOT NULL si une d'entre elles l'est.

L'héritage de table est établi à la création de la table enfant, à l'aide de la clause `INHERITS` de l'instruction CREATE TABLE(7). Alternativement, il est possible d'ajouter à une table, définie de façon compatible, une nouvelle relation de parenté à l'aide de la clause `INHERIT` de ALTER TABLE(7). Pour cela, la nouvelle table enfant doit déjà inclure des colonnes de mêmes nom et type que les colonnes de la table parent. Elle doit aussi contenir des contraintes de vérification de mêmes nom et expression que celles de la table parent.

De la même façon, un lien d'héritage peut être supprimé d'un enfant à l'aide de la variante `NO INHERIT` d'**ALTER TABLE**. Ajouter et supprimer dynamiquement des liens d'héritage de cette façon est utile quand cette relation d'héritage est utilisée pour le partitionnement des tables (voir Section 5.9, « Partitionnement »).

Un moyen pratique de créer une table compatible en vue d'en faire ultérieurement une table enfant est d'utiliser la clause `LIKE` dans **CREATE TABLE**. Ceci crée une nouvelle table avec les même colonnes que la table source. S'il existe des contraintes `CHECK` définies sur la table source, l'option `INCLUDING CONSTRAINTS` de `LIKE` doit être indiquée car le nouvel enfant doit avoir des contraintes qui correspondent à celles du parent pour être considérée compatible.

Une table mère ne peut pas être supprimée tant qu'elle a des enfants. Pas plus que les colonnes ou les contraintes de vérification des tables enfants ne peuvent être supprimées ou modifiées si elles sont héritées. La suppression d'une table et de tous ces descendants peut être aisément obtenue en supprimant la table mère avec l'option `CASCADE`.

ALTER TABLE(7) propage toute modification dans les définitions des colonnes et contraintes de vérification à travers la hiérarchie d'héritage. Là encore, supprimer des colonnes qui dépendent d'autres tables mères n'est possible qu'avec l'option `CASCADE`. **ALTER TABLE** suit les mêmes règles d'assemblage de colonnes dupliquées et de rejet que l'instruction **CREATE TABLE**.

Notez comment sont gérés les droits d'accès aux tables. Exécuter une requête sur une table parent permet automatiquement d'accéder aux données des tables enfants sans vérification supplémentaire sur les droits. Ceci préserve l'apparence que les données proviennent de la table parent. L'accès aux tables enfants directement est, néanmoins, pas automatiquement permis et nécessitera la vérification des droits sur ces tables.

5.8.1. Restrictions

Notez que toutes les commandes SQL fonctionnent avec les héritages. Les commandes utilisées pour récupérer des données, pour modifier des données ou pour modifier le schéma (autrement dit `SELECT`, `UPDATE`, `DELETE`, la plupart des variantes de `ALTER TABLE`, mais pas `INSERT` ou `ALTER TABLE ... RENAME`) incluent par défaut les tables filles et supportent la notation `ONLY` pour les exclure. Les commandes qui font de la maintenance de bases de données et de la configuration (par exemple `REINDEX`, `VACUUM`) fonctionnent typiquement uniquement sur les tables physiques, individuelles et ne supportent pas la récursion sur les tables de l'héritage. Le comportement respectif de chaque commande individuelle est documenté dans la référence (Commandes SQL).

Il existe une réelle limitation à la fonctionnalité d'héritage : les index (dont les contraintes d'unicité) et les contraintes de clés étrangères ne s'appliquent qu'aux tables mères, pas à leurs héritiers. Cela est valable pour le côté référençant et le côté référencé d'une contrainte de clé étrangère. Ce qui donne, dans les termes de l'exemple ci-dessus :

- si villes.*nom* est déclarée `UNIQUE` ou clé primaire (`PRIMARY KEY`), cela n'empêche pas la table capitales de posséder des lignes avec des noms dupliqués dans villes. Et ces lignes upliquées s'affichent par défaut dans les requêtes sur villes. En fait, par défaut, capitales n'a pas de contrainte d'unicité du tout et, du coup, peut contenir plusieurs lignes avec le même nom. Une contrainte d'unicité peut être ajoutée à capitales mais cela n'empêche pas la duplication avec villes ;
- de façon similaire, si villes.*nom* fait référence (`REFERENCES`) à une autre table, cette contrainte n'est pas automatiquement propagée à capitales. Il est facile de contourner ce cas de figure en ajoutant manuellement la même contrainte `REFERENCES` à capitales ;
- si une autre table indique `REFERENCES villes(nom)`, cela l'autorise à contenir les noms des villes mais pas les noms des capitales. Il n'existe pas de contournement efficace de ce cas.

Ces déficiences seront probablement corrigées dans une version future, mais, en attendant, il est obligatoire de réfléchir consciencieusement à l'utilité de l'héritage pour une application donnée.

5.9. Partitionnement

PostgreSQL™ offre un support basique du partitionnement de table. Cette section explique pourquoi et comment implanter le partitionnement lors de la conception de la base de données.

5.9.1. Aperçu

Le partitionnement fait référence à la division d'une table logique volumineuse en plusieurs parties physiques plus petites. Le partitionnement comporte de nombreux avantages :

- les performances des requêtes peuvent être significativement améliorées dans certaines situations, particulièrement lorsque la plupart des lignes fortement accédées d'une table se trouvent sur une seule partition ou sur un petit nombre de partitions. Le partitionnement se substitue aux colonnes principales des index, réduisant ainsi la taille des index et facilitant la tenue en mémoire des parties les plus utilisées de l'index ;
- lorsque les requêtes ou les mises à jour accèdent à un important pourcentage d'une seule partition, les performances peuvent être grandement améliorées par l'utilisation avantageuse de parcours séquentiels sur cette partition plutôt que d'utiliser un index et des lectures aléatoires réparties sur toute la table ;
- les chargements et suppressions importants de données peuvent être obtenus par l'ajout ou la suppression de partitions, sous réserve que ce besoin ait été pris en compte lors de la conception du partitionnement. **ALTER TABLE NO INHERIT** et **DROP TABLE** sont bien plus rapide qu'une opération de masse. Cela supprime également la surcharge due au **VACUUM** causé par un **DELETE** massif ;
- les données peu utilisées peuvent être déplacées sur un média de stockage moins cher et plus lent.

Les bénéfices ne sont réellement intéressants que si cela permet d'éviter une table autrement plus volumineuse. Le point d'équilibre exact à partir duquel une table tire des bénéfices du partitionnement dépend de l'application. Toutefois, le partitionnement doit être envisagé si la taille de la table peut être amenée à dépasser la taille de la mémoire physique du serveur.

Actuellement, PostgreSQL™ supporte le partitionnement à travers l'héritage de tables. Chaque partition doit être créée comme une table enfant d'une unique table parent. La table parent est, elle, habituellement vide ; elle n'existe que pour représenter l'ensemble

complet des données. Il est impératif de maîtriser les concepts de l'héritage (voir Section 5.8, « L'héritage ») avant de tenter d'implanter le partitionnement.

Les formes suivantes de partitionnement peuvent être implantées dans PostgreSQL™ :

Partitionnement par échelon
: La table est partitionnée en « intervalles » (ou échelles) définis par une colonne clé ou par un ensemble de colonnes, sans recouvrement entre les échelles de valeurs affectées aux différentes partitions. Il est possible, par exemple, de partitionner par échelles de date ou par échelles d'identifiants pour des objets métier particuliers.

Partitionnement par liste
: La table est partitionnée en listant explicitement les valeurs clés qui apparaissent dans chaque partition.

5.9.2. Partitionner

Pour partionner une table, la procédure est la suivante :

1. Créer la table « maître ». C'est de celle-ci qu'héritent toutes les partitions.

   Cette table ne contient pas de données. Les contraintes de vérification ne doivent être définies sur cette table que si elles sont appliquées à toutes les partitions. Il n'y a de plus aucune raison de définir des index ou des contraintes d'unicité sur cette table.
2. Créer plusieurs tables « filles » (ou enfants) qui héritent chacune de la table maître. Normalement, ces tables n'ajoutent pas de colonnes à l'ensemble hérité du maître.

   Par la suite, les tables enfants sont appelées partitions, bien qu'elles soient, en tout point, des tables PostgreSQL™ normales.
3. Ajouter les contraintes de tables aux tables de partitions pour définir les valeurs des clés autorisées dans chacune.

   Quelques exemples typiques :

   ```
   CHECK ( x = 1 )
   CHECK ( comté IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( ID >= 100 AND ID < 200 )
   ```

   Les contraintes doivent garantir qu'il n'y a pas de recouvrement entre les valeurs clés autorisées dans les différentes partitions. Une erreur commune est de configurer des contraintes d'échelle de cette façon :

   ```
   CHECK ( comté BETWEEN 100 AND 200 )
   CHECK ( comté BETWEEN 200 AND 300 )
   ```

   Il est dans ce cas difficile de savoir à quelle partition appartient la clé 200.

   Il n'y a aucune différence entre les syntaxes de partitionnement par échelon et de partitionnement par liste ; ces termes ne sont que descriptifs.
4. Pour chaque partition, créer un index sur la (ou les) colonne(s) clé(s), ainsi que tout autre index nécessaire. (L'index clé n'est pas vraiment nécessaire mais, dans la plupart des scénarios, il est utile. Si les valeurs clés doivent être uniques, alors il faut toujours créer une contrainte d'unicité ou de clé primaire pour chaque partition.)
5. Optionnellement, définir un déclencheur ou une règle pour rediriger les données insérées dans la table maître vers la partition appropriée.
6. S'assurer que le paramètre de configuration constraint_exclusion n'est pas désactivé dans `postgresql.conf`. S'il l'est, les requêtes ne sont pas optimisées.

Soit la base de données d'une grande fabrique de glaces. La compagnie mesure le pic de température journalier ainsi que les ventes de glaces dans chaque région. Conceptuellement, la table ressemble à :

```
CREATE TABLE mesure (
    id_ville        int not null,
    date_trace      date not null,
    temperature     int,
    ventes          int
);
```

La plupart des requêtes n'accèdent qu'aux données de la dernière semaine, du dernier mois ou du dernier trimestre car cette table est essentiellement utilisée pour préparer des rapports en ligne pour la direction. Pour réduire le nombre de données anciennes à stocker, seules les trois dernières années sont conservées. Au début de chaque mois, les données du mois le plus ancien sont supprimées.

Dans cette situation, le partitionnement permet de répondre aux différents besoins identifiés sur la table des mesures. En suivant les étapes indiquées ci-dessus, le partitionnement peut être configuré de la façon suivante :

1. la table maître est la table mesure, déclarée exactement comme ci-dessus ;

2. une partition est ensuite créée pour chaque mois actif :

```
CREATE TABLE mesure_a2006m02 ( ) INHERITS (mesure);
CREATE TABLE mesure_a2006m03 ( ) INHERIT (mesure);
...
CREATE TABLE mesure_a2007m11 ( ) INHERITS (mesure);
CREATE TABLE mesure_a2007m12 ( ) INHERITS (mesure);
CREATE TABLE mesure_a2008m01 ( ) INHERITS (mesure);
```

   Chaque partition est une table à part entière mais sa définition est héritée de la table mesure.

   Ceci résoud un des problèmes : la suppression d'anciennes données. Chaque mois, il suffit d'effectuer un **DROP TABLE** sur la table enfant la plus ancienne et de créer une nouvelle table enfant pour les données du nouveau mois.
3. Il est nécessaire de fournir des contraintes de table qui interdisent les recouvrements. Plutôt que de simplement créer les tables de la partition comme ci-dessus, le script de création de tables ressemble à ;

```
CREATE TABLE mesure_a2006m02 (
    CHECK ( date_trace >= DATE '2006-02-01' AND date_trace < DATE '2006-03-01' )
) INHERITS (mesure);
CREATE TABLE mesure_a2006m03 (
    CHECK ( date_trace >= DATE '2006-03-01' AND date_trace < DATE '2006-04-01' )
) INHERITS (mesure);
...
CREATE TABLE mesure_a2007m11 (
    CHECK ( date_trace >= DATE '2007-11-01' AND date_trace < DATE '2007-12-01' )
) INHERITS (mesure);
CREATE TABLE mesure_a2007m12 (
    CHECK ( date_trace >= DATE '2007-12-01' AND date_trace < DATE '2007-01-01' )
) INHERITS (mesure);
CREATE TABLE mesure_a2008m01 (
    CHECK ( date_trace >= DATE '2008-01-01' AND date_trace < DATE '2008-02-01' )
) INHERITS (mesure);
```

4. Des index sur les colonnes clés sont probablement nécessaires :

```
CREATE INDEX mesure_a2006m02_date_trace ON mesure_a2006m02 (date_trace);
CREATE INDEX mesure_a2006m03_date_trace ON mesure_a2006m03 (date_trace);
...
CREATE INDEX mesure_a2007m11_date_trace ON mesure_a2007m11 (date_trace);
CREATE INDEX mesure_a2007m12_date_trace ON mesure_a2007m12 (date_trace);
CREATE INDEX mesure_a2008m01_date_trace ON mesure_a2008m01 (date_trace);
```

   À ce stade, c'est suffisant.
5. L'application doit dire `INSERT INTO mesure...` et les données être redirigées dans la table de partition appropriée. Pour cela une fonction déclencheur est attachée à la table maître. Si les données ne sont ajoutées que dans la dernière partition, la fonction est très simple.

```
CREATE OR REPLACE FUNCTION mesure_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO mesure_a2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

   Le déclencheur qui appelle la fonction est créé à sa suite :

```
CREATE TRIGGER insert_mesure_trigger
    BEFORE INSERT ON mesure
    FOR EACH ROW EXECUTE PROCEDURE mesure_insert_trigger();
```

   La fonction déclencheur doit être redéfinie chaque mois pour qu'elle pointe toujours sur la partition active. La définition du déclencheur n'a pas besoin d'être redéfinie.

   Il est également possible de laisser le serveur localiser la partition dans laquelle doit être insérée la ligne proposée en entrée. Une fonction déclencheur plus complexe peut être utilisée pour cela :

```
CREATE OR REPLACE FUNCTION mesure_insert_trigger()
RETURNS TRIGGER AS $$
```

```
BEGIN
    IF ( NEW.date_trace >= DATE '2006-02-01' AND
         NEW.date_trace < DATE '2006-03-01' ) THEN
        INSERT INTO mesure_a2006m02 VALUES (NEW.*);
    ELSIF ( NEW.date_trace >= DATE '2006-03-01' AND
            NEW.date_trace < DATE '2006-04-01' ) THEN
        INSERT INTO mesure_a2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.date_trace >= DATE '2008-01-01' AND
            NEW.date_trace < DATE '2008-02-01' ) THEN
        INSERT INTO mesure_a2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date en dehors de l''échelle. Corrigez la fonction
mesure_insert_trigger() !';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

La définition du déclencheur ne change pas. Chaque test `IF` doit correspondre exactement à la contrainte `CHECK` de cette partition.

Bien que cette fonction soit plus complexe que celle du mois seul, il n'est pas nécessaire de l'actualiser aussi fréquemment, les branches pouvant être ajoutées avant d'être utiles.

Note

En pratique, il pourrait préférable de vérifier prioritairement la dernière partition créée si la plupart des insertions lui sont destinées. Pour des raisons de simplicité, les tests du déclencheur sont présentés dans le même ordre que les autres parties de l'exemple.

Un schéma complexe de partitionnement peut amener à écrire une grande quantité de DDL. Dans l'exemple ci-dessus, une nouvelle partition est écrite chaque mois. Il est donc conseillé d'écrire un script qui engendre automatiquement la DDL requise.

5.9.3. Gérer les partitions

Généralement, l'ensemble des partitions établies lors de la définition initiale de la table n'a pas pour but de rester statique. Il n'est pas inhabituel de supprimer d'anciennes partitions de données et d'en ajouter périodiquement de nouvelles pour de nouvelles données. Un des principaux avantages du partitionnement est précisément qu'il autorise une exécution quasi-instantanée de cette tâche, bien plus difficile autrement, en permettant la manipulation de la structure de la partition, plutôt que de déplacer physiquement de grands volumes de données.

L'option la plus simple pour supprimer d'anciennes données consiste à supprimer la partition qui n'est plus nécessaire :

```
DROP TABLE mesure_a2006m02;
```

Cela permet de supprimer très rapidement des millions d'enregistrements car il n'est nul besoin de supprimer séparément chaque enregistrement.

Une autre option, souvent préférable, consiste à supprimer la partition de la table partitionnée mais de conserver l'accès à la table en tant que telle :

```
ALTER TABLE mesure_a2006m02 NO INHERIT mesure;
```

Ceci permet la réalisation d'opérations ultérieures sur les données avant qu'elles ne soient supprimées. Par exemple, c'est souvent le bon moment pour sauvegarder les données en utilisant **COPY**, pg_dump ou tout autres outil. C'est aussi le moment d'agréger des données en des formats plus denses, de réaliser d'autres opérations sur les données ou de créer des rapports.

De façon similaire, une nouvelle partition peut être ajoutée pour gérer les nouvelles données. Une partition vide peut être créée dans la table partitionnée de la même façon que les partitions individuelles créées plus haut :

```
CREATE TABLE mesure_a2008m02 (
    CHECK ( date_trace >= DATE '2008-02-01' AND date_trace < DATE '2008-03-01' )
    ) INHERITS (mesure);
```

Alternativement, il est parfois plus intéressant de créer la nouvelle table en dehors de la structure de partitionnement et de la transformer en une partition adéquate plus tard. Cela permet de charger les données, les vérifier et les transformer avant leur apparition

dans la table partitionnée :

```
CREATE TABLE mesure_a2008m02
  (LIKE mesure INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE mesure_a2008m02 ADD CONSTRAINT y2008m02
   CHECK ( date_trace >= DATE '2008-02-01' AND date_trace < DATE '2008-03-01' );
\copy mesure_a2008m02 from 'mesure_a2008m02'
-- quelques travaux de préparation des données
ALTER TABLE mesure_a2008m02 INHERIT mesure;
```

5.9.4. Partitionnement et exclusion de contrainte

L'*exclusion de contrainte* est une technique d'optimisation des requêtes pour améliorer les performances sur les tables partitionnées telles que décrites plus haut. Par exemple :

```
SET constraint_exclusion = on;
SELECT count(*) FROM mesure WHERE date_trace >= DATE '2008-01-01';
```

Sans exclusion de contrainte, la requête ci-dessus parcourt chacune des partitions de la table mesure. Avec l'exclusion de contrainte activée, le planificateur examine les contraintes de chaque partition et tente de prouver que la partition qui n'a pas besoin d'être parcourue parce qu'elle ne peut pas contenir de lignes correspondant à la clause `WHERE` de la requête. Quand le planificateur peut le prouver, il exclut la partition du plan de requête.

La commande **EXPLAIN** permet d'afficher la différence entre un plan avec `constraint_exclusion` activé (*on*) et un plan avec ce paramètre désactivé (*off*). Un plan typique non optimisé pour ce type de table est :

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM mesure WHERE date_trace >= DATE '2008-01-01';

                                        QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
         ->  Seq Scan on mesure_a2006m02 mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
         ->  Seq Scan on mesure_ay2006m03 mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
...
         ->  Seq Scan on mesure_a2007m12 mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
         ->  Seq Scan on mesure_a2008m01 mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
```

Quelques partitions, voire toutes, peuvent utiliser des parcours d'index à la place des parcours séquentiels de la table complète mais le fait est qu'il n'est pas besoin de parcourir les anciennes partitions pour répondre à cette requête. Lorsque l'exclusion de contrainte est activée, un plan significativement moins coûteux est obtenu, qui délivre la même réponse :

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM mesure WHERE date_trace >= DATE '2008-01-01';
                                        QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
         ->  Seq Scan on mesure_a2008m01 mesure  (cost=0.00..30.38 rows=543 width=0)
               Filter: (date_trace >= '2008-01-01'::date)
```

L'exclusion de contraintes n'est pilotée que par les contraintes `CHECK`, pas par la présence d'index. Il n'est donc pas nécessaire de définir des index sur les colonnes clés. Le fait qu'un index doive être créé pour une partition donnée dépend de ce que les requêtes qui parcourent la partition parcourent en général une grande partie de la partition ou seulement une petite partie. Un index est utile dans le dernier cas, pas dans le premier.

La valeur par défaut (et donc recommandée) de constraint_exclusion n'est ni `on` ni `off`, mais un état intermédiaire appelé `partition`, qui fait que la technique est appliquée seulement aux requêtes qui semblent fonctionner avec des tables partitionnées. La valeur `on` fait que le planificateur examine les contraintes `CHECK` dans chaque requête, y compris les requêtes simples qui ont peu de chance d'en profiter.

5.9.5. Autre méthode de partitionnement

Une approche différente pour la redirection des insertions dans la table fille appropriée est de configurer des règles, à la place d'un déclencheur, sur la table maître. Par exemple :

```
CREATE RULE mesure_insert_a2006m02 AS
ON INSERT TO mesure WHERE
    ( date_trace >= DATE '2006-02-01' AND date_trace < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO mesure_a2006m02 VALUES (NEW.*);
...
CREATE RULE mesure_insert_a2008m01 AS
ON INSERT TO mesure WHERE
    ( date_trace >= DATE '2008-01-01' AND date_trace < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO mesure_a2008m01 VALUES (NEW.*);
```

Une règle est plus coûteuse qu'un déclencheur mais ce coût est payé une fois par requête au lieu d'une fois par ligne, cette méthode peut donc s'avérer avantageuse lors de grosses insertions. Néanmoins, dans la majorité des cas, la méthode du trigger offre de meilleures performances.

La commande **COPY** ignore les règles. Si **COPY** est utilisé pour insérer des données, la copie doit être effectuée sur la partition adéquate plutôt que dans la table maître. **COPY** active les déclencheurs. Elle peut donc être utilisée normalement lorsque cette approche est choisie.

Un autre inconvénient de la méthode des règles est qu'il n'existe pas de moyens simples de forcer une erreur si l'ensemble des règles ne couvre pas la date d'insertion. La donnée est alors silencieusement insérée dans la table maître.

Le partitionnement peut aussi être arrangé à l'aide d'une vue `UNION ALL`, en lieu et place de l'héritage. Par exemple :

```
CREATE VIEW mesure AS
          SELECT * FROM mesure_a2006m02
UNION ALL SELECT * FROM mesure_a2006m03
...
UNION ALL SELECT * FROM mesure_a2007m11
UNION ALL SELECT * FROM mesure_a2007m12
UNION ALL SELECT * FROM mesure_a2008m01;
```

Néanmoins, le besoin de recréer la vue ajoute une étape supplémentaire à l'ajout et à la suppression de partitions individuelles de l'ensemble des données. En pratique, cette méthode a peu d'intérêt au regard de l'héritage.

5.9.6. Restrictions

Les restrictions suivantes s'appliquent aux tables partitionnées :

- il n'existe pas de moyen automatique de vérifier que toutes les contraintes de vérification (`CHECK`) sont mutuellement exclusives. Il est plus sûr de créer un code qui fabrique les partitions et crée et/ou modifie les objets associés plutôt que de les créer manuellement ;
- les schémas montrés ici supposent que les colonnes clés du partitionnement d'une ligne ne changent jamais ou, tout du moins, ne changent pas suffisamment pour nécessiter un déplacement vers une autre partition. Une commande **UPDATE** qui tente de le faire échoue à cause des contraintes `CHECK`. Pour gérer ce type de cas, des déclencheurs peuvent être convenablement positionnés pour la mise à jour sur les tables de partition mais cela rend la gestion de la structure beaucoup plus complexe.
- si **VACUUM** ou **ANALYZE** sont lancés manuellement, il est obligatoire de les utiliser sur chaque partition. Une commande comme :

  ```
  ANALYZE mesure;
  ```

  ne traite que la table maître.

Les restrictions suivantes s'appliquent à l'exclusion de contraintes :

- l'exclusion de contrainte ne fonctionne que si la clause `WHERE` de la requête contient des constantes. Une requête avec paramètre n'est pas optimisée car le planificateur ne peut avoir connaissance au préalable des partitions sélectionnées par la valeur du paramètre à l'exécution. Pour la même raison, il faut éviter les fonctions « stable »s comme `CURRENT_DATE` ;

- les contraintes de partitionnement doivent rester simples. Dans le cas contraire, le planificateur peut rencontrer des difficultés à déterminer les partitions qu'il n'est pas nécessaire de parcourir. Des conditions simples d'égalité pour le partitionnement de liste ou des tests d'échelle simples lors de partitionnement d'échelle sont recommandées, comme cela est illustré dans les exemples précédents. Une bonne règle consiste à s'assurer que les comparaisons entre colonnes de partitionnement et constantes utilisées par les contraintes de partitionnement se fassent uniquement à l'aide d'opérateurs utilisables par les index B-tree.
- toutes les contraintes de toutes les partitions de la table maître sont examinées lors de l'exclusion de contraintes. De ce fait, un grand nombre de partitions augmente considérablement le temps de planification de la requête. Un partitionnement qui utilise ces techniques fonctionne assez bien jusqu'environ une centaine de partitions ; il est impensable de vouloir atteindre des milliers de partitions.

5.10. Autres objets de la base de données

Les tables sont les objets centraux dans une structure de base de données relationnelles, car ce sont elles qui stockent les données. Mais ce ne sont pas les seuls objets qui existent dans une base de données. De nombreux autres types d'objets peuvent être créés afin de rendre l'utilisation et la gestion des données plus efficace ou pratique. Ils ne sont pas abordés dans ce chapitre mais une liste en est dressée à titre d'information.

- Vues
- Fonctions et opérateurs
- Types de données et domaines
- Déclencheurs et règles de réécriture

Des informations détaillées sur ces sujets apparaissent dans la Partie V, « Programmation serveur ».

5.11. Gestion des dépendances

Lorsque des structures de base complexes sont créées qui impliquent beaucoup de tables avec des contraintes de clés étrangères, des vues, des déclencheurs, des fonctions, etc., un réseau de dépendances entre les objets est implicitement créé. Par exemple, une table avec une contrainte de clé étrangère dépend de la table à laquelle elle fait référence.

Pour garantir l'intégrité de la structure entière de la base, PostgreSQL™ s'assure qu'un objet dont d'autres objets dépendent ne peut pas être supprimé. Ainsi, toute tentative de suppression de la table des produits utilisée dans la Section 5.3.5, « Clés étrangères », sachant que la table des commandes en dépend, lève un message d'erreur comme celui-ci :

```
DROP TABLE produits;

NOTICE:  constraint commandes_no_produit_fkey on table commandes depends on table
produits
ERROR:  cannot drop table produits because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

ou en français :

```
DROP TABLE produits;

NOTICE:  la contrainte commandes_no_produit_fkey sur la table commandes dépend
de la table produits
ERREUR:  la table produits ne peut pas être supprimée, car d'autre objets en
dépendent
HINT:  Utiliser DROP ... CASCADE pour supprimer également les objets
dépendants.
```

Le message d'erreur contient un indice utile : pour ne pas avoir à supprimer individuellement chaque objet dépendant, on peut lancer

```
DROP TABLE produits CASCADE;
```

et tous les objets dépendants sont ainsi effacés. Dans ce cas, la table des commandes n'est pas supprimée, mais seulement la contrainte de clé étrangère. (Pour vérifier ce que fait **DROP ... CASCADE**, on peut lancer **DROP** sans `CASCADE` et lire les messages `NOTICE`.)

Toutes les commandes de suppression dans PostgreSQL™ supportent l'utilisation de `CASCADE`. La nature des dépendances est évidemment fonction de la nature des objets. On peut aussi écrire `RESTRICT` au lieu de `CASCADE` pour obtenir le comportement

par défaut, à savoir interdire les suppressions d'objets dont dépendent d'autres objets.

Note

D'après le standard SQL, il est nécessaire d'indiquer `RESTRICT` ou `CASCADE`. Aucun système de base de donnée ne force cette règle, en réalité, mais le choix du comportement par défaut, `RESTRICT` ou `CASCADE`, varie suivant le système.

Note

Les dépendances de contraintes de clés étrangères et de colonnes *serial* des versions de PostgreSQL™ antérieures à 7.3 *ne* sont *pas* maintenues ou créées pendant le processus de mise à jour. Tout autre type de dépendance est proprement créé pendant une mise à jour à partir d'une base de données antérieure à la 7.3.

Chapitre 6. Manipulation de données

Ce chapitre est toujours assez incomplet.

Le chapitre précédent présente la création des tables et des autres structures de stockage des données. Il est temps de remplir ces tables avec des données. Le présent chapitre couvre l'insertion, la mise à jour et la suppression des données des tables. Après cela, le chapitre présente l'élimination des données perdues.

6.1. Insérer des données

Quand une table est créée, elle ne contient aucune donnée. La première chose à faire, c'est d'y insérer des données. Sans quoi la base de données n'est pas d'une grande utilité. Les données sont conceptuellement insérées ligne par ligne. Il est évidemment possible d'insérer plus d'une ligne, mais il n'est pas possible d'entrer moins d'une ligne. Même lorsque seules les valeurs d'une partie des colonnes sont connues, une ligne complète doit être créée.

Pour créer une nouvelle ligne, la commande INSERT(7) est utilisée. La commande a besoin du nom de la table et des valeurs de colonnes.

Soit la table des produits du Chapitre 5, Définition des données :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric
);
```

Une commande d'insertion d'une ligne peut être :

```
INSERT INTO produits VALUES (1, 'Fromage', 9.99);
```

Les données sont listées dans l'ordre des colonnes de la table, séparées par des virgules. Souvent, les données sont des libellés (constantes) mais les expressions scalaires sont aussi acceptées.

La syntaxe précédente oblige à connaître l'ordre des colonnes. Pour éviter cela, les colonnes peuvent être explicitement listées. Les deux commandes suivantes ont, ainsi, le même effet que la précédente :

```
INSERT INTO produits (no_produit, nom, prix) VALUES (1, 'Fromage', 9.99);
INSERT INTO produits (nom, prix, no_produit) VALUES ('Fromage', 9.99, 1);
```

Beaucoup d'utilisateurs recommandent de toujours lister les noms de colonnes.

Si les valeurs de certaines colonnes ne sont pas connues, elles peuvent être omises. Dans ce cas, elles sont remplies avec leur valeur par défaut. Par exemple :

```
INSERT INTO produits (no_produit, nom) VALUES (1, 'Fromage');
INSERT INTO produits VALUES (1, 'Fromage');
```

La seconde instruction est une extension PostgreSQL™. Elle remplit les colonnes de gauche à droite avec toutes les valeurs données, et les autres prennent leur valeur par défaut.

Il est possible, pour plus de clarté, d'appeler explicitement les valeurs par défaut pour des colonnes particulières ou pour la ligne complète.

```
INSERT INTO produits (no_produit, nom, prix) VALUES (1, 'Fromage', DEFAULT);
INSERT INTO produits DEFAULT VALUES;
```

Plusieurs lignes peuvent être insérées en une seule commande :

```
INSERT INTO produits (no_produit, nom, prix) VALUES
    (1, 'Fromage', 9.99),
    (2, 'Pain', 1.99),
    (3, 'Lait', 2.99);
```

Astuce

Lors de l'insertion d'une grande quantité de données en même temps, il est préférable d'utiliser la commande COPY(7). Elle n'est pas aussi flexible que la commande INSERT(7) mais elle est plus efficace. Se référer à Section 14.4, « Remplir une base de données » pour plus d'informations sur l'amélioration des performances lors de

gros chargements de données.

6.2. Actualiser les données

La modification de données présentes en base est appelée mise à jour ou actualisation (*update* en anglais). Il est possible de mettre à jour une ligne spécifique, toutes les lignes ou un sous-ensemble de lignes de la table. Chaque colonne peut être actualisée séparément ; les autres colonnes ne sont alors pas modifiées.

Pour mettre à jour les lignes existantes, utilisez la commande UPDATE(7). Trois informations sont nécessaires :

1. le nom de la table et de la colonne à mettre à jour ;
2. la nouvelle valeur de la colonne ;
3. les lignes à mettre à jour.

Comme cela a été vu dans le Chapitre 5, Définition des données, le SQL ne donne pas, par défaut, d'identifiant unique pour les lignes. Il n'est, de ce fait, pas toujours possible d'indiquer directement la ligne à mettre à jour. On précise plutôt les conditions qu'une ligne doit remplir pour être mise à jour. Si la table possède une clé primaire (qu'elle soit déclarée ou non), une ligne unique peut être choisie en précisant une condition sur la clé primaire. Les outils graphiques d'accès aux bases de données utilisent ce principe pour permettre les modifications de lignes individuelles.

La commande suivante, par exemple, modifie tous les produits dont le prix est 5 en le passant à 10.

```
UPDATE produits SET prix = 10 WHERE prix = 5;
```

Cela peut mettre à jour zéro, une, ou plusieurs lignes. L'exécution d'une commande `UPDATE` qui ne met à jour aucune ligne ne représente pas une erreur.

Dans le détail de la commande, on trouve tout d'abord, le mot clé `UPDATE` suivi du nom de la table. Le nom de la table peut toujours être préfixé par un nom de schéma dans le cas contraire elle est recherchée dans le chemin. On trouve ensuite le mot clé `SET` suivi du nom de la colonne, un signe égal et la nouvelle valeur de la colonne, qui peut être une constante ou une expression scalaire.

Par exemple, pour augmenter de 10% le prix de tous les produits, on peut exécuter :

```
UPDATE produits SET prix = prix * 1.10;
```

L'expression donnant la nouvelle valeur peut faire référence aux valeurs courantes de la ligne.

Il n'a pas été indiqué ici de clause `WHERE`. Si elle est omise, toutes les lignes de la table sont modifiées. Si elle est présente, seules les lignes qui remplissent la condition `WHERE` sont mises à jour. Le signe égal dans la clause `SET` réalise une affectation, alors que celui de la clause `WHERE` permet une comparaison. Pour autant, cela ne crée pas d'ambiguïté. La condition `WHERE` n'est pas nécessairement un test d'égalité de nombreux autres opérateurs existent (voir le Chapitre 9, Fonctions et opérateurs). Mais le résultat de l'expression est booléen.

Il est possible d'actualiser plusieurs colonnes en une seule commande **UPDATE** par l'indication de plusieurs colonnes dans la clause `SET`.

Par exemple :

```
UPDATE ma_table SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Supprimer des données

Les parties précédentes présentent l'ajout et la modification de données. Il reste à voir leur suppression quand elles ne sont plus nécessaires. Comme pour l'insertion, la suppression ne peut se faire que par ligne entière. Le SQL ne propose pas de moyen d'accéder à une ligne particulière. C'est pourquoi la suppression de lignes se fait en indiquant les conditions à remplir par les lignes à supprimer. S'il y a une clé primaire dans la table, alors il est possible d'indiquer précisément la ligne à supprimer. Mais on peut aussi supprimer un groupe de lignes qui remplissent une condition, ou même toutes les lignes d'une table en une fois.

Pour supprimer des lignes, on utilise la commande DELETE(7) ; la syntaxe est très similaire à la commande **UPDATE**.

Par exemple, pour supprimer toutes les lignes de la table produits qui ont un prix de 10, on exécute :

```
DELETE FROM produits WHERE prix = 10;
```

En indiquant simplement :

```
DELETE FROM produits;
```

on supprime toutes les lignes de la table. Attention aux mauvaises manipulations !

Chapitre 7. Requêtes

Les précédents chapitres ont expliqué comme créer des tables, comment les remplir avec des données et comment manipuler ces données. Maintenant, nous discutons enfin de la façon de récupérer ces données depuis la base de données.

7.1. Aperçu

Le processus et la commande de récupération des données sont appelés une *requête*. En SQL, la commande SELECT(7) est utilisée pour spécifier des requêtes. La syntaxe générale de la commande **SELECT** est

```
[WITH with_requêtes] SELECT liste_select FROM expression_table [specification_tri]
```

Les sections suivantes décrivent le détail de la liste de sélection, l'expression des tables et la spécification du tri. Les requêtes `WITH` sont traitées en dernier car il s'agit d'une fonctionnalité avancée.

Un type de requête simple est de la forme :

```
SELECT * FROM table1;
```

En supposant qu'il existe une table appelée `table1`, cette commande récupérera toutes les lignes et toutes les colonnes de `table1`. La méthode de récupération dépend de l'application cliente. Par exemple, le programme psql affichera une table, façon art ASCII, alors que les bibliothèques du client offriront des fonctions d'extraction de valeurs individuelles à partir du résultat de la requête. `*` comme liste de sélection signifie que toutes les colonnes de l'expression de table seront récupérées. Une liste de sélection peut aussi être un sous-ensemble des colonnes disponibles ou effectuer un calcul en utilisant les colonnes. Par exemple, si `table1` dispose des colonnes nommées `a`, `b` et `c` (et peut-être d'autres), vous pouvez lancer la requête suivante :

```
SELECT a, b + c FROM table1;
```

(en supposant que `b` et `c` soient de type numérique). Voir la Section 7.3, « Listes de sélection » pour plus de détails.

`FROM table1` est un type très simple d'expression de tables : il lit une seule table. En général, les expressions de tables sont des constructions complexes de tables de base, de jointures et de sous-requêtes. Mais vous pouvez aussi entièrement omettre l'expression de table et utiliser la commande **SELECT** comme une calculatrice :

```
SELECT 3 * 4;
```

Ceci est plus utile si les expressions de la liste de sélection renvoient des résultats variants. Par exemple, vous pouvez appeler une fonction de cette façon :

```
SELECT random();
```

7.2. Expressions de table

Une *expression de table* calcule une table. L'expression de table contient une clause `FROM` qui peut être suivie des clauses `WHERE`, `GROUP BY` et `HAVING`. Les expressions triviales de table font simplement référence à une table sur le disque, une table de base, mais des expressions plus complexes peuvent être utilisées pour modifier ou combiner des tables de base de différentes façons.

Les clauses optionnelles `WHERE`, `GROUP BY` et `HAVING` dans l'expression de table spécifient un tube de transformations successives réalisées sur la table dérivée de la clause `FROM`. Toutes ces transformations produisent une table virtuelle fournissant les lignes à passer à la liste de sélection qui choisira les lignes à afficher de la requête.

7.2.1. Clause `FROM`

La la section intitulée « Clause `FROM` » dérive une table à partir d'une ou plusieurs tables données dans une liste de référence dont les tables sont séparées par des virgules.

```
FROM reference_table [, reference_table [, ...]]
```

Une référence de table pourrait être un nom de table (avec en option le nom du schéma) ou une table dérivée comme une sous-requête, une table jointe ou une combinaison complexe de celles-ci. Si plus d'une référence de tables est listée dans la clause `FROM`, elle sont jointes pour former une table virtuelle intermédiaire qui pourrait être le sujet des transformations des clauses `WHERE`, `GROUP BY` et `HAVING`, et est finalement le résultat des expressions de table.

Lorsqu'une référence de table nomme une table qui est la table parent d'une table suivant la hiérarchie de l'héritage, la référence de table produit les lignes non seulement de la table mais aussi des descendants de cette table sauf si le mot clé `ONLY` précède le nom de la table. Néanmoins, la référence produit seulement les colonnes qui apparaissent dans la table nommée... toute colonne

ajoutée dans une sous-table est ignorée.

Au lieu d'écrire `ONLY` avant le nom de la table, vous pouvez écrire `*` après le nom de la table pour indiquer spécifiquement que les tables filles sont incluses. Écrire `*` n'est pas nécessaire car il s'agit du comportement par défaut (sauf si vous avez choisi de modifier la configuration de sql_inheritance). Néanmoins, écrire `*` peut être utile pour indiquer fortement que les tables filles seront parcourues.

7.2.1.1. Tables jointes

Une table jointe est une table dérivée de deux autres tables (réelles ou dérivées) suivant les règles du type de jointure particulier. Les jointures internes (inner), externes (outer) et croisées (cross) sont disponibles.

Types de jointures

Jointure croisée (cross join)

```
T1 CROSS JOIN T2
```

Pour chaque combinaison possible de lignes provenant de *T1* et *T2* (c'est-à-dire un produit cartésien), la table jointe contiendra une ligne disposant de toutes les colonnes de *T1* suivies par toutes les colonnes de *T2*. Si les tables ont respectivement N et M lignes, la table jointe en aura N * M.

`FROM T1 CROSS JOIN T2` est équivalent à `FROM T1, T2`. C'est aussi équivalent à `FROM T1 INNER JOIN T2 ON TRUE` (voir ci-dessous).

Jointures qualifiées (qualified joins)

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON expression_booleenne
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( liste des colonnes
jointes )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Les mots `INNER` et `OUTER` sont optionnels dans toutes les formes. `INNER` est la valeur par défaut ; `LEFT`, `RIGHT` et `FULL` impliquent une jointure externe.

La *condition de la jointure* est spécifiée dans la clause `ON` ou `USING`, ou implicitement par le mot `NATURAL`. La condition de jointure détermine les lignes des deux tables source considérées comme « correspondante », comme l'explique le paragraphe ci-dessous.

La clause `ON` est le type le plus général de condition de jointure : il prend une expression booléenne du même genre que celle utilisée dans une clause `WHERE`. Une paires de lignes de *T1* et *T2* correspondent si l'expression `ON` est évaluée à vraie (true) pour ces deux lignes.

`USING` est la notation raccourcie : elle prend une liste de noms de colonnes, séparés par des virgules, que les tables jointes ont en commun, et forme une condition de jointure spécifiant l'égalité de chacune de ces paires de colonnes. De plus, la sortie de `JOIN USING` a une colonne pour chaque paires égales des colonnes en entrée, suivies par toutes les autres colonnes de chaque table. Du coup, `USING (a, b, c)` est équivalent à `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)` avec l'exception que si `ON` est utilisé, il y aura deux colonnes `a`, `b`, puis `c` dans le résultat, alors qu'avec `USING`, il n'y en aurait eu qu'une de chaque (et elles apparaîtront en premier si **SELECT *** est utilisé).

Enfin, `NATURAL` est un format raccourci de `USING` : il forme une liste `USING` consistant de tous les noms de colonnes apparaissant à la fois dans les deux tables en entrée. Comme avec `USING`, ces colonnes apparaissent seulement une fois dans la table de sortie.

Les types possibles de jointures qualifiées sont :

`INNER JOIN`
Pour chaque ligne R1 de T1, la table jointe a une ligne pour chaque ligne de T2 satisfaisant la condition de jointure avec R1.

`LEFT OUTER JOIN`

Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T1 qui ne satisfait pas la condition de jointure avec les lignes de T2, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T2. Du coup, la table jointe a toujours au moins une ligne pour chaque ligne de T1 quelque soient les conditions.

`RIGHT OUTER JOIN`

Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T2 qui ne satisfait pas la condition de jointure avec les lignes de T1, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T1. C'est l'inverse d'une jointure gauche : la table résultante aura toujours une ligne pour chaque ligne de T2 quelque soient les conditions.

`FULL OUTER JOIN`
Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T1 qui ne satisfait pas la condition de jointure avec les lignes de T2, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T2. De plus, pour chaque ligne de T2 qui ne satisfait pas la condition de jointure avec les lignes de T1, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T1.

Les jointures de tous les types peuvent être chaînées ensemble ou imbriquées : soit les deux soit une des deux, parmi *T1* et *T2*, peuvent être des tables. Les parenthèses peuvent être utilisées autour des clauses `JOIN` pour contrôler l'ordre de jointure. En l'absence de parenthèses, les clauses `JOIN` sont imbriquées de gauche à droite.

Pour rassembler tout ceci, supposons que nous avons une table `t1` :

```
 no | nom
----+------
  1 | a
  2 | b
  3 | c
```

et une table `t2` :

```
 no | valeur
----+-------
  1 | xxx
  3 | yyy
  5 | zzz
```

nous obtenons les résultats suivants pour les différentes jointures :

```
=> SELECT * FROM t1 CROSS JOIN t2;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  1 | a   |  3 | yyy
  1 | a   |  5 | zzz
  2 | b   |  1 | xxx
  2 | b   |  3 | yyy
  2 | b   |  5 | zzz
  3 | c   |  1 | xxx
  3 | c   |  3 | yyy
  3 | c   |  5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  3 | c   |  3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (no);
 no | nom | valeur
----+-----+-------
  1 | a   | xxx
  3 | c   | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 no | nom | valeur
----+-----+-------
  1 | a   | xxx
  3 | c   | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  2 | b   |    |
  3 | c   |  3 | yyy
(3 rows)
```

```
=> SELECT * FROM t1 LEFT JOIN t2 USING (no);
 no | nom | valeur
----+-----+-------
  1 | a   | xxx
  2 | b   |
  3 | c   | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  3 | c   |  3 | yyy
    |     |  5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  2 | b   |    |
  3 | c   |  3 | yyy
    |     |  5 | zzz
(4 rows)
```

La condition de jointure spécifiée avec `ON` peut aussi contenir des conditions sans relation directe avec la jointure. Ceci est utile pour quelques requêtes mais son utilisation doit avoir été réfléchie. Par exemple :

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.no = t2.no AND t2.valeur = 'xxx';
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  2 | b   |    |
  3 | c   |    |
(3 rows)
```

Notez que placer la restriction dans la clause `WHERE` donne un résultat différent :

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

Ceci est dû au fait qu'une restriction placée dans la clause `ON` est traitée *avant* la jointure alors qu'une restriction placée dans la clause `WHERE` est traitée *après* la jointure.

7.2.1.2. Alias de table et de colonne

Un nom temporaire peut être donné aux tables et aux références de tables complexe, qui sera ensuite utilisé pour référencer la table dérivée dans la suite de la requête. Cela s'appelle un *alias de table*.

Pour créer un alias de table, écrivez

```
FROM reference_table AS alias
```

ou

```
FROM reference_table alias
```

Le mot clé `AS` n'est pas obligatoire. *`alias`* peut être tout identifiant.

Une application typique des alias de table est l'affectation d'identifieurs courts pour les noms de tables longs, ce qui permet de garder des clauses de jointures lisibles. Par exemple :

```
SELECT * FROM nom_de_table_tres_tres_long s
  JOIN un_autre_nom_tres_long a ON s.id = a.no;
```

L'alias devient le nouveau nom de la table en ce qui concerne la requête en cours -- il n'est pas autorisé de faire référence à la table par son nom original où que ce soit dans la requête. Du coup, ceci n'est pas valide :

```
SELECT * FROM mon_table AS m WHERE mon_table.a > 5;    -- mauvais
```

Les alias de table sont disponibles principalement pour aider à l'écriture de requête mais ils deviennent nécessaires pour joindre une table avec elle-même, par exemple :

```
SELECT * FROM personnes AS mere JOIN personnes AS enfant ON mere.id = enfant.mere_id;
```

De plus, un alias est requis si la référence de la table est une sous-requête (voir la Section 7.2.1.3, « Sous-requêtes »).

Les parenthèses sont utilisées pour résoudre les ambiguïtés. Dans l'exemple suivant, la première instruction affecte l'alias `b` à la deuxième instance de `ma_table` mais la deuxième instruction affecte l'alias au résultat de la jonction :

```
SELECT * FROM ma_table AS a CROSS JOIN ma_table AS b ...
SELECT * FROM (ma_table AS a CROSS JOIN ma_table) AS b ...
```

Une autre forme d'alias de tables donne des noms temporaires aux colonnes de la table ainsi qu'à la table :

```
FROM reference_table [AS] alias ( colonne1 [, colonne2 [, ...]] )
```

Si le nombre d'alias de colonnes spécifié est plus petit que le nombre de colonnes dont dispose la table réelle, les colonnes suivantes ne sont pas renommées. Cette syntaxe est particulièrement utile dans le cas de jointure avec la même table ou dans le cas de sous-requêtes.

Quand un alias est appliqué à la sortie d'une clause `JOIN`, l'alias cache le nom original référencé à l'intérieur du `JOIN`. Par exemple :

```
SELECT a.* FROM ma_table AS a JOIN ta_table AS b ON ...
```

est du SQL valide mais :

```
SELECT a.* FROM (ma_table AS a JOIN ta_table AS b ON ...) AS c
```

n'est pas valide l'alias de table `a` n'est pas visible en dehors de l'alias `c`.

7.2.1.3. Sous-requêtes

Une sous-requête spécifiant une table dérivée doit être enfermée dans des parenthèses et *doit* se voir affecté un alias de table (voir la Section 7.2.1.2, « Alias de table et de colonne »). Par exemple :

```
FROM (SELECT * FROM table1) AS nom_alias
```

Cet exemple est équivalent à `FROM table1 AS nom_alias`. Des cas plus intéressants, qui ne peuvent pas être réduit à une jointure pleine, surviennent quand la sous-requête implique un groupement ou un agrégat.

Uns sous-requête peut aussi être une liste **VALUES** :

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS noms(prenom, nom)
```

De nouveau, un alias de table est requis. Affecter des noms d'alias aux colonnes de la liste **VALUES** est en option mais c'est une bonne pratique. Pour plus d'informations, voir Section 7.7, « Listes `VALUES` ».

7.2.1.4. Fonctions de table

Les fonctions de table sont des fonctions produisant un ensemble de lignes composées de types de données de base (types scalaires) ou de types de données composites (lignes de table). Elles sont utilisées comme une table, une vue ou une sous-requête de la clause `FROM` d'une requête. Les colonnes renvoyées par les fonctions de table peuvent être incluses dans une clause `SELECT`, `JOIN` ou `WHERE` de la même manière qu'une colonne de table, vue ou sous-requête.

Si une fonction de table renvoie un type de données de base, la nom de la colonne de résultat correspond à celui de la fonction. Si la fonction renvoie un type composite, les colonnes résultantes ont le même nom que les attributs individuels du type.

Une fonction de table peut avoir un alias dans la clause `FROM` mais elle peut être laissée sans alias. Si une fonction est utilisée dans la clause `FROM` sans alias, le nom de la fonction est utilisé comme nom de table résultante.

Quelques exemples :

```
CREATE TABLE truc (trucid int, trucsousid int, trucnom text);

CREATE FUNCTION recuptruc(int) RETURNS SETOF foo AS $$
    SELECT * FROM truc WHERE trucid = $1;
```

```
$$ LANGUAGE SQL;

SELECT * FROM recuptruc(1) AS t1;

SELECT * FROM truc
    WHERE trucsousid IN (
                         SELECT trucsousid
                         FROM recuptruc(truc.trucid) z
                         WHERE z.trucid = truc.trucid);

CREATE VIEW vue_recuptruc AS SELECT * FROM recuptruc(1);
SELECT * FROM vue_recuptruc;
```

Dans certains cas, il est utile de définir des fonctions de table pouvant renvoyer des ensembles de colonnes différentes suivant la façon dont elles sont appelées. Pour supporter ceci, la fonction de table est déclarée comme renvoyant le pseudotype record. Quand une telle fonction est utilisée dans une requête, la structure de ligne attendue doit être spécifiée dans la requête elle-même, de façon à ce que le système sache comment analyser et planifier la requête. Considérez cet exemple :

```
SELECT *
    FROM dblink('dbname=mabd', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname nom, prosrc text)
    WHERE proname LIKE 'bytea%';
```

La fonction `dblink` exécute une requête distante (voir `contrib/dblink`). Elle déclare renvoyer le type record car elle pourrait être utilisée pour tout type de requête. L'ensemble de colonnes réelles doit être spécifié dans la requête appelante de façon à ce que l'analyseur sache, par exemple, comment étendre `*`.

7.2.2. Clause `WHERE`

La syntaxe de la la section intitulée « Clause `WHERE` » est

```
WHERE condition_recherche
```

où *`condition_recherche`* est toute expression de valeur (voir la Section 4.2, « Expressions de valeurs ») renvoyant une valeur de type boolean.

Après le traitement de la clause `FROM`, chaque ligne de la table virtuelle dérivée est vérifiée avec la condition de recherche. Si le résultat de la vérification est positif (true), la ligne est conservée dans la table de sortie, sinon (c'est-à-dire si le résultat est faux ou nul), la ligne est abandonnée. La condition de recherche référence typiquement au moins une colonne de la table générée dans la clause `FROM` ; ceci n'est pas requis mais, dans le cas contraire, la clause `WHERE` n'aurait aucune utilité.

Note

La condition de jointure d'une jointure interne peut être écrite soit dans la clause `WHERE` soit dans la clause `JOIN`. Par exemple, ces expressions de tables sont équivalentes :

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

et :

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

ou même peut-être :

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Laquelle vous utilisez est plutôt une affaire de style. La syntaxe `JOIN` dans la clause `FROM` n'est probablement pas aussi portable vers les autres systèmes de gestion de bases de données SQL, même si cela fait partie du standard SQL. Pour les jointures externes, il n'y a pas d'autres choix : elles doivent être faites dans la clause `FROM`. La clause `ON` ou `USING` d'une jointure externe n'est *pas* équivalente à une condition `WHERE` parce qu'elle détermine l'ajout de lignes (pour les lignes qui ne correspondent pas en entrée) ainsi que pour la suppression de lignes dans le résultat final.

Voici quelques exemples de clauses `WHERE` :

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` est la table dérivée dans la clause `FROM`. Les lignes qui ne correspondent pas à la condition de recherche de la clause `WHERE` sont éliminées de la table `fdt`. Notez l'utilisation de sous-requêtes scalaires en tant qu'expressions de valeurs. Comme n'importe quelle autre requête, les sous-requêtes peuvent employer des expressions de tables complexes. Notez aussi comment `fdt` est référencée dans les sous-requêtes. Qualifier `c1` comme `fdt.c1` est seulement nécessaire si `c1` est aussi le nom d'une colonne dans la table d'entrée dérivée de la sous-requête. Mais qualifier le nom de colonne ajoute à la clarté même lorsque cela n'est pas nécessaire. Cet exemple montre comment le nom de colonne d'une requête externe est étendue dans les requêtes internes.

7.2.3. Clauses `GROUP BY` et `HAVING`

Après avoir passé le filtre `WHERE`, la table d'entrée dérivée peut être sujette à un regroupement en utilisant la clause `GROUP BY` et à une élimination de groupe de lignes avec la clause `HAVING`.

```
SELECT liste_selection
    FROM ...
    [WHERE ...]
    GROUP BY reference_colonne_regroupement[,reference_colonne_regroupement]...
```

La la section intitulée « Clause `GROUP BY` » est utilisée pour regrouper les lignes d'une table qui ont les mêmes valeurs dans toutes les colonnes précisées. L'ordre dans lequel ces colonnes sont indiquées importe peu. L'effet est de combiner chaque ensemble de lignes partageant des valeurs communes en un seul groupe de ligne représentant toutes les lignes du groupe. Ceci est fait pour éliminer les redondances dans la sortie et/ou pour calculer les agrégats s'appliquant à ces groupes. Par exemple :

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Dans la seconde requête, nous n'aurions pas pu écrire `SELECT * FROM test1 GROUP BY x` parce qu'il n'existe pas une seule valeur pour la colonne `y` pouvant être associé avec chaque autre groupe. Les colonnes de regroupement peuvent être référencées dans la liste de sélection car elles ont une valeur constante unique par groupe.

En général, si une table est groupée, les colonnes qui ne sont pas listées dans le `GROUP BY` ne peuvent pas être référencées sauf dans les expressions d'agrégats. Voici un exemple d'expressions d'agrégat :

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Ici, `sum` est la fonction d'agrégat qui calcule une seule valeur pour le groupe entier. La Section 9.18, « Fonctions d'agrégat » propose plus d'informations sur les fonctions d'agrégats disponibles.

Astuce

Le regroupement sans expressions d'agrégats calcule effectivement l'ensemble les valeurs distinctes d'une colonne. Ceci peut aussi se faire en utilisant la clause `DISTINCT` (voir la Section 7.3.3, « `DISTINCT` »).

Voici un autre exemple : il calcule les ventes totales pour chaque produit (plutôt que le total des ventes sur tous les produits) :

```
SELECT produit_id, p.nom, (sum(v.unite) * p.prix) AS ventes
    FROM produits p LEFT JOIN ventes v USING (produit_id)
    GROUP BY produit_id, p.nom, p.prix;
```

Dans cet exemple, les colonnes `produit_id`, `p.nom` et `p.prix` doivent être dans la clause `GROUP BY` car elles sont référencées dans la liste de sélection de la requête (suivant la façon dont est conçue la table produits, le nom et le prix pourraient être totalement dépendants de l'ID du produit, donc des regroupements supplémentaires pourraient théoriquement être inutiles mais ceci n'est pas encore implémenté). La colonne `s.unite` n'a pas besoin d'être dans la liste `GROUP BY` car elle est seulement utilisée dans l'expression de l'agrégat (`sum(...)`) représentant les ventes d'un produit. Pour chaque produit, la requête renvoie une ligne de résumé sur les ventes de ce produit.

En SQL strict, `GROUP BY` peut seulement grouper les colonnes de la table source mais PostgreSQL™ étend ceci en autorisant `GROUP BY` à grouper aussi les colonnes de la liste de sélection. Grouper par expressions de valeurs au lieu de simples noms de colonnes est aussi permis.

Si une table a été groupée en utilisant la clause `GROUP BY` mais que seuls certains groupes sont intéressants, la clause `HAVING` peut être utilisée, comme une clause `WHERE`, pour éliminer les groupes du résultat. Voici la syntaxe :

```
SELECT liste_selection FROM ... [WHERE ...] GROUP BY ... HAVING expression_booléenne
```

Les expressions de la clause `HAVING` peuvent référer à la fois aux expressions groupées et aux expressions non groupées (ce qui impliquent nécessairement une fonction d'agrégat).

Exemple :

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

De nouveau, un exemple plus réaliste :

```
SELECT produit_id, p.nom, (sum(v.unite) * (p.prix - p.cout)) AS profit
    FROM produits p LEFT JOIN ventes v USING (produit_id)
    WHERE v.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY produit_id, p.nom, p.prix, p.cout
    HAVING sum(p.prix * s.unite) > 5000;
```

Dans l'exemple ci-dessus, la clause `WHERE` sélectionne les lignes par une colonne qui n'est pas groupée (l'expression est vraie seulement pour les ventes des quatre dernières semaines) alors que la clause `HAVING` restreint la sortie aux groupes dont le total des ventes dépasse 5000. Notez que les expressions d'agrégats n'ont pas besoin d'être identiques dans toutes les parties d'une requête.

Si une requête contient des appels à des fonctions d'aggrégat, mais pas de clause `GROUP BY`, le regroupement a toujours lieu : le résultat est une seule ligne de regroupement (ou peut-être pas de ligne du tout si la ligne unique est ensuite éliminée par la clause `HAVING`). Ceci est vrai aussi si elle comporte une clause `HAVING`, même sans fonction d'aggrégat ou `GROUP BY`.

7.2.4. Traitement de fonctions Window

Si la requête contient une des fonctions Window (voir Section 3.5, « Fonctions de fenêtrage », Section 9.19, « Fonctions Window » et Section 4.2.8, « Appels de fonction de fenêtrage »), ces fonctions sont évaluées après que soient effectués les regroupements, les aggrégations, les filtrages par `HAVING`. C'est-à-dire que si la requête comporte des aggrégat, `GROUP BY` ou `HAVING`, alors les enregistrements vus par les fonctions window sont les lignes regroupées à la place des enregistrements originaux provenant de `FROM`/`WHERE`.

Quand des fonctions Window multiples sont utilisées, toutes les fonctions Window ayant des clauses `PARTITION BY` et `ORDER BY` syntaxiquement équivalentes seront à coup sûr évaluées en une seule passe sur les données. Par conséquent, elles verront le même ordre de tri, même si `ORDER BY` ne détermine pas de façon unique un tri. Toutefois, aucune garantie n'est faite à propos de l'évaluation de fonctions ayant des spécifications de `PARTITION BY` ou `ORDER BY` différentes. (Dans ces cas, une étape de tri

est généralement nécessaire entre les passes d'évaluations de fonctions Window, et le tri ne garantit pas la préservation de l'ordre des enregistrements que son `ORDER BY` estime comme identiques.)

À l'heure actuelle, les fonctions window nécessitent toujours des données pré-triées, ce qui fait que la sortie de la requête sera triée suivant l'une ou l'autre des clauses `PARTITION BY`/`ORDER BY` des fonctions Window. Il n'est toutefois pas recommandé de s'en servir. Utilisez une clause `ORDER BY` au plus haut niveau de la requête si vous voulez être sûr que vos résultats soient triés d'une certaine façon.

7.3. Listes de sélection

Comme montré dans la section précédente, l'expression de table pour la commande **SELECT** construit une table virtuelle intermédiaire en combinant les tables, vues, en éliminant les lignes, en groupant, etc. Cette table est finalement passée à la réalisation de la *liste de sélection*. Cette liste détermine les *colonnes* de la table intermédiaire à afficher.

7.3.1. Éléments de la liste de sélection

La forme la plus simple de liste de sélection est `*`. C'est un raccourci pour indiquer toutes les colonnes que l'expression de table produit. Sinon, une liste de sélection est une liste d'expressions de valeurs séparées par des virgules (comme défini dans la Section 4.2, « Expressions de valeurs »). Par exemple, cela pourrait être une liste des noms de colonnes :

```
SELECT a, b, c FROM ...
```

Les noms de colonnes `a`, `b` et `c` sont soit les noms actuels des colonnes des tables référencées dans la clause `FROM` soit les alias qui leur ont été donnés (voir l'explication dans Section 7.2.1.2, « Alias de table et de colonne »). L'espace de nom disponible dans la liste de sélection est le même que dans la clause `WHERE` sauf si le regroupement est utilisé, auquel cas c'est le même que dans la clause `HAVING`.

Si plus d'une table a une colonne du même nom, le nom de la table doit aussi être donné comme dans :

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

En travaillant avec plusieurs tables, il est aussi utile de demander toutes les colonnes d'une table particulière :

```
SELECT tbl1.*, tbl2.a FROM ...
```

(voir aussi la Section 7.2.2, « Clause `WHERE` »)

Si une expression de valeur arbitraire est utilisée dans la liste de sélection, il ajoute conceptuellement une nouvelle colonne virtuelle dans la table renvoyée. L'expression de valeur est évaluée une fois pour chaque ligne avec une substitution des valeurs de lignes avec les références de colonnes. Mais les expressions de la liste de sélection n'ont pas à référencer les colonnes dans l'expression de la table de la clause `FROM` ; elles pourrait être des expressions arithmétiques constantes, par exemple.

7.3.2. Labels de colonnes

Les entrées de la liste de sélection peuvent se voir affecter des noms pour la suite de l'exécution, peut-être pour référence dans une clause `ORDER BY` ou pour affichage par l'application cliente. Par exemple :

```
SELECT a AS valeur, b + c AS sum FROM ...
```

Si aucun nom de colonne en sortie n'est spécifié en utilisant `AS`, le système affecte un nom de colonne par défaut. Pour les références de colonne simple, c'est le nom de la colonne référencée. Pour les appels de fonction, il s'agit du nom de la fonction. Pour les expressions complexes, le système générera un nom générique.

Le mot clé `AS` est optionnel, mais seulement si le nouveau nom de colonne ne correspond à aucun des mots clés PostgreSQL™ (voir Annexe C, Mots-clé SQL). Pour éviter une correspondance accidentelle à un mot clé, vous pouvez mettre le nom de colonne entre guillemets. Par exemple, `VALUE` est un mot clé, ce qui fait que ceci ne fonctionne pas :

```
SELECT a valeur, b + c AS somme FROM ...
```

mais ceci fonctionne :

```
SELECT a "valeur", b + c AS somme FROM ...
```

Pour vous protéger de possibles ajouts futurs de mots clés, il est recommandé de toujours écrire `AS` ou de mettre le nom de colonne de sortie entre guillemets.

Note

> Le nom des colonnes en sortie est différent ici de ce qui est fait dans la clause `FROM` (voir la Section 7.2.1.2, « Alias de table et de colonne »). Il est possible de renommer deux fois la même colonne mais le nom affecté dans la liste de sélection est celui qui sera passé.

7.3.3. `DISTINCT`

Après le traitement de la liste de sélection, la table résultant pourrait être optionnellement sujet à l'élimination des lignes dupliquées. Le mot clé `DISTINCT` est écrit directement après `SELECT` pour spécifier ceci :

```
SELECT DISTINCT liste_selection ...
```

(au lieu de `DISTINCT`, le mot clé `ALL` peut être utilisé pour spécifier le comportement par défaut, la récupération de toutes les lignes)

Évidemment, les deux lignes sont considérées distinctes si elles diffèrent dans au moins une valeur de colonne. Les valeurs NULL sont considérées égales dans cette comparaison.

Autrement, une expression arbitraire peut déterminer quelles lignes doivent être considérées distinctes :

```
SELECT DISTINCT ON (expression [, expression ...]) liste_selection ...
```

Ici, *`expression`* est une expression de valeur arbitraire, évaluée pour toutes les lignes. Les lignes dont toutes les expressions sont égales sont considérées comme dupliquées et seule la première ligne de cet ensemble est conservée dans la sortie. Notez que la « première ligne » d'un ensemble est non prévisible sauf si la requête est triée sur assez de colonnes pour garantir un ordre unique des colonnes arrivant dans le filtre `DISTINCT` (le traitement de `DISTINCT ON` parvient après le tri de `ORDER BY`).

La clause `DISTINCT ON` ne fait pas partie du standard SQL et est quelque fois considérée comme étant un mauvais style à cause de la nature potentiellement indéterminée de ses résultats. Avec l'utilisation judicieuse de `GROUP BY` et de sous-requêtes dans `FROM`, la construction peut être évitée mais elle représente souvent l'alternative la plus agréable.

7.4. Combiner des requêtes

Les résultats de deux requêtes peuvent être combinés en utilisant les opérations d'ensemble : union, intersection et différence. La syntaxe est

```
requete1 UNION [ALL] requete2
requete1 INTERSECT [ALL] requete2
requete1 EXCEPT [ALL] requete2
```

`requete1` et *`requete2`* sont les requêtes pouvant utiliser toutes les fonctionnalités discutées ici. Les opérations d'ensemble peuvent aussi être combinées et chaînées, par exemple

```
requete1 UNION requete2 UNION requete3
```

est exécuté ainsi :

```
(requete1 UNION requete2) UNION requete3
```

`UNION` ajoute effectivement le résultat de *`requete2`* au résultat de *`requete1`* (bien qu'il n'y ait pas de garantie qu'il s'agit de l'ordre dans lequel les lignes sont réellement renvoyées). De plus, il élimine les lignes dupliquées du résultat, de la même façon que `DISTINCT`, sauf si `UNION ALL` est utilisée.

`INTERSECT` renvoie toutes les lignes qui sont à la fois dans le résultat de *`requete1`* et dans le résultat de *`requete2`*. Les lignes dupliquées sont éliminées sauf si `INTERSECT ALL` est utilisé.

`EXCEPT` renvoie toutes les lignes qui sont dans le résultat de *`requete1`* mais pas dans le résultat de *`requete2`* (ceci est quelque fois appelé la *différence* entre deux requêtes). De nouveau, les lignes dupliquées sont éliminées sauf si `EXCEPT ALL` est utilisé.

Pour calculer l'union, l'intersection ou la différence de deux requêtes, les deux requêtes doivent être « compatibles pour une union », ce qui signifie qu'elles doivent renvoyer le même nombre de colonnes et que les colonnes correspondantes doivent avoir des types de données compatibles, comme décrit dans la Section 10.5, « Constructions `UNION`, `CASE` et constructions relatives ».

7.5. Tri des lignes

Après qu'une requête ait produit une table en sortie (après que la liste de sélection ait été traitée), elle peut être optionnellement triée. Si le tri n'a pas été choisi, les lignes sont renvoyées dans un ordre non spécifié. Dans ce cas, l'ordre réel dépendra des types de plan de parcours et de jointure et de l'ordre sur le disque mais vous ne devez pas vous y fier. Un tri particulier en sortie peut

seulement être garantie si l'étape de tri est choisie explicitement.

La clause ORDER BY spécifie l'ordre de tri :

```
SELECT liste_selection
    FROM expression_table
    ORDER BY expression_tri1 [ASC | DESC] [NULLS { FIRST | LAST }]
[, expression_tri2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

Les expressions de tri peuvent être toute expression qui serait valide dans la liste de sélection des requêtes. Voici un exemple :

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

Quand plus d'une expression est indiquée, les valeurs suivantes sont utilisées pour trier les lignes qui sont identiques aux valeurs précédentes. Chaque expression pourrait être suivie d'un ASC ou DESC optionnel pour configurer la direction du tri (ascendant ou descendant). L'ordre ASC est la valeur par défaut. L'ordre ascendant place les plus petites valeurs en premier où « plus petit » est défini avec l'opérateur <. De façon similaire, l'ordre descendant est déterminé avec l'opérateur >. [1]

Les options NULLS FIRST et NULLS LAST sont utilisées pour déterminer si les valeurs NULL apparaissent avant ou après les valeurs non NULL après un tri. Par défaut, les valeurs NULL sont triées comme si elles étaient plus grandes que toute valeur non NULL. Autrement dit, NULLS FIRST est la valeur par défaut pour l'ordre descendant (DESC) et NULLS LAST est la valeur utilisée sinon.

Notez que les options de tri sont considérées indépendament pour chaque colonne triée. Par exemple, ORDER BY x, y DESC signifie en fait ORDER BY x ASC, y DESC, ce qui est différent de ORDER BY x DESC, y DESC.

Une *expression_tri* peut aussi être à la place le nom ou le numéro d'une colonne en sortie, par exemple :

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

les deux triant par la première colonne en sortie. Notez qu'un nom de colonne en sortie doit être unique, il ne doit pas être utilisé dans une expression -- par exemple, ceci n'est *pas* correct :

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- mauvais
```

Cette restriction est là pour réduire l'ambiguïté. Il y en a toujours si un élément ORDER BY est un simple nom qui pourrait correspondre soit à un nom de colonne en sortie soit à une colonne d'une expression de table. La colonne en sortie est utilisée dans de tels cas. Cela causera seulement de la confusion si vous utilisez AS pour renommer une colonne en sortie qui correspondra à un autre nom de colonne d'une table.

ORDER BY peut être appliqué au résultat d'une combinaison UNION, d'une combinaisonINTERSECT ou d'une combinaison EXCEPT mais, dans ce cas, il est seulement permis de trier par les noms ou numéros de colonnes, pas par les expressions.

7.6. LIMIT et OFFSET

LIMIT et OFFSET vous permet de retrouver seulement une portion des lignes générées par le reste de la requête :

```
SELECT liste_selection
    FROM expression_table
    [ ORDER BY ...]
    [ LIMIT { nombre | ALL } ] [OFFSET nombre]
```

Si un nombre limite est donné, pas plus que ce nombre de lignes sera renvoyé (mais peut-être moins si la requête récupère moins de lignes). LIMIT ALL revient à ne pas spécifier la clause LIMIT.

OFFSET indique de passer ce nombre de lignes avant de renvoyer les lignes restantes. OFFSET 0 revient à oublier la clause OFFSET, et LIMIT NULL revient à oublier la clause LIMIT. Si à la fois OFFSET et LIMIT apparaissent, alors les OFFSET lignes sont laissées avant de commencer le renvoi des LIMIT lignes.

Lors de l'utilisation de LIMIT, il est important d'utiliser une clause ORDER BY contraignant les lignes résultantes dans un ordre unique. Sinon, vous obtiendrez un sous-ensemble non prévisible de lignes de la requête. Vous pourriez demander les lignes de 10 à 20 mais dans quel ordre ? L'ordre est inconnu si vous ne spécifiez pas ORDER BY.

[1] En fait, PostgreSQL™ utilise la *classe d'opérateur B-tree par défaut* pour le type de données de l'expression pour déterminer l'ordre de tri avec ASC et DESC. De façon conventionnelle, les types de données seront initialisés de façon à ce que les opérateurs < et > correspondent à cet ordre de tri mais un concepteur des types de données définis par l'utilisateur pourrait choisir de faire quelque chose de différent.

L'optimiseur de requêtes prend LIMIT en compte lors de la génération des plans de requêtes, de façon à ce que vous obteniez différents plans (avec différents ordres de lignes) suivant ce que vous donnez à LIMIT et OFFSET. Du coup, utiliser des valeurs LIMIT/OFFSET différentes pour sélectionner des sous-ensembles différents d'un résultat de requête *donnera des résultats inconsistants* sauf si vous forcez un ordre de résultat prévisible avec ORDER BY. Ceci n'est pas un bogue ; c'est une conséquence inhérente du fait que le SQL ne promette par de délivrer les résultats d'une requête dans un ordre particulier sauf si ORDER BY est utilisé pour contraindre l'ordre.

Les lignes passées par une clause OFFSET devront toujours être traitées à l'intérieur du serveur ; du coup, un OFFSET important peut être inefficace.

7.7. Listes VALUES

VALUES fournit une façon de générer une table de « constantes » qui peut être utilisé dans une requête sans avoir à réellement créer et peupler une table sur disque. La syntaxe est

```
VALUES ( expression [, ...] ) [, ...]
```

Chaque liste d'expressions entre parenthèses génère une ligne dans la table. Les listes doivent toutes avoir le même nombre d'éléments (c'est-à-dire une liste de colonnes dans la table), et les entrées correspondantes dans chaque liste doivent avoir des types compatibles. Le type réel affecté à chaque colonne du résultat est déterminé en utilisant les mêmes règles que pour UNION (voir Section 10.5, « Constructions UNION, CASE et constructions relatives »).

Voici un exemple :

```
VALUES (1, 'un'), (2, 'deux'), (3, 'trois');
```

renverra une table de deux colonnes et trois lignes. C'est équivalent à :

```
SELECT 1 AS column1, 'un' AS column2
UNION ALL
SELECT 2, 'deux'
UNION ALL
SELECT 3, 'trois';
```

Par défaut, PostgreSQL™ affecte les noms column1, column2, etc. aux colonnes d'une table VALUES. Les noms des colonnes ne sont pas spécifiés par le standard SQL et les différents SGBD le font de façon différente. Donc, il est généralement mieux de surcharger les noms par défaut avec une liste d'alias.

Syntaxiquement, VALUES suivi par une liste d'expressions est traité de la même façon que

```
SELECT liste_select FROM expression_table
```

et peut apparaître partout où un SELECT le peut. Par exemple, vous pouvez l'utiliser comme élément d'un UNION ou y attacher une *spécification de tri* (ORDER BY, LIMIT et/ou OFFSET). VALUES est habituellement utilisée comme source de données dans une commande **INSERT** command, mais aussi dans une sous-requête.

Pour plus d'informations, voir VALUES(7).

7.8. Requêtes WITH (*Common Table Expressions*)

WITH fournit une façon d'écrire les sous-requêtes pour utilisation dans une requête SELECT plus étendue. Les sous-requêtes, qui sont souvent appelées des expressions communes de table (*Common Table Expressions* ou CTE), peuvent être considérées comme la déclaration d'une table temporaire n'existant que pour la requête. Une utilisation de cette fonctionnalité est de découper des requêtes complexes en parties plus simples. En voici un exemple :

```
WITH ventes_regionales AS (
        SELECT region, SUM(montant) AS ventes_totales
        FROM commandes
        GROUP BY region
     ), meilleures_regions AS (
        SELECT region
        FROM ventes_regionales
        WHERE ventes_totales > (SELECT SUM(ventes_totales)/10 FROM ventes_regionales)
     )
SELECT region,
       produit,
       SUM(quantite) AS unites_produit,
       SUM(montant) AS ventes_produit
FROM commandes
```

```
WHERE region IN (SELECT region FROM meilleures_regions)
GROUP BY region, produit;
```

qui affiche les totaux de ventes par produit dans seulement les régions ayant les meilleures ventes. Cet exemple aurait pu être écrit sans `WITH`, mais aurait alors nécessité deux niveaux de sous-SELECT imbriqués. Les choses sont un peu plus faciles à suivre de cette façon.

Le modificateur optionnel `RECURSIVE` fait passer `WITH` du statut de simple aide syntaxique à celui de quelque chose qu'il serait impossible d'accomplir avec du SQL standard. Grâce à `RECURSIVE`, une requête `WITH` peut utiliser sa propre sortie. Un exemple très simple se trouve dans cette requête, qui ajoute les nombres de 1 à 100 :

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

La forme générale d'une requête `WITH` est toujours un *terme non-recursif*, puis `UNION` (ou `UNION ALL`), puis un *terme récursif*. Seul le terme récursif peut contenir une référence à la sortie propre de la requête. Une requête de ce genre est exécutée comme suit :

Procédure 7.1. Évaluation de requête récursive

1. Évaluer le terme non récursif. Pour `UNION` (mais pas `UNION ALL`), supprimer les enregistrements en double. Inclure le reste dans le résultat de la requête récursive et le mettre aussi dans une table temporaire de travail (*working table*.)
2. Tant que la table de travail n'est pas vide, répéter ces étapes :
   a. Évaluer le terme récursif, en substituant à la référence récursive le contenu courant de la table de travail. Pour `UNION` (mais pas `UNION ALL`), supprimer les doublons, ainsi que les enregistrements en doublon des enregistrements déjà obtenus. Inclure les enregistrements restants dans le résultat de la requête récursive, et les mettre aussi dans une table temporaire intermédiaire (*intermediate table*).
   b. Remplacer le contenu de la table de travail par celui de la table intermédiaire, puis supprimer la table intermédiaire.

Note

Dans son appellation stricte, ce processus est une itération, pas une récursion, mais `RECURSIVE` est la terminologie choisie par le comité de standardisation de SQL.

Dans l'exemple précédent, la table de travail a un seul enregistrement à chaque étape, et il prend les valeurs de 1 à 100 en étapes successives. À la centième étape, il n'y a plus de sortie en raison de la clause `WHERE`, ce qui met fin à la requête.

Les requêtes récursives sont utilisées généralement pour traiter des données hiérarchiques ou sous forme d'arbres. Cette requête est un exemple utile pour trouver toutes les sous-parties directes et indirectes d'un produit, si seule une table donne toutes les inclusions immédiates :

```
WITH RECURSIVE parties_incluses(sous_partie, partie, quantite) AS (
    SELECT sous_partie, partie, quantite FROM parties WHERE partie = 'notre_produit'
  UNION ALL
    SELECT p.sous_partie, p.partie, p.quantite
    FROM parties_incluses pr, parties p
    WHERE p.partie = pr.sous_partie
  )
SELECT sous_partie, SUM(quantite) as quantite_totale
FROM parties_incluses
GROUP BY sous_partie
```

Quand on travaille avec des requêtes récursives, il est important d'être sûr que la partie récursive de la requête finira par ne retourner aucun enregistrement, au risque sinon de voir la requête boucler indéfiniment. Quelquefois, utiliser `UNION` à la place de `UNION ALL` peut résoudre le problème en supprimant les enregistrements qui doublonnent ceux déjà retournés. Toutefois, souvent, un cycle ne met pas en jeu des enregistrements de sortie qui sont totalement des doublons : il peut s'avérer nécessaire de vérifier juste un ou quelques champs, afin de s'assurer que le même point a déjà été atteint précédemment. La méthode standard pour

gérer ces situations est de calculer un tableau de valeurs déjà visitées. Par exemple, observez la requête suivante, qui parcourt une table graphe en utilisant un champ *lien* :

```
WITH RECURSIVE parcourt_graphe(id, lien, donnee, profondeur) AS (
        SELECT g.id, g.lien, g.donnee, 1
        FROM graphe g
      UNION ALL
        SELECT g.id, g.lien, g.donnee, sg.profondeur + 1
        FROM graphe g, parcourt_graphe sg
        WHERE g.id = sg.lien
)
SELECT * FROM parcourt_graphe;
```

Cette requête va boucler si la liaison *lien* contient des boucles. Parce que nous avons besoin de la sortie « profondeur », simplement remplacer `UNION ALL` par `UNION` ne résoudra pas le problème. À la place, nous avons besoin d'identifier si nous avons atteint un enregistrement que nous avons déjà traité pendant notre parcours des liens. Nous ajoutons deux colonnes *chemin* et *boucle* à la requête :

```
WITH RECURSIVE parcourt_graphe(id, lien, donnee, profondeur, chemin, boucle) AS (
        SELECT g.id, g.lien, g.donnee, 1,
          ARRAY[g.id],
          false
        FROM graphe g
      UNION ALL
        SELECT g.id, g.lien, g.donnee, sg.profondeur + 1,
          chemin || g.id,
          g.id = ANY(chemin)
        FROM graphe g, parcourt_graphe sg
        WHERE g.id = sg.lien AND NOT boucle
)
SELECT * FROM parcourt_graphe;
```

En plus de prévenir les boucles, cette valeur de tableau est souvent pratique en elle-même pour représenter le « chemin » pris pour atteindre chaque enregistrement.

De façon plus générale, quand plus d'un champ a besoin d'être vérifié pour identifier une boucle, utilisez un tableau d'enregistrements. Par exemple, si nous avions besoin de comparer les champs *f1* et *f2* :

```
WITH RECURSIVE parcourt_graphe(id, lien, donnee, profondeur, chemin, boucle) AS (
        SELECT g.id, g.lien, g.donnee, 1,
          ARRAY[ROW(g.f1, g.f2)],
          false
        FROM graphe g
      UNION ALL
        SELECT g.id, g.lien, g.donnee, sg.profondeur + 1,
          chemin || ROW(g.f1, g.f2),
          ROW(g.f1, g.f2) = ANY(path)
        FROM graphe g, parcourt_graphe sg
        WHERE g.id = sg.link AND NOT boucle
)
SELECT * FROM parcourt_graphe;
```

Astuce

Omettez la syntaxe `ROW()` dans le cas courant où un seul champ a besoin d'être testé pour déterminer une boucle. Ceci permet, par l'utilisation d'un tableau simple plutôt que d'un tableau de type composite, de gagner en efficacité.

Astuce

L'algorithme d'évaluation récursive de requête produit sa sortie en ordre de parcours en largeur (algorithme *breadth-first*). Vous pouvez afficher les résultats en ordre de parcours en profondeur (*depth-first*) en faisant sur la requête externe un `ORDER BY` sur une colonne « chemin » construite de cette façon.

Si vous n'êtes pas certain qu'une requête peut boucler, une astuce pratique pour la tester est d'utiliser `LIMIT` dans la requête parente. Par exemple, cette requête bouclerait indéfiniment sans un `LIMIT` :

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Ceci fonctionne parce que l'implémentation de PostgreSQL™ n'évalue que le nombre d'enregistrements de la requête `WITH` récupérés par la requête parente. L'utilisation de cette astuce en production est déconseillée parce que d'autres systèmes pourraient fonctionner différemment. Par ailleurs, cela ne fonctionnera pas si vous demandez à la requête externe de trier les résultats de la requête récursive, ou si vous les joignez à une autre table.

Une propriété intéressante des requêtes `WITH` est qu'elles ne sont évaluées qu'une seule fois par exécution de la requête parente ou des requêtes `WITH` sœurs. Par conséquent, les calculs coûteux qui sont nécessaires à plusieurs endroits peuvent être placés dans une requête `WITH` pour éviter le travail redondant. Un autre intérêt peut être d'éviter l'exécution multiple d'une fonction ayant des effets de bord. Toutefois, le revers de la médaille est que l'optimiseur est moins capable d'extrapoler les restrictions de la requête parente vers une requête `WITH` que vers une sous-requête classique. La requête `WITH` sera généralement exécutée telle quelle, sans suppression d'enregistrements, que la requête parente devra supprimer ensuite. (Mais, comme mentionné précédemment, l'évaluation pourrait s'arrêter rapidement si la (les) référence(s) à la requête ne demande(nt) qu'un nombre limité d'enregistrements).

Chapitre 8. Types de données

PostgreSQL™ offre un large choix de types de données disponibles nativement. Les utilisateurs peuvent ajouter de nouveaux types à PostgreSQL™ en utilisant la commande CREATE TYPE(7).

Le Tableau 8.1, « Types de données » montre tous les types de données généraux disponibles nativement. La plupart des types de données alternatifs listés dans la colonne « Alias » sont les noms utilisés en interne par PostgreSQL™ pour des raisons historiques. Il existe également d'autres types de données internes ou obsolètes, mais ils ne sont pas listés ici.

Tableau 8.1. Types de données

Nom	Alias	Description
bigint	int8	Entier signé sur 8 octets
bigserial	serial8	Entier sur 8 octets à incrémentation automatique
bit [(*n*)]		Suite de bits de longueur fixe
bit varying [(*n*)]	varbit	Suite de bits de longueur variable
boolean	bool	Booléen (Vrai/Faux)
box		Boîte rectangulaire dans le plan
bytea		Donnée binaire (« tableau d'octets »)
character varying [(*n*)]	varchar [(*n*)]	Chaîne de caractères de longueur variable
character [(*n*)]	char [(*n*)]	Chaîne de caractères de longueur fixe
cidr		Adresse réseau IPv4 ou IPv6
circle		Cercle dans le plan
date		Date du calendrier (année, mois, jour)
double precision	float8	Nombre à virgule flottante de double précision (sur huit octets)
inet		Adresse d'ordinateur IPv4 ou IPv6
integer	int, int4	Entier signé sur 4 octets
interval [*champs*] [(*p*)]		Intervalle de temps
line		Droite (infinie) dans le plan
lseg		Segment de droite dans le plan
macaddr		Adresse MAC (pour *Media Access Control*)
money		Montant monétaire
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	Nombre exact dont la précision peut être précisée
path		Chemin géométrique dans le plan
point		Point géométrique dans le plan
polygon		Chemin géométrique fermé dans le plan
real	float4	Nombre à virgule flottante de simple précision (sur quatre octets)
smallint	int2	Entier signé sur 2 octets
serial	serial4	Entier sur 4 octets à incrémentation automatique
text		Chaîne de caractères de longueur variable
time [(*p*)] [without time zone]		Heure du jour (pas du fuseau horaire)
time [(*p*)] with time zone	timetz	Heure du jour, avec fuseau horaire
timestamp [(*p*)] [without time zone]		Date et heure (pas du fuseau horaire)
timestamp [(*p*) with time zone	timestamptz	Date et heure, avec fuseau horaire
tsquery		requête pour la recherche plein texte
tsvector		document pour la recherche plein texte
txid_snapshot		image de l'identifiant de transaction au niveau utilisateur

Nom	Alias	Description
uuid		identifiant unique universel
xml		données XML

Compatibilité

Les types suivants sont conformes à la norme SQL: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (avec et sans fuseau horaire), timestamp (avec et sans fuseau horaire), xml, .

Chaque type de données a une représentation externe déterminée par ses fonctions d'entrée et de sortie. De nombreux types de données internes ont un format externe évident. Cependant, certains types sont spécifiques à PostgreSQL™, comme les chemins géométriques, ou acceptent différents formats, comme les types de données de date et d'heure. Certaines fonctions d'entrée et de sortie ne sont pas inversables : le résultat de la fonction de sortie peut manquer de précision comparé à l'entrée initiale.

8.1. Types numériques

Les types numériques sont constitués d'entiers de 2, 4 ou 8 octets, de nombres à virgule flottante de 4 ou 8 octets et de décimaux dont la précision peut être indiquée. Le Tableau 8.2, « Types numériques » précise les types disponibles.

Tableau 8.2. Types numériques

Nom	Taille de stockage	Description	Étendue
smallint	2 octets	entier de faible étendue	de -32768 à +32767
integer	4 octets	entier habituel	de -2147483648 à +2147483647
bigint	8 octets	grand entier	de -9223372036854775808 à 9223372036854775807
decimal	variable	précision indiquée par l'utilisateur, valeur exacte	pas de limite
numeric	variable	précision indiquée par l'utilisateur, valeur exacte	pas de limite
real	4 octets	précision variable, valeur inexacte	précision de 6 décimales
double precision	8 octets	précision variable, valeur inexacte	précision de 15 décimales
serial	4 octets	entier à incrémentation automatique	de 1 à 2147483647
bigserial	8 octets	entier de grande taille à incrémentation automatique	de 1 à 9223372036854775807

La syntaxe des constantes pour les types numériques est décrite dans la Section 4.1.2, « Constantes ». Les types numériques ont un ensemble complet d'opérateurs arithmétiques et de fonctions. On peut se référer au Chapitre 9, Fonctions et opérateurs pour plus d'informations. Les sections suivantes décrivent ces types en détail.

8.1.1. Types entiers

Les types smallint, integer et bigint stockent des nombres entiers, c'est-à-dire sans décimale, de différentes étendues. Toute tentative d'y stocker une valeur en dehors de l'échelle produit une erreur.

Le type integer est le plus courant. Il offre un bon compromis entre capacité, espace utilisé et performance. Le type smallint n'est utilisé que si l'économie d'espace disque est le premier critère de choix. Le type bigint ne doit être utilisé que si le type integer n'offre pas une étendue suffisante car le type integer est nettement plus rapide.

Sur les très petits systèmes, le type bigint peut ne pas fonctionner correctement car il repose sur la capacité du compilateur à supporter les entiers de 8 octets. Sur une machine qui ne les supporte pas, bigint se comporte comme integer (mais prend bien huit octets d'espace de stockage). Cela dit, les auteurs n'ont pas connaissance de plate-forme sur laquelle il en va ainsi.

SQL ne définit que les types de données integer (ou int), smallint et bigint. Les noms de types int2, int4, et int8 sont des extensions, partagées par d'autres systèmes de bases de données SQL.

8.1.2. Nombres à précision arbitraire

Le type numeric peut stocker des nombres contenant jusqu'à 1000 chiffres significatifs et effectuer des calculs exacts. Il est spécialement recommandé pour stocker les montants financiers et autres quantités pour lesquelles l'exactitude est indispensable. Néanmoins, l'arithmétique sur les valeurs numeric est très lente comparée aux types entiers ou aux types à virgule flottante décrits dans la section suivante.

Dans ce qui suit, on utilise les termes suivants : l'*échelle* d'un numeric est le nombre de chiffres décimaux de la partie fractionnaire, à droite du séparateur de décimales. La *précision* d'un numeric est le nombre total de chiffres significatifs dans le nombre complet, c'est-à-dire le nombre de chiffres de part et d'autre du séparateur. Donc, le nombre 23.5141 a une précision de 6 et une échelle de 4. On peut considérer que les entiers ont une échelle de 0.

La précision maximale et l'échelle maximale d'une colonne numeric peuvent être toutes deux réglées. Pour déclarer une colonne de type numérique, il faut utiliser la syntaxe :

```
NUMERIC(précision, échelle)
```

La précision doit être strictement positive, l'échelle positive ou NULL. Alternativement :

```
NUMERIC(précision)
```

indique une échelle de 0.

```
NUMERIC
```

sans précision ni échelle crée une colonne dans laquelle on peut stocker des valeurs de n'importe quelle précision ou échelle, dans la limite de la précision implantée. Une colonne de ce type n'impose aucune précision à la valeur entrée, alors que les colonnes numeric ayant une échelle forcent les valeurs entrées à cette échelle. (Le standard SQL demande une précision par défaut de 0, c'est-à-dire de forcer la transformation en entiers. Les auteurs trouvent cela inutile. Dans un souci de portabilité, il est préférable de toujours indiquer explicitement la précision et l'échelle.)

Si l'échelle d'une valeur à stocker est supérieure à celle de la colonne, le système arrondit la valeur au nombre de décimales indiqué pour la colonne. Si le nombre de chiffres à gauche du point décimal est supérieur à la différence entre la précision déclarée et l'échelle déclarée, une erreur est levée.

Les valeurs numériques sont stockées physiquement sans zéro avant ou après. Du coup, la précision déclarée et l'échelle de la colonne sont des valeurs maximales, pas des allocations fixes (en ce sens, le type numérique est plus proche de varchar(*n*) que de char(*n*)). Le besoin pour le stockage réel est de deux octets pour chaque groupe de quatre chiffres décimaux, plus cinq à huit octets d'en-tête.

En plus des valeurs numériques ordinaires, le type numeric autorise la valeur spéciale `NaN` qui signifie « not-a-number » (NdT : pas un nombre). Toute opération sur `NaN` retourne `NaN`. Pour écrire cette valeur comme une constante dans une requête SQL, elle doit être placée entre guillemets. Par exemple, `UPDATE table SET x = 'NaN'`. En saisie, la chaîne `NaN` est reconnue quelque soit la casse utilisée.

Note

Dans la plupart des implémentations du concept « not-a-number », `NaN` est considéré différent de toute valeur numérique (ceci incluant `NaN`). Pour autoriser le tri des valeurs de type numeric et les utiliser dans des index basés sur le tri, PostgreSQL™ traite les valeurs `NaN` comme identiques entre elles, mais toutes supérieures aux valeurs non `NaN`.

Les types decimal et numeric sont équivalents. Les deux types sont dans le standard SQL.

8.1.3. Types à virgule flottante

Les types de données real et double precision sont des types numériques inexacts de précision variable. En pratique, ils sont généralement conformes à la norme IEEE 754 pour l'arithmétique binaire à virgule flottante (respectivement simple et double précision), dans la mesure où les processeurs, le système d'exploitation et le compilateur les supportent.

Inexact signifie que certaines valeurs ne peuvent être converties exactement dans le format interne. Elles sont, de ce fait, stockées sous une forme approchée. Ainsi, stocker puis réafficher ces valeurs peut faire apparaître de légers écarts. Prendre en compte ces erreurs et la façon dont elles se propagent au cours des calculs est le sujet d'une branche entière des mathématiques et de l'informatique, qui n'est pas le sujet de ce document, à l'exception des points suivants :

- pour un stockage et des calculs exacts, comme pour les valeurs monétaires, le type numeric doit être privilégié ;
- pour des calculs compliqués avec ces types pour quoi que ce soit d'important, et particulièrement pour le comportement aux limites (infini, zéro), l'implantation spécifique à la plate-forme doit être étudié avec soin ;

- tester l'égalité de deux valeurs à virgule flottante peut ne pas donner le résultat attendu.

Sur la plupart des plates-formes, le type real a une étendue d'au moins 1E-37 à 1E37 avec une précision d'au moins 6 chiffres décimaux. Le type double precision a généralement une étendue de 1E-307 à 1E+308 avec une précision d'au moins 15 chiffres. Les valeurs trop grandes ou trop petites produisent une erreur. Un arrondi peut avoir lieu si la précision d'un nombre en entrée est trop grande. Les nombres trop proches de zéro qui ne peuvent être représentés autrement que par zéro produisent une erreur (underflow).

> **Note**
>
> Le paramètre extra_float_digits contrôle le nombre de chiffres significatifs inclus lorsqu'une valeur à virgule flottante est convertie en texte. Avec la valeur par défaut de `0`, la sortie est la même sur chaque plateforme supportée par PostgreSQL. L'augmenter va produire une sortie représentant plus précisément la valeur stockée mais il est possible que la sortie soit différente suivant les plateformes.

En plus des valeurs numériques ordinaires, les types à virgule flottante ont plusieurs valeurs spéciales :

```
Infinity
-Infinity
NaN
```

Elles représentent les valeurs spéciales de l'IEEE 754, respectivement « infinity » (NdT : infini), « negative infinity » (NdT : infini négatif) et « not-a-number » (NdT : pas un nombre) (sur une machine dont l'arithmétique à virgule flottante ne suit pas l'IEEE 754, ces valeurs ne fonctionnent probablement pas comme espéré). Lorsqu'elles sont saisies en tant que constantes dans une commande SQL, ces valeurs doivent être placées entre guillemets. Par exemple, `UPDATE table SET x = 'Infinity'`. En entrée, ces valeurs sont reconnues quelque soit la casse utilisée.

> **Note**
>
> IEEE754 spécifie que `NaN` ne devrait pas être considéré égale à toute autre valeur en virgule flottante (ceci incluant `NaN`). Pour permettre le tri des valeurs en virgule flottante et leur utilisation dans des index basés sur des arbres, PostgreSQL™ traite les valeurs `NaN` comme identiques entre elles, mais supérieures à toute valeur différente de `NaN`.

PostgreSQL™ autorise aussi la notation float du standard SQL, ainsi que float(*p*) pour indiquer des types numériques inexacts. *p* indique la précision minimale acceptable en *chiffres binaires*. PostgreSQL™ accepte de float(1) à float(24), qu'il transforme en type real, et de float(25) à float(53), qu'il transforme en type double precision. Toute valeur de *p* hors de la zone des valeurs possibles produit une erreur. float sans précision est compris comme double precision.

> **Note**
>
> Avant PostgreSQL™ 7.4, la précision d'un float(*p*) était supposée indiquer une précision en *chiffres décimaux*. Cela a été corrigé pour respecter le standard SQL, qui indique que la précision est indiquée en chiffres binaires. L'affirmation que les real et les double precision ont exactement 24 et 53 bits dans la mantisse est correcte pour les implémentations des nombres à virgule flottante respectant le standard IEEE. Sur les plates-formes non-IEEE, c'est peut-être un peu sous-estimé mais, pour plus de simplicité, la gamme de valeurs pour *p* est utilisée sur toutes les plates-formes.

8.1.4. Types seriés

Les types de données serial et bigserial ne sont pas de vrais types, mais plutôt un raccourci de notation pour créer des colonnes d'identifiants uniques (similaires à la propriété `AUTO_INCREMENT` utilisée par d'autres SGBD). Dans la version actuelle, indiquer :

```
CREATE TABLE nom_de_table (
    nom_de_colonne SERIAL
);
```

est équivalent à écrire :

```
CREATE SEQUENCE nom_de_table_nom_de_colonne_seq;
CREATE TABLE nom_de_table (
    nom_de_colonne integer NOT NULL DEFAULT nextval('nom_de_table_nom_de_colonne_seq')
NOT NULL
```

```
);
ALTER SEQUENCE nom_de_table_nom_de_colonne_seq OWNED BY nom_de_table.nom_de_colonne;
```

Une colonne d'entiers a ainsi été créée dont la valeur par défaut est assignée par un générateur de séquence. Une contrainte `NOT NULL` est ajoutée pour s'assurer qu'une valeur NULL ne puisse pas être insérée. (Dans la plupart des cas, une contrainte `UNIQUE` ou `PRIMARY KEY` peut être ajoutée pour interdire que des doublons soient créés par accident, mais ce n'est pas automatique.) Enfin, la séquence est marquée « owned by » (possédée par) la colonne pour qu'elle soit supprimée si la colonne ou la table est supprimée.

> **Note**
>
> Avant PostgreSQL™ 7.3, serial sous-entendait `UNIQUE`. Ce n'est plus automatique. Pour qu'une colonne de type serial soit unique ou soit une clé primaire, il faut le préciser, comme pour les autres types.

Pour insérer la valeur suivante de la séquence dans la colonne serial, il faut préciser que la valeur par défaut de la colonne doit être utilisée. Cela peut se faire de deux façons : soit en excluant cette colonne de la liste des colonnes de la commande **INSERT** soit en utilisant le mot clé `DEFAULT`.

Les types serial et serial4 sont identiques : ils créent tous les deux des colonnes integer. Les types bigserial et serial8 fonctionnent de la même façon mais créent des colonnes bigint. bigserial doit être utilisé si plus de 2^{31} identifiants sont prévus sur la durée de vie de la table.

La séquence créée pour une colonne serial est automatiquement supprimée quand la colonne correspondante est supprimée. La séquence peut être détruite sans supprimer la colonne, mais la valeur par défaut de la colonne est alors également supprimée.

8.2. Types monétaires

Le type money stocke un montant en devise avec un nombre fixe de décimales. Voir le Tableau 8.3, « Types monétaires ». La précision de la partie fractionnée est déterminée par le paramètre lc_monetary de la base de données. De nombreux formats sont acceptés en entrée, dont les entiers et les nombre à virgule flottante, ainsi que les formats classiques de devises, comme `'$1,000.00'`. Le format de sortie est généralement dans le dernier format, mais dépend de la locale. Les valeurs numériques qui ne sont pas entre guillemets peuvent être converties dans le type money en convertissant la valeur numérique en text puis en money, par exemple :

```
SELECT 1234::text::money;
```

Il n'existe pas de moyen simple de faire l'inverse d'une façon indépendante de la locale, donc en convertissant une valeur de type money en un type numérique. Si vous connaissez le symbole de la monnaie et le séparateur des milliers, vous pouvez utiliser `regexp_replace()` :

```
SELECT regexp_replace('52093.89'::money::text, '[$,]', '', 'g')::numeric;
```

Comme l'affichage de ce type de données est sensible à la casse, charger des données du type money dans une base de données qui a une configuration différente de `lc_monetary` pourrait ne pas fonctionner. Pour éviter les problèmes, assurez-vous, avant de restaurer une sauvegarde dans une nouvelle base de données, que la variable `lc_monetary` a la même valeur ou une valeur équivalente à celle qui était configurée lors de la sauvegarde.

Tableau 8.3. Types monétaires

Nom	Taille de stockage	Description	Étendue
money	8 octets	montant monétaire	-92233720368547758.08 à +92233720368547758.07

8.3. Types caractère

Tableau 8.4. Types caractère

Nom	Description
character varying(*n*), varchar(*n*)	Longueur variable avec limite
character(*n*), char(*n*)	longueur fixe, complété par des espaces

Nom	Description
text	longueur variable illimitée

Le Tableau 8.4, « Types caractère » présente les types génériques disponibles dans PostgreSQL™.

SQL définit deux types de caractères principaux : character varying(*n*) et character(*n*) où *n* est un entier positif. Ces deux types permettent de stocker des chaînes de caractères de taille inférieure ou égale à *n* (ce ne sont pas des octets). Toute tentative d'insertion d'une chaîne plus longue conduit à une erreur, à moins que les caractères en excès ne soient tous des espaces, auquel cas la chaîne est tronquée à la taille maximale (cette exception étrange est imposée par la norme SQL). Si la chaîne à stocker est plus petite que la taille déclarée, les valeurs de type character sont complétées par des espaces, celles de type character varying sont stockées en l'état.

Si une valeur est explicitement transtypée en character varying(*n*) ou en character(*n*), une valeur trop longue est tronquée à *n* caractères sans qu'aucune erreur ne soit levée (ce comportement est aussi imposé par la norme SQL.)

Les notations varchar(*n*) et char(*n*) sont des alias de character varying(*n*) et character(*n*), respectivement. character sans indication de taille est équivalent à character(1). Si character varying est utilisé sans indicateur de taille, le type accepte des chaînes de toute taille. Il s'agit là d'une spécificité de PostgreSQL™.

De plus, PostgreSQL™ propose aussi le type text, qui permet de stocker des chaînes de n'importe quelle taille. Bien que le type text ne soit pas dans le standard SQL, plusieurs autres systèmes de gestion de bases de données SQL le proposent également.

Les valeurs de type character sont complétées physiquement à l'aide d'espaces pour atteindre la longueur *n* indiquée. Ces valeurs sont également stockées et affichées de cette façon. Les espaces de remplissage n'ont, toutefois, aucune signification sémantique. Les espaces finales sont ignorées lors de la comparaison de deux valeurs de type character et sont supprimées lors de la conversion d'une valeur character en un des autres types chaîne. Ces espaces *ont* une signification sémantique pour les valeurs de type character varying et text.

L'espace nécessaire pour une chaîne de caractères courte (jusqu'à 126 octets) est de un octet, plus la taille de la chaîne qui inclut le remplissage avec des espaces dans le cas du type character. Les chaînes plus longues ont quatre octets d'en-tête au lieu d'un seul. Les chaînes longues sont automatiquement compressées par le système, donc le besoin pourrait être moindre. Les chaînes vraiment très longues sont stockées dans des tables supplémentaires, pour qu'elles n'empêchent pas d'accéder rapidement à des valeurs plus courtes. Dans tous les cas, la taille maximale possible pour une chaîne de caractères est de l'ordre de 1 Go. (La taille maximale pour *n* dans la déclaration de type est inférieure. Il ne sert à rien de modifier ce comportement, car avec les encodages sur plusieurs octets, les nombres de caractères et d'octets peuvent être très différents. Pour stocker de longues chaînes sans limite supérieure précise, il est préférable d'utiliser les types text et character varying sans taille, plutôt que d'indiquer une limite de taille arbitraire.)

Astuce

Il n'y a aucune différence de performance parmi ces trois types, si ce n'est la place disque supplémentaire requise pour le type à remplissage et quelques cycles CPU supplémentaires pour vérifier la longueur lors du stockage dans une colonne contrainte par la taille. Bien que character(*n*) ait des avantages en terme de performance sur certains autres systèmes de bases de données, il ne dispose pas de ce type d'avantages dans PostgreSQL™ ; en fait, character(*n*) est habituellement le plus lent des trois à cause des coûts de stockage supplémentaires. Dans la plupart des situations, les types text et character varying peuvent être utilisés à leur place.

On peut se référer à la Section 4.1.2.1, « Constantes de chaînes » pour obtenir plus d'informations sur la syntaxe des libellés de chaînes, et le Chapitre 9, Fonctions et opérateurs pour des informations complémentaires sur les opérateurs et les fonctions. Le jeu de caractères de la base de données détermine celui utilisé pour stocker les valeurs texte ; pour plus d'informations sur le support des jeux de caractères, se référer à la Section 22.2, « Support des jeux de caractères ».

Exemple 8.1. Utilisation des types caractère

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; --
  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
```

```
INSERT INTO test2 VALUES ('bien      ');
INSERT INTO test2 VALUES ('trop long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('trop long'::varchar(5)); -- troncature explicite
SELECT b, char_length(b) FROM test2;

   b   | char_length
-------+-------------
 ok    |           2
 bien  |           5
 trop  |           5
```

1 La fonction `char_length` est décrite dans la Section 9.4, « Fonctions et opérateurs de chaînes ».

Il y a deux autres types caractère de taille fixe dans PostgreSQL™. Ils sont décrits dans le Tableau 8.5, « Types caractères spéciaux ». Le type name existe *uniquement* pour le stockage des identifiants dans les catalogues systèmes et n'est pas destiné à être utilisé par les utilisateurs normaux. Sa taille est actuellement définie à 64 octets (63 utilisables plus le terminateur) mais doit être référencée en utilisant la constante NAMEDATALEN en code source `C`. La taille est définie à la compilation (et est donc ajustable pour des besoins particuliers). La taille maximale par défaut peut éventuellement être modifiée dans une prochaine version. Le type "char" (attention aux guillemets) est différent de char(1) car il n'utilise qu'un seul octet de stockage. Il est utilisé dans les catalogues systèmes comme un type d'énumération simpliste.

Tableau 8.5. Types caractères spéciaux

Nom	Taille de stockage	Description
"char"	1 octet	type interne d'un octet
name	64 octets	type interne pour les noms d'objets

8.4. Types de données binaires

Le type de données bytea permet de stocker des chaînes binaires ; voir le Tableau 8.6, « Types de données binaires ».

Tableau 8.6. Types de données binaires

Nom	Espace de stockage	Description
bytea	un à quatre octets plus la taille de la chaîne binaire à stocker	Chaîne binaire de longueur variable

Une chaîne binaire est une séquence d'octets. Les chaînes binaires se distinguent des chaînes de caractères de deux façons : tout d'abord, les chaînes binaires permettent de stocker des octets de valeurs zéro ainsi que les autres caractères « non imprimables » (habituellement, les octets en dehors de l'échelle de 32 à 126). Les chaînes de caractères interdisent les octets de valeur zéro et interdisent aussi toute valeur d'octet ou séquence d'octets invalide selon l'encodage sélectionné pour la base de données. Ensuite, les opérations sur les chaînes binaires traitent réellement les octets alors que le traitement de chaînes de caractères dépend de la configuration de la locale. En résumé, les chaînes binaires sont appropriées pour le stockage de données que le développeur considère comme des « octets bruts » alors que les chaînes de caractères sont appropriées pour le stockage de texte.

Le type bytea supporte deux formats externes pour l'entrée et la sortie : le format d'échappement (« escape ») historique de PostgreSQL™ et le format hexadécimal (« hex »). Les deux sont acceptés en entrée. Le format de sortie dépend du paramètre de configuration bytea_output ; ce dernier sélectionne par défaut le format hexadécimal. (Notez que le format hexadécimal est disponible depuis PostgreSQL™ 9.0 ; les versions antérieures et certains outils ne le comprennent pas.)

Le standard SQL définit un type de chaîne binaire différent, appelé BLOB ou BINARY LARGE OBJECT. Le format en entrée est différent du bytea, mais les fonctions et opérateurs fournis sont pratiquement les mêmes.

8.4.1. Le format hexadécimal bytea

Le format « hex » code les données binaires sous la forme de deux chiffres hexadécimaux par octet, le plus significatif en premier. La chaîne complète est précédée par la séquence `\x` (pour la distinguer du format d'échappement). Dans la majorité des cas (exactement les mêmes pour lesquelles les antislashs sont doublés dans le format d'échappement), l'antislash initial peut avoir besoin d'être échappé par un doublage du caractère ; les détails sont disponibles plus bas. Les chiffres hexadécimaux peuvent être soit en majuscule, soit en minuscule, et les espaces blancs sont permis entre les paires de chiffres (mais pas à l'intérieur d'une paire

ni dans la séquence `\x` de début). Le format hexadécimal est compatible avec une grande variété d'applications et de protocoles externes, et il a tendance à être plus rapide à convertir que le format d'échappement. Son utilisation est donc préférée.

Exemple :

```
SELECT E'\\xDEADBEEF';
```

8.4.2. Le format d'échappement bytea

Le format d'échappement (« escape ») est le format traditionnel de PostgreSQL™ pour le type bytea. Son approche est de représenter une chaîne binaire comme un séquence de caractères ASCII et de convertir les données qui ne peuvent pas être représentés en ASCII en une séquence spéciale d'échappement. Si, du point de vue de l'application, représenter les octets sous la forme de caractères revet un sens, alors cette représentation est intéressante. En pratique, c'est généralement source de confusion car cela diminue la distinction entre chaînes binaires et chaînes textuelles. De plus le mécanisme particulier de l'échappement qui a été choisi est quelque peu unwieldy. Donc ce format devrait probablement être évité pour la plupart des nouvelles applications.

Lors de la saisie de valeurs bytea dans le format d'échappement, les octets de certaines valeurs *doivent* être échappés alors que les autres valeurs d'octet *peuvent* être échappés. En général, pour échapper un octet, il suffit de le convertir dans sa valeur octal composée de trois chiffres et de la faire précéder d'un antislash (ou de deux antislashs s'il faut utiliser la syntaxe d'échappement de chaînes). L'antislash lui-même (octet 92) peut alternativement être représenté par un double antislashs. Le Tableau 8.7, « Octets littéraux bytea à échapper » affiche les caractères qui doivent être échappés, et donne les séquences d'échappement possibles.

Tableau 8.7. Octets littéraux bytea à échapper

Valeur décimale de l'octet	Description	Représentation échappée en entrée	Exemple	Représentation en sortie
0	octet zéro	`E'\\000'`	`SELECT E'\\000'::bytea;`	`\000`
39	apostrophe	`''''` or `E'\\047'`	`SELECT E'\''::bytea;`	`'`
92	antislash	`E'\\\\'` or `E'\\134'`	`SELECT E'\\\\'::bytea;`	`\\`
de 0 à 31 et de 127 à 255	octets « non affichables »	`E'\\xxx'` (octal value)	`SELECT E'\\001'::bytea;`	`\001`

La nécessité d'échapper les octets *non affichables* dépend des paramétrages de la locale. Il est parfois possible de s'en sortir sans échappement. Le résultat de chacun des exemples du Tableau 8.7, « Octets littéraux bytea à échapper » fait exactement un octet, même si la représentation en sortie fait plus d'un caractère.

S'il faut écrire tant d'antislashs, comme indiqué dans le Tableau 8.7, « Octets littéraux bytea à échapper », c'est qu'une chaîne binaire doit passer à travers deux phases d'analyse dans le serveur PostgreSQL™. Le premier antislash de chaque paire est vu comme un caractère d'échappement par l'analyseur de chaîne (en supposant que la syntaxe d'échappement des chaînes soit utilisée) et est donc consommé, laissant le second antislash de la paire. (Les chaînes à guillemets dollar peuvent être utilisées pour éviter ce niveau d'échappement.) L'antislash restant est compris par la fonction d'entrée de PostgreSQL™ comme le début d'une valeur octale sur trois caractères ou comme l'échappement d'un autre antislash. Par exemple, une chaîne littérale passée au serveur comme `E'\\001'` devient `\001` après être passée au travers de l'analyseur d'échappement de chaîne. Le `\001` est envoyé à la fonction d'entrée de bytea, qui le convertit en un octet simple ayant une valeur décimale de 1. Le guillemet simple n'est pas traité spécialement par bytea et suit les règles normales des chaînes littérales de chaîne. Voir aussi la Section 4.1.2.1, « Constantes de chaînes ».

Les octets de bytea sont également échappés en sortie. En général, tout octet « non-imprimable » est converti en son équivalent octal sur trois caractères et précédé d'un antislash. La plupart des caractères « imprimables » sont affichés avec leur représentation standard dans le jeu de caractères du client. Les octets de valeur décimale 92 (antislash) sont doublés. Les détails sont dans le Tableau 8.8, « Octets échappés en sortie pour bytea ».

Tableau 8.8. Octets échappés en sortie pour bytea

Valeur décimale de l'octet	Description	Représentation de sortie échappée	Exemple	Résultat en sortie
92	antislash	`\\`	`SELECT E'\\134'::bytea;`	`\\`
0 à 31 et 127 à 255	octets« non affichables »	`\xxx` (valeur octale)	`SELECT`	`\001`

Valeur décimale de l'octet	Description	Représentation de sortie échappée	Exemple	Résultat en sortie
			`E'\\001'::bytea;`	
32 à 126	octets « affichables »	Représentation dans le jeu de caractères du client	`SELECT E'\\176'::bytea;`	~

En fonction de l'interface utilisée pour accéder à PostgreSQL™, un travail supplémentaire d'échappement/de « déséchappement » des chaînes bytea peut être nécessaire. Il faut également échapper les sauts de lignes et retours à la ligne si l'interface les traduit automatiquement, par exemple.

8.5. Types date/heure

PostgreSQL™ supporte l'ensemble des types date et heure du SQL. Ces types sont présentés dans le Tableau 8.9, « Types date et heure ». Les opérations disponibles sur ces types de données sont décrites dans la Section 9.9, « Fonctions et opérateurs sur date/heure ».

Tableau 8.9. Types date et heure

Nom	Taille de stockage	Description	Valeur minimale	Valeur maximale	Résolution
timestamp [(*p*)] [without time zone]	8 octets	date et heure (sans fuseau horaire)	4713 avant JC	294276 après JC	1 microseconde / 14 chiffres
timestamp [(*p*)] with time zone	8 octets	date et heure, avec fuseau horaire	4713 avant JC	294276 après JC	1 microseconde / 14 chiffres
date	4 octets	date seule (pas d'heure)	4713 avant JC	5874897 après JC	1 jour
time [(*p*)] [without time zone]	8 octets	heure seule (pas de date)	00:00:00.00	24:00:00	1 microseconde / 14 chiffres
time [(*p*)] with time zone	12 octets	heure seule, avec fuseau horaire	00:00:00+1459	24:00:00-1459	1 microseconde / 14 chiffres
interval [`champs`] [(*p*)]	16 octets	intervalles de temps	-178000000 années	178000000 années	1 microseconde / 14 chiffres

Note

Le standard SQL impose que timestamp soit un équivalent de timestamp without time zone. PostgreSQL™ force ce comportement à partir de la version 7.3. Les versions antérieures traitaient ce type de données comme le type timestamp with time zone.)

time, timestamp, et interval acceptent une précision optionnelle *p*, qui indique le nombre de décimales pour les secondes. Il n'y a pas, par défaut, de limite explicite à cette précision. Les valeurs acceptées pour *p* s'étendent de 0 à 6 pour les types timestamp et interval.

Note

Quand des valeurs de type timestamp sont stockées sur des entiers de 8 octets (ce qui est la valeur par défaut actuelle), la précision à la microseconde près est disponible sur tout le spectre des valeurs. Quand les timestamp sont stockés en nombres à virgule flottante double précision à la place (une option de compilation obsolète), la limite effective de précision peut être inférieure à 6. Les valeurs de type timestamp sont stockées en secondes avant ou après le 01/01/2000 à minuit. Quand les valeurs timestamp sont implémentées avec des nombres à virgule flottante, la précision à la microseconde n'est obtenue que sur les quelques années autour du 01/01/2000, et décroît pour les dates plus éloignées. Notez qu'utiliser des types date à virgule flottante permet d'avoir une plus grande étendue de timestamp : de 4713 av. J.-C. à 5874897 ap. J.-C., à la différence de ce qui est écrit plus haut.

La même option de compilation détermine aussi si les valeurs de type time et interval sont stockées en tant que nombres à virgule flottante ou entiers de 8 octets. Dans le cas de la virgule flottante, la précision des valeurs de type interval se dégradent avec leur accroissement.

Pour les types time, l'intervalle accepté pour *p* s'étend de 0 à 6 pour les entiers sur 8 octets et de 0 à 10 pour les nombres à virgule flottante.

Le type interval a une option supplémentaire, qui permet de restreindre le jeu de champs stockés en écrivant une de ces expressions :

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Notez que si *champs* et *p* sont tous les deux indiqués, *champs* doit inclure `SECOND`, puisque la précision s'applique uniquement aux secondes.

Le type time with time zone est défini dans le standard SQL mais sa définition lui prête des propriétés qui font douter de son utilité. Dans la plupart des cas, une combinaison de date, time, timestamp without time zone et timestamp with time zone devrait permettre de résoudre toutes les fonctionnalités de date et heure nécessaires à une application.

Les types abstime et reltime sont des types de précision moindre, utilisés en interne. Il n'est pas recommandé de les utiliser dans de nouvelles applications car ils pourraient disparaître dans une prochaine version.

8.5.1. Saisie des dates et heures

La saisie de dates et heures peut se faire dans la plupart des formats raisonnables, dont ISO8601, tout format compatible avec SQL, le format POSTGRES™ traditionnel ou autres. Pour certains formats, l'ordre des jours, mois et années en entrée est ambigu. Il est alors possible de préciser l'ordre attendu pour ces champs. Le paramètre datestyle peut être positionné à `MDY` pour choisir une interprétation mois-jour-année, à `DMY` pour jour-mois-année ou à `YMD` pour année-mois-jour.

PostgreSQL™ est plus flexible que la norme SQL ne l'exige pour la manipulation des dates et des heures. Voir l'Annexe B, Support de date/heure pour connaître les règles exactes de reconnaissance des dates et heures et les formats reconnus pour les champs texte comme les mois, les jours de la semaine et les fuseaux horaires.

Tout libellé de date ou heure saisi doit être placé entre apostrophes, comme les chaînes de caractères. La Section 4.1.2.7, « Constantes d'autres types » peut être consultée pour plus d'information. SQL requiert la syntaxe suivante :

```
type [ (p) ] 'valeur'
```

où *p*, précision optionnelle, est un entier correspondant au nombre de décimales du champ secondes. La précision peut être précisée pour les types time, timestamp, et interval. Les valeurs admissibles sont mentionnées plus haut. Si aucune précision n'est indiquée dans une déclaration de constante, celle de la valeur littérale est utilisée.

8.5.1.1. Dates

Le Tableau 8.10, « Saisie de date » regroupe les formats de date possibles pour la saisie de valeurs de type date.

Tableau 8.10. Saisie de date

Exemple	Description
1999-01-08	ISO-8601 ; 8 janvier, quel que soit le mode (format recommandé)
January 8, 1999	sans ambiguïté quel que soit le style de date (`datestyle`)
1/8/1999	8 janvier en mode `MDY` ; 1er août en mode `DMY`
1/18/1999	18 janvier en mode `MDY` ; rejeté dans les autres modes
01/02/03	2 janvier 2003 en mode `MDY` ; 1er février 2003 en mode `DMY` ; 3 février 2001 en mode `YMD`
1999-Jan-08	8 janvier dans tous les modes
Jan-08-1999	8 janvier dans tous les modes

Exemple	Description
08-Jan-1999	8 janvier dans tous les modes
99-Jan-08	8 janvier en mode `YMD`, erreur sinon
08-Jan-99	8 janvier, sauf en mode `YMD` : erreur
Jan-08-99	8 janvier, sauf en mode `YMD` : erreur
19990108	ISO-8601 ; 8 janvier 1999 dans tous les modes
990108	ISO-8601 ; 8 janvier 1999 dans tous les modes
1999.008	Année et jour de l'année
J2451187	Jour du calendrier Julien
January 8, 99 BC	année 99 avant Jésus Christ

8.5.1.2. Heures

Les types heure-du-jour sont time [(*p*)] without time zone et time [(*p*)] with time zone. time est équivalent à time without time zone.

Les saisies valides pour ces types sont constituées d'une heure suivie éventuellement d'un fuseau horaire (voir le Tableau 8.11, « Saisie d'heure » et le Tableau 8.12, « Saisie des fuseaux horaires »). Si un fuseau est précisé pour le type time without time zone, il est ignoré sans message d'erreur. Si une date est indiquée, elle est ignorée sauf si un fuseau horaire impliquant une règle de changement d'heure (heure d'été/heure d'hiver) est précisé, `America/New_York` par exemple. Dans ce cas, la date est nécessaire pour pouvoir déterminer la règle de calcul de l'heure qui s'applique. Le décalage approprié du fuseau horaire est enregistré dans la valeur de time with time zone.

Tableau 8.11. Saisie d'heure

Exemple	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	Identique à 04:05 ; AM n'affecte pas la valeur
`04:05 PM`	Identique à 16:05 ; l'heure doit être <= 12
`04:05:06.789-8`	ISO 8601
`04:05:06-08:00`	ISO 8601
`04:05-08:00`	ISO 8601
`040506-08`	ISO 8601
`04:05:06 PST`	fuseau horaire abrégé
`2003-04-12 04:05:06 America/New_York`	fuseau horaire en nom complet

Tableau 8.12. Saisie des fuseaux horaires

Exemple	Description
PST	Abréviation pour l'heure standard du Pacifique (Pacific Standard Time)
`America/New_York`	Nom complet du fuseau horaire
`PST8PDT`	Nommage POSIX du fuseau horaire
-8:00	Décalage ISO-8601 pour la zone PST
-800	Décalage ISO-8601 pour la zone PST
-8	Décalage ISO-8601 pour la zone PST

Exemple	Description
`zulu`	Abréviation militaire de GMT
`z`	Version courte de `zulu`

L'Section 8.5.3, « Fuseaux horaires » apporte des précisions quant à la façon d'indiquer les fuseaux horaires.

8.5.1.3. Horodatage

Les saisies valides sont constituées de la concaténation d'une date et d'une heure, éventuellement suivie d'un fuseau horaire et d'un qualificatif `AD` (après Jésus Christ) ou `BC` (avant Jésus Christ). (`AD`/`BC` peut aussi apparaître avant le fuseau horaire mais ce n'est pas l'ordre préféré.) Ainsi :

```
1999-01-08 04:05:06
```

et :

```
1999-01-08 04:05:06 -8:00
```

sont des valeurs valides, qui suivent le standard ISO 8601. Le format très courant :

```
January 8 04:05:06 1999 PST
```

est également supporté.

Le standard SQL différencie les libéllés timestamp without time zone et timestamp with time zone par la présence d'un symbole « + » ou d'un « - » et le déclage du fuseau horaire après l'indication du temps. De ce fait, d'après le standard,

```
TIMESTAMP '2004-10-19 10:23:54'
```

est du type timestamp without time zone alors que

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

est du type timestamp with time zone. PostgreSQL™ n'examine jamais le contenu d'un libellé avant de déterminer son type. Du coup, il traite les deux ci-dessus comme des valeurs de type timestamp without time zone. Pour s'assurer qu'un littéral est traité comme une valeur de type timestamp with time zone, il faut préciser explicitement le bon type :

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

Dans un libellé de type timestamp without time zone, PostgreSQL™ ignore silencieusement toute indication de fuseau horaire. C'est-à-dire que la valeur résultante est dérivée des champs date/heure de la valeur saisie et n'est pas corrigée par le fuseau horaire.

Pour timestamp with time zone, la valeur stockée en interne est toujours en UTC (*Universal Coordinated Time* ou Temps Universel Coordonné), aussi connu sous le nom de GMT (*Greenwich Mean Time*). Les valeurs saisies avec un fuseau horaire explicite sont converties en UTC à l'aide du décalage approprié. Si aucun fuseau horaire n'est précisé, alors le système considère que la date est dans le fuseau horaire indiqué par le paramètre système timezone, et la convertit en UTC en utilisant le décalage de la zone `timezone`.

Quand une valeur timestamp with time zone est affichée, elle est toujours convertie de l'UTC vers le fuseau horaire courant (variable `timezone`), et affichée comme une heure locale. Pour voir l'heure dans un autre fuseau horaire, il faut, soit changer la valeur de `timezone`, soit utiliser la construction `AT TIME ZONE` (voir la Section 9.9.3, « `AT TIME ZONE` »).

Les conversions entre timestamp without time zone et timestamp with time zone considèrent normalement que la valeur timestamp without time zone utilise le fuseau horaire `timezone`. Un fuseau différent peut être choisi en utilisant `AT TIME ZONE`.

8.5.1.4. Valeurs spéciales

PostgreSQL™ supporte plusieurs valeurs de dates spéciales, dans un souci de simplification. Ces valeurs sont présentées dans le Tableau 8.13, « Saisie de dates/heures spéciales ». Les valeurs `infinity` et `-infinity` ont une représentation spéciale dans le système et sont affichées ainsi ; les autres ne sont que des raccourcies de notation convertis en dates/heures ordinaires lorsqu'ils sont lus. (En particulier, `now` et les chaînes relatives sont converties en une valeur de temps spécifique à leur lecture). Toutes ces valeurs doivent être écrites entre simples quotes lorsqu'elles sont utilisées comme des constantes dans les commandes SQL.

Tableau 8.13. Saisie de dates/heures spéciales

Saisie	Types valides
`epoch`	date, timestamp
`infinity`	date, timestamp
`-infinity`	date, timestamp

Saisie	Types valides
`now`	date, time, timestamp
`today`	date, timestamp
`tomorrow`	date, timestamp
`yesterday`	date, timestamp
`allballs`	time

Les fonctions suivantes, compatibles avec le standard SQL, peuvent aussi être utilisées pour obtenir l'heure courante pour le type de données correspondant : `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`. Les quatre derniers acceptent une indication optionnelle de précision en dessous de la seconde (voir la Section 9.9.4, « Date/Heure courante »). Ce sont là des fonctions SQL qui ne sont *pas* reconnues comme chaînes de saisie de données.

8.5.2. Affichage des dates et heures

Le format de sortie des types date/heure peut être positionné à l'un des quatre formats de date suivants : ISO 8601, SQL (Ingres), traditionnel POSTGRES™ (date au format Unix date) ou German (germanique). Le format par défaut est le format ISO. (Le standard `SQL` impose l'utilisation du format ISO 8601. Le nom du format d'affichage « SQL » est mal choisi, un accident historique.) Le Tableau 8.14, « Styles d'affichage de date/heure » présente des exemples de chaque format d'affichage. La sortie d'un type date ou time n'est évidemment composée que de la partie date ou heure, comme montré dans les exemples.

Tableau 8.14. Styles d'affichage de date/heure

Spécification de style	Description	Exemple
ISO	standard ISO 8601/SQL	1997-12-17 07:37:16-08
SQL	style traditionnel	12/17/1997 07:37:16.00 PST
POSTGRES	style original	Wed Dec 17 07:37:16 1997 PST
German	style régional	17.12.1997 07:37:16.00 PST

Dans les styles SQL et POSTGRES, les jours apparaissent avant le mois si l'ordre des champs DMY a été précisé, sinon les mois apparaissent avant les jours (voir la Section 8.5.1, « Saisie des dates et heures » pour savoir comment ce paramètre affecte l'interprétation des valeurs en entrée). Le Tableau 8.15, « Convention de présentation des dates » présente un exemple.

Tableau 8.15. Convention de présentation des dates

Valeur de `datestyle` (style de date)	Ordre de saisie	Exemple d'affichage
`SQL, DMY`	*jour*/*mois*/*année*	17/12/1997 15:37:16.00 CET
`SQL, MDY`	*mois*/*jour*/*année*	12/17/1997 07:37:16.00 PST
`Postgres, DMY`	*jour*/*mois*/*année*	Wed 17 Dec 07:37:16 1997 PST

Les styles de date/heure peuvent être sélectionnés à l'aide de la commande **SET datestyle**, du paramètre datestyle du fichier de configuration `postgresql.conf` ou par la variable d'environnement `PGDATESTYLE` sur le serveur ou le client. La fonction de formatage `to_char` (voir Section 9.8, « Fonctions de formatage des types de données ») permet de formater les affichages de date/heure de manière plus flexible.

8.5.3. Fuseaux horaires

Les fuseaux horaires et les conventions liées sont influencées par des décisions politiques, pas uniquement par la géométrie de la terre. Les fuseaux horaires se sont quelque peu standardisés au cours du vingtième siècle mais continuent à être soumis à des changements arbitraires, particulièrement en respect des règles de changement d'heure (heure d'été/heure d'hiver). PostgreSQL™ utilise la très répandue base de données de fuseaux horaires `zoneinfo` pour gérer les informations sur les règles de fuseau horaire historiques. Pour les dates se situant dans le futur, PostgreSQL™ part de l'assomption que les dernières règles connues pour un fuseau continueront à s'appliquer dans le futur.

PostgreSQL™ se veut compatible avec les définitions standard SQL pour un usage typique. Néanmoins, le standard SQL possède un mélange étrange de types de date/heure et de possibilités. Deux problèmes évidents sont :

- bien que le type date ne peut pas se voir associer un fuseau horaire, le type heure peut en avoir un. Les fuseaux horaires, dans le monde réel, ne peuvent avoir de sens qu'associés à une date et à une heure, vu que l'écart peut varier avec l'heure d'été ;
- le fuseau horaire par défaut est précisé comme un écart numérique constant avec l'UTC. Il n'est, de ce fait, pas possible de s'adapter à l'heure d'été ou d'hiver lorsque l'on fait des calculs arithmétiques qui passent les limites de l'heure d'été et de l'heure d'hiver.

Pour éviter ces difficultés, il est recommandé d'utiliser des types date/heure qui contiennent à la fois une date et une heure lorsque les fuseaux horaires sont utilisés. Il est également préférable de *ne pas* utiliser le type time with time zone. (Ce type est néanmoins proposé par PostgreSQL™ pour les applications existantes et pour assurer la compatibilité avec le standard SQL.) PostgreSQL™ utilise le fuseau horaire local pour tous les types qui ne contiennent qu'une date ou une heure.

Toutes les dates et heures liées à un fuseau horaire sont stockées en interne en UTC. Elles sont converties en heure locale dans le fuseau indiqué par le paramètre de configuration timezone avant d'être affiché sur le client.

PostgreSQL™ permet d'indiquer les fuseaux horaires de trois façons différentes :

- un nom complet de fuseau horaire, par exemple `America/New_York`. Les noms reconnus de fuseau horaire sont listés dans la vue `pg_timezone_names` (voir Section 45.60, « pg_timezone_names »). PostgreSQL™ utilise les données `zoneinfo` pour cela, les mêmes noms sont donc reconnus par de nombreux autres logiciels ;
- une abréviation de fuseau horaire, par exemple `PST`. Une telle indication ne définit qu'un décalage particulier à partir d'UTC, en contraste avec les noms complets de fuseau horaire qui peuvent aussi impliquer un ensemble de dates pour le changement d'heure. Les abréviations reconnues sont listées dans la vue `pg_timezone_abbrevs` (voir Section 45.59, « pg_timezone_abbrevs »). Les paramètres de configuration timezone et log_timezone ne peuvent pas être configurés à l'aide d'une abréviation de fuseau horaire, mais ces abréviations peuvent être utilisées dnas les saisies de date/heure et avec l'opérateur `AT TIME ZONE` ;
- une spécification POSIX de fuseau sous la forme *`STDdécalage`* ou *`STDdécalageDST`* avec *`STD`* une abréviation de fuseau et *`décalage`* un décalage numérique en nombre d'heures à l'ouest d'UTC et *`DST`* une abréviation optionnelle de changement d'heure, à interpréter comme une heure avant le décalage donné. Par exemple si `EST5EDT` n'est pas déjà reconnu comme fuseau horaire, il est accepté et est fonctionnellement équivalent à l'heure du fuseau de la côte est des USA. Si un nom de changement d'heure est présent, il est interprété selon les règles régissant les changements d'heure utilisées dans l'entrée `posixrules` de la base de données des fuseaux horaires, `zoneinfo`. Dans une installation PostgreSQL™ standard, `posixrules` est identique à `US/Eastern`, pour que les spécifications POSIX des fuseaux horaires correspondent aux règles de changements d'heure aux États-Unis. Ce comportement peut, au besoin, être ajusté en remplaçant le fichier `posixrules`.

En résumé, il y a une différence entre les abréviations et les noms complets : les abréviations représentent toujours un décalage fixe par rapport à UTC alors que la plupart des noms complets impliquent une règle de changement d'heure et donc deux décalages possibles.

La fonctionnalité des fuseaux horaires POSIX peut accepter silencieusement des saisies erronées car il n'y a pas de vérification des abréviations de fuseaux horaires. Par exemple, `SET TIMEZONE TO FOOBAR0` fonctionne mais conduit le système à utiliser en réalité une abréviation très particulière d'UTC. Un autre problème à conserver en tête est que, pour les noms des fuseaux horaires POSIX, les décalages positifs sont utilisés pour les emplacements situés à l'*ouest* de Greenwich. Partout ailleurs, PostgreSQL™ suit la convention ISO-8601 pour qui les décalages positifs de fuseaux horaires concernent l'*est* de Greenwich.

Dans tous les cas, les noms des fuseaux horaires sont insensibles à la casse. (C'est un changement par rapport aux versions de PostgreSQL™ antérieures à la 8.2 qui étaient sensibles à la casse dans certains cas et pas dans d'autres.)

Ni les noms complets ni les abréviations ne sont codés en dur dans le serveur ; ils sont obtenus à partir des fichiers de configuration stockés sous `.../share/timezone/` et `.../share/timezonesets/` du répertoire d'installation (voir Section B.3, « Fichiers de configuration date/heure »).

Le paramètre de configuration timezone peut être fixé dans le fichier `postgresql.conf` ou par tout autre moyen standard décrit dans le Chapitre 18, Configuration du serveur. Il existe aussi quelques manières spéciales de le configurer :

- si `timezone` n'est précisé ni dans `postgresql.conf` ni comme une option en ligne de commande du serveur, le serveur tente d'utiliser la valeur de la variable d'environnement `TZ` comme fuseau horaire par défaut. Si `TZ` n'est pas définie ou ne fait pas partie des noms de fuseau horaire connus par PostgreSQL™, le serveur tente de déterminer le fuseau horaire par défaut du système d'exploitation en vérifiant le comportement de la fonction `localtime()` de la bibliothèque C. Le fuseau horaire par défaut est sélectionné comme la correspondance la plus proche parmi les fuseaux horaires connus par PostgreSQL™ ; (Ces règles sont aussi utilisées pour choisir la valeur par défaut de log_timezone, si elle n'est pas précisée.)
- la commande SQL **SET TIME ZONE** configure le fuseau horaire pour une session. C'est une autre façon d'indiquer **SET TIMEZONE TO** avec une syntaxe plus compatible avec les spécifications SQL ;

- la variable d'environnement PGTZ est utilisée par les applications clientes fondées sur libpq pour envoyer une commande **SET TIME ZONE** au serveur lors de la connexion.

8.5.4. Saisie d'intervalle

Les valeurs de type interval peuvent être saisies en utilisant la syntaxe verbeuse suivante :

```
[@] quantité
unité [quantité
unité...]
[direction]
```

où *quantité* est un nombre (éventuellement signé) ; *unité* est microsecond millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, ou des abréviations ou pluriels de ces unités ; *direction* peut être ago (pour indiquer un intervalle négatif) ou vide. Le signe @ est du bruit optionnel. Les quantités de chaque unité différente sont implicitement ajoutées, avec prise en compte appropriée des signes (+ et -). ago inverse tous les champs. Cette syntaxe est aussi utilisée pour les sorties d'intervalles, si IntervalStyle est positionné à postgres_verbose.

Les quantités de jours, heures, minutes et secondes peuvent être spécifiées sans notations explicites d'unités. Par exemple '1 12:59:10' est comprise comme '1 day 12 hours 59 min 10 sec'. Par ailleurs, une combinaison d'années et de mois peut être spécifiée avec un tiret ; par exemple, '200-10' est compris comme '200 years 10 months'. (Ces formes raccourcies sont en fait les seules autorisées par le standard SQL, et sont utilisées pour la sortie quand la variable IntervalStyle est positionnée à sql_standard.)

Les valeurs d'intervalles peuvent aussi être écrites en tant qu'intervalles de temps ISO 8601, en utilisant soit le « format avec désignateurs » de la section 4.4.3.2 ou le « format alternatif » de la section 4.4.3.3. Le format avec désignateurs ressemble à ceci :

```
P quantité unité [ quantité unité ...] [ T [ quantité unité ...]]
```

La chaîne doit commencer avec un P, et peut inclure un T qui introduit les unités de ce type. Les abréviations d'unité disponibles sont données dans Tableau 8.16, « Abréviations d'unités d'intervalle ISO 8601 ». Des unités peuvent être omises, et peuvent être spécifiées dans n'importe quel ordre, mais les unités inférieures à 1 jour doivent apparaître après T. En particulier, la signification de M dépend de son emplacement, c'est-à-dire avant ou après T.

Tableau 8.16. Abréviations d'unités d'intervalle ISO 8601

Abréviation	Signification
Y	Années
M	Mois (dans la zone de date)
W	Semaines
D	Jours
H	Heures
M	Minutes (dans la zone de temps)
S	Secondes

Dans le format alternatif :

```
P [ années-mois-jours ] [ T heures:minutes:secondes ]
```

la chaîne doit commencer par P, et un T sépare la zone de date et la zone de temps de l'intervalle. Les valeurs sont données comme des nombres, de façon similaire aux dates ISO 8601.

Lors de l'écriture d'une constante d'intervalle avec une spécification de *champs*, ou lors de l'assignation d'une chaîne à une colonne d'intervalle qui a été définie avec une spécification de *champs*, l'interprétation de quantité sans unité dépend des *champs*. Par exemple, INTERVAL '1' YEAR est interprété comme 1 an, alors que INTERVAL '1' est interprété comme 1 seconde. De plus, les valeurs du champ « à droite » du champ le moins significatif autorisé par la spécification de *champs* sont annulées de façon silencieuse. Par exemple, écrire INTERVAL '1 day 2:03:04' HOUR TO MINUTE implique la suppression du champ des secondes, mais pas celui des journées.

D'après le standard SQL toutes les valeurs de tous les champs d'un intervalle doivent avoir le même signe, ce qui entraîne qu'un signe négatif initial s'applique à tous les champs ; par exemple, le signe négatif dans l'expression d'intervalle '-1 2:03:04'

s'applique à la fois aux jours et aux heures/minutes/secondes. PostgreSQL™ permet que les champs aient des signes différents, et traditionnellement traite chaque champ de la représentation textuelle comme indépendamment signé, ce qui fait que la partie heure/minute/seconde est considérée comme positive dans l'exemple. Si `IntervalStyle` est positionné à `sql_standard`, alors un signe initial est considéré comme s'appliquant à tous les champs (mais seulement si aucun autre signe n'apparaît). Sinon, l'interprétation traditionnnelle de PostgreSQL™ est utilisée. Pour éviter les ambiguïtés, il est recommandé d'attacher un signe explicite à chaque partie, si au moins un champ est négatif.

De façon interne, les valeurs de type interval sont stockées comme mois, jours et secondes. C'est ainsi parce que le nombre de jours d'un mois varie, et un jour peut avoir 23 ou 25 heures si des changements d'heures sont impliqués. Les champs mois et jours sont des entiers, alors que le champ secondes peut stocker des nombres décimaux. Les intervalles étant habituellement créés à partir de chaînes constantes ou de soustractions de timestamps, cette méthode fonctionne bien dans la plupart des cas. Les fonctions `justify_days` et `justify_hours` sont disponibles pour ajuster les jours et heures qui dépassent leurs portées habituelles.

Dans le format verbeux de saisie, et dans certains champs des formats plus compacts, les valeurs de champs peuvent avoir des parties décimales ; par exemple, `'1.5 week'` ou `'01:02:03.45'`. Ces entrées sont converties en un nombre approprié de mois, jours et secondes pour être stockées. Quand ceci entraînerait le stockage d'une valeur décimale pour les mois ou les jours, la partie décimale est ajoutée aux champs d'ordre inférieur en utilisant les facteurs de conversion suivants : 1 mois = 30 jours, 1 jour = 24heures. Par exemple, `'1.5 month'` devient 1 mois et 15 jours. Seules les secondes pourront apparaître comme décimales en sortie.

Tableau 8.17, « Saisie d'intervalle » présente des exemples de saisies d'interval valides.

Tableau 8.17. Saisie d'intervalle

Exemple	Description
1-2	Format SQL standard : 1 an 2 mois
3 4:05:06	Format SQL standard : 3 jours 4 heures 5 minutes 6 secondes
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Format PostgreSQL traditionnel : 1 an 2 mois 3 jours 4 heures 5 minutes 6 secondes
P1Y2M3DT4H5M6S	« format avec désignateurs » ISO 8601 : signification identique à ci-dessus
P0001-02-03T04:05:06	« format alternatif » ISO 8601 : signification identique à ci-dessus

8.5.5. Affichage d'intervalles

Le format de sortie du type interval peut être positionné à une de ces quatre valeurs : `sql_standard`, `postgres`, `postgres_verbose` ou `iso_8601`, en utilisant la commande `SET intervalstyle`. La valeur par défaut est le format `postgres`. Tableau 8.18, « Exemples de styles d'affichage d'intervalles » donne des exemples de chaque style de format de sortie.

Le style `sql_standard` produit une sortie qui se conforme à la spécification du standard SQL pour les chaînes littérales d'intervalle, si la valeur de l'intervalle reste dans les restrictions du standard (soit année-mois seul, ou jour-temps seul, et sans mélanger les composants positifs et négatifs). Sinon, la sortie ressemble au standard littéral année-mois suivi par une chaîne jour-temps littérale, avec des signes explicites ajoutés pour désambiguer les intervalles dont les signes seraient mélangés.

La sortie du style `postgres` correspond à la sortie des versions de PostgreSQL™ précédant la 8.4, si le paramètre datestyle était positionné à `ISO`.

La sortie du style `postgres_verbose` correspond à la sortie des versions de PostgreSQL™ précédant la 8.4, si le paramètre datestyle était positionné à autre chose que `ISO`.

La sortie du style `iso_8601` correspond au « format avec designateurs » décrit dans la section 4.4.3.2 du standard ISO 8601.

Tableau 8.18. Exemples de styles d'affichage d'intervalles

Spécification de style	Intervalle année-mois	Intervalle date-temps	Interval Mixte
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06
`postgres`	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
`postgres_verbose`	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago

Spécification de style	Intervalle année-mois	Intervalle date-temps	Interval Mixte
`iso_8601`	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.5.6. Types internes

PostgreSQL™ utilise les dates du calendrier Julien pour tous les calculs de date/heure. Elles ont la propriété intéressante de permettre le calcul de toute date depuis 4713 avant Jésus Christ jusque loin dans le futur, avec pour seule hypothèse que l'année dure 365,2425 jours.

Les conventions pour les dates antérieures au 19ème siècle sont d'une lecture intéressante mais ne sont pas assez cohérentes pour être codées dans un gestionnaire de dates.

8.6. Type booléen

PostgreSQL™ fournit le type boolean du standard SQL ; voir Tableau 8.19, « Type de données booléen ». Ce type ne dispose que deux états :« true » (vrai) et « false » (faux). Un troisième état, « unknown » (inconnu), est représenté par la valeur SQL NULL.

Tableau 8.19. Type de données booléen

Nom	Taille du stockage	Description
boolean	1 octet	état vrai ou faux

Les libellés valides pour l'état « vrai » sont :

```
TRUE
't'
'true'
'y'
'yes'
'on'
'1'
```

Pour l'état « faux », il s'agit de :

```
FALSE
'f'
'false'
'n'
'no'
'off'
'0'
```

Les espaces avant ou après, ainsi que la casse, sont ignorés. Il est recommandé d'utiliser `TRUE` et `FALSE` (qui sont compatibles avec la norme SQL).

L'Exemple 8.2, « Utilisation du type boolean. » montre que les valeurs booléennes sont affichées avec les lettres `t` et `f`.

Exemple 8.2. Utilisation du type boolean.

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

8.7. Types énumération

Les types énumérés (enum) sont des types de données qui comprennent un ensemble statique, prédéfini de valeurs dans un ordre spécifique. Ils sont équivalents aux types enum dans de nombreux langages de programmation. Les jours de la semaine ou un ensemble de valeurs de statut pour un type de données sont de bons exemples de type enum.

8.7.1. Déclaration de types énumérés

Les types enum sont créés en utilisant la commande CREATE TYPE(7). Par exemple :

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Une fois créé, le type enum peut être utilisé dans des définitions de table et de fonction, comme tous les autres types :

```
CREATE TYPE humeur AS ENUM ('triste', 'ok', 'heureux');
CREATE TABLE personne (
    nom text,
    humeur_actuelle humeur
);
INSERT INTO personne VALUES ('Moe', 'heureux');
SELECT * FROM personne WHERE humeur_actuelle = 'heureux';
 name | humeur_actuelle
------+-----------------
 Moe  | heureux
(1 row)
```

8.7.2. Tri

L'ordre des valeurs dans un type enum correspond à l'ordre dans lequel les valeurs sont créées lors de la déclaration du type. Tous les opérateurs de comparaison et les fonctions d'agrégats relatives peuvent être utilisés avec des types enum. Par exemple :

```
INSERT INTO personne VALUES ('Larry', 'triste');
INSERT INTO personne VALUES ('Curly', 'heureux');
SELECT * FROM personne WHERE humeur_actuelle > 'triste';
 nom   | humeur_actuelle
-------+-----------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM personne WHERE humeur_actuelle > 'triste' ORDER BY humeur_actuelle;
 nom   | humeur_actuelle
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT nom
FROM personne
WHERE humeur_actuelle = (SELECT MIN(humeur_actuelle) FROM personne);
 nom
-------
 Larry
(1 row)
```

8.7.3. Surêté du type

Chaque type de données énuméré est séparé et ne peut pas être comparé aux autres types énumérés. Par exemple :

```
CREATE TYPE niveau_de_joie AS ENUM ('heureux', 'très heureux', 'ecstatique');
CREATE TABLE vacances (
    nombre_de_semaines integer,
    niveau_de_joie niveau_de_joie
);
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (4, 'heureux');
```

```
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (6, 'très heureux');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (8, 'ecstatique');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (2, 'triste');
ERROR:  invalid input value for enum niveau_de_joie: "triste"
SELECT personne.nom, vacances.nombre_de_semaines FROM personne, vacances
  WHERE personne.humeur_actuelle = vacances.niveau_de_joie;
ERROR:  operator does not exist: humeur = niveau_de_joie
```

Si vous avez vraiment besoin de ce type de conversion, vous pouvez soit écrire un opérateur personnalisé soit ajouter des conversions explicites dans votre requête :

```
SELECT personne.nom, vacances.nombre_de_semaines FROM personne, vacances
  WHERE personne.humeur_actuelle::text = vacances.niveau_de_joie::text;
 nom  | nombre_de_semaines
------+--------------------
 Moe  |          4
(1 row)
```

8.7.4. Détails d'implémentation

Une valeur enum occupe quatre octets sur disque. La longueur du label texte d'une valeur enum est limité au paramètre NAMEDATALEN codé en dur dans PostgreSQL™ ; dans les constructions standards, cela signifie un maximum de 63 octets.

Les labels enum sont sensibles à la casse, donc 'heureux' n'est pas identique à 'HEUREUX'. Les espaces blancs sont significatifs dans les labels.

Les traductions des valeurs enum internes vers des labels texte sont gardées dans le catalogue système pg_enum. Interroger ce catalogue directement peut s'avérer utile.

8.8. Types géométriques

Les types de données géométriques représentent des objets à deux dimensions. Le Tableau 8.20, « Types géométriques » liste les types disponibles dans PostgreSQL™. Le type le plus fondamental, le point, est à la base de tous les autres types.

Tableau 8.20. Types géométriques

Nom	Taille de stockage	Représentation	Description
point	16 octets	Point du plan	(x,y)
line	32 octets	Ligne infinie (pas entièrement implanté)	((x1,y1),(x2,y2))
lseg	32 octets	Segment de droite fini	((x1,y1),(x2,y2))
box	32 octets	Boîte rectangulaire	((x1,y1),(x2,y2))
path	16+16n octets	Chemin fermé (similaire à un polygone)	((x1,y1),...)
path	16+16n octets	Chemin ouvert	[(x1,y1),...]
polygon	40+16n octets	Polygone (similaire à un chemin fermé)	((x1,y1),...)
circle	24 octets	Cercle	<(x,y),r> (point central et rayon)

Un large ensemble de fonctions et d'opérateurs permettent d'effectuer différentes opérations géométriques, comme l'échelonnage, la translation, la rotation, la détermination des intersections. Elles sont expliquées dans la Section 9.11, « Fonctions et opérateurs géométriques ».

8.8.1. Points

Les points sont les briques fondamentales des types géométriques. Les valeurs de type point sont indiquées à l'aide d'une des syntaxes suivantes :

```
( x , y )
  x , y
```

où *x* et *y* sont les coordonnées respectives sous forme de nombre à virgule flottante.

Les points sont affichés en utilisant la première syntaxe.

8.8.2. Segments de droite

Les segments de droite (lseg) sont représentés sous la forme de paires de points à l'aide d'une des syntaxes suivantes :

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont les points aux extrémités du segment.

Les segments de ligne sont affichés en utilisant la première syntaxe.

8.8.3. Boîtes

Les boîtes (rectangles) sont représentées par les paires de points des coins opposés de la boîte selon une des syntaxes suivantes :

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont les coins opposés du rectangle.

Les rectangles sont affichés selon la deuxième syntaxe.

Les deux coins opposés peuvent être fournis en entrée mais les valeurs seront ré-ordonnés pour stocker les coins en haut à droite et en bas à gauche, dans cet ordre.

8.8.4. Chemins

Les chemins (type path) sont représentés par des listes de points connectés. Ils peuvent être *ouverts*, si le premier et le dernier point ne sont pas considérés connectés, ou *fermés*, si le premier et le dernier point sont considérés connectés.

Les valeurs de type path sont saisies selon une des syntaxes suivantes :

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

où les points sont les extrémités des segments de droite qui forment le chemin. Les crochets (`[]`) indiquent un chemin ouvert alors que les parenthèses (`()`) indiquent un chemin fermé. Quand les parenthèses externes sont omises, comme dans les syntaxes trois à cinq, un chemin fermé est utilisé.

Les chemins sont affichés selon la première ou la seconde syntaxe appropriée.

8.8.5. Polygones

Les polygones (type polygon) sont représentés par des listes de points (les vertex du polygone). Ils sont très similaires à des chemins fermés, mais ils sont stockés différemment et disposent de leurs propres routines de manipulation.

Les valeurs de type polygon sont saisies selon une des syntaxes suivantes :

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

où les points sont les extrémités des segments de droite qui forment les limites du polygone.

Les polygones sont affichés selon la première syntaxe.

8.8.6. Cercles

Les cercles (type circle) sont représentés par un point central et un rayon. Les valeurs de type circle sont saisies selon une des syntaxes suivantes :

```
< ( x , y ) , r >
```

```
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

où `(x,y)` est le point central et `r` le rayon du cercle.

Les cercles sont affichés selon la première syntaxe.

8.9. Types adresses réseau

PostgreSQL™ propose des types de données pour stocker des adresses IPv4, IPv6 et MAC. Ceux-ci sont décrits dans le Tableau 8.21, « Types d'adresses réseau ». Il est préférable d'utiliser ces types plutôt que des types texte standard pour stocker les adresses réseau car ils offrent un contrôle de syntaxe lors de la saisie et plusieurs opérateurs et fonctions spécialisées (voir la Section 9.12, « Fonctions et opérateurs sur les adresses réseau »).

Tableau 8.21. Types d'adresses réseau

Nom	Taille de stockage	Description
cidr	7 ou 19 octets	réseaux IPv4 et IPv6
inet	7 ou 19 octets	hôtes et réseaux IPv4 et IPv6
macaddr	6 octets	adresses MAC

Lors du tri de données de types inet ou cidr, les adresses IPv4 apparaissent toujours avant les adresses IPv6, y compris les adresses IPv4 encapsulées, comme ::10.2.3.4 ou ::ffff:10.4.3.2.

8.9.1. inet

Le type inet stocke une adresse d'hôte IPv4 ou IPv6 et, optionnellement, son sous-réseau, le tout dans un seul champ. Le sous-réseau est représentée par le nombre de bits de l'adresse hôte constituent l'adresse réseau (le « masque réseau »). Si le masque réseau est 32 et l'adresse de type IPv4, alors la valeur n'indique pas un sous-réseau, juste un hôte. En IPv6, la longueur de l'adresse est de 128 bits, si bien que 128 bits définissent une adresse réseau unique. Pour n'accepter que des adresses réseau, il est préférable d'utiliser le type cidr plutôt que le type inet.

Le format de saisie pour ce type est `adresse/y` où `adresse` est une adresse IPv4 ou IPv6 et `y` est le nombre de bits du masque réseau. Si `y` est omis, alors le masque vaut 32 pour IPv4 et 128 pour IPv6, et la valeur représente un hôte unique. À l'affichage, la portion `/y` est supprimée si le masque réseau indique un hôte unique.

8.9.2. cidr

Le type cidr stocke une définition de réseau IPv4 ou IPv6. La saisie et l'affichage suivent les conventions Classless Internet Domain Routing. Le format de saisie d'un réseau est `address/y` où `address` est le réseau représenté sous forme d'une adresse IPv4 ou IPv6 et `y` est le nombre de bits du masque réseau. Si `y` est omis, il calculé en utilisant les règles de l'ancien système de classes d'adresses, à ceci près qu'il est au moins assez grand pour inclure tous les octets saisis. Saisir une adresse réseau avec des bits positionnés à droite du masque indiqué est une erreur.

Tableau 8.22, « Exemples de saisie de types cidr » présente quelques exemples.

Tableau 8.22. Exemples de saisie de types cidr

Saisie cidr	Affichage cidr	`abbrev(cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24

Saisie cidr	Affichage cidr	abbrev(cidr)
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inet vs cidr

La différence principale entre les types de données inet et cidr réside dans le fait que inet accepte des valeurs avec des bits non nuls à droite du masque de réseau, alors que cidr ne l'accepte pas.

Astuce

Les fonctions `host`, `text` et `abbrev` permettent de modifier le format d'affichage des valeurs inet et cidr.

8.9.4. macaddr

Le type macaddr stocke des adresses MAC, connues par exemple à partir des adresses de cartes réseau Ethernet (mais les adresses MAC sont aussi utilisées dans d'autres cas). Les saisies sont acceptées dans les formats suivants :

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08002b010203'
```

Ces exemples indiquent tous la même adresse. Les majuscules et les minuscules sont acceptées pour les chiffres `a` à `f`. L'affichage se fait toujours selon le premier des formats ci-dessus.

Le standard IEEE 802-2001 spécifie la seconde forme affichée (avec les tirets) comme forme canonique pour les adresses MAC, et que la première forme (avec les :) est la notation à bits retournés, ce qui donne l'équivalence 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. Cette convention est largement ignorée aujourd'hui, et n'a de sens que pour des protocoles réseaux obsolètes (comme Token Ring). PostgreSQL ne tient pas compte des bits retournés, et tous les formats acceptés utilisent l'ordre canonique LSB.

Les quatre derniers formats ne font partie d'aucun standard.

8.10. Type chaîne de bits

Les chaînes de bits sont des chaînes de 0 et de 1. Elles peuvent être utilisées pour stocker ou visualiser des masques de bits. Il y a deux types bits en SQL : bit(*n*) et bit varying(*n*), avec *n* un entier positif.

Les données de type bit doivent avoir une longueur de *n* bits exactement. Essayer de lui affecter une chaîne de bits plus longue ou plus courte déclenche une erreur. Les données de type bit varying ont une longueur variable, d'au maximum *n* bits ; les chaînes plus longues sont rejetées. Écrire bit sans longueur est équivalent à `bit(1)`, alors que bit varying sans longueur indique une taille illimitée.

Note

Lors du transtypage explicite (cast) d'une chaîne de bits en champ de type bit(*n*), la chaîne obtenue est complétée avec des zéros ou bien tronquée pour obtenir une taille de *n* bits exactement, sans que cela produise une erreur. De la même façon, si une chaîne de bits est explicitement transtypée en un champ de type bit varying(*n*), elle est tronquée si sa longueur dépasse *n* bits.

Voir la Section 4.1.2.5, « Constantes de chaînes de bits » pour plus d'information sur la syntaxe des constantes en chaîne de bits. Les opérateurs logiques et les fonctions de manipulation de chaînes sont décrits dans la Section 9.6, « Fonctions et opérateurs sur les chaînes de bits ».

Exemple 8.3. Utiliser les types de chaînes de bits

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Une valeur pour une chaîne de bit nécessite un octet pour chaque groupe de huit bits, plus cinq ou huit octets d'en-tête suivant la longueur de la chaîne (les valeurs longues peuvent être compressées ou déplacées, comme expliqué dans Section 8.3, « Types caractère » pour les chaînes de caractères).

8.11. Types de recherche plein texte

PostgreSQL™ fournit deux types de données conçus pour supporter la recherche plein texte qui est l'activité de recherche via une collection de *documents* en langage naturel pour situer ceux qui correspondent le mieux à une *requête*. Le type tsvector représente un document dans une forme optimisée pour la recherche plein texte alors que le type tsquery représente de façon similaire une requête. Chapitre 12, Recherche plein texte fournit une explication détaillée de cette capacité et Section 9.13, « Fonctions et opérateurs de la recherche plein texte » résumé les fonctions et opérateurs en relation.

8.11.1. tsvector

Une valeur tsvector est une liste triée de *lexemes* distincts, qui sont des mots qui ont été *normalisés* pour fusionner différentes variantes du même mot apparaissent (voir Chapitre 12, Recherche plein texte pour plus de détails). Trier et éliminer les duplicats se font automatiquement lors des entrées, comme indiqué dans cet exemple :

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
-----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Pour représenter des lexemes contenant des espaces blancs ou des signes de ponctuation, entourez-les avec des guillemets simples :

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                  tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(Nous utilisons les valeurs litérales entre guillemets simples dans cet exemple et dans le prochain pour éviter une confusion en ayant à doubler les guillemets à l'intérieur des valeurs litérales.) Les guillemets imbriqués et les antislashs doivent être doublés :

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                     tsvector
-----------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

En option, les *positions* peuvent être attachées aux lexemes :

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                   tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

Une position indique normalement l'emplacement du mot source dans le document. Les informations de position sont utilisables

pour avoir un *score de proximité*. Les valeurs des positions peuvent aller de 1 à 16383 ; les grands nombres sont limités silencieusement à 16383. Les positions dupliquée du même lexeme sont rejetées.

Les lexemes qui ont des positions peuvent aussi avoir un label d'un certain *poids*. Les labels possibles sont `A`, `B`, `C` ou `D`. `D` est la valeur par défaut et n'est du coup pas affiché en sortie :

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Les poids sont typiquement utilisés pour refléter la structure du document en marquant les mots du titre de façon différente des mots du corps. Les fonctions de score de la recherche plein texte peuvent assigner des priorités différentes aux marqueurs de poids différents.

Il est important de comprendre que le type tsvector lui-même ne réalise aucune normalisation ; il suppose que les mots qui lui sont fournis sont normalisés correctement pour l'application. Par exemple,

```
select 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

Pour la plupart des applications de recherche en anglais, les mots ci-dessus seraient considérés comme non normalisés mais tsvector n'y prête pas attention. Le texte des documents bruts doit habituellement passer via `to_tsvector` pour normaliser les mots de façon appropriée pour la recherche :

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

De nouveau, voir Chapitre 12, Recherche plein texte pour plus de détails.

8.11.2. tsquery

Une valeur tsquery enregistre les lexemes qui doivent être recherchés et les combine en utilisant les opérateurs booléens `&` (AND), `|` (OR) et `!` (NOT). Les parenthèses peuvent être utilisées pour forcer le regroupement des opérateurs :

```
 SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
----------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

En l'absence de ces parenthèses, `!` (NOT) est lié plus fortement, et `&` (AND) est lié plus fortement que `|` (OR).

En option, les lexemes dans une tsquery peuvent être labelisés avec une lettre de poids ou plus, ce qui les restreint à une correspondance avec les seuls lexemes tsvector pour un de ces poids :

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Par ailleurs, les lexemes d'une tsquery peuvent être marqués avec `*` pour spécifier une correspondance de préfixe :

```
SELECT 'super:*'::tsquery;
```

```
  tsquery
-----------
 'super':*
```

Cette requête fera ressortir tout mot dans un tsvector qui commence par « super ».

Les règles de guillemets pour les lexemes sont identiques à celles décrites ci-dessus pour les lexemes de tsvector ; et, comme avec tsvector, toute normalisation requise des mots doit se faire avant de les placer dans le type tsquery. La fonction `to_tsquery` est convenable pour réaliser une telle normalisation :

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

8.12. Type UUID

Le type de données uuid stocke des identifiants universels uniques (UUID, acronyme de *Universally Unique Identifiers*) décrits dans les standards RFC 4122, ISO/IEC 9834-8:2005, et d'autres encore. (Certains systèmes font référence à ce type de données en tant qu'identifiant unique global (ou GUID).) Un identifiant de ce type est une quantité sur 128 bits généré par un algorithme adéquat qui a peu de chances d'être reproduit par quelqu'un d'autre utilisant le même algorithme. Du coup, pour les systèmes distribués, ces identifiants fournissent une meilleure garantie d'unicité que ce que pourrait fournir une séquence, dont la valeur est unique seulement au sein d'une base de données.

Un UUID est écrit comme une séquence de chiffres hexadécimaux en minuscule, répartis en différents groupes séparés par un tiret. Plus précisément, il s'agit d'un groupe de huit chiffres suivis de trois groupes de quatre chiffres terminés par un groupe de douze chiffres, ce qui fait un total de 32 chiffres représentant les 128 bits. Voici un exemple d'UUID dans sa forme standard :

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL accepte aussi d'autres formes en entrée : utilisation des majuscules, de crochets englobant le nombre, suppression d'une partie ou de tous les tirets, ajout d'un tiret après n'importe quel groupe de quatre chiffres. Voici quelques exemples :

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

L'affichage est toujours dans la forme standard.

Pour générer des UUID, le module contrib `uuid-ossp` fournit des fonctions qui implémentent les algorithmes standards. Sinon, les UUID peuvent être générés par des applications clientes ou par d'autres bibliothèques appelées par une fonction serveur.

8.13. Type XML

Le type de données xml est utilisé pour stocker des données au format XML. Son avantage sur un champ de type text est qu'il vérifie que les valeurs sont bien formées. De plus, il existe de nombreuses fonctions pour réaliser des opérations de vérification à partir de ce type ; voir la Section 9.14, « Fonctions XML ». L'utilisation de ce type de données requiert que l'étape de compilation a utilisé l'option `--with-libxml`.

Le type xml peut stocker des « documents » bien formés, suivant la définition du standard XML, ainsi que des fragments de contenu (« content »), qui sont définis par `XMLDecl? content` du standard XML. Cela signifie que les fragments de contenu peuvent avoir plus d'un élément racine ou nœud caractère. L'expression *`valeurxml`* `IS DOCUMENT` permet d'évaluer si une valeur xml particulière est un document complet ou seulement un fragment de contenu.

8.13.1. Créer des valeurs XML

Pour produire une valeur de type xml à partir d'une donnée de type caractère, utilisez la fonction `xmlparse` :

```
XMLPARSE ( { DOCUMENT | CONTENT } valeur)
```

Quelques exemples :

```
XMLPARSE (DOCUMENT '<?xml
version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Bien que cela soit la seule façon de convertir des chaînes de caractère en valeurs XML d'après le standard XML, voici des syntaxes spécifiques à PostgreSQL :

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

Le type xml ne valide pas les valeurs en entrée par rapport à une déclaration de type de document (DTD), même quand la valeur en entrée indique une DTD. Il n'existe pas encore de support pour la validation avec d'autres langages de schéma XML, comme XML Schema.

L'opération inverse, produisant une chaîne de caractères à partir d'une valeur au type xml, utilise la fonction `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

`type` peut être character, character varying ou text (ou un alias de ces derniers). Encore une fois, d'après le standard SQL, c'est le seul moyen de convertir le type xml vers les types caractère mais PostgreSQL autorise aussi la conversion simple de la valeur.

Lorsque les valeurs des chaînes de caractère sont converties vers ou à partir du type xml sans passer par XMLPARSE ou XMLSERIALIZE, respectivement, le choix de `DOCUMENT` ou de `CONTENT` est déterminé par un paramètre de configuration niveau session, « XML OPTION » , qui peut être configuré par la commande habituelle :

```
SET XML OPTION { DOCUMENT | CONTENT };
```

ou la syntaxe PostgreSQL :

```
SET xmloption TO { DOCUMENT | CONTENT };
```

La valeur par défaut est `CONTENT`, donc toutes les formes de données XML sont autorisées.

Note

Avec le paramètrage par défaut des options XML, vous ne pouvez pas convertir directement des chaînes de caractères dans le type xml si elles contiennent une déclaration du type de document car la définition du fragment de contenu XML ne les accepte pas. Si vous avez besoin de changer cela, soit vous utilisez `XMLPARSE` soit vous changez l'option XML.

8.13.2. Gestion de l'encodage

Une grande attention doit prévaloir lors de la gestion de plusieurs encodages sur le client, le serveur ou dans les données XML qui passent entre eux. Lors de l'utilisation du mode texte pour passer les requêtes au serveur et pour renvoyer les résultats au client (qui se trouve dans le mode normal), PostgreSQL convertit toutes les données de type caractère passées entre le client et le serveur et vice-versa suivant l'encodage spécifique du bout final ; voir la Section 22.2, « Support des jeux de caractères ». Ceci inclut les représentations textuelles des valeurs XML, comme dans les exemples ci-dessus. Ceci signifie que les déclarations d'encodage contenues dans les données XML pourraient devenir invalide lorsque les données sont converties vers un autre encodage lors du transfert entre le client et le serveur, alors que la déclaration de l'encodage n'est pas modifiée. Pour s'en sortir, une déclaration d'encodage contenue dans une chaîne de caractères présentée en entrée du type xml est *ignorée*, et le contenu est toujours supposé être de l'encodage du serveur. En conséquence, pour un traitement correct, ces chaînes de caractères de données XML doivent être envoyées du client dans le bon encodage. C'est de la responsabilité du client de soit convertir le document avec le bon encodage client avant de l'envoyer au serveur soit d'ajuster l'encodage client de façon appropriée. En sortie, les valeurs du type xml n'auront pas une déclaration d'encodage et les clients devront supposer que les données sont dans l'encodage du client.

Lors de l'utilisation du mode binaire pour le passage des paramètres de la requête au serveur et des résultats au client, aucune conversion de jeu de caractères n'est réalisée, donc la situation est différente. Dans ce cas, une déclaration d'encodage dans les données XML sera observée et, si elle est absente, les données seront supposées être en UTF-8 (comme requis par le standard XML ; notez que PostgreSQL ne supporte pas du tout UTF-16). En sortie, les données auront une déclaration d'encodage spécifiant l'encodage client sauf si l'encodage client est UTF-8, auquel case elle sera omise.

Le traitement des données XML avec PostgreSQL sera moins complexe et plus efficace si l'encodage des données, l'encodage client et serveur sont identiques. Comme les données XML sont traitées en interne en UTF-8, les traitements seront plus efficaces si l'encodage serveur est aussi en UTF-8.

> **Attention**
>
> Certaines fonctions relatives à XML pourraient ne pas fonctionner du tout sur des données non ASCII quand l'encodage du serveur n'est pas UTF-8. C'est un problème connu pour `xpath()` en particulier.

8.13.3. Accéder aux valeurs XML

Le type de données xml est inhabituel dans le sens où il ne dispose pas d'opérateurs de comparaison. Ceci est dû au fait qu'il n'existe pas d'algorithme de comparaison bien défini et utile pour des données XML. Une conséquence de ceci est que vous ne pouvez pas récupérer des lignes en comparant une colonne xml avec une valeur de recherche. Les valeurs XML doivent du coup être typiquement accompagnées par un champ clé séparé comme un identifiant. Une autre solution pour la comparaison de valeurs XML est de les convertir en des chaînes de caractères, mais notez que la comparaison de chaînes n'a que peu à voir avec une méthode de comparaison XML utile.

Comme il n'y a pas d'opérateurs de comparaison pour le type de données xml, il n'est pas possible de créer un index directement sur une colonne de ce type. Si une recherche rapide est souhaitée dans des données XML, il est toujours possible de convertir l'expression en une chaîne de caractères et d'indexer cette conversion. Il est aussi possible d'indexer une expression XPath. La vraie requête devra bien sûr être ajustée à une recherche sur l'expression indexée.

La fonctionnalité de recherche plein texte peut aussi être utilisée pour accélérer les recherches dans des données XML. Le support du pré-traitement nécessaire n'est cependant pas disponible dans la distribution PostgreSQL.

8.14. Tableaux

PostgreSQL™ permet de définir des colonnes de table comme des tableaux multidimensionnels de longueur variable. Il est possible de créer des tableaux de n'importe quel type utilisateur : de base, composé, enum. Toutefois, les tableaux de domaines ne sont pas encore supportés.

8.14.1. Déclaration des types tableaux

La création de la table suivante permet d'illustrer l'utilisation des types tableaux :

```
CREATE TABLE sal_emp (
    nom              text,
    paye_par_semaine integer[],
    planning         text[][]
);
```

Comme indiqué ci-dessus, un type de données tableau est nommé en ajoutant des crochets (`[]`) au type de données des éléments du tableau. La commande ci-dessus crée une table nommée sal_emp avec une colonne de type text (*`nom`*), un tableau à une dimension de type integer (*`paye_par_semaine`*), représentant le salaire d'un employé par semaine et un tableau à deux dimensions de type text (*`planning`*), représentant le planning hebdomadaire de l'employé.

La syntaxe de **CREATE TABLE** permet de préciser la taille exacte des tableaux, par exemple :

```
CREATE TABLE tictactoe (
    carres   integer[3][3]
);
```

Néanmoins, l'implantation actuelle ignore toute limite fournie pour la taille du tableau, c'est-à-dire que le comportement est identique à celui des tableaux dont la longueur n'est pas précisée.

De plus, l'implantation actuelle n'oblige pas non plus à déclarer le nombre de dimensions. Les tableaux d'un type d'élément particulier sont tous considérés comme étant du même type, quels que soient leur taille ou le nombre de dimensions. Déclarer la taille du tableau ou le nombre de dimensions dans **CREATE TABLE** n'a qu'un but documentaire. Le comportement de l'application n'en est pas affecté.

Une autre syntaxe, conforme au standard SQL via l'utilisation du mot clé `ARRAY`, peut être employée pour les tableaux à une dimension. *`paye_par_semaine`* peut être défini ainsi :

```
paye_par_semaine  integer ARRAY[4],
```

ou si aucune taille du tableau n'est spécifiée :

```
    paye_par_semaine  integer ARRAY,
```

Néanmoins, comme indiqué précédemment, PostgreSQL™ n'impose aucune restriction sur la taille dans tous les cas.

8.14.2. Saisie de valeurs de type tableau

Pour écrire une valeur de type tableau comme une constante littérale, on encadre les valeurs des éléments par des accolades et on les sépare par des virgules (ce n'est pas différent de la syntaxe C utilisée pour initialiser les structures). Des guillemets doubles peuvent être positionnés autour des valeurs des éléments. C'est d'ailleurs obligatoire si elles contiennent des virgules ou des accolades (plus de détails ci-dessous). Le format général d'une constante de type tableau est donc le suivant :

```
'{ val1 delim val2 delim ... }'
```

où *delim* est le caractère de délimitation pour ce type, tel qu'il est enregistré dans son entrée `pg_type`. Parmi les types de données standard fournis par la distribution PostgreSQL™, tous utilisent une virgule (`,`), sauf pour le type box qui utilise un point-virgule (`;`). Chaque *val* est soit une constante du type des éléments du tableau soit un sous-tableau.

Exemple de constante tableau :

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Cette constante a deux dimensions, un tableau 3 par 3 consistant en trois sous-tableaux d'entiers.

Pour initialiser un élément d'un tableau à NULL, on écrit `NULL` pour la valeur de cet élément. (Toute variante majuscule et/ou minuscule de `NULL` est acceptée.) Si « NULL » doit être utilisé comme valeur de chaîne, on place des guillemets doubles autour.

Ces types de constantes tableau sont en fait un cas particulier des constantes de type générique abordées dans la Section 4.1.2.7, « Constantes d'autres types ». La constante est traitée initialement comme une chaîne et passée à la routine de conversion d'entrées de tableau. Une spécification explicite du type peut être nécessaire.

Quelques instructions **INSERT** :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"rendez-vous", "repas"}, {"entrainement", "présentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"petit-déjeuner", "consultation"}, {"rendez-vous", "repas"}}');
```

Le résultat des deux insertions précédentes ressemble à :

```
SELECT * FROM sal_emp;
 nom   |      paye_par_semaine     |      planning
-------+---------------------------+--------------------
Bill   | {10000,10000,10000,10000} | {{rendez-vous,repas},{entrainement,présentation}}
Carol  | {20000,25000,25000,25000} |
{{petit-déjeuner,consultation},{rendez-vous,repas}}
(2 rows)
```

Les tableaux multi-dimensionnels doivent avoir des échelles correspondantes pour chaque dimension. Une différence cause la levée d'une erreur. Par exemple :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"rendez-vous", "repas"}, {"rendez-vous"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

La syntaxe du constructeur `ARRAY` peut aussi être utilisée :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['rendez-vous', 'repas'], ['entrainement','présentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['petit-déjeuner', 'consultation'], ['rendez-vous', 'repas']]);
```

Les éléments du tableau sont des constantes SQL ordinaires ou des expressions ; par exemple, les chaînes de caractères littérales

sont encadrées par des guillemets simples au lieu de guillemets doubles comme cela est le cas dans un tableau littéral. La syntaxe du constructeur ARRAY est discutée plus en profondeur dans la Section 4.2.11, « Constructeurs de tableaux ».

8.14.3. Accès aux tableaux

Quelques requêtes lancées sur la table permettent d'éclairer le propos précédent. Tout d'abord, l'accès à un seul élément du tableau. Cette requête retrouve le nom des employés dont la paye a changé au cours de la deuxième semaine :

```
SELECT nom FROM sal_emp WHERE paye_par_semaine[1] <> paye_par_semaine[2];

 nom
-------
 Carol
(1 row)
```

Les indices du tableau sont écrits entre crochets. Par défaut, PostgreSQL™ utilise la convention des indices commençant à 1 pour les tableaux, c'est-à-dire un tableau à *n* éléments commence avec `array[1]` et finit avec `array[n]`.

Récupérer la paye de la troisième semaine de tous les employés :

```
SELECT paye_par_semaine[3] FROM sal_emp;

 paye_par_semaine
------------------
          10000
          25000
(2 rows)
```

Il est également possible d'accéder à des parties rectangulaires arbitraires ou à des sous-tableaux. Une partie de tableau est indiquée par l'écriture *extrémité basse*:*extrémité haute* sur n'importe quelle dimension. Ainsi, la requête suivante retourne le premier élément du planning de Bill pour les deux premiers jours de la semaine :

```
SELECT planning[1:2][1:1] FROM sal_emp WHERE nom = 'Bill';

      planning
--------------------
 {{rendez-vous},{entrainement}}
(1 row)
```

Si l'une des dimensions est écrite comme une partie, c'est-à-dire si elle contient le caractère deux-points, alors toutes les dimensions sont traitées comme des parties. Toute dimension qui n'a qu'un numéro (pas de deux-points), est traitée comme allant de `1` au nombre indiqué. Par exemple, `[2]` est traitée comme `[1:2]`, comme le montre cet exemple :

```
SELECT planning[1:2][2] FROM sal_emp WHERE nom = 'Bill';

         planning
---------------------------
 {{rendez-vous,repas},{entrainement,présentation}}
(1 row)
```

Pour éviter la confusion avec le cas sans indice, il est meilleur d'utiliser la syntaxe avec indice pour toutes les dimensions, c'est-à-dire `[1:2][1:1]` et non pas `[2][1:1]`.

Une expression indicée de tableau retourne NULL si le tableau ou une des expressions est NULL. De plus, NULL est renvoyé si un indice se trouve en dehors de la plage du tableau (ce cas n'amène pas d'erreur). Par exemple, si `planning` a les dimensions `[1:3][1:2]`, faire référence à `planning[3][3]` donne un résultat NULL. De la même façon, une référence sur un tableau avec une valeur d'indices incorrecte retourne une valeur NULL plutôt qu'une erreur.

Une expression de découpage d'un tableau est aussi NULL si, soit le tableau, soit une des expressions indicées est NULL. Néanmoins, dans certains cas particuliers comme la sélection d'une partie d'un tableau complètement en dehors de la plage de ce dernier, l'expression de cette partie est un tableau vide (zéro dimension) et non pas un tableau NULL. (Ceci ne correspond pas au comportement sans indice, et est fait pour des raisons historiques.) Si la partie demandée surcharge partiellement les limites du tableau, alors elle est réduite silencieusement à la partie surchargée au lieu de renvoyer NULL.

Les dimensions actuelles de toute valeur de type tableau sont disponibles avec la fonction `array_dims` :

```
SELECT array_dims(planning) FROM sal_emp WHERE nom = 'Carol';

 array_dims
------------
 [1:2][1:2]
```

```
(1 row)
```

`array_dims` donne un résultat de type text, ce qui est pratique à lire mais peut s'avérer plus difficile à interpréter par les programmes. Les dimensions sont aussi récupérables avec `array_upper` et `array_lower`, qui renvoient respectivement la limite haute et et la limite basse du tableau précisé :

```
SELECT array_upper(planning, 1) FROM sal_emp WHERE nom = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` renverra la longueur de la dimension indiquée pour le tableau :

```
SELECT array_length(planning, 1) FROM sal_emp WHERE nom = 'Carol';

 array_length
--------------
            2
(1 row)
```

8.14.4. Modification de tableaux

La valeur d'un tableau peut être complètement remplacée :

```
UPDATE sal_emp SET paye_par_semaine = '{25000,25000,27000,27000}'
    WHERE nom = 'Carol';
```

ou en utilisant la syntaxe de l'expression ARRAY :

```
UPDATE sal_emp SET paye_par_semaine = ARRAY[25000,25000,27000,27000]
    WHERE nom = 'Carol';
```

On peut aussi mettre à jour un seul élément d'un tableau :

```
UPDATE sal_emp SET paye_par_semaine[4] = 15000
    WHERE nom = 'Bill';
```

ou faire une mise à jour par tranche :

```
UPDATE sal_emp SET paye_par_semaine[1:2] = '{27000,27000}'
    WHERE nom = 'Carol';
```

Un tableau peut être agrandi en y stockant des éléments qui n'y sont pas déjà présents. Toute position entre ceux déjà présents et les nouveaux éléments est remplie avec la valeur NULL. Par exemple, si le tableau `mon_tableau` a actuellement quatre éléments, il en aura six après une mise à jour qui affecte `mon_tableau[6]` car `mon_tableau[5]` est alors rempli avec une valeur NULL. Actuellement, l'agrandissement de cette façon n'est autorisé que pour les tableaux à une dimension, pas pour les tableaux multidimensionnels.

L'affectation par parties d'un tableau permet la création de tableaux dont l'indice de départ n'est pas 1. On peut ainsi affecter, par exemple, `mon_tableau[-2:7]` pour créer un tableau avec des valeurs d'indices allant de -2 à 7.

Les valeurs de nouveaux tableaux peuvent aussi être construites en utilisant l'opérateur de concaténation, `||` :

```
SELECT ARRAY[1,2] || ARRAY[3,4];
   ?column?
---------------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

L'opérateur de concaténation autorise un élément à être placé au début ou à la fin d'un tableau à une dimension. Il accepte aussi deux tableaux à *N* dimensions, ou un tableau à *N* dimensions et un à *N+1* dimensions.

Quand un élément seul est poussé soit au début soit à la fin d'un tableau à une dimension, le résultat est un tableau avec le même

indice bas que l'opérande du tableau. Par exemple :

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

Lorsque deux tableaux ayant un même nombre de dimensions sont concaténés, le résultat conserve la limite inférieure de l'opérande gauche. Le résultat est un tableau comprenant chaque élément de l'opérande gauche suivi de chaque élément de l'opérande droit. Par exemple :

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

Lorsqu'un tableau à *N* dimensions est placé au début ou à la fin d'un tableau à *N+1* dimensions, le résultat est analogue au cas ci-dessus. Chaque sous-tableau de dimension *N* est en quelque sorte un élément de la dimension externe d'un tableau à *N+1* dimensions. Par exemple :

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

Un tableau peut aussi être construit en utilisant les fonctions `array_prepend`, `array_append` ou `array_cat`. Les deux premières ne supportent que les tableaux à une dimension alors que `array_cat` supporte les tableaux multidimensionnels. L'opérateur de concaténation vu plus haut est préférable à l'utilisation directe de ces fonctions. En fait, les fonctions existent principalement pour l'implantation de l'opérateur de concaténation. Néanmoins, elles peuvent être directement utiles dans la création d'agrégats utilisateur. Quelques exemples :

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
   array_cat
---------------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)
```

```
SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

8.14.5. Recherche dans les tableaux

Pour rechercher une valeur dans un tableau, il faut vérifier chaque valeur dans le tableau. Ceci peut se faire à la main lorque la taille du tableau est connue. Par exemple :

```
SELECT * FROM sal_emp WHERE paye_par_semaine[1] = 10000 OR
                            paye_par_semaine[2] = 10000 OR
                            paye_par_semaine[3] = 10000 OR
                            paye_par_semaine[4] = 10000;
```

Ceci devient toutefois rapidement fastidieux pour les gros tableaux et n'est pas très utile si la taille du tableau n'est pas connue. Une autre méthode est décrite dans la Section 9.21, « Comparaisons de lignes et de tableaux ». La requête ci-dessus est remplaçable par :

```
SELECT * FROM sal_emp WHERE 10000 = ANY (paye_par_semaine);
```

De la même façon, on trouve les lignes où le tableau n'a que des valeurs égales à 10000 avec :

```
SELECT * FROM sal_emp WHERE 10000 = ALL (paye_par_semaine);
```

Sinon, la fonction `generate_subscripts` peut être utilisée. Par exemple :

```
SELECT * FROM
   (SELECT paye_par_semaine,
           generate_subscripts(paye_par_semaine, 1) AS s
      FROM sal_emp) AS foo
 WHERE paye_par_semaine[s] = 10000;
```

Cette fonction est décrite dans Tableau 9.46, « Fonctions de génération d'indices ».

Astuce

Les tableaux ne sont pas des ensembles ; rechercher des éléments spécifiques dans un tableau peut être un signe d'une mauvaise conception de la base de données. On utilise plutôt une table séparée avec une ligne pour chaque élément faisant parti du tableau. Cela simplifie la recherche et fonctionne mieux dans le cas d'un grand nombre d'éléments.

8.14.6. Syntaxe d'entrée et de sortie des tableaux

La représentation externe du type texte d'une valeur de type tableau consiste en des éléments interprétés suivant les règles de conversion d'entrées/sorties pour le type de l'élément du tableau, plus des décorations indiquant la structure du tableau. L'affichage est constitué d'accolades (`{` et `}`) autour des valeurs du tableau et de caractères de délimitation entre éléments adjacents. Le caractère délimiteur est habituellement une virgule (`,`) mais peut différer : il est déterminé par le paramètre `typdelim` du type de l'élément tableau. Parmi les types de données standard supportés par l'implantation de PostgreSQL™, seul le type box utilise un point-virgule (`;`), tous les autres utilisant la virgule. Dans un tableau multidimensionnel, chaque dimension (row, plane, cube, etc.) utilise son propre niveau d'accolades et les délimiteurs doivent être utilisés entre des entités adjacentes au sein d'accolades de même niveau.

La routine de sortie du tableau place des guillemets doubles autour des valeurs des éléments si ce sont des chaînes vides, si elles contiennent des accolades, des caractères délimiteurs, des guillemets doubles, des antislashs ou des espaces ou si elles correspondent à `NULL`. Les guillemets doubles et les antislashs intégrés aux valeurs des éléments sont échappés à l'aide d'un antislash. Pour les types de données numériques, on peut supposer sans risque que les doubles guillemets n'apparaissent jamais, mais pour les types de données texte, il faut être préparé à gérer la présence et l'absence de guillemets.

Par défaut, la valeur de la limite basse d'un tableau est initialisée à 1. Pour représenter des tableaux avec des limites basses différentes, les indices du tableau doivent être indiqués explicitement avant d'écrire le contenu du tableau. Cet affichage est consititué de crochets (`[]`) autour de chaque limite basse et haute d'une dimension avec un délimiteur deux-points (`:`) entre les deux. L'affichage des dimensions du tableau est suivie par un signe d'égalité (`=`). Par exemple :

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
```

```
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

La routine de sortie du tableau inclut les dimensions explicites dans le resultat uniquement lorsqu'au moins une limite basse est différente de 1.

Si la valeur écrite pour un élément est `NULL` (toute variante), l'élément est considéré NULL. La présence de guillemets ou d'antislashs désactive ce fonctionnement et autorise la saisie de la valeur litérale de la chaîne « NULL ». De plus, pour une compatibilité ascendante avec les versions antérieures à la version 8.2 de PostgreSQL™, le paramètre de configuration array_nulls doit être désactivé (`off`) pour supprimer la reconnaissance de `NULL` comme un NULL.

Comme indiqué précédemment, lors de l'écriture d'une valeur de tableau, des guillemets doubles peuvent être utilisés autour de chaque élément individuel du tableau. Il *faut* le faire si leur absence autour d'un élément induit en erreur l'analyseur de tableau. Par exemple, les éléments contenant des crochets, virgules (ou tout type de données pour le caractère délimiteur correspondant), guillemets doubles, antislashs ou espace (en début comme en fin) doivent avoir des guillemets doubles. Les chaînes vides et les chaînes `NULL` doivent aussi être entre guillemets. Pour placer un guillemet double ou un antislash dans une valeur d'élément d'un tableau, on utilise la syntaxe d'échappement des chaînes en le précédant d'un antislash. Alternativement, il est possible de se passer de guillemets et d'utiliser l'échappement par antislash pour protéger tous les caractères de données qui seraient autrement interprétés en tant que caractères de syntaxe de tableau.

Des espaces peuvent être ajoutées avant un crochet gauche ou après un crochet droit. Comme avant tout élément individuel. Dans tous ces cas-là, les espaces sont ignorées. En revanche, les espaces à l'intérieur des éléments entre guillemets doubles ou entourées de caractères autres que des espaces ne sont pas ignorées.

Note

Toute ce qui est écrit dans une commande SQL est d'abord interprété en tant que chaîne littérale puis en tant que tableau. Ceci double le nombre d'antislash nécessaire. Par exemple, pour insérer une valeur de tableau de type text contenant un antislash et un guillemet double, il faut écrire :

```
INSERT ... VALUES (E'{"\\\\","\\""}');
```

Le processeur de la chaîne d'échappement supprime un niveau d'antislash, donc l'analyseur de tableau reçoit `{"\\","\""}`. En conséquence, les chaînes remplissant l'entrée du type de données text deviennent respectivement `\` et `"`. (Si la routine d'entrée du type de données utilisé traite aussi les antislash de manière spéciale, bytea par exemple, il peut être nécessaire d'avoir jusqu'à huit antislash dans la commande pour en obtenir un dans l'élément stocké.) Les guillemets dollar (voir Section 4.1.2.4, « Constantes de chaînes avec guillemet dollar ») peuvent être utilisés pour éviter de doubler les antislash.

Astuce

La syntaxe du constructeur `ARRAY` (voir Section 4.2.11, « Constructeurs de tableaux ») est souvent plus facile à utiliser que la syntaxe de tableau littéral lors de l'écriture des valeurs du tableau en commandes SQL. Avec `ARRAY`, les valeurs de l'élément individuel sont écrites comme elles le seraient si elles ne faisaient pas partie d'un tableau.

8.15. Types composites

Un *type composite* représente la structure d'une ligne ou d'un enregistrement ; il est en essence une simple liste de noms de champs et de leur types de données. PostgreSQL™ autorise l'utilisation de types composite identiques de plusieurs façons à l'utilisation des types simples. Par exemple, une colonne d'une table peut être déclarée comme étant de type composite.

8.15.1. Déclaration de types composite

Voici deux exemples simples de définition de types composite :

```
CREATE TYPE complexe AS (
    r       double precision,
    i       double precision
);

CREATE TYPE element_inventaire AS (
```

```
    nom             text,
    id_fournisseur  integer,
    prix            numeric
);
```

La syntaxe est comparable à **CREATE TABLE** sauf que seuls les noms de champs et leur types peuvent être spécifiés ; aucune contrainte (telle que `NOT NULL`) ne peut être inclus actuellement. Notez que le mot clé `AS` est essentiel ; sans lui, le système penserait à un autre genre de commande **CREATE TYPE** et vous obtiendriez d'étranges erreurs de syntaxe.

Après avoir défini les types, nous pouvons les utiliser pour créer des tables :

```
CREATE TABLE disponible (
    element   element_inventaire,
    nombre    integer
);

INSERT INTO disponible VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

ou des fonctions :

```
CREATE FUNCTION prix_extension(element_inventaire, integer) RETURNS numeric
AS 'SELECT $1.prix * $2' LANGUAGE SQL;

SELECT prix_extension(element, 10) FROM disponible;
```

Quand vous créez une table, un type composite est automatiquement créé, avec le même nom que la table, pour représenter le type de ligne de la table. Par exemple, si nous avions dit :

```
CREATE TABLE element_inventaire (
    nom             text,
    id_fournisseur  integer REFERENCES fournisseur,
    prix            numeric CHECK (prix > 0)
);
```

alors le même type composite `element_inventaire` montré ci-dessus aurait été créé et pourrait être utilisé comme ci-dessus. Néanmoins, notez une restriction importante de l'implémentation actuelle : comme aucune contrainte n'est associée avec un type composite, les contraintes indiquées dans la définition de la table *ne sont pas appliquées* aux valeurs du type composite en dehors de la table. (Un contournement partiel est d'utiliser les types de domaine comme membres de types composites.)

8.15.2. Entrée d'une valeur composite

Pour écrire une valeur composite comme une constante littérale, englobez les valeurs du champ dans des parenthèses et séparez-les par des virgules. Vous pouvez placer des guillemets doubles autour de chaque valeur de champ et vous devez le faire si elle contient des virgules ou des parenthèses (plus de détails ci-dessous). Donc, le format général d'une constante composite est le suivant :

```
'( val1 , val2 , ... )'
```

Voici un exemple :

```
'("fuzzy dice",42,1.99)'
```

qui serait une valeur valide du type `element_inventaire` défini ci-dessus. Pour rendre un champ NULL, n'écrivez aucun caractère dans sa position dans la liste. Par exemple, cette constante spécifie un troisième champ NULL :

```
'("fuzzy dice",42,)'
```

Si vous voulez un champ vide au lieu d'une valeur NULL, saisissez deux guillemets :

```
'("",42,)'
```

Ici, le premier champ est une chaîne vide non NULL alors que le troisième est NULL.

(Ces constantes sont réellement seulement un cas spécial de constantes génériques de type discutées dans la Section 4.1.2.7, « Constantes d'autres types ». La constante est initialement traitée comme une chaîne et passée à la routine de conversion de l'entrée de type composite. Une spécification explicite de type pourrait être nécessaire.)

La syntaxe d'expression `ROW` pourrait aussi être utilisée pour construire des valeurs composites. Dans la plupart des cas, ceci est considérablement plus simple à utiliser que la syntaxe de chaîne littérale car vous n'avez pas à vous inquiéter des multiples couches de guillemets. Nous avons déjà utilisé cette méthode ci-dessus :

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Le mot clé ROW est optionnel si vous avez plus d'un champ dans l'expression, donc ceci peut être simplifié avec

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

La syntaxe de l'expression `ROW` est discutée avec plus de détails dans la Section 4.2.12, « Constructeurs de lignes ».

8.15.3. Accéder aux types composite

Pour accéder à un champ d'une colonne composite, vous pouvez écrire un point et le nom du champ, un peu comme la sélection d'un champ à partir d'un nom de table. En fait, c'est tellement similaire que vous pouvez souvent utiliser des parenthèses pour éviter une confusion de l'analyseur. Par exemple, vous pouvez essayer de sélectionner des sous-champs à partir de notre exemple de table, `disponible`, avec quelque chose comme :

```
SELECT element.nom FROM disponible WHERE element.prix > 9.99;
```

Ceci ne fonctionnera pas car le nom `element` est pris pour le nom d'une table, et non pas d'une colonne de `disponible`, suivant les règles de la syntaxe SQL. Vous devez l'écrire ainsi :

```
SELECT (element).nom FROM disponible WHERE (element).prix > 9.99;
```

ou si vous avez aussi besoin d'utiliser le nom de la table (par exemple dans une requête multi-table), de cette façon :

```
SELECT (disponible.element).nom FROM disponible WHERE (disponible.element).prix > 9.99;
```

Maintenant, l'objet entre parenthèses est correctement interprété comme une référence à la colonne `element`, puis le sous-champ peut être sélectionné à partir de lui.

Des problèmes syntaxiques similaires s'appliquent quand vous sélectionnez un champ à partir d'une valeur composite. En fait, pour sélectionner un seul champ à partir du résultat d'une fonction renvoyant une valeur composite, vous aurez besoin d'écrire quelque chose comme :

```
SELECT (ma_fonction(...)).champ FROM ...
```

Sans les parenthèses supplémentaires, ceci provoquera une erreur.

8.15.4. Modifier les types composite

Voici quelques exemples de la bonne syntaxe pour insérer et mettre à jour des colonnes composites. Tout d'abord pour insérer ou modifier une colonne entière :

```
INSERT INTO matab (col_complexe) VALUES((1.1,2.2));

UPDATE matab SET col_complexe = ROW(1.1,2.2) WHERE ...;
```

Le premier exemple omet `ROW`, le deuxième l'utilise ; nous pouvons le faire des deux façons.

Nous pouvons mettre à jour un sous-champ individuel d'une colonne composite :

```
UPDATE matab SET col_complexe.r = (col_complexe).r + 1 WHERE ...;
```

Notez ici que nous n'avons pas besoin de (et, en fait, ne pouvons pas) placer des parenthèses autour des noms de colonnes apparaissant juste après `SET`, mais nous avons besoin de parenthèses lors de la référence à la même colonne dans l'expression à droite du signe d'égalité.

Et nous pouvons aussi spécifier des sous-champs comme cibles de la commande **INSERT** :

```
INSERT INTO matab (col_complexe.r, col_complexe.i) VALUES(1.1, 2.2);
```

Si tous les sous-champs d'une colonne ne sont pas spécifiés, ils sont remplis avec une valeur NULL.

8.15.5. Syntaxe en entrée et sortie d'un type composite

La représentation texte externe d'une valeur composite consiste en des éléments qui sont interprétés suivant les règles de conversion d'entrées/sorties pour les types de champs individuels, plus des décorations indiquant la structure composite. Cette décoration consiste en des parenthèses (`(` et `)`) autour de la valeur entière ainsi que des virgules (`,`) entre les éléments adjacents. Des espace blancs en dehors des parenthèses sont ignorés mais à l'intérieur des parenthèses, ils sont considérés comme faisant partie de la valeur du champ et pourrait ou non être significatif suivant les règles de conversion de l'entrée pour le type de données du champ. Par exemple, dans :

```
'(  42)'
```

l'espace blanc sera ignoré si le type du champ est un entier, mais pas s'il s'agit d'un champ de type texte.

Comme indiqué précédemment, lors de l'écriture d'une valeur composite, vous pouvez utiliser des guillemets doubles autour de chaque valeur de champ individuel. Vous *devez* le faire si la valeur du champ pourrait sinon gêner l'analyseur de la valeur du champ composite. En particulier, les champs contenant des parenthèses, des virgules, des guillemets doubles ou des antislashs doivent être entre guillemets doubles. Pour placer un guillemet double ou un antislash dans la valeur d'un champ composite entre guillemets, faites-le précéder d'un antislash. (De plus, une paire de guillemets doubles à l'intérieur d'une valeur de champ à guillemets doubles est pris pour représenter un caractère guillemet double, en analogie aux règles des guillemets simples dans les chaînes SQL littérales.) Autrement, vous pouvez éviter les guillemets et utiliser l'échappement par antislash pour protéger tous les caractères de données qui auraient été pris pour une syntaxe composite.

Une valeur de champ composite vide (aucun caractère entre les virgules ou parenthèses) représente une valeur NULL. Pour écrire une valeur qui est une chaîne vide plutôt qu'une valeur NULL, écrivez `""`.

La routine de sortie composite placera des guillemets doubles autour des valeurs de champs s'ils sont des chaînes vides ou s'ils contiennent des parenthèses, virgules, guillemets doubles, antislash ou espaces blancs. (Faire ainsi pour les espaces blancs n'est pas essentiel mais aide à la lecture.) Les guillemets doubles et antislashs dans les valeurs des champs seront doublés.

Note

Rappelez-vous que ce que vous allez saisir dans une commande SQL sera tout d'abord interprété comme une chaîne littérale, puis comme un composite. Ceci double le nombre d'antislash dont vous avez besoin (en supposant que la syntaxe d'échappement des chaînes est utilisée). Par exemple, pour insérer un champ text contenant un guillemet double et un antislash dans une valeur composite, vous devez écrire :

```
INSERT ... VALUES (E('("\\"\\\\")'));
```

Le processeur des chaînes littérales supprime un niveau d'antislash de façon à ce qui arrive à l'analyseur de valeurs composites ressemble à `("\"\\")`. À son tour, la chaîne remplie par la routine d'entrée du type de données text devient `"\`. (Si nous étions en train de travailler avec un type de données dont la routine d'entrée traite aussi les antislashs spécialement, bytea par exemple, nous pourrions avoir besoin d'au plus huit antislashs dans la commande pour obtenir un antislash dans le champ composite stocké.) Le guillemet dollar (voir Section 4.1.2.4, « Constantes de chaînes avec guillemet dollar ») pourrait être utilisé pour éviter le besoin des antislashs doublés.

Astuce

La syntaxe du constructeur `ROW` est habituellement plus simple à utiliser que la syntaxe du littérale composite lors de l'écriture de valeurs composites dans des commandes SQL. Dans `ROW`, les valeurs individuelles d'un champ sont écrits de la même façon qu'ils l'auraient été en étant pas membres du composite.

8.16. Types identifiant d'objet

Les identifiants d'objets (OID) sont utilisés en interne par PostgreSQL™ comme clés primaires de différentes tables système. Les OID ne sont pas ajoutés aux tables utilisateur à moins que `WITH OIDS` ne soit indiqué lors de la création de la table ou que la variable de configuration default_with_oids ne soit activée. Le type oid représente un identifiant d'objet. Il existe également différents types alias du type oid : regproc, regprocedure, regoper, regoperator, regclass, regtype, regconfig et regdictionary. Le Tableau 8.23, « Types identifiant d'objet » en donne un aperçu.

Le type oid est à ce jour un entier non-signé sur quatre octets. Il n'est, de ce fait, pas suffisamment large pour garantir l'unicité au sein d'une base de données volumineuse, voire même au sein d'une très grosse table. Il est donc déconseillé d'utiliser une colonne OID comme clé primaire d'une table utilisateur. Les OID sont avant-tout destinés à stocker des références vers les tables système.

Le type oid lui-même dispose de peu d'opérations en dehors de la comparaison. Il peut toutefois être converti en entier (integer) et manipulé par les opérateurs habituels des entiers (attention aux possibles confusions entre les entiers signés et non signés dans ce cas).

Les types alias d'OID ne disposent pas d'opérations propres à l'exception des routines spécialisées de saisie et d'affichage. Ces routines acceptent et affichent les noms symboliques des objets systèmes, plutôt que la valeur numérique brute que le type oid utilise. Les types alias permettent de simplifier la recherche des valeurs OID des objets. Par exemple, pour examiner les lignes pg_attribute en relation avec une table `ma_table`, on peut écrire :

```
SELECT * FROM pg_attribute WHERE attrelid = 'ma_table'::regclass;
```

plutôt que :

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'ma_table');
```

Bien que cela semble une bonne solution, c'est un peu trop simplifié. Un sous-select bien plus compliqué peut être nécessaire pour sélectionner le bon OID s'il existe plusieurs tables nommées `ma_table` dans différents schémas. Le convertisseur de saisie regclass gère la recherche de la table en fonction du paramétrage du parcours des schémas et effectue donc la « bonne recherche » automatiquement. De façon similaire, la conversion d'un OID de table en regclass pour l'affichage d'un OID numérique est aisée.

Tableau 8.23. Types identifiant d'objet

Nom	Référence	Description	Exemple
oid	tous	identifiant d'objet numérique	`564182`
regproc	pg_proc	nom de fonction	`sum`
regprocedure	pg_proc	fonction avec types d'arguments	`sum(int4)`
regoper	pg_operator	nom d'opérateur	`+`
regoperator	pg_operator	opérateur avec types d'arguments	`*(integer,integer)` ou `-(NONE,integer)`
regclass	pg_class	nom de relation	`pg_type`
regtype	pg_type	nom de type de données	`integer`
regconfig	pg_ts_config	configuration de la recherche plein texte	`english`
regdictionary	pg_ts_dict	dictionnaire de la recherche plein texte	`simple`

Tous les types alias d'OID acceptent des noms qualifiés par le schéma, et affichent des noms préfixés par un schéma si l'objet ne peut être trouvé dans le chemin de recherche courant sans être qualifié. Les types alias regproc et regoper n'acceptent que des noms uniques en entrée (sans surcharge), si bien qu'ils sont d'un usage limité ; dans la plupart des cas, regprocedure et regoperator sont plus appropriés. Pour regoperator, les opérateurs unaires sont identifiés en écrivant `NONE` pour les opérandes non utilisés.

Une propriété supplémentaire des types alias d'OID est la création de dépendances. Si une constante d'un de ces types apparaît dans une expression stockée (telle que l'expression par défaut d'une colonne ou une vue), elle crée une dépendance sur l'objet référencé. Par exemple, si une colonne a une expression par défaut `nextval('ma_seq'::regclass)`, PostgreSQL™ comprend que l'expression par défaut dépend de la séquence `ma_seq` ; le système ne permet alors pas la suppression de la séquence si l'expression par défaut n'est pas elle-même supprimée au préalable.

Un autre type d'identifiant utilisé par le système est xid, ou identifiant de transaction (abrégée xact). C'est le type de données des colonnes système *`xmin`* et *`xmax`*. Les identifiants de transactions sont stockés sur 32 bits.

Un troisième type d'identifiant utilisé par le système est cid, ou identifiant de commande. C'est le type de données des colonnes systèmes *`cmin`* et *`cmax`*. Les identifiants de commandes sont aussi stockés sur 32 bits.

Le dernier type d'identifiant utilisé par le système est tid, ou identifiant de ligne (tuple). C'est le type de données des colonnes système *`ctid`*. Un identifiant de tuple est une paire (numéro de bloc, index de tuple dans le bloc) qui identifie l'emplacement physique de la ligne dans sa table.

Les colonnes systèmes sont expliquées plus en détail dans la Section 5.4, « Colonnes système ».

8.17. Pseudo-Types

Le système de types de PostgreSQL™ contient un certain nombre de types à usage spécial qui sont collectivement appelés des *pseudo-types*. Un pseudo-type ne peut être utilisé comme type d'une colonne de table, mais peut l'être pour déclarer un argument de fonction ou un type de résultat. Tous les pseudo-types disponibles sont utiles dans des situations où une fonction ne se contente pas d'accepter et retourner des valeurs d'un type de données SQL particulier. Le Tableau 8.24, « Pseudo-Types » liste les différents pseudo-types.

Tableau 8.24. Pseudo-Types

Nom	Description
any	Indique qu'une fonction accepte tout type de données, quel qu'il soit.
anyarray	Indique qu'une fonction accepte tout type tableau (voir la Section 35.2.5, « Types et fonctions polymorphes »).
anyelement	Indique qu'une fonction accepte tout type de données (voir la Section 35.2.5, « Types et fonctions polymorphes »).
anyenum	Indique que la fonction accepte tout type de données enum (voir Section 35.2.5, « Types et fonctions poly-

Nom	Description
	morphes » et Section 8.7, « Types énumération »).
anynonarray	Indique que la fonction accepte tout type de données non-array (voir Section 35.2.5, « Types et fonctions polymorphes »).
cstring	Indique qu'une fonction accepte ou retourne une chaîne de caractères C (terminée par un NULL).
internal	Indique qu'une fonction accepte ou retourne un type de données interne du serveur de bases de données.
language_handler	Une fonction d'appel de langage procédural est déclarée retourner un language_handler.
record	Identifie une fonction qui retourne un type de ligne non spécifié.
trigger	Une fonction déclencheur est déclarée comme retournant un type trigger.
void	Indique qu'une fonction ne retourne aucune valeur.
opaque	Un type de données obsolète qui servait précédemment à tous les usages cités ci-dessus.

Les fonctions codées en C (incluses ou chargées dynamiquement) peuvent être déclarées comme acceptant ou retournant tout pseudo-type. Il est de la responsabilité de l'auteur de la fonction de s'assurer du bon comportement de la fonction lorsqu'un pseudo-type est utilisé comme type d'argument.

Les fonctions codées en langage procédural ne peuvent utiliser les pseudo-types que dans les limites imposées par l'implantation du langage. À ce jour, tous les langages procéduraux interdisent l'usage d'un pseudo-type comme argument et n'autorisent que void et record comme type de retours (plus trigger lorsque la fonction est utilisée comme déclencheur). Certains supportent également les fonctions polymorphes qui utilisent les types anyarray, anyelement, anyenum, anynonarray.

Le pseudo-type internal sert à déclarer des fonctions qui ne sont appelées que par le système en interne, et non pas directement par une requête SQL. Si une fonction accepte au minimum un argument de type internal, alors elle ne peut être appelée depuis SQL. Pour préserver la sécurité du type de cette restriction, il est important de suivre la règle de codage suivante : ne jamais créer de fonction qui retourne un internal si elle n'accepte pas au moins un argument de type internal.

Chapitre 9. Fonctions et opérateurs

PostgreSQL™ fournit un grand nombre de fonctions et d'opérateurs pour les types de données intégrés. Les utilisateurs peuvent aussi définir leurs propres fonctions et opérateurs comme décrit dans la Partie V, « Programmation serveur ».

Les commandes **\df** et **\do** de psql sont utilisées pour afficher respectivement la liste des fonctions et des opérateurs.

Du point de vue de la portabilité, il faut savoir que la plupart des fonctions et opérateurs décrits dans ce chapitre, à l'exception des opérateurs arithmétiques et logiques les plus triviaux et de quelques fonctions spécifiquement indiquées, ne font pas partie du standard SQL. Quelques fonctionnalités étendues sont présentes dans d'autres systèmes de gestion de bases de données SQL et dans la plupart des cas, ces fonctionnalités sont compatibles et cohérentes à de nombreuses implantations. Ce chapitre n'est pas exhaustif ; des fonctions supplémentaires apparaissent dans les sections adéquates du manuel.

9.1. Opérateurs logiques

Opérateurs logiques habituels :

`AND`
`OR`
`NOT`

SQL utilise une logique à trois valeurs, true, false et NULL (ce dernier représentant une valeur « inconnue »). Les tables de vérité à considérer sont les suivantes :

a	***b***	***a* AND *b***	***a* OR *b***
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT *a***
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Les opérateurs `AND` et `OR` sont commutatifs, la permutation des opérandes gauche et droit n'affecte pas le résultat. Voir la Section 4.2.13, « Règles d'évaluation des expressions » pour plus d'informations sur l'ordre d'évaluation des sous-expressions.

9.2. Opérateurs de comparaison

Les opérateurs de comparaison habituels sont disponibles, comme l'indique le Tableau 9.1, « Opérateurs de comparaison ».

Tableau 9.1. Opérateurs de comparaison

Opérateur	**Description**
<	inférieur à
>	supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à
=	égal à
<> ou !=	différent de

> **Note**
>
> L'opérateur `!=` est converti en `<>` au moment de l'analyse. Il n'est pas possible d'implanter des opérateurs `!=` et `<>` réalisant des opérations différentes.

Les opérateurs de comparaison sont disponibles pour tous les types de données pour lesquels cela a du sens. Tous les opérateurs de comparaison sont des opérateurs binaires renvoyant des valeurs du type boolean ; des expressions comme `1 < 2 < 3` ne sont pas valides (car il n'existe pas d'opérateur `<` de comparaison d'une valeur booléenne avec `3`).

En plus des opérateurs de comparaison, on trouve la construction spéciale BETWEEN.

```
a BETWEEN x AND y
```

est équivalent à

```
a >= x AND a <= y
```

Notez que BETWEEN traite le point final comme inclut dans l'échelle des valeurs. `NOT BETWEEN` fait la comparaison inverse :

```
a NOT BETWEEN x AND y
```

est équivalent à

```
a < x OR a > y
```

`BETWEEN SYMMETRIC` est identique à `BETWEEN` sauf qu'il n'est pas nécessaire que l'argument à gauche de `AND` soit plus petit ou égal à l'argument à droite. SI ce n'est pas le cas, ces deux arguments sont automatiquement inversés, pour qu'une échelle non vide soit toujours supposée.

Pour vérifier si une valeur est NULL ou non, on utilise les constructions

```
expression IS NULL
expression IS NOT NULL
```

ou la construction équivalente, non standard,

```
expression ISNULL
expression NOTNULL
```

On *ne* peut *pas* écrire `expression = NULL` parce que `NULL` n'est pas « égal à » `NULL`. (La valeur NULL représente une valeur inconnue et il est impossible de dire si deux valeurs inconnues sont égales.) Ce comportement est conforme au standard SQL.

> **Astuce**
>
> Il se peut que des applications s'attendent à voir `expression = NULL` évaluée à vrai (*true*) si `expression` s'évalue comme la valeur NULL. Il est chaudement recommandé que ces applications soient modifiées pour se conformer au standard SQL. Néanmoins, si cela n'est pas possible, le paramètre de configuration transform_null_equals peut être utilisé. S'il est activé, PostgreSQL™ convertit les clauses `x = NULL` en `x IS NULL`.

> **Note**
>
> Si l'`expression` est une valeur de ligne, alors `IS NULL` est vrai quand l'expression même de la ligne est NULL ou quand tous les champs de la ligne sont NULL alors que `IS NOT NULL` est vrai quand l'expression même de la ligne est non NULL et que tous les champs de la ligne sont non NULL. À cause de ce comportement, `IS NULL` et `IS NOT NULL` ne renvoient pas toujours des résultats inversés pour les expressions de lignes, c'est-à-dire une expression de ligne qui contient à la fois des valeurs NULL et des valeurs non NULL retournera faux pour les deux tests. Cette définition, conforme au standard SQL, est une modification du comportement incohérent des versions de PostgreSQL™ antérieures à la 8.2.

L'opérateur standard de comparaison renvoie NULL (ce qui signifie « inconnu ») si l'une des entrées est NULL, ni true ni false, c'est-à-dire `7 = NULL` renvoie NULL. Quand ce comportement n'est pas convenable, utilisez la syntaxe `IS [NOT] DISTINCT FROM` :

```
expression IS DISTINCT FROM expression
expression IS NOT DISTINCT FROM expression
```

Pour des entrées non NULL, `IS DISTINCT FROM` est identique à l'opérateur `<>`. Cependant, si les deux entrées sont NULL, alors cela retourne faux et si une des deux entrées est NULL, alors cela retourne vrai. De la même façon, `IS NOT DISTINCT`

`FROM` est identique à `=` pour les entrées non NULL mais il renvoie true si les deux entrées sont NULL et false quand une seule est NULL. Dans ces constructions, NULL n'est plus considéré comme un état inconnu mais comme une valeur.

Les valeurs booléennes peuvent aussi être testées en utilisant les constructions

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

Elles retournent toujours true ou false, jamais une valeur NULL, même si l'opérande est NULL. Une entrée NULL est traitée comme la valeur logique « inconnue ». `IS UNKNOWN` et `IS NOT UNKNOWN` sont réellement identiques à `IS NULL` et `IS NOT NULL`, respectivement, sauf que l'expression en entrée doit être de type booléen.

9.3. Fonctions et opérateurs mathématiques

Des opérateurs mathématiques sont fournis pour un grand nombre de types PostgreSQL™. Pour les types sans conventions mathématiques standards (les types dates/time, par exemple), le comportement réel est décrit dans les sections appropriées.

Le Tableau 9.2, « Opérateurs mathématiques » affiche les opérateurs mathématiques disponibles.

Tableau 9.2. Opérateurs mathématiques

Opérateur	Description	Exemple	Résultat
`+`	addition	`2 + 3`	`5`
`-`	soustraction	`2 - 3`	`-1`
`*`	multiplication	`2 * 3`	`6`
`/`	division (la division entière tronque les résultats)	`4 / 2`	`2`
`%`	modulo (reste)	`5 % 4`	`1`
`^`	exposant	`2.0 ^ 3.0`	`8`
`|/`	racine carrée	`|/ 25.0`	`5`
`||/`	racine cubique	`||/ 27.0`	`3`
`!`	factoriel	`5 !`	`120`
`!!`	factoriel (opérateur préfixe)	`!! 5`	`120`
`@`	valeur absolue	`@ -5.0`	`5`
`&`	AND bit à bit	`91 & 15`	`11`
`|`	OR bit à bit	`32 | 3`	`35`
`#`	XOR bit à bit	`17 # 5`	`20`
`~`	NOT bit à bit	`~1`	`-2`
`<<`	décalage gauche	`1 << 4`	`16`
`>>`	décalage droit	`8 >> 2`	`2`

Les opérateurs bit à bit ne fonctionnent que sur les types de données entiers alors que les autres sont disponibles pour tous les types de données numériques. Les opérateurs bit à bit sont aussi disponibles pour les types de chaînes de bits bit et bit varying comme le montre le Tableau 9.10, « Opérateurs sur les chaînes de bits ».

Le Tableau 9.3, « Fonctions mathématiques » affiche les fonctions mathématiques disponibles. Dans ce tableau, `dp` signifie double precision. Beaucoup de ces fonctions sont fournies dans de nombreuses formes avec différents types d'argument. Sauf précision contraire, toute forme donnée d'une fonction renvoie le même type de données que son argument. Les fonctions utilisant des données de type double precision sont pour la plupart implantées avec la bibliothèque C du système hôte ; la précision et le comportement dans les cas particuliers peuvent varier en fonction du système hôte.

Tableau 9.3. Fonctions mathématiques

Fonction	Type renvoyé	Description	Exemple	Résultat
`abs(`*`x`*`)`	(identique à l'entrée)	valeur absolue	`abs(-17.4)`	`17.4`

Fonction	Type renvoyé	Description	Exemple	Résultat
`cbrt(dp)`	dp	racine cubique	`cbrt(27.0)`	`3`
`ceil(dp ou numeric)`	(identique à l'argument)	plus petit entier supérieur à l'argument	`ceil(-42.8)`	`-42`
`ceiling(dp ou numeric)`	(identique à l'argument)	plus petit entier supérieur à l'argument (alias de `ceil`)	`ceiling(-95.3)`	`-95`
`degrees(dp)`	dp	radians vers degrés	`degrees(0.5)`	`28.6478897565412`
`div(`*y* `numeric,` *x* `numeric)`	numeric	quotient entier de *y*/*x*	`div(9,4)`	`2`
`exp(dp ou numeric)`	(identique à l'argument)	exponentiel	`exp(1.0)`	`2.71828182845905`
`floor(dp ou numeric)`	(identique à l'argument)	plus grand entier inférieur à l'argument	`floor(-42.8)`	`-43`
`ln(dp ou numeric)`	(identique à l'argument)	logarithme	`ln(2.0)`	`0.693147180559945`
`log(dp ou numeric)`	(identique à l'argument)	logarithme base 10	`log(100.0)`	`2`
`log(`*b* `numeric,` *x* `numeric)`	numeric	logarithme en base *b*	`log(2.0, 64.0)`	`6.0000000000`
`mod(`*y*`,` *x*`)`	(identique au type des arguments)	reste de *y*/*x*	`mod(9,4)`	`1`
`pi()`	dp	constante « pi »	`pi()`	`3.14159265358979`
`power(`*a* `dp,` *b* `dp)`	dp	*a* élevé à la puissance *b*	`power(9.0, 3.0)`	`729`
`power(`*a* `numeric,` *b* `numeric)`	numeric	*a* élevé à la puissance *b*	`power(9.0, 3.0)`	`729`
`radians(dp)`	dp	degrés vers radians	`radians(45.0)`	`0.785398163397448`
`random()`	dp	valeur au hasard entre 0.0 (compris) et 1.0 (non compris)	`random()`	
`round(dp ou numeric)`	(identique à l'argument)	arrondi à l'entier le plus proche	`round(42.4)`	`42`
`round(`*v* `numeric,` *s* `int)`	numeric	arrondi pour *s* décimales	`round(42.4382, 2)`	`42.44`
`setseed(dp)`	void	initialise les appels à `random()` à suivre (valeur entre -1.0 et 1.0, inclusif)	`setseed(0.54823)`	
`sign(dp ou numeric)`	(identique à l'argument)	signe de l'argument (-1, 0, +1)	`sign(-8.4)`	`-1`
`sqrt(dp ou numeric)`	(identique à l'argument)	racine carré	`sqrt(2.0)`	`1.4142135623731`
`trunc(dp ou numeric)`	(identique à l'argument)	tronque vers zéro	`trunc(42.8)`	`42`
`trunc(`*v* `numeric,` *s* `int)`	numeric	tronque sur *s* décimales	`trunc(42.4382, 2)`	`42.43`
`width_bucket(`*opérande* `numeric,` *b1* `numeric,` *b2*	int	renvoie le compartiment auquel l'*opérande* est affecté dans un histogramme d'équidistance à *nombre* compartiments, les va-	`width_bucket(5.35, 0.024, 10.06, 5)`	`3`

Fonction	Type renvoyé	Description	Exemple	Résultat
numeric, *nombre* int)		leurs allant de *b1* à *b2*		

Pour finir, le Tableau 9.4, « Fonctions trigonométriques » affiche les fonctions trigonométriques disponibles. Toutes les fonctions trigonométriques prennent des arguments et renvoient des valeurs de type double precision. Les arguments des fonctions trigonométriques sont exprimés en radian. Voir les fonctions de transformation d'unité `radians()` et `degrees()` ci-dessus.

Tableau 9.4. Fonctions trigonométriques

Fonction	Description
acos(*x*)	arccosinus
asin(*x*)	arcsinus
atan(*x*)	arctangente
atan2(*y*, *x*)	arctangente de *y/x*
cos(*x*)	cosinus
cot(*x*)	cotangente
sin(*x*)	sinus
tan(*x*)	tangente

9.4. Fonctions et opérateurs de chaînes

Cette section décrit les fonctions et opérateurs d'examen et de mainpulation des valeurs de type chaîne de caractères. Dans ce contexte, les chaînes incluent les valeurs des types character, character varying et text. Sauf lorsque cela est précisé différement, toutes les fonctions listées ci-dessous fonctionnent sur tous ces types, mais une attention particulière doit être portée aux effets potentiels du remplissage automatique lors de l'utilisation du type character. Quelques fonctions existent aussi nativement pour le type chaîne bit à bit.

SQL définit quelques fonctions de type chaîne qui utilisent des mots clés, à la place de la virgule, pour séparer les arguments. Des détails sont disponibles dans le Tableau 9.5, « Fonctions et opérateurs SQL pour le type chaîne ». PostgreSQL™ fournit aussi des versions de ces fonctions qui utilisent la syntaxe standard d'appel des fonctions (voir le Tableau 9.6, « Autres fonctions de chaîne »).

Note

Avant PostgreSQL™ 8.3, ces fonctions acceptent silencieusement des valeurs de types de données différents de chaînes de caractères. Cela parce qu'existent des transtypages implicites de ces types en text. Ces forçages ont été supprimés parce que leur comportement est souvent surprenant. Néanmoins, l'opérateur de concaténation de chaîne (||) accepte toujours des éléments qui ne sont pas du type chaîne de caractères, dès lors qu'au moins un des éléments est de type chaîne, comme montré dans Tableau 9.5, « Fonctions et opérateurs SQL pour le type chaîne ». Dans tous les autres cas, il faut insérer un transtypage explicite en text pour mimer le comportement précédent.

Tableau 9.5. Fonctions et opérateurs SQL pour le type chaîne

Fonction	Type renvoyé	Description	Exemple	Résultat
chaîne || *chaîne*	text	Concaténation de chaînes	'Post' || 'greSQL'	PostgreSQL
chaîne || *autre-que-chaîne* ou *autre-que-chaîne* || *chaîne*	text	Concaténation de chaînes avec un argument non-chaîne	'Value: ' || 42	Value: 42
bit_length (*chaîne*)	int	Nombre de bits de la chaîne	bit_length('jose')	32
char_length (*chaîne*) ou character_length (*chaîne*)	int	Nombre de caractères de la chaîne	char_length('jose')	4
lower (*chaîne*)	text	Convertit une chaîne en minus-	lower('TOM')	tom

Fonction	Type renvoyé	Description	Exemple	Résultat
		cule		
octet_length(*chaîne*)	int	Nombre d'octets de la chaîne	octet_length('jose')	4
overlay(*chaîne* placing *chaîne* from int [for int])	text	Remplace la sous-chaîne	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(*substring* in *chaîne*)	int	Emplacement de la sous-chaîne indiquée	position('om' in 'Thomas')	3
substring(*chaîne* [from int] [for int])	text	Extrait une sous-chaîne	substring('Thomas' from 2 for 3)	hom
substring(*chaîne* from *modele*)	text	Extrait la sous-chaîne correspondant à l'expression rationnelle POSIX. Voir Section 9.7, « Correspondance de motif » pour plus d'informations sur la correspondance de modèles.	substring('Thomas' from '...$')	mas
substring(*chaîne* from *pattern* for *escape*)	text	Extrait la sous-chaîne correspondant à l'expression rationnelle SQL. Voir Section 9.7, « Correspondance de motif » pour plus d'informations sur la correspondance de modèles.	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [*caractères*] from *chaîne*)	text	Supprime la plus grande chaîne qui ne contient que les *caractères* (une espace par défaut) à partir du début, de la fin ou des deux extrémités (respectivement leading, trailing, both) de la *chaîne*.	trim(both 'x' from 'xTomxx')	Tom
upper (*chaîne*)	text	Convertit une chaîne en majuscule	upper('tom')	TOM

D'autres fonctions de manipulation de chaînes sont disponibles et listées dans le Tableau 9.6, « Autres fonctions de chaîne ». Certaines d'entre elles sont utilisées en interne pour implanter les fonctions de chaîne répondant au standard SQL listées dans le Tableau 9.5, « Fonctions et opérateurs SQL pour le type chaîne ».

Tableau 9.6. Autres fonctions de chaîne

Fonction	Type renvoyé	Description	Exemple	Résultat
ascii(*chaîne*)	int	Code ASCII du premier octet de l'argument. Pour UTF8, renvoie le code Unicode du caractère. Pour les autres codages multi-octets, l'argument doit impérativement être un caractère ASCII.	ascii('x')	120
btrim(*chaîne* text [, *caractères* text])	text	Supprime la chaîne la plus longue constituée uniquement de caractères issus de *caractères* (une espace par défaut) à partir du début et de la fin de *chaîne*.	btrim('xyxtrimyyx', 'xy')	trim
chr(int)	text	Caractère correspondant au code donné. Pour UTF8, l'argument est traité comme un code Unicode. Pour les autres codages	chr(65)	A

Fonction	Type renvoyé	Description	Exemple	Résultat
		multi-octets, l'argument doit impérativement désigner un caractère ASCII. Le caractère NULL (0) n'est pas autorisé car les types de données texte ne peuvent pas stocker ce type d'octets.		
convert(*chaine* bytea, *encodage_source* name, *encodage_destination* name)	bytea	Convertit la chaîne en encodage *encodage_destination*. L'encodage d'origine est indiqué par *encodage_source*. La *chaine* doit être valide pour cet encodage. Les conversions peuvent être définies avec **CREATE CONVERSION**. De plus, il existe quelques conversions pré-définies. Voir Tableau 9.7, « Conversions intégrées » pour les conversions disponibles.	convert('texte_en_utf8', 'UTF8', 'LATIN1')	texte_en_utf8 représenté dans le codage LATIN1
convert_from(*chaine* bytea, *encodage_source* nom)	text	Convertit la chaîne dans l'encodage de la base. L'encodage original est indiqué par *encodage_source*. La *chaîne* doit être valide pour cet encodage.	convert_from('texte_en_utf8', 'UTF8')	texte_en_utf8 représenté dans le codage de la base en cours
convert_to(*chaine* text, *encodage_destination* nom)	bytea	Convertit une chaîne en encodage *encodage_destination*.	convert_to('un texte', 'UTF8')	un texte représenté dans l'encodage UTF8
decode(*chaîne* text, *format* text)	bytea	Décode les données binaires à partir de sa représentation textuelle dans *chaîne*. Les options pour *format* sont identiques à celles de la fonction encode.	decode('MTIzAAE=', 'base64')	x3132330001
encode(*données* bytea, *format* text)	text	Code les données binaires en une représentation textuelle. Les formats supportés sont : base64, hex, escape. escape convertit les octets nuls et les octets dont le bit de poids fort est à 1, en séquence octal (*nnn*) et des antislashs doubles.	encode(E'123\\000\\001', 'base64')	MTIzAAE=
initcap (*chaîne*)	text	Convertit la première lettre de chaque mot en majuscule et le reste en minuscule. Les mots sont des séquences de caractères alphanumériques séparés par des caractères non alphanumériques.	initcap('bonjour THOMAS')	Bonjour Thomas
length (*chaîne*)	int	Nombre de caractères de *chaîne*	length('jose')	4
length(*chaine*bytea, *encodage* nom)	int	Nombre de caractères de *chaine* dans l'*encodage* donné. La *chaine* doit être valide dans cet encodage.	length('jose', 'UTF8')	4
lpad(*chaîne* text,	text	Complète *chaîne* à *lon-*	lpad('hi', 5, 'xy')	xyxhi

Fonction	Type renvoyé	Description	Exemple	Résultat
longueur int [, *remplissage* text])		*gueur* en ajoutant les caractères *remplissage* en début de chaîne (une espace par défaut). Si *chaîne* a une taille supérieure à *longueur*, alors elle est tronquée (sur la droite).		
ltrim(*chaîne* text [, *caractères* text])	text	Supprime la chaîne la plus longue constituée uniquement de caractères issus de *caractères* (une espace par défaut) à partir du début de la chaîne.	ltrim('zzzytrim', 'xyz')	trim
md5(*chaîne*)	text	Calcule la clé MD5 de *chaîne* et retourne le résultat en hexadécimal.	md5('abc')	900150983cd24fb0d6963f7d28e17f72
pg_client_encoding()	name	Nom de l'encodage client courant.	pg_client_encoding()	SQL_ASCII
quote_ident(*chaîne* text)	text	Renvoie la chaîne correctement placée entre guillemets pour utilisation comme identifiant dans une chaîne d'instruction SQL. Les guillemets ne sont ajoutés que s'ils sont nécessaires (c'est-à-dire si la chaîne contient des caractères autres que ceux de l'identifiant ou qu'il peut y avoir un problème de casse). Les guillemets compris dans la chaîne sont correctement doublés. Voir aussi Exemple 39.1, « Mettre entre guillemets des valeurs dans des requêtes dynamiques ».	quote_ident('Foo bar')	"Foo bar"
quote_literal(*chaîne* text)	text	Renvoie la chaîne correctement placée entre guillemets pour être utilisée comme libellé dans un chaîne d'instruction SQL. Les guillemets simples compris dans la chaîne et les antislash sont correctement doublés. Notez que quote_literal renvoit NULL si son argument est NULL ; si l'argument peut être NULL, la fonction quote_nullable convient mieux. Voir aussi Exemple 39.1, « Mettre entre guillemets des valeurs dans des requêtes dynamiques ».	quote_literal(E'O\'Reilly')	'O''Reilly'
quote_literal(*valeur* anyelement)	text	Convertit la valeur donnée en texte, puis la place entre guillemets suivant la méthode appropriée pour une valeur litérale. Les guillemets simples et antislashs faisant partie de cette valeur sont doublés proprement.	quote_literal(42.5)	'42.5'

Fonction	Type renvoyé	Description	Exemple	Résultat
regexp_matches(*chaine* text, *modele* text [, *options* text])	setof text[]	Renvoie toutes les sous-chaînes capturées résultant d'une correspondance entre l'expression rationnelle POSIX et *chaine*. Voir Section 9.7.3, « Expressions rationnelles POSIX » pour plus d'informations.	regexp_matches('foobar bequebaz', '(bar)(beque)')	{bar,beque}
quote_nullable(*chaine* text)	text	Renvoit la chaîne donnée convenablement mise entre guillemets pour être utilisée comme une chaîne litérale dans une instruction SQL ; or si l'argument est NULL, elle renvoit NULL. Les guillemets simples et antislashs dans la chaîne sont doublés correctement. Voir aussi Exemple 39.1, « Mettre entre guillemets des valeurs dans des requêtes dynamiques ».	quote_nullable(NULL)	NULL
quote_nullable(*valeur* anyelement)	text	Renvoit la valeur donnée en texte, puis la met entre guillemets comme un litéral ; or, si l'argument est NULL, elle renvoit NULL.Les guillemets simples et antislashs dans la chaîne sont doublés correctement.	quote_nullable(42.5)	'42.5'
regexp_replace(*chaîne* text, *modele* text, *remplacement* text [,*options* text])	text	Remplace la sous-chaîne correspondant à l'expression rationnelle POSIX. Voir Section 9.7.3, « Expressions rationnelles POSIX » pour plus d'informations.	regexp_replace('Thomas', '.[mN]a.', 'M')	ThM
regexp_split_to_array (*chaine* text, *modele* text [, *options* text])	text[]	Divise une *chaine* en utilisant une expression rationnelle POSIX en tant que délimiteur. Voir Section 9.7.3, « Expressions rationnelles POSIX » pour plus d'informations.	regexp_split_to_array('hello world', E'\\s+')	{hello,world}
regexp_split_to_table (*chaine* text, *modele* text [, *options* text])	setof text	Divise la *chaine* en utilisant une expression rationnelle POSIX comme délimiteur. Voir Section 9.7.3, « Expressions rationnelles POSIX » pour plus d'informations.	regexp_split_to_table('hello world', E'\\s+')	hello world (2 rows)
repeat(*chaîne* text, *nombre* int)	text	Répète le texte *chaîne* *nombre* fois	repeat('Pg', 4)	PgPgPgPg
replace(*chaîne* text, *àpartirde* text, *vers* text)	text	Remplace dans *chaîne* toutes les occurrences de la sous-chaîne *àpartirde* par la sous-chaîne *vers*.	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
rpad(*chaîne* text, *longueur* int [, *remplissage* text])	text	Complète *chaîne* à *longueur* caractères en ajoutant les caractères *remplissage* à la fin (une espace par défaut). Si la *chaîne* a une taille supérieure à *longueur*, elle est tronquée.	rpad('hi', 5, 'xy')	hixyx

Fonction	Type renvoyé	Description	Exemple	Résultat
rtrim(*chaîne* text [, *caractères* text])	text	Supprime la chaîne la plus longue contenant uniquement les caractères provenant de *caractères* (une espace par défaut) depuis la fin de *chaîne*.	rtrim('trimxxxx', 'x')	trim
split_part(*chaîne* text, *délimiteur* text, *champ* int)	text	Divise *chaîne* par rapport au *délimiteur* et renvoie le champ donné (en comptant à partir de 1).	split_part('abc~@~def~@~ghi', '~@~', 2)	def
strpos(*chaîne*, *sous-chaîne*)	int	Emplacement de la sous-chaîne indiquée (identique à position(*sous-chaîne* in *sous-chaîne*), mais avec les arguments en ordre inverse).	strpos('high', 'ig')	2
substr(*chaîne*, *from* [, *nombre*])	text	Extrait la sous-chaîne (identique à substring(*chaîne* from *àpartirde* for *nombre*))	substr('alphabet', 3, 2)	ph
to_ascii(*chaîne* text [, *encodage* text])	text	Convertit la *chaîne* en ASCII à partir de n'importe quelle autre encodage (ne supporte que les conversions à partir de LATIN1, LATIN2, LATIN9 et WIN1250).	to_ascii('Karel')	Karel
to_hex(*nombre* int ou bigint)	text	Convertit *nombre* dans sa représentation hexadécimale équivalente.	to_hex(2147483647)	7fffffff
translate(*chaîne* text, *àpartirde* text, *vers* text)	text	Tout caractère de *chaîne* qui correspond à un caractère de l'ensemble *àpartirde* est remplacé par le caractère correspondant de l'ensemble *vers*.	translate('12345', '14', 'ax')	a23x5

Voir aussi la fonction d'agrégat string_agg dans Section 9.18, « Fonctions d'agrégat ».

Tableau 9.7. Conversions intégrées

Nom de la conversion [a]	Codage source	Codage destination
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL

Nom de la conversion [a]	Codage source	Codage destination
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5

Nom de la conversion [a]	Codage source	Codage destination
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
tcvn_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_tcvn	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255

Nom de la conversion [a]	Codage source	Codage destination
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] Les noms des conversions suivent un schéma de nommage standard : le nom officiel de l'encodage source avec tous les caractères non alpha-numériques remplacés par des tirets bas suivi de `_to_` suivi du nom de l'encodage cible ayant subit le même traitement que le nom de l'encodage source. Il est donc possible que les noms varient par rapport aux noms d'encodage personnalisés.

9.5. Fonctions et opérateurs de chaînes binaires

Cette section décrit les fonctions et opérateurs d'examen et de manipulation des valeurs de type bytea.

SQL définit quelques fonctions de chaînes qui utilise des mots clés qui sont employés à la place de virgules pour séparer les arguments. Les détails sont présentés dans Tableau 9.8, « Fonctions et opérateurs SQL pour chaînes binaires ». PostgreSQL™ fournit aussi des versions de ces fonctions qui utilisant la syntaxe standard de l'appel de fonction (voir le Tableau 9.9, « Autres fonctions sur les chaînes binaires »).

Note

Les résultats montrés en exemple sur cette page supposent que le paramètre serveur `bytea_output` est configuré à `escape` (le format traditionnel de PostgreSQL).

Tableau 9.8. Fonctions et opérateurs SQL pour chaînes binaires

Fonction	Type renvoyé	Description	Exemple	Résultat
chaîne `||` *chaîne*	bytea	Concaténation de chaîne	`E'\\\\Post'::bytea || E'\\047gres\\000'::bytea`	`\\Post'gres\000`

Fonction	Type renvoyé	Description	Exemple	Résultat
octet_length(*chaîne*)	int	Nombre d'octets d'une chaîne binaire	octet_length(E'jo\\000se'::bytea)	5
overlay(*string* placing *string* from int [for int])	bytea	Remplace une sous-chaîne	overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea from 2 for 3)	T\\002\\003mas
position(*sous-chaîne* in *chaîne*)	int	Emplacement de la sous-chaîne indiquée	position(E'\\000om'::bytea in E'Th\\000omas'::bytea)	3
substring(*chaîne* [from int] [for int])	bytea	Extrait une sous-chaîne	substring(E'Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] *octets* from *chaîne*)	bytea	Supprime la plus longue chaîne composée uniquement d'octets de *octets* à partir du début et de la fin de *chaîne*	trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea)	Tom

Des fonctions supplémentaires de manipulations de chaînes binaires sont listées dans le Tableau 9.9, « Autres fonctions sur les chaînes binaires ». Certaines sont utilisées en interne pour coder les fonctions de chaînes suivant le standard SQL et sont listées dans le Tableau 9.8, « Fonctions et opérateurs SQL pour chaînes binaires ».

Tableau 9.9. Autres fonctions sur les chaînes binaires

Fonction	Type retourné	Description	Exemple	Résultat
btrim(*chaîne* bytea, *octets* bytea)	bytea	Supprime la plus longue chaîne constituée uniquement d'octets de *octets* à partir du début et de la fin de *chaîne*.	btrim(E'\\000trim\\000'::bytea, E'\\000'::bytea)	trim
decode(*chaîne* text, *format* text)	bytea	Décode les données binaires de leur représentation textuelle dans *chaîne* auparavant codée. Les options pour *format* sont les mêmes que pour encode.	decode(E'123\\000456', 'escape')	123\000456
encode(*chaîne* bytea, *type* text)	text	Code les données binaires en une représentation textuelle. Les formats supportés sont : base64, hex, escape. escape convertit les octets nuls et les octets dont le bit de poids fort est à 1, en séquence octal (*nnn*) et des antislashs doubles.	encode(E'123\\000456'::bytea, 'escape')	123\000456
get_bit(*string*, *offset*)	int	Extrait un bit d'une chaîne	get_bit(E'Th\\000omas'::bytea, 45)	1
get_byte(*string*, *offset*)	int	Extrait un octet d'une chaîne	get_byte(E'Th\\000omas'::bytea, 4)	109
length (*chaîne*)	int	Longueur de la chaîne binaire	length(E'jo\\000se'::bytea)	5

Fonction	Type retourné	Description	Exemple	Résultat
`md5(chaîne)`	text	Calcule le hachage MD5 de la *chaîne* et retourne le résultat en hexadécimal	`md5(E'Th\\000omas'::bytea)`	`8ab2d3c9689aaf18b4958c334c82d8b1`
`set_bit(string, offset, newvalue)`	bytea	Positionne un bit dans une chaîne	`set_bit(E'Th\\000omas'::bytea, 45, 0)`	`Th\000omAs`
`set_byte(string, offset, newvalue)`	bytea	Positionne un octet dans une chaîne	`set_byte(E'Th\\000omas'::bytea, 4, 64)`	`Th\000o@as`

`get_byte` et `set_byte` prennent en compte le premier octet d'une chaîne binaire comme l'octet numéro zéro. `get_bit` et `set_bit` comptent les bits à partir de la droite pour chaque octet. Par exemple, le bit 0 est le bit le moins significatif du premier octet et le bit 15 est le bit le plus significatif du second octet.

9.6. Fonctions et opérateurs sur les chaînes de bits

Cette section décrit les fonctions et opérateurs d'examen et de manipulation des chaînes de bits, c'est-à-dire des valeurs de types bit et bit varying. En dehors des opérateurs de comparaison habituels, les opérateurs présentés dans le Tableau 9.10, « Opérateurs sur les chaînes de bits » peuvent être utilisés. Les opérandes de chaînes de bits utilisés avec `&`, `|` et `#` doivent être de même longueur. Lors d'un décalage de bits, la longueur originale de la chaîne est préservée comme le montrent les exemples.

Tableau 9.10. Opérateurs sur les chaînes de bits

Opérateur	Description	Exemple	Résultat
`||`	concaténation	`B'10001' || B'011'`	`10001011`
`&`	AND bit à bit	`B'10001' & B'01101'`	`00001`
`|`	OR bit à bit	`B'10001' | B'01101'`	`11101`
`#`	XOR bit à bit	`B'10001' # B'01101'`	`11100`
`~`	NOT bit à bit	`~ B'10001'`	`01110`
`<<`	décalage gauche bit à bit	`B'10001' << 3`	`01000`
`>>`	décalage droit bit à bit	`B'10001' >> 2`	`00100`

Les fonctions SQL suivantes fonctionnent sur les chaînes de bits ainsi que sur les chaînes de caractères : `length`, `bit_length`, `octet_length`, `position`, `substring`, `overlay`.

Les fonctions suivantes fonctionnent sur les chaînes de bits ainsi que sur les chaînes binaires : `get_bit`, `set_bit`. En travaillant sur des chaînes de bits, ces fonctions numérotent le premier bit (le plus à gauche) comme le bit 0.

De plus, il est possible de convertir des valeurs intégrales vers ou depuis le type bit. Quelques exemples :

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Le transtypage « bit » signifie transtyper en `bit(1)` et, de ce fait, seul le bit de poids faible de l'entier est rendu.

Note

Avant PostgreSQL™ 8.0, la conversion d'un entier en bit(n) copiait les `n` bits les plus à gauche de l'entier. Désormais, ce sont les `n` bits les plus à droite qui sont copiés. De plus, la conversion d'un entier en une chaîne de bits plus grande que l'entier lui-même étend l'entier, avec signature, vers la gauche.

9.7. Correspondance de motif

PostgreSQL™ fournit trois approches différentes à la correspondance de motif : l'opérateur SQL traditionnel `LIKE`, le plus récent

SIMILAR TO (ajouté dans SQL:1999) et les expressions rationnelles de type POSIX. En dehors des opérateurs basiques du style « est-ce que cette chaîne correspond à ce modèle ? », les fonctions sont disponibles pour extraire ou remplacer des sous-chaînes correspondantes ou pour diviser une chaîne aux emplacements correspondants.

> **Astuce**
>
> Si un besoin de correspondances de motif va au-delà, il faut considérer l'écriture d'une fonction en Perl ou Tcl.

9.7.1. LIKE

```
chaîne LIKE motif [ESCAPE caractère d'échappement]
chaîne NOT LIKE motif [ESCAPE caractère d'échappement]
```

L'expression LIKE renvoie true si la *chaîne* est contenue dans l'ensemble de chaînes représenté par le *motif*. (L'expression NOT LIKE renvoie false si LIKE renvoie true et vice versa. Une expression équivalente est NOT (*chaîne* LIKE *motif*).)

Si le *motif* ne contient ni signe pourcent ni tiret bas, alors il ne représente que la chaîne elle-même ; dans ce cas, LIKE agit exactement comme l'opérateur d'égalité. Un tiret bas (_) dans *motif* correspond à un seul caractère, un signe pourcent (%) à toutes les chaînes de zéro ou plusieurs caractères.

Quelques exemples :

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

Le modèle LIKE correspond toujours à la chaîne entière. Du coup, pour faire correspondre une séquence à l'intérieur d'une chaîne, le motif doit donc commencer et finir avec un signe pourcent.

Pour faire correspondre un vrai tiret bas ou un vrai signe de pourcentage sans correspondance avec d'autres caractères, le caractère correspondant dans *motif* doit être précédé du caractère d'échappement. Par défaut, il s'agit de l'antislash, mais un autre caractère peut être sélectionné en utilisant la clause ESCAPE. Pour un correspondance avec le caractère d'échappement lui-même, on écrit deux fois ce caractère.

L'antislash a déjà une signification particulière dans les chaînes littérales. De ce fait, pour écrire un motif qui contient un antislash, il faut écrire deux antislashs dans l'instruction SQL (en supposant que la syntaxe d'échappement de chaînes soit utilisée, voir Section 4.1.2.1, « Constantes de chaînes »). Ainsi, pour écrire un motif qui corresponde effectivement à un antislash littéral nécessite l'écriture de quatre antislash dans l'instruction. Tout cela peut être évité en sélectionnant un autre caractère d'échappement avec ESCAPE ; un antislash n'est alors plus spécial au regard de LIKE (mais il est toujours spécial pour l'analyseur de chaînes littérales, et nécessite toujours d'être doublé pour correspondre à un antislash).

Il est aussi possible de ne sélectionner aucun caractère d'échappement en écrivant ESCAPE ''. Ceci désactive complètement le mécanisme d'échappement, ce qui rend impossible la désactivation de la signification particulière du tiret bas et du signe de pourcentage dans le motif.

Le mot clé ILIKE est utilisé à la place de LIKE pour faire des correspondances sans tenir compte de la casse mais en tenant compte de la locale active. Ceci ne fait pas partie du standard SQL mais est une extension PostgreSQL™.

L'opérateur ~~ est équivalent à LIKE alors que ~~* correspond à ILIKE. Il existe aussi les opérateurs !~~ et !~~* représentant respectivement NOT LIKE et NOT ILIKE. Tous ces opérateurs sont spécifiques à PostgreSQL™.

9.7.2. Expressions rationnelles SIMILAR TO

```
chaîne SIMILAR TO motif [ESCAPE caractère d'échappement]
chaîne NOT SIMILAR TO motif [ESCAPE caractère d'échappement]
```

L'opérateur SIMILAR TO renvoie true ou false selon que le motif correspond ou non à la chaîne donnée. Il se rapproche de LIKE à la différence qu'il interprète le motif en utilisant la définition SQL d'une expression rationnelle. Les expressions rationnelles SQL sont un curieux mélange de la notation LIKE et de la notation habituelle des expressions rationnelles.

À l'instar de LIKE, l'opérateur SIMILAR TO ne réussit que si son motif correspond à la chaîne entière ; ceci en désaccord avec les pratiques habituelles des expressions rationnelles où le modèle peut se situer n'importe où dans la chaîne. Tout comme LIKE, SIMILAR TO utilise _ et % comme caractères joker représentant respectivement tout caractère unique et toute chaîne (ils sont comparables à . et .* des expressions rationnelles compatibles POSIX).

En plus de ces fonctionnalités empruntées à LIKE, SIMILAR TO supporte trois méta-caractères de correspondance de motif em-

pruntés aux expressions rationnelles de POSIX :

- `|` représente une alternative (une des deux alternatives) ;
- `*` représente la répétition des éléments précédents, 0 ou plusieurs fois ;
- `+` représente la répétition des éléments précédents, une ou plusieurs fois ;
- `?` dénote une répétition du précédent élément zéro ou une fois.
- `{m}` dénote une répétition du précédent élément exactement *m* fois.
- `{m,}` dénote une répétition du précédent élément *m* ou plusieurs fois.
- `{m,n}` dénote une répétition du précédent élément au moins *m* et au plus *n* fois.
- les parenthèses `()` peuvent être utilisées pour grouper des éléments en un seul élément logique ;
- une expression entre crochets `[...]` spécifie une classe de caractères, comme dans les expressions rationnelles POSIX.

Notez que le point (`.`) n'est pas un méta-caractère pour `SIMILAR TO`.

Comme avec `LIKE`, un antislash désactive la signification spéciale de tous les méta-caractères ; un autre caractère d'échappement peut être indiqué avec `ESCAPE`.

Quelques exemples :

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

La fonction `substring` avec trois paramètres, `substring(`*`chaîne`* `from` *`motif`* `for` *`caractère d'échappement`*`)`, fournit l'extraction d'une sous-chaîne correspondant à un motif d'expression rationnelle SQL. Comme avec `SIMILAR TO`, le motif fourni doit correspondre à la chaîne de données entière, sinon la fonction échoue et renvoie NULL. Pour indiquer la partie du motif à retourner en cas de succès, le motif doit contenir deux occurrences du caractère d'échappement suivi d'un guillemet double (`"`). Le texte correspondant à la portion du motif entre ces deux marqueurs est renvoyé.

Quelques exemples, avec `#"` délimitant la chaîne en retour :

```
substring('foobar' from '%#"o_b#"%' for '#')
oob
substring('foobar' from '#"o_b#"%' for '#')
NULL
```

9.7.3. Expressions rationnelles POSIX

Le Tableau 9.11, « Opérateurs de correspondance des expressions rationnelles » liste les opérateurs disponibles pour la correspondance de motifs à partir d'expressions rationnelles POSIX.

Tableau 9.11. Opérateurs de correspondance des expressions rationnelles

Opérateur	Description	Exemple
`~`	Correspondance d'expression rationnelle, en tenant compte de la casse	`'thomas' ~ '.*thomas.*'`
`~*`	Correspondance d'expression rationnelle, sans tenir compte de la casse	`'thomas' ~* '.*Thomas.*'`
`!~`	Non-correspondance d'expression rationnelle, en tenant compte de la casse	`'thomas' !~ '.*Thomas.*'`
`!~*`	Non-correspondance d'expression rationnelle, sans tenir compte de la casse	`'thomas' !~* '.*vadim.*'`

Les expressions rationnelles POSIX sont un outil de correspondance de motifs plus puissant que les opérateurs `LIKE` et `SIMILAR TO`. Beaucoup d'outils Unix comme **egrep**, **sed** ou **awk** utilisent un langage de correspondance de modèles similaire à celui décrit ici.

Une expression rationnelle est une séquence de caractères représentant une définition abrégée d'un ensemble de chaînes (un *ensemble rationnel*). Une chaîne est déclarée correspondre à une expression rationnelle si elle est membre de l'ensemble rationnel décrit par l'expression rationnelle. Comme avec `LIKE`, les caractères du motif correspondent exactement aux caractères de le chaîne sauf s'ils représentent des caractères spéciaux dans le langage des expressions rationnelles -- mais les expressions ration-

nelles utilisent des caractères spéciaux différents de ceux utilisés par `LIKE`. Contrairement aux motifs de `LIKE`, une expression rationnelle peut avoir une correspondance en toute place de la chaîne, sauf si l'expression rationnelle est explicitement ancrée au début ou à la fin de la chaîne.

Quelques exemples :

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

Le langage modèle POSIX est décrit avec plus de détail ci-dessous.

La fonction `substring` avec deux paramètres, `substring(`*`chaîne`* `from` *`motif`*`)`, extrait une sous-chaîne qui correspond à un motif d'expression rationnelle POSIX. Elle renvoie NULL s'il n'y a pas de correspondance, la portion de texte correspondant au modèle dans le cas contraire. Mais si le motif contient des parenthèses, c'est la portion de texte qui correspond à la première sous-expression entre parenthèses (la première dont la parenthèse gauche apparaît) qui est renvoyée. Il est possible de placer toute l'expression entre parenthèses pour pouvoir utiliser des parenthèses à l'intérieur sans déclencher cette exception. Si des parenthèses sont nécessaires dans le motif avant la sous-expression à extraire, il faut utiliser les propriétés des parenthèses non-capturantes décrites plus bas.

Quelques exemples :

```
substring('foubar' from 'o.b')     oub
substring('foubar' from 'o(.)b')   u
```

La fonction `regexp_replace` substitue un nouveau texte aux sous-chaînes correspondantes des motifs d'expressions rationnelles. Elle a la syntaxe `regexp_replace(`*`source`*, *`motif`*, *`remplacement`* [, *`options`*]). La chaîne *`source`* est renvoyée non modifiée s'il n'existe pas de correspondance avec *`motif`*. S'il existe une correspondance, la chaîne *`source`* est renvoyée avec la chaîne *`remplacement`* substituée à la sous-chaîne correspondante. La chaîne *`remplacement`* peut contenir `\`*`n`*, avec *`n`* de `1` à `9`, pour indiquer que la *`n`*-ième sous-chaîne source correspondante doit être insérée. Elle peut aussi contenir `\&` pour indiquer que la sous-chaîne qui correspond au motif entier doit être insérée. On écrit `\\` pou placer un antislash littéral dans le texte de remplacement (comme toujours, les antislashs doivent être doublés dans les libellés, si la syntaxe d'échappement de chaînes est utilisée). Le paramètre *`options`* est une chaîne optionnelle de drapeaux (0 ou plus) d'une lettre qui modifie le comportement de la fonction. Le drapeau `i` indique une recherche insensible à la casse, le drapeau `g` un remplacement de chaque sous-chaîne correspondante (pas uniquement la première). Les autres options supportées sont décrites dans Tableau 9.19, « Lettres d'option intégrées à une ERA ».

Quelques exemples :

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', E'X\\1Y', 'g')
                                   fooXarYXazY
```

La fonction `regexp_matches` renvoie un tableau de texte contenant toutes les sous-chaînes capturées résultant de la correspondance avec une expression rationnelle POSIX. Elle a la syntaxe : `regexp_matches(`*`chaine`*, *`modele`* [, *`options`*]). La fonction peut ne renvoyer aucune ligne, une ligne ou plusieurs lignes (voir le drapeau `g` ci-dessous). Si le *`motif`* ne correspond pas, la fonction ne renvoit aucune ligne. Si le motif ne contient aucune sous-expressions entre parenthèses, alors chaque ligne renvoyée est un tableau de texte à un seul élément contenant la sous-chaîne correspondant au motif complet. Si le motif contient des sous-expressions entre parenthèses, la fonction renvoie un tableau de texte dont l'élément *`n`* est la sous-chaîne en correspondance avec la *`n`*-ième sous-expression entre parenthèses du modèle (sans compter les parenthèses « non capturantes » ; voir ci-dessous pour les détails). Le paramètre *`options`* est une chaîne optionnelle contenant zéro ou plus options d'une lettre, modifiant ainsi le comportement de la fonction. L'option `g` indique que la fonction trouve chaque correspondance dans la chaîne, pas seulement la première, et renvoie une ligne pour chaque correspondance. Les autres options supportées sont décrites dans Tableau 9.19, « Lettres d'option intégrées à une ERA ».

Quelques exemples :

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
 regexp_matches
----------------
 {bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
```

```
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)

SELECT regexp_matches('foobarbequebaz', 'barbeque');
 regexp_matches
----------------
 {barbeque}
(1 row)
```

Il est possible de forcer `regexp_matches()` à toujours renvoyer une ligne en utilisant une sous-sélection ; ceci est particulièrement utile dans une liste cible `SELECT` lorsque vous voulez renvoyer toutes les lignes, y compris celles qui ne correspondent pas :

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

La fonction `regexp_split_to_table` divise une chaîne en utilisant une expression rationnelle POSIX comme délimiteur. Elle a la syntaxe suivante : `regexp_split_to_table(`*`chaine`*, *`modele`* [, *`options`*]). S'il n'y a pas de correspondance avec le *`modele`*, la fonction renvoie la *`chaine`*. S'il y a au moins une correspondance, pour chaque correspondance, elle renvoie le texte à partir de la fin de la dernière correspondance (ou le début de la chaîne) jusqu'au début de la correspondance. Quand il ne reste plus de correspondance, elle renvoie le texte depuis la fin de la dernière correspondance jusqu'à la fin de la chaîne. Le paramètre *`options`* est une chaîne optionelle contenant zéro ou plus options d'un caractère, modifiant ainsi le comportement de la fonction. `regexp_split_to_table` supporte les options décrites dans Tableau 9.19, « Lettres d'option intégrées à une ERA ».

La fonction `regexp_split_to_array` se comporte de la même façon que `regexp_split_to_table`, sauf que `regexp_split_to_array` renvoie son résultat en tant que tableau de text. Elle a comme syntaxe `regexp_split_to_array(`*`chaine`*, *`modele`* [, *`options`*]). Les paramètres sont les mêmes que pour `regexp_split_to_table`.

Quelques exemples :

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog',
E'\\s+') AS foo;
  foo
--------
 the
 quick
 brown
 fox
 jumped
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog', E'\\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumped,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
```

```
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

Comme le montre le dernier exemple, les fonctions de division des expressions rationnelles ignorent les correspondances de longueur nulle qui surviennent au début ou à la fin de la chaîne ou immédiatement après une correspondance. C'est contraire à la définition stricte de la correspondance des expressions rationnelles implantée par `regexp_matches`, mais c'est habituellement le comportement le plus pratique. Les autres systèmes comme Perl utilisent des définitions similaires.

9.7.3.1. Détails des expressions rationnelles

Les expressions rationnelles de PostgreSQL™ sont implantées à l'aide d'un paquetage écrit par Henry Spencer. Une grande partie de la description des expressions rationnelles ci-dessous est une copie intégrale de son manuel.

Les expressions rationnelles (ERs), telles que définies dans POSIX 1003.2, existent sous deux formes : les ER *étendues* ou ERE (en gros celles de **egrep**) et les ER *basiques* ou ERB (BRE en anglais) (en gros celles d'**ed**). PostgreSQL™ supporte les deux formes et y ajoute quelques extensions ne faisant pas partie du standard POSIX mais largement utilisées du fait de leur disponibilité dans les langages de programmation tels que Perl et Tcl. Les ER qui utilisent ces extensions non POSIX sont appelées des ER *avancées* ou ERA (ARE en anglais) dans cette documentation. Les ERA sont un surensemble exact des ERE alors que les ERB ont des incompatibilités de notation (sans parler du fait qu'elles sont bien plus limitées). En premier lieu sont décrits les formats ERA et ERE, en précisant les fonctionnalités qui ne s'appliquent qu'aux ERA. L'explication des différences des ERB vient ensuite.

> **Note**
>
> PostgreSQL™ always initially presumes that a regular expression follows the ARE rules. However, the more limited ERE or BRE rules can be chosen by prepending an *embedded option* to the RE pattern, as described in Section 9.7.3.4, « Métasyntaxe des expressions rationnelles ». This can be useful for compatibility with applications that expect exactly the POSIX 1003.2 rules.

Une expression rationnelle est définie par une ou plusieurs *branches* séparées par des caractères `|`. Elle établit une correspondance avec tout ce qui correspond à une des branches.

Une branche contient des *atomes quantifiés*, ou *contraintes*, concaténés. Elle établit une correspondance pour le premier suivi d'une correspondance pour le second, etc ; une branche vide établit une correspondance avec une chaîne vide.

Un atome quantifié est un *atome* éventuellement suivi d'un *quantificateur* unique. Sans quantificateur, il établit une correspondance avec l'atome. Avec un quantificateur, il peut établir un certain nombre de correspondances avec l'atome. Un *atome* est une des possibilités du Tableau 9.12, « Atomes d'expressions rationnelles ». Les quantificateurs possibles et leurs significations sont disponibles dans le Tableau 9.13, « quantificateur d'expressions rationnelles ».

Une *contrainte* établit une correspondance avec une chaîne vide, mais cette correspondance n'est établie que lorsque des conditions spécifiques sont remplies. Une contrainte peut être utilisée là où un atome peut l'être et ne peut pas être suivie d'un quantificateur. Les contraintes simples sont affichées dans le Tableau 9.14, « Contraintes des expressions rationnelles » ; quelques contraintes supplémentaires sont décrites plus loin.

Tableau 9.12. Atomes d'expressions rationnelles

Atome	Description
`(`*`re`*`)`	(où *`re`* est toute expression rationnelle) établit une correspondance avec *`re`*, la correspondance étant conservée en vue d'un éventuel report
`(?:`*`re`*`)`	comme ci-dessus mais la correspondance n'est pas conservée pour report (un ensemble de parenthèses « sans capture ») (seulement pour les ERA)
`.`	correpondance avec tout caractère unique
`[`*`caractères`*`]`	une *expression entre crochets*, qui établit une correspondance avec tout caractère de *`caractères`* (voir la Section 9.7.3.2, « Expressions avec crochets » pour plus de détails)
`\`*`k`*	(où *`k`* n'est pas un caractère alpha-numérique) établit une correspondance avec ce caractère, considéré comme caractère ordinaire. Par exemple, `\\` établit une correspondance avec un caractère antislash

Atome	Description
`\c`	avec `c` un caractère alphanumérique (éventuellement suivi d'autres caractères) est un *échappement*, voir la Section 9.7.3.3, « Échappement d'expressions rationnelles » (ERA seulement ; pour les ERE et ERB, établit une correspondance avec `c`)
`{`	lorsqu'il est suivi d'un caractère autre qu'un chiffre, établit une correspondance avec l'accolade ouvrante `{` ; suivi d'un chiffre, c'est le début d'une `limite` (voir ci-dessous)
`x`	où `x` est un caractère unique sans signification, établit une correspondance avec ce caractère

Une ER ne peut pas se terminer par `\`.

> **Note**
>
> L'antislash (`\`) a déjà une signification particulière dans les chaînes littérales PostgreSQL™. Pour écrire un motif contenant un antislash, il faut écrire deux antislashs dans l'instruction, en supposant que la syntaxe de chaîne d'échappement est utilisée (voir Section 4.1.2.1, « Constantes de chaînes »).

Tableau 9.13. quantificateur d'expressions rationnelles

quantificateur	Correspondance
`*`	une séquence de 0 ou plus correspondance(s) de l'atome
`+`	une séquence de 1 ou plus correspondance(s) de l'atome
`?`	une séquence de 0 ou 1 correspondance de l'atome
`{m}`	une séquence d'exactement `m` correspondances de l'atome
`{m,}`	une séquence de `m` ou plus correspondances de l'atome
`{m,n}`	une séquence de `m` à `n` (inclus) correspondances de l'atome ; `m` ne doit pas être supérieur à `n`
`*?`	version non gourmande de `*`
`+?`	version non gourmande de `+`
`??`	version non gourmande de `?`
`{m}?`	version non gourmande de `{m}`
`{m,}?`	version non gourmande de `{m,}`
`{m,n}?`	version non gourmande de `{m,n}`

Les formes qui utilisent `{...}` sont appelées *limites*. Les nombres `m` et `n` à l'intérieur d'une limite sont des entiers non signés dont les valeurs vont de 0 à 255 inclus.

Les quantificateurs *non gourmands* (disponibles uniquement avec les ERA) correspondent aux même possibilités que leurs équivalents normaux (*gourmand*), mais préfèrent le plus petit nombre de correspondances au plus grand nombre. Voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles » pour plus de détails.

> **Note**
>
> Un quantificateur ne peut pas immédiatement suivre un autre quantificateur, autrement dit `**` est invalide. Il ne peut pas non plus débuter une expression ou sous-expression ni suivre `^` ou `|`.

Tableau 9.14. Contraintes des expressions rationnelles

Contrainte	Description
`^`	correspondance de début de chaîne
`$`	correspondance de fin de chaîne
`(?=er)`	*positive lookahead* (recherche positive) établit une correspondance avec tout point où une sous-chaîne qui correspond à `er` débute (uniquement pour les ERA)
`(?!er)`	*negative lookahead* (recherche négative) établit une correspondance avec tout point où aucune sous-chaîne qui correspond à `re` ne débute (uniquement pour les ERA)

Les contraintes « lookahead » ne doivent pas contenir de *références arrières* (voir la Section 9.7.3.3, « Échappement d'expressions rationnelles »), et toutes les parenthèses contenues sont considérées comme non capturantes.

9.7.3.2. Expressions avec crochets

Une *expression entre crochets* est une liste de caractères contenue dans `[]`. Une correspondance est habituellement établie avec tout caractère de la liste (voir cependant plus bas). Si la liste débute par `^`, la correspondance est établie avec tout caractère *non* compris dans la liste. Si deux caractères de la liste sont séparés par un tiret (`-`), il s'agit d'un raccourci pour représenter tous les caractères compris entre ces deux-là, c'est-à-dire qu'en ASCII, `[0-9]` correspond à tout chiffre. Deux séquences ne peuvent pas partager une limite, par exemple `a-c-e`. Les plages étant fortement liées à la séquence de tri (*collate*), il est recommandé de ne pas les utiliser dans les programmes portables.

Un `]` peut être inclus dans la liste s'il en est le premier caractère (éventuellement précédé de `^`). Un `-` peut être inclus dans la liste s'il en est le premier ou le dernier caractère ou s'il est la deuxième borne d'une plage. Un `-` peut être utilisé comme première borne d'une plage s'il est entouré par `[.` et `.]` et devient de ce fait un élément d'interclassement (*collating element*). À l'exception de ces caractères, des combinaisons utilisant `[` (voir les paragraphes suivants) et des échappements (uniquement pour les ERA), tous les autres caractères spéciaux perdent leur signification spéciale à l'intérieur d'une expression entre crochets. En particulier, `\` n'est pas spécial lorsqu'il suit les règles des ERE ou des ERB bien qu'il soit spécial (en tant qu'introduction d'un échappement) dans les ERA.

Dans une expression entre crochets, un élément d'interclassement (un caractère, une séquence de caractères multiples qui s'interclasse comme un élément unique, ou le nom d'une séquence d'interclassement) entouré de `[.` et `.]` représente la séquence de caractères de cet élément d'interclassement. La séquence est un élément unique de la liste dans l'expression entre crochets. Une expression entre crochets contenant un élément d'interclassement multi-caractères peut donc correspondre à plusieurs caractères (par exemple, si la séquence d'interclassement inclut un élément d'interclassement `ch`, alors l'ER `[[.ch.]]*c` établit une correspondance avec les cinq premiers caractères de `chchcc`.

> **Note**
>
> PostgreSQL™ n'a pas, à ce jour, d'éléments d'interclassement multi-caractères. L'information portée ici décrit un éventuel comportement futur.

Dans une expression entre crochets, un élément d'interclassement écrit entre `[=` et `=]` est une classe d'équivalence qui représente les séquences de caractères de tous les éléments d'interclassement équivalents à celui-là, lui compris. (En l'absence d'élément d'interclassement équivalent, le traitement correspond à celui obtenu avec les délimiteurs `[.` et `.]`). Par exemple, si `o` et `^` sont les membres d'une classe d'équivalence, alors `[[=o=]]`, `[[=^=]]` et `[o^]` sont tous synonymes. Une classe d'équivalence ne peut pas être borne d'une plage.

Dans une expression entre crochets, le nom d'une classe de caractères écrit entre `[:` et `:]` représente la liste de tous les caractères appartenant à cette classe. Les noms de classes de caractères standard sont `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. Ils correspondent aux classes de caractères définies dans ctype(3). Une locale peut en fournir d'autres. Une classe de caractères ne peut pas être utilisée comme borne d'une plage.

Il existe deux cas spéciaux d'expressions entre crochets : les expressions entre crochets `[[:<:]]` et `[[:>:]]` sont des contraintes, qui établissent une correspondance avec des chaînes vides respectivement au début et à la fin d'un mot. Un mot est défini comme une séquence de caractères de mot qui n'est ni précédée ni suivie de caractères de mot. Un caractère de mot est un caractère `alnum` (comme défini par ctype(3)) ou un tiret bas. C'est une extension, compatible avec, mais non spécifiée dans POSIX 1003.2, et devant être utilisée avec précaution dans les logiciels conçus pour être portables sur d'autres systèmes. Les échappements de contraintes décrits ci-dessous sont généralement préférables (ils ne sont pas plus standard mais certainement plus simples à saisir).

9.7.3.3. Échappement d'expressions rationnelles

Les *échappements* sont des séquences spéciales débutant avec `\` suivi d'un caractère alphanumérique. Il existe plusieurs sortes d'échappements : entrée de caractère, raccourci de classe, échappement de contraintes et rétro-références. Un `\` suivi d'un caractère alphanumérique qui ne constitue pas un échappement valide est illégal dans une ERA. Pour les ERE, il n'y pas d'échappement : en dehors d'une expression entre crochets, un `\` suivi d'un caractère alphanumérique représente simplement ce caractère (comme ordinaire) et, à l'intérieur d'une expression entre crochets, `\` est un caractère ordinaire. (C'est dans ce dernier cas que se situe réellement l'incompatibilité entre les ERE et les ERA.)

Les *échappements de caractère* (*character-entry escapes*) permettent d'indiquer des caractères non affichables et donc indésirables dans les ER. Ils sont présentés dans le Tableau 9.15, « Échappements de caractère dans les expressions rationnelles ».

Les *échappements de raccourci de classe* (*class-shorthand escapes*) fournissent des raccourcis pour certaines classes de caractères communément utilisées. Ils sont présentés dans le Tableau 9.16, « Échappement de raccourcis de classes dans les expressions ra-

tionnelles ».

Un *échappement de contrainte* (*constraint escape*) est une contrainte, qui correspond à la chaîne vide sous certaines conditions, écrite comme un échappement. Ces échappements sont présentés dans le Tableau 9.17, « Échappements de contrainte dans les expressions rationnelles ».

Une *rétro-référence* (*back reference*) (`\n`) correspond à la même chaîne que la sous-expression entre parenthèses précédente indiquée par le nombre *n* (voir le Tableau 9.18, « Rétro-références dans les expressions rationnelles »). Par exemple, `([bc])\1` peut correspondre à `bb` ou `cc`, mais ni à `bc` ni à `cb`. La sous-expression doit précéder complètement la référence dans l'ER. Les sous-expressions sont numérotées dans l'ordre des parenthèses ouvrantes. Les parenthèses non capturantes ne définissent pas de sous-expressions.

Note

Le symbole `\` qui débute une séquence d'échappement doit être obligatoirement doublé pour saisir le motif comme une chaîne SQL constante. Par exemple :

```
'123' ~ E'^\\d{3}' true
```

Tableau 9.15. Échappements de caractère dans les expressions rationnelles

Échappement	Description
`\a`	caractère alerte (cloche), comme en C
`\b`	effacement (*backspace*), comme en C
`\B`	synonyme de `\` pour éviter les doublements d'antislash
`\c`*X*	(où *X* est un caractère quelconque) le caractère dont les cinq bits de poids faible sont les mêmes que ceux de *X* et dont tous les autres bits sont à zéro
`\e`	le caractère dont le nom de séquence d'interclassement est `ESC`, ou le caractère de valeur octale 033
`\f`	retour chariot (*form feed*), comme en C
`\n`	retour à la ligne (*newline*), comme en C
`\r`	retour chariot (*carriage return*), comme en C
`\t`	tabulation horizontale, comme en C
`\u`*wxyz*	(où *wxyz* représente exactement quatre chiffres hexadécimaux) le caractère UTF16 (Unicode, 16 bits) `U+`*wxyz* dans l'ordre local des octets
`\U`*stuvwxyz*	(où *stuvwxyz* représente exactement huit chiffres hexadécimaux) réservé pour une extension, hypothétique, de l'Unicode vers le 32 bits
`\v`	tabulation verticale, comme en C
`\x`*hhh*	(où *hhh* représente toute séquence de chiffres hexadécimaux) le caractère dont la valeur hexadécimale est `0x`*hhh* (un simple caractère, peu importe le nombre de chiffres hexadécimaux utilisés)
`\0`	le caractère dont la valeur est `0`
`\`*xy*	(où *xy* représente exactement deux chiffres octaux et n'est pas une *rétro-référence*) le caractère dont la valeur octale est `0`*xy*
`\`*xyz*	(où *xyz* représente exactement trois chiffres octaux et n'est pas une *rétro-référence*) le caractère dont la valeur octale est `0`*xyz*

Les chiffres hexadécimaux sont `0-9`, `a-f` et `A-F`. Les chiffres octaux sont `0-7`.

Les échappements de caractère sont toujours pris comme des caractères ordinaires. Par exemple, `\135` est `]` en ASCII mais `\135` ne termine pas une expression entre crochets.

Tableau 9.16. Échappement de raccourcis de classes dans les expressions rationnelles

Échappement	Description
`\d`	`[[:digit:]]`
`\s`	`[[:space:]]`

Échappement	Description
`\w`	`[[:alnum:]_]` (le tiret bas est inclus)
`\D`	`[^[:digit:]]`
`\S`	`[^[:space:]]`
`\W`	`[^[:alnum:]_]` (le tiret bas est inclus)

Dans les expressions entre crochets, `\d`, `\s`, et `\w` perdent leurs crochets externes et `\D`, `\S` et `\W` ne sont pas autorisés. (Ainsi, par exemple, `[a-c\d]` est équivalent à `[a-c[:digit:]]`. Mais `[a-c\D]`, qui est équivalent à `[a-c^[:digit:]]`, est interdit.)

Tableau 9.17. Échappements de contrainte dans les expressions rationnelles

Échappement	Description
`\A`	n'établit la correspondance qu'au début de la chaîne (voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles » pour comprendre la différence avec `^`)
`\m`	n'établit la correspondance qu'au début d'un mot
`\M`	n'établit la correspondance qu'à la fin d'un mot
`\y`	n'établit la correspondance qu'au début ou à la fin d'un mot
`\Y`	n'établit la correspondance qu'en dehors du début et de la fin d'un mot
`\Z`	n'établit la correspondance qu'à la fin d'une chaîne (voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles » pour comprendre la différence avec `$`)

Un mot est défini selon suivant la spécification de `[[:<:]]` et `[[:>:]]` donnée ci-dessus. Les échappement de contrainte sont interdits dans les expressions entre crochets.

Tableau 9.18. Rétro-références dans les expressions rationnelles

Échappement	Description
`\`*`m`*	(où *`m`* est un chiffre différent de zéro) référence à la *`m`*-ième sous-expression
`\`*`mnn`*	(où *`m`* est un chiffre différent de zéro et *`nn`* quelques chiffres supplémentaires, et la valeur décimale *`mnn`* n'est pas plus grande que le nombre de parenthèses fermantes capturantes vues jusque là) référence à la *`mnn`*-ième sous-expression

> **Note**
>
> Une ambiguïté persiste entre les échappements de caractère octal et les rétro-références. Cette ambiguïté est résolue par les heuristiques suivantes, comme montré ci-dessus. Un zéro en début de chaîne indique toujours un échappement octal. Un caractère seul différent de zéro, qui n'est pas suivi d'un autre caractère, est toujours pris comme une rétro-référence. Une séquence à plusieurs chiffres qui ne débute pas par zéro est prise comme une référence si elle suit une sous-expression utilisable (c'est-à-dire que le nombre est dans la plage autorisée pour les rétro-références). Dans le cas contraire, il est pris comme nombre octal.

9.7.3.4. Métasyntaxe des expressions rationnelles

En plus de la syntaxe principale décrite ci-dessus, il existe quelques formes spéciales et autres possibilités syntaxiques.

Une ER peut commencer avec un des deux préfixes *director* spéciaux. Si une ER commence par `***:`, le reste de l'ER est considéré comme une ERA. (Ceci n'a normalement aucun effet dans PostgreSQL™ car les ER sont supposées être des ERA mais il a un effet si le mode ERE ou BRE a été spécifié par le paramètre *`flags`* à une fonction d'expression rationnelle.) Si une ER commence par `***=`, le reste de l'ER est considéré comme une chaîne littérale, tous les caractères étant considérés ordinaires.

Une ERA peut débuter par des *options intégrées* : une séquence `(?`*`xyz`*`)` (où *`xyz`* correspond à un ou plusieurs caractères alphabétiques) spécifie les options affectant le reste de l'ER. Ces options surchargent toutes les options précédemment déterminées -- en particulier, elles peuvent surcharger le comportement sur la sensibilité à la casse d'un opérateur d'une ER ou le paramètre *`flags`* vers une fonction d'expression rationnelle. Les lettres d'options disponibles sont indiquées dans le Tableau 9.19, « Lettres d'option intégrées à une ERA ». Notez que ces mêmes lettres d'option sont utilisées dans les paramètres *`flags`* des fonctions d'expressions

rationnelles.

Tableau 9.19. Lettres d'option intégrées à une ERA

Option	Description
b	le reste de l'ER est une ERB
c	activation de la sensibilité à la casse (surcharge l'opérateur type)
e	le reste de l'ER est une ERE
i	désactivation de la sensibilité à la casse (voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles ») (surcharge l'opérateur type)
m	synonyme historique pour n
n	activation de la sensibilité aux nouvelles lignes (voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles »)
p	activation de la sensibilité partielle aux nouvelles lignes (voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles »)
q	le reste de l'ER est une chaîne littérale (« entre guillemets »), composé uniquement de caractères ordinaires
s	désactivation de la sensibilité aux nouvelles lignes (par défaut)
t	syntaxe compacte (par défaut ; voir ci-dessous)
w	activation de la sensibilité partielle inverse aux nouvelles lignes (« étrange ») (voir la Section 9.7.3.5, « Règles de correspondance des expressions rationnelles »)
x	syntaxe étendue (voir ci-dessous)

Les options intégrées prennent effet à la) qui termine la séquence. Elles ne peuvent apparaître qu'au début d'une ERA (après le directeur ***: s'il y en a un).

En plus de la syntaxe habituelle d'une ER (*compacte*), dans laquelle tous les caractères ont une signification, il existe une syntaxe *étendue*, accessible en signifiant l'option intégrée x. Avec la syntaxe étendue, les caractères espace dans l'ER sont ignorés comme le sont tous les caractères entre un # et le retour-chariot qui suit (ou la fin de l'ER). Ceci permet de mettre en paragraphe et de commenter une ER complexe. Il existe trois exceptions à cette règle de base :

- un caractère espace ou # suivi d'un \ est retenu
- un caractère espace ou # à l'intérieur d'une expression entre crochets est retenu
- caractère espace et commentaires ne peuvent pas apparaître dans les symboles multi-caractères, tels que (?:

Pour cela, les caractères espace sont l'espace, la tabulation, le retour chariot et tout caractère appartenant à la classe de caractère *space*.

Enfin, dans une ERA, en dehors d'expressions entre crochets, la séquence (?#*ttt*) (où *ttt* est tout texte ne contenant pas)) est un commentaire, totalement ignoré. Là encore, cela n'est pas permis entre les caractères des symboles multi-caractères comme (?:. De tels commentaires sont plus un artefact historique qu'une fonctionnalité utile et leur utilisation est obsolète ; on utilise plutôt la syntaxe étendue.

Aucune de ces extensions métasyntaxique n'est disponible si un directeur initial ***= indique que la saisie utilisateur doit être traitée comme une chaîne littérale plutôt que comme une ER.

9.7.3.5. Règles de correspondance des expressions rationnelles

Dans l'hypothèse où une ER peut correspondre à plusieurs sous-chaînes d'une chaîne donnée, l'ER correspond à celle qui apparaît la première dans la chaîne. Si l'ER peut correspondre à plusieurs sous-chaînes à partir de ce point, c'est soit la correspondance la plus longue possible, soit la correspondance la plus courte possible, qui est retenue selon que l'ER est *gourmande* ou *non-gourmande* (*greedy/non-greedy*).

La gourmandise d'une ER est déterminée par les règles suivantes :

- la plupart des atomes, et toutes les contraintes, n'ont pas d'attribut de gourmandise (parce qu'ils ne peuvent, en aucune façon, établir de correspondance avec des quantités variables de texte) ;
- l'ajout de parenthèses autour d'une ER ne change pas sa gourmandise ;

- un atome quantifié avec un quantificateur à répétition fixe (`{m}` ou `{m}?`) a la même gourmandise (éventuellement aucune) que l'atome lui-même ;
- un atome quantifié avec d'autres quantificateurs standard (dont `{m,n}` avec *m* égal à *n*) est gourmand (préfère la plus grande correspondance) ;
- un atome quantifié avec un quantificateur non gourmand (dont `{m,n}?` avec *m* égal à *n*) n'est pas gourmand (préfère la plus courte correspondance) ;
- une branche -- c'est-à-dire une ER dépourvue d'opérateur `|` au sommet -- est aussi gourmande que le premier atome quantifié qu'elle contient qui possède un attribut de gourmandise ;
- une ER constituée au minimum de deux branches connectées par l'opérateur `|` est toujours gourmande.

Les règles ci-dessus associent les attributs de gourmandise non seulement avec les atomes quantifiés individuels, mais aussi avec les branches et les ER complètes qui contiennent des atomes quantifiés. Cela signifie que la correspondance est établie de sorte que la branche, ou l'ER complète, corresponde à la sous-chaîne la plus longue ou la plus courte possible *comme un tout*. Une fois la longueur de la correspondance complète déterminée, la partie de cette correspondance qui établit une correspondance avec une sous-expression particulière est déterminée sur la base de l'attribut de gourmandise de cette sous-expression, priorité étant donnée aux sous-expressions commençant le plus tôt dans l'ER.

Exemple de signification de tout cela :

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Resultat : 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Resultat : 1
```

Dans le premier cas, l'ER dans son intégralité est gourmande parce que `Y*` est gourmand. Il peut établir une correspondance qui débute à `Y` et correspondre à la chaîne la plus longue à partir de là, soit `Y123`. La sortie reprend la partie entre parenthèses, soit `123`. Dans le second cas, l'ER dans son ensemble n'est pas gourmande car `Y*?` ne l'est pas. Il peut établir une correspondance qui débute à `Y` et correspond à la chaîne la plus courte à partir de là, soit `Y1`. La sous-expression `[0-9]{1,3}` est gourmande mais elle ne peut pas changer la décision sur la longueur totale de la correspondance ; elle ne peut donc correspondre qu'à `1`.

En résumé, quand une ER contient à la fois des sous-expressions gourmandes et non gourmandes, la longueur de la correspondance totale est soit aussi longue que possible soit aussi courte que possible, en fonction de l'attribut affecté à l'ER complète. Les attributs assignés aux sous-expressions permettent uniquement de déterminer la partie de la correspondance qu'elles peuvent incorporer les unes par rapport aux autres.

Les quantificateurs `{1,1}` et `{1,1}?` peuvent être utilisés pour forcer, respectivement, la préférence la plus longue (gourmandise) ou la plus courte (retenue), sur une sous-expression ou une ER complète.

Les longueurs de correspondance sont mesurés en caractères, et non en éléments d'interclassement. Une chaîne vide est considérée plus grande que pas de correspondance du tout. Par exemple : `bb*` correspond aux trois caractères du milieu de `abbbc` ; `(week|wee)(night|knights)` correspond aux dix caractères de `weeknights` ; lorsque une correspondance est recherchée entre `(.*).*` et `abc`, la sous-expression entre parenthèses correspond aux trois caractères ; et lorsqu'une correspondance est recherchée entre `(a*)*` et `bc`, à la fois l'ER et la sous-expression entre parenthèses correspondent à une chaîne vide.

Lorsqu'il est précisé que la recherche de correspondance ne tient pas compte de la casse, cela revient à considérer que toutes les distinctions de casse ont disparu de l'alphabet. Quand un caractère alphabétique, pour lequel existent différentes casses, apparaît comme un caractère ordinaire en dehors d'une expression entre crochets, il est en fait transformé en une expression entre crochets contenant les deux casses, c'est-à-dire que `x` devient `[xX]`. Quand il apparaît dans une expression entre crochets, toutes les transformations de casse sont ajoutées à l'expression entre crochets, c'est-à-dire que `[x]` devient `[xX]` et que `[^x]` devient `[^xX]`.

Si la sensibilité aux retours chariots est précisée, `.` et les expressions entre crochets utilisant `^` n'établissent jamais de correspondance avec le caractère de retour à la ligne (de cette façon, les correspondances ne franchissent jamais les retours chariots sauf si l'ER l'explicite), et `^` et `$` établissent une correspondance avec la chaîne vide, respectivement après et avant un retour chariot, en plus d'établir une correspondance respectivement au début et à la fin de la chaîne. Mais les échappements d'ERA `\A` et `\Z` n'établissent toujours de correspondance *qu'*au début ou à la fin de la chaîne.

Si une sensibilité partielle aux retours chariot est indiquée, cela affecte `.` et les expressions entre crochets, comme avec la sensibilité aux retours chariot, mais pas `^` et `$`.

Si une sensibilité partielle inverse aux retours chariot est indiquée, cela affecte `^` et `$`, comme avec la sensibilité aux retours chariot, mais pas `.` et les sous-expressions. Ceci n'est pas très utile mais est toutefois fourni pour des raisons de symétrie.

9.7.3.6. Limites et compatibilité

Aucune limite particulière n'est imposée sur la longueur des ER dans cette implantation. Néanmoins, les programmes prévus pour

être portables ne devraient pas employer d'ER de plus de 256 octets car une implantation POSIX peut refuser d'accepter de telles ER.

La seule fonctionnalité des ERA qui soit incompatible avec les ERE POSIX est le maintien de la signification spéciale de \ dans les expressions entre crochets. Toutes les autres fonctionnalités ERA utilisent une syntaxe interdite, à effets indéfinis ou non spécifiés dans les ERE POSIX ; la syntaxe *** des directeurs ne figure pas dans la syntaxe POSIX pour les ERB et les ERE.

Un grand nombre d'extensions ERA sont empruntées à Perl mais certaines ont été modifiées et quelques extensions Perl ne sont pas présentes. Les incompatibilités incluent \b, \B, le manque de traitement spécial pour le retour à la ligne en fin de chaîne, l'ajout d'expressions entre crochets aux expressions affectées par les correspondances avec retour à la ligne, les restrictions sur les parenthèses et les références dans les contraintes et la sémantique de correspondance chaînes les plus longues/les plus courtes (au lieu de la première rencontrée).

Deux incompatibilités importantes existent entre les syntaxes ERA et ERE reconnues par les versions antérieures à PostgreSQL™ 7.4 :

- dans les ERA, \ suivi d'un caractère alphanumérique est soit un échappement soit une erreur alors que dans les versions précédentes, c'était simplement un autre moyen d'écrire un caractère alphanumérique. Ceci ne devrait pas poser trop de problèmes car il n'y avait aucune raison d'écrire une telle séquence dans les versions plus anciennes ;
- dans les ERA, \ reste un caractère spécial à l'intérieur de [], donc un \ à l'intérieur d'une expression entre crochets doit être écrit \\.

9.7.3.7. Expressions rationnelles élémentaires

Les ERB diffèrent des ERE par plusieurs aspects. Dans les BRE, |, + et ? sont des caractères ordinaires et il n'existe pas d'équivalent pour leur fonctionnalité. Les délimiteurs de frontières sont \{ et \}, avec { et } étant eux-même des caractères ordinaires. Les parenthèses pour les sous-expressions imbriquées sont \(et \), (et) restent des caractères ordinaires. ^ est un caractère ordinaire sauf au début d'une ER ou au début d'une sous-expression entre parenthèses, $ est un caractère ordinaire sauf à la fin d'une ER ou à la fin d'une sous-expression entre parenthèses et * est un caractère ordinaire s'il apparaît au début d'une ER ou au début d'une sous-expression entre parenthèses (après un possible ^). Enfin, les rétro-références à un chiffre sont disponibles, et \< et \> sont des synonymes pour respectivement [[:<:]] et [[:>:]] ; aucun autre échappement n'est disponible dans les BRE.

9.8. Fonctions de formatage des types de données

Les fonctions de formatage de PostgreSQL™ fournissent un ensemble d'outils puissants pour convertir différents types de données (date/heure, entier, nombre à virgule flottante, numérique) en chaînes formatées et pour convertir des chaînes formatées en types de données spécifiques. Le Tableau 9.20, « Fonctions de formatage » les liste. Ces fonctions suivent toutes une même convention d'appel : le premier argument est la valeur à formater et le second argument est un modèle définissant le format de sortie ou d'entrée.

La fonction to_timestamp à un argument est aussi disponible ; elle accepte un argument double precision unique pour convertir une valeur de type epoch Unix en timestamp with time zone (secondes depuis 1970-01-01 00:00:00+00) en timestamp with time zone. (Les types epoch Unix (entier) sont implicitement convertis en double precision.)

Tableau 9.20. Fonctions de formatage

Fonction	Type en retour	Description	Exemple
to_char(timestamp, text)	text	convertit un champ de type timestamp en chaîne	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	convertit un champ de type interval en chaîne	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	convertit un champ de type integer en chaîne	to_char(125, '999')
to_char(double precision, text)	text	convertit un champ de type real/double precision en chaîne	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	convertit un champ de type numeric en chaîne	to_char(-125.8, '999D99S')
to_date(text, text)	date	convertit une chaîne en date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text,	numeric	convertit une chaîne en champ	to_number('12,454.8-',

Fonction	Type en retour	Description	Exemple
text)		de type numeric	'99G999D9S')
to_timestamp(text, text)	timestamp with time zone	convertit une chaîne string en champ de type timestamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
(doubl to_timestampe precision)	timestamp with time zone	convertit une valeur de type epoch UNIX en valeur de type timestamp	to_timestamp(1284352323)

Dans une chaîne de motif pour `to_char`, il existe certains motifs qui sont reconnus et remplacés avec des données correctement formatées basées sur la valeur. Tout texte qui n'est pas un motif est copié sans modification. De façon similaire, dans toute chaîne de motif en entrée (tout sauf `to_char`), les motifs identifient les valeurs à fournir à la chaîne de données en entrée.

Le Tableau 9.21, « Modèles pour le formatage de champs de type date/heure » affiche les motifs disponibles pour formater les valeurs de types date et heure.

Tableau 9.21. Modèles pour le formatage de champs de type date/heure

Modèle	Description
HH	heure du jour (01-12)
HH12	heure du jour (01-12)
HH24	heure du jour (00-23)
MI	minute (00-59)
SS	seconde (00-59)
MS	milliseconde (000-999)
US	microseconde (000000-999999)
SSSS	secondes écoulées depuis minuit (0-86399)
AM ou am ou PM ou pm	indicateur du méridien (sans point)
A.M. ou a.m. ou P.M. ou p.m.	indicateur du méridien (avec des points)
am ou a.m. ou pm ou p.m.	indicateur du méridien (en minuscules)
Y,YYY	année (quatre chiffres et plus) avec virgule
YYYY	année (quatre chiffres et plus)
YYY	trois derniers chiffres de l'année
YY	deux derniers chiffres de l'année
Y	dernier chiffre de l'année
IYYY	année ISO (quatre chiffres ou plus)
IYY	trois derniers chiffres de l'année ISO
IY	deux derniers chiffres de l'année ISO
I	dernier chiffre de l'année ISO
BC, bc, AD ou ad	indicateur de l'ère (sans point)
B.C., b.c., A.D. ou a.d.	indicateur de l'ère (avec des points)
MONTH	nom complet du mois en majuscules (espaces de complètement pour arriver à neuf caractères)
Month	nom complet du mois en casse mixte (espaces de complètement pour arriver à neuf caractères)
month	nom complet du mois en minuscules (espaces de complètement pour arriver à neuf caractères)
MON	abréviation du nom du mois en majuscules (trois caractères en anglais, la longueur des versions localisées peut varier)
Mon	abréviation du nom du mois avec la première lettre en majuscule et les deux autres en minuscule (trois caractères en anglais, la longueur des versions localisées peut varier)
mon	abréviation du nom du mois en minuscules (trois caractères en anglais, la longueur des versions locali-

Modèle	Description
	sées peut varier)
MM	numéro du mois (01-12)
DAY	nom complet du jour en majuscules (espaces de complètement pour arriver à neuf caractères)
Day	nom complet du jour avec la première lettre en majuscule et les deux autres en minuscule (espaces de complètement pour arriver à neuf caractères)
day	nom complet du jour en minuscules (espaces de complètement pour arriver à neuf caractères)
DY	abréviation du nom du jour en majuscules (trois caractères en anglais, la longueur des versions localisées peut varier)
Dy	abréviation du nom du jour avec la première lettre en majuscule et les deux autres en minuscule (trois caractères en anglais, la longueur des versions localisées peut varier)
dy	abréviation du nom du jour en minuscules (trois caractères en anglais, la longueur des versions localisées peut varier)
DDD	jour de l'année (001-366)
IDDD	jour ISO de l'année (001-371 ; le jour 1 de l'année est le lundi de la première semaine ISO.)
DD	jour du mois (01-31)
D	jour de la semaine du dimanche (1) au samedi (7)
ID	jour ISO de la semaine du lundi (1) au dimanche (7)
W	numéro de semaine du mois, de 1 à 5 (la première semaine commence le premier jour du mois)
WW	numéro de la semaine dans l'année, de 1 à 53 (la première semaine commence le premier jour de l'année)
IW	numéro ISO de la semaine dans l'année (01 - 53 ; le premier jeudi de la nouvelle année est dans la semaine 1)
CC	siècle (deux chiffres) (le 21è siècle commence le 1er janvier 2001)
J	Jour dans le calendrier Julien (nombre de jours depuis le 24 novembre -4714 à minuit)
Q	trimestre (ignoré par to_date and to_timestamp)
RM	mois en majuscule en nombre romain (I-XII ; I étant janvier) (en majuscules)
rm	mois en minuscule en nombre romain (i-xii; i étant janvier) (en minuscules)
TZ	nom en majuscule du fuseau horaire (en majuscules)
tz	nom en minuscule du fuseau horaire (en minuscules)

Les modificateurs peuvent être appliqués à tous les motifs pour en changer le comportement. Par exemple, FMMonth est le motif Month avec le modificateur FM. Le Tableau 9.22, « Modificateurs de motifs pour le formatage des dates/heures » affiche les modificateurs de motifs pour le formatage des dates/heures.

Tableau 9.22. Modificateurs de motifs pour le formatage des dates/heures

Modificateur	Description	Exemple
préfixe FM	mode remplissage (*Fill Mode*) (supprime les espaces et les zéros de complétion)	FMMonth
suffixe TH	suffixe du nombre ordinal en majuscules, c'est-à-dire 12TH	DDTH
suffixe th	suffixe du nombre ordinal en minuscules, c'est-à-dire 12th	DDth
préfixe FX	option globale de format fixe (voir les notes d'utilisation)	FX Month DD Day
préfixe TM	mode de traduction (affiche les noms des jours et mois localisés en fonction de lc_time)	TMMonth
suffixe SP	mode épelé (*Spell Mode*) (non implanté)	DDSP

Notes d'utilisation pour le formatage date/heure :

- FM supprime les zéros de début et les espaces de fin qui, autrement, sont ajoutés pour fixer la taille du motif de sortie ; Dans

PostgreSQL™, FM modifie seulement la prochaine spécification alors qu'avec Oracle, FM affecte toutes les spécifications suivantes et des modifieurs FM répétés bascule l'activation du mode de remplissage.

- TM n'inclut pas les espaces de compéltion en fin de chaîne ;
- to_timestamp et to_date ignorent les espaces multiples de la chaîne en entrée si l'option FX n'est pas utilisée. Par exemple, to_timestamp('2000 JUN', 'YYYY MON') fonctionne mais to_timestamp('2000 JUN', 'FXYYYY MON') renvoie une erreur car to_timestamp n'attend qu'une seule espace ; FX doit être indiqué comme premier élément du modèle.
- il est possible d'insérer du texte ordinaire dans les modèles to_char. il est alors littéralement remis en sortie. Une sous-chaîne peut être placée entre guillemets doubles pour forcer son interprétation en tant que libellé même si elle contient des mots clés de motif. Par exemple, dans '"Hello Year "YYYY', les caractères YYYY sont remplacés par l'année mais l'Y isolé du mot Year ne l'est pas ; Dans to_date, to_number et to_timestamp, les chaînes entre guillemets doubles ignorent le nombre de caractères en entrée contenus dans la chaîne, par exemple "XX" ignorent les deux caractères en entrée.
- pour afficher un guillemet double dans la sortie, il faut le faire précéder d'un antislash. E'\\"YYYY Month\\"', par exemple. (Deux antislashs sont nécessaire parce que l'antislash a déjà une signification spéciale lors de l'utilisation de la syntaxe d'échappement des chaînes) ;
- la conversion YYYY d'une chaîne en champ de type timestamp ou date comporte une restriction avec les années à plus de quatre chiffres. Il faut alors utiliser un modèle ou un caractère non-numérique après YYYY, sans quoi l'année est toujours interprétée sur quatre chiffres. Par exemple, pour l'année 20000 : to_date('200001131', 'YYYYMMDD') est interprété comme une année à quatre chiffres ; il faut alors utiliser un séparateur non décimal après l'année comme to_date('20000-1131', 'YYYY-MMDD') ou to_date('20000Nov31', 'YYYYMonDD') ;
- dans les conversions de chaîne en timestamp ou date, le champ CC (siècle) est ignoré s'il y a un champ YYY, YYYY ou Y,YYY. Si CC est utilisé avec YY ou Y, alors l'année est calculée comme (CC-1)*100+YY ;
- Une date de semaine ISO (distinct de la date grégorienne) peut être passée à to_timestamp et to_date de deux façons :
  - Année, semaine et jour de la semaine. Par exemple, to_date('2006-42-4', 'IYYY-IW-ID') renvoie la date 2006-10-19. En cas d'omission du jour de la semaine, lundi est utilisé.
  - Année et jour de l'année. Par exemple, to_date('2006-291', 'IYYY-IDDD') renvoie aussi 2006-10-19.

  Essayer de construire une date en utilisant un mix de champs de semaine ISO et de date grégorienne n'a pas de sens et renverra du coup une erreur. Dans le contexte d'une année ISO, le concept d'un « mois » ou du « jour d'un mois » n'a pas de signification. Dans le contexte d'une année grégorienne, la semaine ISO n'a pas de signification. Les utilisateurs doivent éviter de mixer les spécifications grégoriennes et les dates ISO.
- les valeurs en millisecondes (MS) et microsecondes (US) dans une conversion de chaîne en champ de type timestamp sont utilisées comme partie décimale des secondes. Par exemple, to_timestamp('12:3', 'SS:MS') n'est pas 3 millisecondes mais 300 car la conversion le compte comme 12 + 0,3 secondes. Cela signifie que pour le format SS:MS, les valeurs d'entrée 12:3, 12:30 et 12:300 indiquent le même nombre de millisecondes. Pour obtenir trois millisecondes, il faut écrire 12:003 que la conversion compte comme 12 + 0,003 = 12,003 secondes.

  Exemple plus complexe : to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US') représente 15 heures, 12 minutes et (2 secondes + 20 millisecondes + 1230 microsecondes =) 2,021230 secondes ;
- la numérotation du jour de la semaine de to_char(..., 'ID') correspond à la fonction extract(isodow from ...) mais to_char(..., 'D') ne correspond pas à la numération des jours de extract(dow from ...).
- to_char(interval) formate HH et HH12 comme indiqué dans une horloge sur 12 heures, c'est-à-dire que l'heure 0 et l'heure 36 sont affichées 12, alors que HH24 affiche la valeur heure complète, qui peut même dépasser 23 pour les

Le Tableau 9.23, « Motifs de modèle pour le formatage de valeurs numériques » affiche les motifs de modèle disponibles pour le formatage des valeurs numériques.

Tableau 9.23. Motifs de modèle pour le formatage de valeurs numériques

Motif	Description
9	valeur avec le nombre indiqué de chiffres
0	valeur avec des zéros de début de chaîne
. (point)	point décimal
, (virgule)	séparateur de groupe (milliers)

Motif	Description
PR	valeur négative entre chevrons
S	signe accroché au nombre (utilise la locale)
L	symbole monétaire (utilise la locale)
D	point décimal (utilise la locale)
G	séparateur de groupe (utilise la locale)
MI	signe moins dans la position indiquée (si le nombre est inférieur à 0)
PL	signe plus dans la position indiquée (si le nombre est supérieur à 0)
SG	signe plus/moins dans la position indiquée
RN	numéro romain (saisie entre 1 et 3999)
TH ou th	suffixe du nombre ordinal
V	décalage du nombre indiqué de chiffres (voir les notes)
EEEE	exposant pour la notation scientifique

Notes d'utilisation pour le formatage des nombres :

- un signe formaté à l'aide de SG, PL ou MI n'est pas ancré au nombre ; par exemple, `to_char(-12, 'S9999')` produit `'  -12'` mais `to_char(-12, 'MI9999')` produit `'-  12'`. L'implantation Oracle n'autorise pas l'utilisation de MI devant 9, mais requiert plutôt que 9 précède MI ;
- 9 est transformé en valeur avec le même nombre de chiffres qu'il y a de 9. Si un chiffre n'est pas disponible, il est remplacé par une espace ;
- TH ne convertit pas les valeurs inférieures à zéro et ne convertit pas les nombres fractionnels ;
- PL, SG et TH sont des extensions PostgreSQL™ ;
- V multiplie effectivement les valeurs en entrée par `10^n`, où *n* est le nombre de chiffres qui suit V. `to_char` ne supporte pas l'utilisation de V combiné avec un point décimal (donc `99.9V99` n'est pas autorisé).
- EEEE (notation scientifique) ne peut pas être utilisé en combinaison avec un des autres motifs de formatage ou avec un autre modifieur, en dehors des motifs chiffre et de point décimal, et doit être placé à la fin de la chaîne de format (par exemple, `9.99EEEE` est valide).

Certains modificateurs peuvent être appliqués à un motif pour modifier son comportement. Par exemple, FM9999 est le motif 9999 avec le modificateur FM. Tableau 9.24, « Modifications de motifs pour le formatage numérique » affiche les motifs pour le formatage numérique.

Tableau 9.24. Modifications de motifs pour le formatage numérique

Modificateur	Description	Exemple
préfixe FM	mode de remplissuage (supprime les blancs et zéris avant et après la chaîne)	FM9999
suffixe TH	suffixe d'un nombre ordinal en majuscule	999TH
suffixe th	suffixe d'un nombre ordinal en minuscule	999th

Le Tableau 9.25, « Exemples avec `to_char` » affiche quelques exemples de l'utilisation de la fonction `to_char`.

Tableau 9.25. Exemples avec `to_char`

Expression	Résultat
`to_char(current_timestamp, 'Day, DD  HH12:MI:SS')`	`'Tuesday  , 06  05:39:18'`
`to_char(current_timestamp, 'FMDay, FMDD  HH12:MI:SS')`	`'Tuesday, 6  05:39:18'`
`to_char(-0.1, '99.99')`	`'  -.10'`
`to_char(-0.1, 'FM9.99')`	`'-.1'`
`to_char(0.1, '0.9')`	`' 0.1'`

Expression	Résultat
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. Fonctions et opérateurs sur date/heure

Le Tableau 9.27, « Fonctions date/heure » affiche les fonctions disponibles pour le traitement des valeurs date et heure. Les détails sont présentés dans les sous-sections qui suivent. Le Tableau 9.26, « Opérateurs date/heure » illustre les comportements des opérateurs arithmétiques basiques (+, *, etc.). Pour les fonctions de formatage, on peut se référer à la Section 9.8, « Fonctions de formatage des types de données ». Il est important d'être familier avec les informations de base concernant les types de données date/heure de la Section 8.5, « Types date/heure ».

Toutes les fonctions et opérateurs décrits ci-dessous qui acceptent une entrée de type time ou timestamp acceptent deux variantes : une avec time with time zone ou timestamp with time zone et une autre avec time without time zone ou timestamp without time zone. Ces variantes ne sont pas affichées séparément. De plus, les opérateurs + et * sont commutatifs (par exemple, date + integer et integer + date) ; une seule possibilité est présentée ici.

Tableau 9.26. Opérateurs date/heure

Opérateur	Exemple	Résultat
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (jours)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Tableau 9.27. Fonctions date/heure

Fonction	Code de retour	Description	Exemple	Résultat
age(timestamp, timestamp)	interval	Soustrait les arguments, ce qui produit un résultat « symbolique » en années, mois, jours	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	Soustrait à la date courante (current_date à minuit)	age(timestamp '1957-06-13')	43 years 8 mons 3 days
clock_timestamp()	timestamp with time zone	Date et heure courantes (change pendant l'exécution de l'instruction) ; voir la Section 9.9.4, « Date/Heure courante »		
current_date	date	Date courante ; voir la Section 9.9.4, « Date/Heure courante »		1
current_time	time with time zone	Heure courante ; voir la Section 9.9.4, « Date/Heure courante »		
current_timestamp	timestamp with time zone	Date et heure courantes (début de la transaction en cours) ; voir la Section 9.9.4, « Date/Heure courante »		

Fonction	Code de retour	Description	Exemple	Résultat
`date_part(text, timestamp)`	double precision	Obtenir un sous-champ (équivalent à `extract`) ; voir la Section 9.9.1, « `EXTRACT`, `date_part` »	`date_part('hour', timestamp '2001-02-16 20:38:40')`	`20`
`date_part(text, interval)`	double precision	Obtenir un sous-champ (équivalent à `extract`) ; voir la Section 9.9.1, « `EXTRACT`, `date_part` »	`date_part('month', interval '2 years 3 months')`	`3`
`date_trunc(text, timestamp)`	timestamp	Tronquer à la précision indiquée ; voir aussi la Section 9.9.2, « `date_trunc` »	`date_trunc('hour', timestamp '2001-02-16 20:38:40')`	`2001-02-16 20:00:00`
`extract(`*`field`*` from timestamp)`	double precision	Obtenir un sous-champ ; voir la Section 9.9.1, « `EXTRACT`, `date_part` »	`extract(hour from timestamp '2001-02-16 20:38:40')`	`20`
`extract(`*`field`*` from interval)`	double precision	Obtenir un sous-champ ; voir la Section 9.9.1, « `EXTRACT`, `date_part` »	`extract(month from interval '2 years 3 months')`	`3`
`isfinite(date)`	boolean	Test si la date est finie (donc différent de +/-infinity)	`isfinite(date '2001-02-16')`	`true`
`isfinite(timestamp)`	boolean	Teste si l'estampille temporelle est finie (donc différent de +/-infinity)	`isfinite(timestamp '2001-02-16 21:28:30')`	`true`
`isfinite(interval)`	boolean	Teste si l'intervalle est fini	`isfinite(interval '4 hours')`	`true`
`justify_days(interval)`	interval	Ajuste l'intervalle pour que les périodes de 30 jours soient représentées comme des mois	`justify_days(interval '35 days')`	`1 mon 5 days`
`justify_hours(interval)`	interval	Ajuste l'intervalle pour que les périodes de 24 heures soient représentées comme des jours	`justify_hours(interval '27 hours')`	`1 day 03:00:00`
`justify_interval(interval)`	interval	Ajuste l'intervalle en utilisant `justify_days` et `justify_hours`, avec des signes supplémentaires d'ajustement	`justify_interval(interval '1 mon -1 hour')`	`29 days 23:00:00`
`localtime`	time	Heure du jour courante ; voir la Section 9.9.4, « Date/Heure courante »		
`localtimestamp`	timestamp	Date et heure courantes (début de la transaction) ; voir la Section 9.9.4, « Date/Heure courante »		
`now()`	timestamp with time zone	Date et heure courantes (début de la transaction) ; voir la Section 9.9.4, « Date/Heure courante »		
`statement_timestamp()`	timestamp with time zone	Date et heure courantes (début de l'instruction en cours) ; voir Section 9.9.4,		

Fonction	Code de retour	Description	Exemple	Résultat
		« Date/Heure courante »		
`timeofday()`	text	Date et heure courantes (comme `clock_timestamp` mais avec une chaîne de type text) ; voir la Section 9.9.4, « Date/Heure courante »		
`transaction_timestamp()`	timestamp with time zone	Date et heure courantes (début de la transaction en cours) ; voir Section 9.9.4, « Date/Heure courante »		

En plus de ces fonctions, l'opérateur SQL `OVERLAPS` est supporté :

```
( début1, fin1 ) OVERLAPS ( début2, fin2 )
( début1, longueur1 ) OVERLAPS ( début2, longueur2 )
```

Cette expression renvoie vrai (true) lorsque les deux périodes de temps (définies par leurs extrémités) se chevauchent, et faux dans le cas contraire. Les limites peuvent être indiquées comme des paires de dates, d'heures ou de timestamps ; ou comme une date, une heure ou un timestamp suivi d'un intervalle. Quand une paire de valeur est fournie, soit le début soit la fin doit être écrit en premier ; `OVERLAPS` prend automatiquement la valeur la plus ancienne dans la paire comme valeur de départ. Chaque période de temps est considéré représentant l'intervalle à moitié ouvert *`début1`* `<=` *`longueur1`* `<` *`fin2`*, sauf si *`début1`* et *`fin2`* sont identiques, auquel cas ils représentent une instant précis. Cela signifie en fait que deux périodes de temps avec seulement un point final en commun ne se surchargent pas.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Résultat :
true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Résultat :
false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Résultat : false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

Lors de l'ajout (ou de la soustraction) d'une valeur de type interval à une valeur de type timestamp with time zone, le composant jours incrémente (ou décremente) la date du timestamp with time zone du nombre de jours indiqués. Avec les modifications occasionnées par les changements d'heure (avec un fuseau horaire de session qui reconnaît DST), cela signifie qu'un `interval '1 day'` n'est pas forcément égal à un `interval '24 hours'`. Par exemple, avec un fuseau horaire configuré à `CST7CDT`, `timestamp with time zone '2005-04-02 12:00-07' + interval '1 day'` produit un `timestamp with time zone '2005-04-03 12:00-06'` alors qu'ajouter `interval '24 hours'` au timestamp with time zone initial produit un `timestamp with time zone '2005-04-03 13:00-06'` parce qu'il y a un changement d'heure le `2005-04-03 02:00` pour le fuseau horaire `CST7CDT`.

Il peut y avoir une ambiguïté dans le nombre de `months` retournés par `age` car les mois n'ont pas tous le même nombre de jours. L'approche de PostgreSQL™ utilise le mois de la date la plus ancienne lors du calcul de mois partiels. Par exemple, `age('2004-06-01', '2004-04-30')` utilise avril pour ramener `1 mon 1 day`, alors qu'utiliser mai aurait ramener `1 mon 2 days` car mai a 31 jours alors qu'avril n'en a que 30.

9.9.1. EXTRACT, `date_part`

```
EXTRACT (champ FROM source)
```

La fonction `extract` récupère des sous-champs de valeurs date/heure, tels que l'année ou l'heure. *`source`* est une expression de valeur de type timestamp, time ou interval. (Les expressions de type date sont converties en timestamp et peuvent aussi être utilisées.) *`champ`* est un identifiant ou une chaîne qui sélectionne le champ à extraire de la valeur source. La fonction `extract` renvoie des valeurs de type double precision. La liste qui suit présente les noms de champs valides :

`century`

Le siècle.

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Résultat : 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 21
```

Le premier siècle commence le 1er janvier de l'an 1 (0001-01-01 00:00:00 AD) bien qu'ils ne le savaient pas à cette époque. Cette définition s'applique à tous les pays qui utilisent le calendrier Grégorien. Le siècle 0 n'existe pas. On passe de -1 siècle à 1 siècle. En cas de désaccord, adresser une plainte à : Sa Sainteté le Pape, Cathédrale Saint-Pierre de Rome, Vatican.

Les versions de PostgreSQL™ antérieures à la 8.0 ne suivaient pas la numérotation conventionnelle des siècles mais renvoyaient uniquement le champ année divisée par 100.

`day`

Le champ jour (du mois) : de 1 à 31.

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 16
```

`decade`

Le champ année divisé par 10.

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 200
```

`dow`

Le jour de la semaine du dimanche (`0`) au samedi (`6`).

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 5
```

Cette numérotation du jour de la semaine est différente de celle de la fonction `to_char(..., 'D')`.

`doy`

Le jour de l'année (de 1 à 365/366).

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 47
```

`epoch`

Pour les valeurs de type date et timestamp, le nombre de secondes depuis le 1er janvier 1970 (exactement depuis le 1970-01-01 00:00:00 UTC). Ce nombre peut être négatif. Pour les valeurs de type interval, il s'agit du nombre total de secondes dans l'intervalle.

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Résultat :
982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Résultat :
442800
```

Convertir une valeur epoch en valeur de type date/heure :

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second';
```

(La fonction `to_timestamp` encapsule la conversion ci-dessus.)

`hour`

Le champ heure (0 - 23).

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 20
```

`isodow`

Le jour de la semaine du lundi (`1`) au dimanche (`7`).

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Résultat : 7
```

Ceci est identique à `dow` sauf pour le dimanche. Cela correspond à la numérotation du jour de la semaine suivant le format ISO 8601.

`isoyear`
L'année ISO 8601 dans laquelle se trouve la date (non applicable aux intervalles).

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Résultat : 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Résultat : 2006
```

Chaque année ISO commence avec le lundi de la semaine contenant le 4 janvier, donc soit début janvier, soit fin décembre. L'année ISO peut être différente de l'année grégorienne. Voir le champ `week` pour plus d'informations.

Ce champ n'est disponible que depuis la version 8.3 de PostgreSQL.

`microseconds`
Le champ secondes, incluant la partie décimale, multiplié par 1 000 000. Cela inclut l'intégralité des secondes.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Résultat :
28500000
```

`millennium`
Le millénaire.

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 3
```

Les années 1900 sont dans le second millénaire. Le troisième millénaire commence le 1er janvier 2001.

Les versions de PostgreSQL™ antérieures à la 8.0 ne suivaient pas les conventions de numérotation des millénaires mais renvoyaient seulement le champ année divisé par 1000.

`milliseconds`
Le champ secondes, incluant la partie décimale, multiplié par 1000. Cela inclut l'intégralité des secondes.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Résultat :
28500
```

`minute`
Le champ minutes (0 - 59).

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 38
```

`month`
Pour les valeurs de type timestamp, le numéro du mois dans l'année (de 1 à 12) ; pour les valeurs de type interval, le nombre de mois, modulo 12 (0 - 11).

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Résultat : 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Résultat : 1
```

`quarter`
Le trimestre (1 - 4) dont le jour fait partie.

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 1
```

`second`
Le champs secondes, incluant la partie décimale (0 - 59[1]).

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Résultat :
28.5
```

`timezone`
Le décalage du fuseau horaire depuis UTC, mesuré en secondes. Les valeurs positives correspondent aux fuseaux horaires à l'est d'UTC, les valeurs négatives à l'ouest d'UTC.

`timezone_hour`
Le composant heure du décalage du fuseau horaire.

`timezone_minute`
Le composant minute du décalage du fuseau horaire.

`week`
Le numéro de la semaine dans l'année, à laquelle appartient le jour. Par définition (ISO 8601), les semaines commencent le lundi et la première semaine d'une année contient le 4 janvier de cette année. Autrement dit, le premier jeudi d'une année se trouve dans la première semaine de cette année.

Dans la définition ISO, il est possible que les premiers jours de janvier fassent partie de la semaine 52 ou 53 de l'année précédente. Il est aussi possibles que les derniers jours de décembre fassent partie de la première semaine de l'année suivante. Par exemple, `2005-01-01` fait partie de la semaine 53 de l'année 2004 et `2006-01-01` fait partie de la semaine 52 de l'année 2005, alors que `2012-12-31` fait partie de la première semaine de 2013. Il est recommandé d'utiliser le champ `isoyear` avec `week` pour obtenir des résultats cohérents.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 7
```

`year`
Le champ année. Il n'y a pas de `0 AD`, la soustraction d'années `BC` aux années `AD` nécessite donc une attention particulière.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat :
2001
```

La fonction `extract` a pour but principal l'exécution de calculs. Pour le formatage des valeurs date/heure en vue de leur affichage, voir la Section 9.8, « Fonctions de formatage des types de données ».

La fonction `date_part` est modelée sur l'équivalent traditionnel Ingres™ de la fonction `extract` du standard SQL :

```
date_part('champ', source)
```

Le paramètre *`champ`* est obligatoirement une valeur de type chaîne et non pas un nom. Les noms de champ valide pour `date_part` sont les mêmes que pour `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Résultat : 4
```

9.9.2. `date_trunc`

La fonction `date_trunc` est conceptuellement similaire à la fonction `trunc` pour les nombres.

```
date_trunc('champ', source)
```

`source` est une expression de type timestamp ou interval. (Les valeurs de type date et time sont converties automatiquement en, respectivement, timestamp ou interval). *`champ`* indique la précision avec laquelle tronquer la valeur en entrée. La valeur de retour est de type timestamp ou interval avec tous les champs moins significatifs que celui sélectionné positionnés à zéro (ou un pour la date et le mois).

[1] 60 si les secondes d'ajustement (*leap second*) sont implantées par le système d'exploitation.

Les valeurs valides pour *champ* sont :

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

Exemples :

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001-02-16
20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001-01-01
00:00:00
```

9.9.3. AT TIME ZONE

La construction `AT TIME ZONE` permet les conversions d'« estampilles temporelles » (*time stamps*) dans les différents fuseaux horaires. Le Tableau 9.28, « Variantes `AT TIME ZONE` » affiche ses variantes.

Tableau 9.28. Variantes `AT TIME ZONE`

Expression	Type de retour	Description
`timestamp without time zone AT TIME ZONE zone`	timestamp with time zone	Traite l'estampille donnée *without time zone* (sans fuseau), comme située dans le fuseau horaire indiqué.
`timestamp with time zone AT TIME ZONE zone`	timestamp without time zone	Convertit l'estampille donnée *with time zone* (avec fuseau) dans le nouveau fuseau horaire, sans désignation du fuseau.
`time with time zone AT TIME ZONE zone`	time with time zone	Convertit l'heure donnée *with time zone* (avec fuseau) dans le nouveau fuseau horaire.

Dans ces expressions, le fuseau horaire désiré *zone* peut être indiqué comme une chaîne texte (par exemple, `'PST'`) ou comme un intervalle (c'est-à-dire `INTERVAL '-08:00'`). Dans le cas textuel, un nom de fuseau peut être indiqué de toute façon décrite dans Section 8.5.3, « Fuseaux horaires ».

Exemples (en supposant que le fuseau horaire local soit `PST8PDT`) :

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Résultat : 2001-02-16
19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Résultat : 2001-02-16
18:38:40
```

Le premier exemple prend une « estampille temporelle sans fuseau » et l'interprète comme une date MST (UTC-7), qui est ensuite convertie en PST (UTC-8) pour l'affichage. Le second exemple prend une estampille indiquée en EST (UTC-5) et la convertit en heure locale, c'est-à-dire en MST (UTC-7).

La fonction `timezone(zone, timestamp)` est équivalente à la construction conforme au standard SQL, *timestamp* `AT TIME ZONE` *zone*.

9.9.4. Date/Heure courante

PostgreSQL™ fournit diverses fonctions qui renvoient des valeurs relatives aux date et heure courantes. Ces fonctions du standard SQL renvoient toutes des valeurs fondées sur l'heure de début de la transaction en cours :

```
CURRENT_DATE ;
CURRENT_TIME ;
CURRENT_TIMESTAMP ;
CURRENT_TIME(precision) ;
CURRENT_TIMESTAMP(precision) ;
LOCALTIME ;
LOCALTIMESTAMP ;
LOCALTIME(precision) ;
LOCALTIMESTAMP(precision).
```

`CURRENT_TIME` et `CURRENT_TIMESTAMP` délivrent les valeurs avec indication du fuseau horaire ; `LOCALTIME` et `LOCALTIMESTAMP` délivrent les valeurs sans indication du fuseau horaire.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, et `LOCALTIMESTAMP` acceptent un paramètre optionnel de précision. Celui-ci permet d'arrondir le résultat au nombre de chiffres indiqués pour la partie fractionnelle des secondes. Sans ce paramètre de précision, le résultat est donné avec toute la précision disponible.

Quelques exemples :

```
SELECT CURRENT_TIME;
Résultat :
14:39:53.662522-05

SELECT CURRENT_DATE;
Résultat :
2001-12-23

SELECT CURRENT_TIMESTAMP;
Résultat : 2001-12-23
14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Résultat : 2001-12-23
14:39:53.66-05

SELECT LOCALTIMESTAMP;
Résultat : 2001-12-23
14:39:53.662522
```

Comme ces fonctions renvoient l'heure du début de la transaction en cours, leurs valeurs ne changent pas au cours de la transaction. Il s'agit d'une fonctionnalité : le but est de permettre à une même transaction de disposer d'une notion cohérente de l'heure « courante ». Les multiples modifications au sein d'une même transaction portent ainsi toutes la même heure.

Note

D'autres systèmes de bases de données actualisent ces valeurs plus fréquemment.

PostgreSQL™ fournit aussi des fonctions qui renvoient l'heure de début de l'instruction en cours, voire l'heure de l'appel de la fonction. La liste complète des fonctions ne faisant pas partie du standard SQL est :

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` est un peu l'équivalent de `CURRENT_TIMESTAMP` mais est nommé ainsi pour expliciter l'information retournée. `statement_timestamp()` renvoie l'heure de début de l'instruction en cours (plus exactement, l'heure de réception du dernier message de la commande en provenance du client). `statement_timestamp()` et `transaction_timestamp()` renvoient la même valeur pendant la première commande d'une transaction, mais leurs résultats peuvent différer pour les commandes suivantes. `clock_timestamp()` renvoie l'heure courante, et, de ce fait, sa valeur change même à l'intérieur d'une commande SQL unique. `timeofday()` est une fonction historique de PostgreSQL™. Comme `clock_timestamp()`, elle renvoie l'heure courante, mais celle-ci est alors formatée comme une chaîne text et non comme une valeur de type timestamp with time zone. `now()` est l'équivalent traditionnel PostgreSQL™ de `CURRENT_TIMESTAMP`.

Tous les types de données date/heure acceptent aussi la valeur littérale spéciale `now` pour indiquer la date et l'heure courantes (interprétés comme l'heure de début de la transaction). De ce fait, les trois instructions suivantes renvoient le même résultat :

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- utilisation incorrecte avec DEFAULT
```

> **Astuce**
>
> La troisième forme ne doit pas être utilisée pour la spécification de la clause `DEFAULT` à la création d'une table. Le système convertirait `now` en valeur de type timestamp dès l'analyse de la constante. À chaque fois que la valeur par défaut est nécessaire, c'est l'heure de création de la table qui est alors utilisée. Les deux premières formes ne sont pas évaluées avant l'utilisation de la valeur par défaut, il s'agit d'appels de fonctions. C'est donc bien le comportement attendu, l'heure d'insertion comme valeur par défaut, qui est obtenu.

9.9.5. Retarder l'exécution

La fonction suivante permet de retarder l'exécution du processus serveur :

```
pg_sleep(seconds)
```

`pg_sleep` endort le processus de la session courante pendant *`seconds`* secondes. *`seconds`* est une valeur de type double precision, ce qui autorise les délais en fraction de secondes. Par exemple :

```
SELECT pg_sleep(1.5);
```

> **Note**
>
> La résolution réelle de l'intervalle est spécifique à la plateforme ; 0,01 seconde est une valeur habituelle. Le délai dure au minimum celui précisé. Il peut toutefois être plus long du fait de certains facteurs tels que la charge serveur.

> **Avertissement**
>
> Il convient de s'assurer que la session courante ne détient pas plus de verrous que nécessaires lors de l'appel à `pg_sleep`. Dans le cas contraire, d'autres sessions peuvent être amenées à attendre que le processus de retard courant ne termine, ralentissant ainsi tout le système.

9.10. Fonctions de support enum

Pour les types enum (décrit dans Section 8.7, « Types énumération »), il existe plusieurs fonctions qui autorisent une programmation plus claire sans coder en dur les valeurs particulières d'un type enum. Elles sont listées dans Tableau 9.29, « Fonctions de support enum ». Les exemples supposent un type enum créé ainsi :

```
CREATE TYPE couleurs AS ENUM ('rouge', 'orange', 'jaune', 'vert', 'bleu', 'violet');
```

Tableau 9.29. Fonctions de support enum

Fonction	Description	Exemple	Résultat de l'exemple
`enum_first(anyenum)`	Renvoie la première valeur du type enum en entrée	`enum_first(null::couleurs)`	`rouge`
`enum_last(anyenum)`	Renvoie la dernière valeur du type enum en entrée	`enum_last(null::couleurs)`	`violet`
`enum_range(anyenum)`	Renvoie toutes les valeurs du type enum en entrée dans un tableau trié	`enum_range(null::couleurs)`	`{rouge,orange,jaune,vert,bleu,violet}`

Fonction	Description	Exemple	Résultat de l'exemple
`enum_range(anyenum, anyenum)`	Renvoie les éléments entre deux valeurs enum données dans un tableau trié. Les valeurs doivent être du même type enum. Si le premier paramètre est NULL, le résultat se termine avec la dernière valeur du type enum.	`enum_range('orange'::couleurs, 'vert'::couleurs)`	`{orange,jaune,vert}`
		`enum_range(NULL, 'vert'::couleurs)`	`{rouge,orange,jaune,vert}`
		`enum_range('orange'::couleurs, NULL)`	`{orange,jaune,vert,bleu,violet}`

En dehors de la forme à deux arguments de `enum_range`, ces fonctions ne tiennent pas compte de la valeur qui leur est fournie ; elles ne s'attachent qu'au type de donnée déclaré. NULL ou une valeur spécifique du type peut être passée, le résultat est le même. Il est plus commun d'appliquer ces fonctions à la colonne d'une table ou à l'argument d'une fonction qu'à un nom de type en dur, comme le suggèrent les exemples.

9.11. Fonctions et opérateurs géométriques

Les types géométriques point, box, lseg, line, path, polygon et circle disposent d'un large ensemble de fonctions et opérateurs natifs. Ils sont listés dans le Tableau 9.30, « Opérateurs géométriques », le Tableau 9.31, « Fonctions géométriques » et le Tableau 9.32, « Fonctions de conversion de types géométriques ».

Attention

L'opérateur « identique à », `~=`, représente la notion habituelle d'égalité pour les types point, box, polygon et circle. Certains disposent également d'un opérateur `=`, mais `=` ne compare que les égalités d'*aires*. Les autres opérateurs de comparaison scalaires (`<=` et autres) comparent de la même façon des aires pour ces types.

Tableau 9.30. Opérateurs géométriques

Opérateur	Description	Exemple
`+`	Translation	`box '((0,0),(1,1))' + point '(2.0,0)'`
`-`	Translation	`box '((0,0),(1,1))' - point '(2.0,0)'`
`*`	Mise à l'échelle/rotation	`box '((0,0),(1,1))' * point '(2.0,0)'`
`/`	Mise à l'échelle/rotation	`box '((0,0),(2,2))' / point '(2.0,0)'`
`#`	Point ou boîte d'intersection	`'((1,-1),(-1,1))' # '((1,1),(-1,-1))'`
`#`	Nombre de points dans le chemin ou le polygone	`# '((1,0),(0,1),(-1,0))'`
`@-@`	Longueur ou circonférence	`@-@ path '((0,0),(1,0))'`
`@@`	Centre	`@@ circle '((0,0),10)'`
`##`	Point de la seconde opérande le plus proche de la première	`point '(0,0)' ## lseg '((2,0),(0,2))'`
`<->`	Distance entre	`circle '((0,0),1)' <-> circle '((5,0),1)'`
`&&`	Recouvrement ? (Un point en commun renvoie la valeur true.)	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
`<<`	Est strictement à gauche de ?	`circle '((0,0),1)' << circle '((5,0),1)'`
`>>`	Est strictement à droite de ?	`circle '((5,0),1)' >> circle '((0,0),1)'`
`&<`	Ne s'étend pas à droite de ?	`box '((0,0),(1,1))' &< box '((0,0),(2,2))'`

Opérateur	Description	Exemple
`&>`	Ne s'étend pas à gauche de ?	`box '((0,0),(3,3))' &> box '((0,0),(2,2))'`
`<<|`	Est strictement en-dessous de ?	`box '((0,0),(3,3))' <<| box '((3,4),(5,5))'`
`|>>`	Est strictement au-dessus de ?	`box '((3,4),(5,5))' |>> box '((0,0),(3,3))'`
`&<|`	Ne s'étend pas au-dessus de ?	`box '((0,0),(1,1))' &<| box '((0,0),(2,2))'`
`|&>`	Ne s'étend pas en-dessous de ?	`box '((0,0),(3,3))' |&> box '((0,0),(2,2))'`
`<^`	Est en-dessous de (peut toucher) ?	`circle '((0,0),1)' <^ circle '((0,5),1)'`
`>^`	Est au-dessus de (peut toucher) ?	`circle '((0,5),1)' >^ circle '((0,0),1)'`
`?#`	Intersection ?	`lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'`
`?-`	Horizontal ?	`?- lseg '((-1,0),(1,0))'`
`?-`	Sont alignés horizontalement ?	`point '(1,0)' ?- point '(0,0)'`
`?|`	Vertical ?	`?| lseg '((-1,0),(1,0))'`
`?|`	Sont verticalement alignés ?	`point '(0,1)' ?| point '(0,0)'`
`?-|`	Perpendiculaires ?	`lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'`
`?||`	Parallèles ?	`lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'`
`@>`	Contient ?	`circle '((0,0),2)' @> point '(1,1)'`
`<@`	Contenu ou dessus ?	`point '(1,1)' <@ circle '((0,0),2)'`
`~=`	Identique à ?	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

Note

Avant PostgreSQL™ 8.2, les opérateurs `@>` et `<@` s'appelaient respectivement `~` et `@`. Ces noms sont toujours disponibles mais, obsolètes, ils seront éventuellement supprimés.

Tableau 9.31. Fonctions géométriques

Fonction	Type de retour	Description	Exemple
`area (`*`object`*`)`	double precision	aire	`area(box '((0,0),(1,1))')`
`center (`*`object`*`)`	point	centre	`center(box '((0,0),(1,2))')`
`diameter(circle)`	double precision	diamètre du cercle	`diameter(circle '((0,0),2.0)')`
`height(box)`	double precision	taille verticale (hauteur) de la boîte	`height(box '((0,0),(1,1))')`
`isclosed(path)`	boolean	chemin fermé ?	`isclosed(path '((0,0),(1,1),(2,0))' )`
`isopen(path)`	boolean	chemin ouvert ?	`isopen(path`

Fonction	Type de retour	Description	Exemple
			`'[(0,0),(1,1),(2,0)]'`)
`length(object)`	double precision	longueur	`length(path '((-1,0),(1,0))')`
`npoints(path)`	int	nombre de points	`npoints(path '[(0,0),(1,1),(2,0)]'`)
`npoints(polygon)`	int	nombre de points	`npoints(polygon '((1,1),(0,0))')`
`pclose(path)`	path	convertit un chemin en chemin fermé	`pclose(path '[(0,0),(1,1),(2,0)]'`)
`popen(path)`	path	convertit un chemin en chemin ouvert	`popen(path '((0,0),(1,1),(2,0))'`)
`radius(circle)`	double precision	rayon du cercle	`radius(circle '((0,0),2.0)')`
`width(box)`	double precision	taille horizontale (largeur) d'une boîte	`width(box '((0,0),(1,1))')`

Tableau 9.32. Fonctions de conversion de types géométriques

Fonction	Type de retour	Description	Exemple
`box(circle)`	box	cercle vers boîte	`box(circle '((0,0),2.0)')`
`box(point, point)`	box	points vers boîte	`box(point '(0,0)', point '(1,1)')`
`box(polygon)`	box	polygone vers boîte	`box(polygon '((0,0),(1,1),(2,0))'`)
`circle(box)`	circle	boîte vers cercle	`circle(box '((0,0),(1,1))')`
`circle(point, double precision)`	circle	centre et rayon vers cercle	`circle(point '(0,0)', 2.0)`
`circle(polygon)`	circle	polygone vers cercle	`circle(polygon '((0,0),(1,1),(2,0))'`)
`lseg(box)`	lseg	diagonale de boîte vers segment de ligne	`lseg(box '((-1,0),(1,0))')`
`lseg(point, point)`	lseg	points vers segment de ligne	`lseg(point '(-1,0)', point '(1,0)')`
`path(polygon)`	path	polygone vers chemin	`path(polygon '((0,0),(1,1),(2,0))'`)
`point(double precision, double precision)`	point	point de construction	`point(23.4, -44.5)`
`point(box)`	point	centre de la boîte	`point(box '((-1,0),(1,0))')`
`point(circle)`	point	centre du cercle	`point(circle '((0,0),2.0)')`
`point(lseg)`	point	centre de segment de ligne	`point(lseg '((-1,0),(1,0))')`

Fonction	Type de retour	Description	Exemple
`point(polygon)`	point	centre de polygone	`point(polygon '((0,0),(1,1),(2,0))' )`
`polygon(box)`	polygon	boîte vers polygone à quatre points	`polygon(box '((0,0),(1,1))')`
`polygon(circle)`	polygon	cercle vers polygone à 12 points	`polygon(circle '((0,0),2.0)')`
`polygon(`*`npts`*`, circle)`	polygon	cercle vers polygone à *`npts`* points	`polygon(12, circle '((0,0),2.0)')`
`polygon(path)`	polygon	chemin vers polygone	`polygon(path '((0,0),(1,1),(2,0))' )`

Il est possible d'accéder aux deux composants d'un point comme si c'était un tableau avec des index 0 et 1. Par exemple, si `t.p` est une colonne de type point, alors `SELECT p[0] FROM t` récupère la coordonnée X et `UPDATE t SET p[1] = ...` modifie la coordonnée Y. De la même façon, une valeur de type box ou lseg peut être traitée comme un tableau de deux valeurs de type point.

La fonction `area` est utilisable avec les types box, circle et path. Elle ne fonctionne avec le type de données path que s'il n'y a pas d'intersection entre les points du path. Le path `'((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH`, par exemple, ne fonctionne pas. Le path, visuellement identique, `'((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH`, quant à lui, fonctionne. Si les concepts de path avec intersection et sans intersection sont sources de confusion, dessiner les deux path ci-dessus côte-à-côte.

9.12. Fonctions et opérateurs sur les adresses réseau

Le Tableau 9.33, « Opérateurs cidr et inet » affiche les opérateurs disponibles pour les types cidr et inet. Les opérateurs `<<`, `<<=`, `>>` et `>>=` testent l'inclusion de sous-réseau. Ils ne considèrent que les parties réseau des deux adresses, ignorant toute partie hôte, et déterminent si une partie réseau est identique ou consitue un sous-réseau de l'autre.

Tableau 9.33. Opérateurs cidr et inet

Opérateur	Description	Exemple
`<`	est plus petit que	`inet '192.168.1.5' < inet '192.168.1.6'`
`<=`	est plus petit que ou égal à	`inet '192.168.1.5' <= inet '192.168.1.5'`
`=`	est égal à	`inet '192.168.1.5' = inet '192.168.1.5'`
`>=`	est plus grand ou égal à	`inet '192.168.1.5' >= inet '192.168.1.5'`
`>`	est plus grand que	`inet '192.168.1.5' > inet '192.168.1.4'`
`<>`	n'est pas égal à	`inet '192.168.1.5' <> inet '192.168.1.4'`
`<<`	est contenu dans	`inet '192.168.1.5' << inet '192.168.1/24'`
`<<=`	est contenu dans ou égal à	`inet '192.168.1/24' <<= inet '192.168.1/24'`
`>>`	contient	`inet'192.168.1/24' >> inet '192.168.1.5'`
`>>=`	contient ou est égal à	`inet '192.168.1/24' >>= inet '192.168.1/24'`
`~`	bitwise NOT	`~ inet '192.168.1.6'`
`&`	bitwise AND	`inet '192.168.1.6' & inet '0.0.0.255'`
`|`	bitwise OR	`inet '192.168.1.6' | inet '0.0.0.255'`
`+`	addition	`inet '192.168.1.6' + 25`
`-`	soustraction	`inet '192.168.1.43' - 36`
`-`	soustraction	`inet '192.168.1.43' - inet '192.168.1.19'`

Le Tableau 9.34, « Fonctions cidr et inet » affiche les fonctions utilisables avec les types cidr et inet. Les fonctions `abbrev`, `host`, `text` ont principalement pour but d'offrir des formats d'affichage alternatifs.

Tableau 9.34. Fonctions cidr et inet

Fonction	Type de retour	Description	Exemple	Résultat
`abbrev(inet)`	text	format textuel d'affichage raccourci	`abbrev(inet '10.1.0.0/16')`	`10.1.0.0/16`
`abbrev(cidr)`	text	format textuel d'affichage raccourci	`abbrev(cidr '10.1.0.0/16')`	`10.1/16`
`broadcast(inet)`	inet	adresse de broadcast pour le réseau	`broadcast('192.168.1.5/24')`	`192.168.1.255/24`
`family(inet)`	int	extraction de la famille d'adresse ; `4` pour IPv4, `6` pour IPv6	`family('::1')`	`6`
`host(inet)`	text	extraction de l'adresse IP en texte	`host('192.168.1.5/24')`	`192.168.1.5`
`hostmask(inet)`	inet	construction du masque d'hôte pour le réseau	`hostmask('192.168.23.20/30')`	`0.0.0.3`
`masklen(inet)`	int	extraction de la longueur du masque réseau	`masklen('192.168.1.5/24')`	`24`
`netmask(inet)`	inet	construction du masque réseau	`netmask('192.168.1.5/24')`	`255.255.255.0`
`network(inet)`	cidr	extraction de la partie réseau de l'adresse	`network('192.168.1.5/24')`	`192.168.1.0/24`
`(inet set_masklen, int)`	inet	configure la longueur du masque réseau pour les valeurs inet	`set_masklen('192.168.1.5/24', 16)`	`192.168.1.5/16`
`(cidr set_masklen, int)`	cidr	configure la longueur du masque réseau pour les valeurs cidr	`set_masklen('192.168.1.0/24'::cidr, 16)`	`192.168.0.0/16`
`text(inet)`	text	extraction de l'adresse IP et de la longueur du masque réseau comme texte	`text(inet '192.168.1.5')`	`192.168.1.5/32`

Toute valeur cidr peut être convertie en inet implicitement ou explicitement ; de ce fait, les fonctions indiquées ci-dessus comme opérant sur le type inet opèrent aussi sur le type cidr. (Lorsque les fonctions sont séparées pour les types inet et cidr, c'est que leur comportement peut différer.) Il est également permis de convertir une valeur inet en cidr. Dans ce cas, tout bit à la droite du masque réseau est silencieusement positionné à zéro pour créer une valeur cidr valide. De plus, une valeur de type texte peut être transtypée en inet ou cidr à l'aide de la syntaxe habituelle de transtypage : par exemple `inet(`*`expression`*`)` ou *`nom_colonne`*`::cidr`.

Le Tableau 9.35, « Fonctions macaddr » affiche les fonctions utilsables avec le type macaddr. La fonction `trunc(macaddr)` renvoie une adresse MAC avec les trois derniers octets initialisés à zéro. Ceci peut être utilisé pour associer le préfixe restant à un manufacturier.

Tableau 9.35. Fonctions macaddr

Fonction	Type de retour	Description	Exemple	Résultat
`trunc(macaddr)`	macaddr	initialiser les trois octets finaux à zéro	`trunc(macaddr '12:34:56:78:90:ab')`	`12:34:56:00:00:00`

Le type macaddr supporte aussi les opérateurs relationnels standard (>, <=, etc.) de tri lexicographique.

9.13. Fonctions et opérateurs de la recherche plein texte

Tableau 9.36, « Opérateurs de recherche plein texte », Tableau 9.37, « Fonctions de la recherche plein texte » et Tableau 9.38, « Fonctions de débogage de la recherche plein texte » résume les fonctions et les opérateurs fournis pour la recherche plein texte. Voir Chapitre 12, Recherche plein texte pour une explication détaillée sur la fonctionnalité de recherche plein texte de PostgreSQL™.

Tableau 9.36. Opérateurs de recherche plein texte

Opérateur	Description	Exemple	Résultat
@@	tsvector correspond à tsquery ?	`to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')`	`t`
@@@	synonym obsolète de @@	`to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')`	`t`
||	concatène tsvectors	`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector`	`'a':1 'b':2,5 'c':3 'd':4`
&&	ET logique des tsquery	`'fat | rat'::tsquery && 'cat'::tsquery`	`( 'fat' | 'rat' ) & 'cat'`
||	OU logique des tsquery	`'fat | rat'::tsquery || 'cat'::tsquery`	`( 'fat' | 'rat' ) | 'cat'`
!!	inverse une tsquery	`!! 'cat'::tsquery`	`!'cat'`
@>	tsquery en contient une autre ?	`'cat'::tsquery @> 'cat & rat'::tsquery`	`f`
<@	tsquery est contenu dans ?	`'cat'::tsquery <@ 'cat & rat'::tsquery`	`t`

Note

Les opérateurs de confinement de tsquery considèrent seulement les lexèmes listés dans les deux requêtes, ignorant les opérateurs de combinaison.

En plus des opérateurs présentés dans la table, les opérateurs de comparaison B-tree habituels (=, <, etc) sont définis pour les types tsvector et tsquery. Ils ne sont pas très utiles dans le cadre de la recherche plein texte mais permettent la construction d'index d'unicité sur ces types de colonne.

Tableau 9.37. Fonctions de la recherche plein texte

Fonction	Type de retour	Description	Exemple	Résultat
`to_tsvector([ *config* regconfig , ] *document* text)`	tsvector	réduire le texte du document en un tsvector	`to_tsvector('english', 'The Fat Rats')`	`'fat':2 'rat':3`
`length(tsvector)`	integer	nombre de lexemes dans tsvector	`length('fat:2,4 cat:3 rat:5A'::tsvector)`	`3`
`(tsvect setweightor, "char")`	tsvector	affecte un poids à chaque élément d'un tsvector	`setweight('fat:2,4 cat:3`	`'cat':3A 'fat':2A,4A 'rat':5A`

Fonction	Type de retour	Description	Exemple	Résultat
			rat:5B'::tsvector, 'A')	
strip(tsvector)	tsvector	supprime les positions et les poids du tsvector	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([*config* regconfig,] *requete* text)	tsquery	normalise les mots et les convertit en un tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
plainto_tsquery([*config* regconfig,] *requete* text)	tsquery	crée un tsquery en ignorant la ponctuation	plainto_tsquery('english', 'The Fat Rats')	'fat' & 'rat'
numnode(tsquery)	integer	nombre de lexèmes et d'opérateurs dans tsquery	numnode('(fat & rat) | cat'::tsquery)	5
querytree(*requete* tsquery)	text	récupère la partie indexable d'un tsquery	querytree('foo & ! bar'::tsquery)	'foo'
ts_rank([*poids* float4[],] *vecteur* tsvector, *requete* tsquery [, *normalisation* integer])	float4	renvoie le score d'un document pour une requête	ts_rank(textsearch, query)	0.818
ts_rank_cd([*poids* float4[],] *vecteur* tsvector, *requete* tsquery [, *normalisation* integer])	float4	renvoie le score d'un document pour une requête en utilisant une densité personnalisée	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_headline([*config* regconfig,] *document* text, *requete* tsquery [, *options* text])	text	affiche la correspondance d'une requête dans un document	ts_headline('x y z', 'z'::tsquery)	x y z
ts_rewrite(*requete* tsquery, *cible* tsquery, *substitution* tsquery)	tsquery	remplace la cible avec la substitution à l'intérieur de la requête	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')
ts_rewrite(*requete* tsquery, *select* text)	tsquery	remplace en utilisant les cibles et substitutions à partir d'une commande **SELECT**	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
get_current_ts_config()	regconfig	récupère la configuration par défaut de la recherche plein texte	get_current_ts_config()	english
tsvector_update_trigger()	trigger	fonction déclencheur pour la mise à jour automatique de colonne tsvector	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swed	

Fonction	Type de retour	Description	Exemple	Résultat
			ish', title, body)	
tsvector_update_trigger_column()	trigger	fonction déclencheur pour la mise à jour automatique de colonne tsvector	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)	

Note

Toutes les fonctions de recherche plein texte qui acceptent un argument regconfig optionel utilisent la configuration indiqée par default_text_search_config en cas d'omission de cet argument.

Les fonctions de Tableau 9.38, « Fonctions de débogage de la recherche plein texte » sont listées séparément, car elles ne font pas partie des fonctions utilisées dans les opérations de recherche plein texte de tous les jours. Elles sont utiles pour le développement et le débogage de nouvelles configurations de recherche plein texte.

Tableau 9.38. Fonctions de débogage de la recherche plein texte

Fonction	Type de retour	Description	Exemple	Résultat
ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *jeton* text, OUT *dictionnaires* regdictionary[], OUT *dictionnaire* regdictionary, OUT *lexemes* text[])	setof record	teste une configuration	ts_debug('english', 'The Brightest supernovaes')	(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...
ts_lexize(*dict* regdictionary, *jeton* text)	text[]	teste un dictionnaire	ts_lexize('english_stem', 'stars')	{star}
nom_ana ts_parse(*lyseur* text, *document* text, OUT *id_jeton* integer, OUT *jeton* text)	setof record	teste un analyseur	ts_parse('default', 'foo - bar')	(1,foo) ...
oid_ana ts_parse(*lyseur* oid, *document* text, OUT *id_jeton* integer, OUT *jeton* text)	setof record	teste un analyseur	ts_parse(3722, 'foo - bar')	(1,foo) ...
nom_analyseu	setof record	obtient les types de jeton définis par l'analyseur	ts_token_type('default')	(1,asciiword,"Word, all ASCII") ...

Fonction	Type de retour	Description	Exemple	Résultat
r text, OUT *id_jeton* integer, OUT *alias* text, OUT *description* text)				
oid_analysyeu ts_token_type(*r*oid, OUT *id_jeton* integer, OUT *alias* text, OUT *description* text)	setof record	obtient les types de jeton définis par l'analyseur	ts_token_type(3722)	(1,asciiword,"Word, all ASCII") ...
ts_stat(*sqlquery* text, [*poids* text,] OUT *mot* text, OUT *ndoc* integer, OUT *nentry* integer)	setof record	obtient des statistiques sur une colonne tsvector	ts_stat('SELECT vector from apod')	(foo,10,15) ...

9.14. Fonctions XML

Les fonctions et expressions décrites dans cette section opèrent sur des valeurs de type xml. Lire la Section 8.13, « Type XML » pour des informations sur le type xml. Les expressions `xmlparse` et `xmlserialize` permettant de convertir vers ou à partir du type xml ne sont pas reprises ici. L'utilisation d'un grand nombre de ces fonctions nécessite que l'installation soit construite avec **configure --with-libxml**.

9.14.1. Produire un contenu XML

Un ensemble de fonctions et expressions de type fonction est disponible pour produire du contenu XML à partir de données SQL. En tant que telles, elles conviennent particulièrement bien pour formater les résultats de requêtes en XML à traiter dans les applications clientes.

9.14.1.1. `xmlcomment`

```
xmlcomment(text)
```

La fonction `xmlcomment` crée une valeur XML contenant un commentaire XML avec, comme contenu, le texte indiqué. Le texte ne peut pas contenir « `--` » ou se terminer par un « `-` » de sorte que que la construction résultante représente un commentaire XML valide. Si l'argument est NULL, le résultat est NULL.

Exemple :

```
SELECT xmlcomment('bonjour');

  xmlcomment
--------------
 <!--bonjour-->
```

9.14.1.2. `xmlconcat`

```
xmlconcat(xml[, ...])
```

La fonction `xmlconcat` concatène une liste de valeurs XML individuelles pour créer une valeur simple contenant un fragment de contenu XML. Les valeurs NULL sont omises ; le résultat est NULL seulement s'il n'y a pas d'arguments non NULL.

Exemple :

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

Les déclarations XML, si elles sont présentes, sont combinées come suit. Si toutes les valeurs en argument ont la même déclaration de version XML, cette version est utilisée dans le résultat. Sinon aucune version n'est utilisée. Si toutes les valeurs en argument ont la valeur de déclaration « standalone » à « yes », alors cette valeur est utilisée dans le résultat. Si toutes les valeurs en argument ont une valeur de déclaration « standalone » et qu'au moins l'une d'entre elles est « no », alors cette valeur est utilisée dans le résultat. Sinon le résultat n'a aucune déclaration « standalone ». Si le résultat nécessite une déclaration « standalone » sans déclaration de version, une déclaration de version 1.0 est utilisée car le standard XML impose qu'une déclaration XML contienne une déclaration de version. Les déclarations d'encodage sont ignorées et supprimées dans tous les cas.

Exemple :

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1"
standalone="no"?><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. `xmlelement`

```
 xmlelement(name nom [, xmlattributes(valeur [AS nom_attribut] [, ... ])] [, contenu,
...])
```

L'expression `xmlelement` produit un élément XML avec le nom, les attributs et le contenu donnés.

Exemples :

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
--------------------------------------
 <foo bar="2007-01-26">content</foo>
```

Les noms d'élément et d'attribut qui ne sont pas des noms XML valides sont modifiés en remplaçant les caractères indésirables par une séquence `_xHHHH_`, où *HHHH* est le codage Unicode du caractère en notation hexadécimal. Par exemple :

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

Un nom explicit d'attribut n'a pas besoin d'être indiqué si la valeur de l'attribut est la référence d'une colonne, auquel cas le nom de la colonne est utilisé comme nom de l'attribut par défaut. Dans tous les autres cas, l'attribut doit avoir un nom explicite. Donc, cet exemple est valide :

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

Mais ceux-ci ne le sont pas :

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Si le contenu de l'élément est précisé, il est formaté en fonction du type de données. Si le contenu est lui-même de type xml, des documents XML complexes peuvent être construits. Par exemple :

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Le contenu des autres types est formaté avec des données XML valides. Cela signifie en particulier que les caractères <, >, et & sont convertis en entités. Les données binaires (type bytea) sont représentées dans un encodage base64 ou hexadécimal, suivant la configuration du paramètre xmlbinary. Le comportement particulier pour les types de données individuels devrait évoluer pour aligner les types de données SQL et PostgreSQL avec la spécification de XML Schema, auquel cas une description plus précise sera ajoutée.

9.14.1.4. `xmlforest`

```
 xmlforest(contenu [AS nom] [, ...])
```

L'expression `xmlforest` produit un arbre XML (autrement dit une séquence) d'éléments utilisant les noms et le contenu donnés.

Exemples :

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                          xmlforest
-----------------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

Comme indiqué dans le second exemple, le nom de l'élément peut être omis si la valeur du contenu est une référence de colonne, auquel cas le nom de la colonne est utilisé par défaut. Sinon, un nom doit être indiqué.

Les noms d'éléments qui ne sont pas des noms XML valides sont échappés comme indiqué pour `xmlelement` ci-dessus. De façon similaire, les données de contenu sont échappées pour rendre le contenu XML valide sauf s'il est déjà de type xml.

Les arbres XML ne sont pas des documents XML valides s'ils sont constitués de plus d'un élément. Il peut donc s'avérer utile d'emballer les expressions `xmlforest` dans `xmlelement`.

9.14.1.5. xmlpi

```
xmlpi(name target [, content])
```

L'expression `xmlpi` crée une instruction de traitement XML. Le contenu, si présent, ne doit pas contenir la séquence de caractères `?>`.

Exemple :

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.14.1.6. xmlroot

```
xmlroot(xml, version text | no value [, standalone yes|no|no value])
```

L'expression `xmlroot` modifie les propriétés du nœud racine d'une valeur XML. Si une version est indiquée, elle remplace la valeur dans la déclaration de version du nœud racine. Si un paramètre « standalone » est spécifié, il remplace la valeur dans la déclaration « standalone » du nœud racine.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.14.1.7. xmlagg

```
xmlagg(xml)
```

La fonction `xmlagg` est, à la différence des fonctions décrites ici, une fonction d'aggrégat. Elle concatène les valeurs en entrée pour les passer en argument à la fonction d'aggrégat, comme le fait la fonction `xmlconcat`, sauf que la concaténation survient entre les lignes plutôt qu'entre les expressions d'une même ligne. Voir Section 9.18, « Fonctions d'agrégat » pour plus d'informations sur les fonctions d'agrégat.

Exemple :

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

Pour déterminer l'ordre de la concaténation, une clause `ORDER BY` peut être ajoutée à l'appel de l'agrégat comme décrit dans Section 4.2.7, « Expressions d'agrégat ». Par exemple :

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

L'approche non standard suivante était recommendée dans les versions précédentes et peut toujours être utiles dans certains cas particuliers :

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.14.1.8. Prédicats XML

```
xml IS DOCUMENT
```

L'expression `IS DOCUMENT` renvoie true si la valeur de l'argument XML est un document XML correct, false dans le cas contraire (c'est-à-dire qu'il s'agit d'un fragment de document) ou NULL si l'argument est NULL. Voir la Section 8.13, « Type XML » pour les différences entre documents et fragments de contenu.

9.14.2. Traiter du XML

Pour traiter les valeurs du type xml, PostgreSQL fournit la fonction `xpath`, qui évalue les expressions XPath 1.0.

```
xpath(xpath, xml[, nsarray])
```

La fonction `xpath` évalue l'expression XPath *`xpath`* avec la valeur XML *`xml`*. Elle renvoie un tableau de valeurs XML correspondant à l'ensemble de nœuds produit par une expression XPath.

Le second argument doit être un document XML bien formé. En particulier, il doit avoir un seul élément de nœud racine.

Le troisième argument de la fonction est un tableau de correspondances de *namespace*. Ce tableau doit avoir deux dimensions dont la seconde a une longueur 2 (en fait, c'est un tableau de tableaux à exactement deux éléments). Le premier élément de chaque entrée du tableau est le nom du *namespace* (alias), le second étant l'URI du *namespace*. Il n'est pas requis que les alias fournis dans ce tableau soient les mêmes que ceux utilisés dans le document XML (autrement dit, que ce soit dans le contexte du document XML ou dans celui de la fonction `xpath`, les alias ont une vue *locale*).

Exemple :

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

Comment gérer des *namespaces* par défaut (anonymes) :

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.14.3. Transformer les tables en XML

Les fonctions suivantes transforment le contenu de tables relationnelles en valeurs XML. Il s'agit en quelque sorte d'un export XML.

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

Le type en retour de ces fonctions est xml.

`table_to_xml` transforme le contenu de la table passée en argument (paramètre *`tbl`*). regclass accepte des chaînes identifiant les tables en utilisant la notation habituelle, incluant les qualifications possibles du schéma et les guillemets doubles. `query_to_xml` exécute la requête dont le texte est passé par le paramètre *`query`* et transforme le résultat. `cursor_to_xml` récupère le nombre indiqué de lignes à partir du curseur indiqué par le paramètre *`cursor`*. Cette variante est recommandée si la transformation se fait sur de grosses tables car la valeur en résultat est construite en mémoire pour chaque fonction.

Si *`tableforest`* vaut false, alors le document XML résultant ressemble à ceci :

```
<tablename>
  <row>
    <columnname1>donnees</columnname1>
    <columnname2>donnees</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

Si *`tableforest`* vaut true, le résultat est un fragment XML qui ressemble à ceci :

```
<tablename>
  <columnname1>donnees</columnname1>
  <columnname2>donnees</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

Si aucune table n'est disponible, c'est-à-dire lors d'une transformation à partir d'une requête ou d'un curseur, la chaîne `table` est utilisée dans le premier format, et la chaîne `row` dans le second.

Le choix entre ces formats dépend de l'utilisateur. Le premier format est un document XML correct, ce qui est important dans beaucoup d'applications. Le second format tend à être plus utile dans la fonction `cursor_to_xml` si les valeurs du résultat sont à rassembler plus tard dans un document. Les fonctions pour produire du contenu XML discutées ci-dessus, en particulier `xmlelement`, peuvent être utilisées pour modifier les résultats.

Les valeurs des données sont transformées de la même façon que ce qui est décrit ci-dessus pour la fonction `xmlelement`.

Le paramètre *`nulls`* détermine si les valeurs NULL doivent être incluses en sortie. À true, les valeurs NULL dans les colonnes sont représentées ainsi :

```
<columnname xsi:nil="true"/>
```

où `xsi` est le préfixe de l'espace de noms XML pour l'instance XML Schema. Une déclaration appropriée d'un espace de noms est ajoutée à la valeur du résultat. À false, les colonnes contenant des valeurs NULL sont simplement omises de la sortie.

Le paramètre *`targetns`* indique l'espace de noms souhaité pour le résultat. Si aucun espace de nom particulier n'est demandé, une chaîne vide doit être passée.

Les fonctions suivantes renvoient des documents XML Schema décrivant la transformation réalisée par les fonctions ci-dessus.

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns
text)
```

Il est essentiel que les mêmes paramètres soient passés pour obtenir les bonnes transformations de données XML et des documents XML Schema.

Les fonctions suivantes réalisent la transformation des données XML et du XML Schema correspondant en un seul document (ou arbre), liés ensemble. Elles sont utiles lorsque les résultats doivent être auto-contenus et auto-descriptifs.

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns
text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns
text)
```

De plus, les fonctions suivantes sont disponibles pour produire des transformations analogues de schémas complets ou de bases de données complètes.

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns
text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

Elles peuvent produire beaucoup de données, qui sont construites en mémoire. Lors de transformations de gros schémas ou de grosses bases, il peut être utile de considérer la transformation séparée des tables, parfois même via un curseur.

Le résultat de la transformation du contenu d'un schéma ressemble à ceci :

```
<nomschema>

transformation-table1

transformation-table2

...

</nomschema>
```

où le format de transformation d'une table dépend du paramètre *`tableforest`* comme expliqué ci-dessus.

Le résultat de la transformation du contenu d'une base ressemble à ceci :

```
<nombase>

<nomschema1>
  ...
</nomschema1>

<nomschema2>
  ...
</nomschema2>

...

</nombase>
```

avec une transformation du schéma identique à celle indiquée ci-dessus.

En exemple de l'utilisation de la sortie produite par ces fonctions, la Figure 9.1, « Feuille de style XSLT pour convertir du SQL/XML en HTML » montre une feuille de style XSLT qui convertit la sortie de `table_to_xml_and_xmlschema` en un document HTML contenant un affichage en tableau des données de la table. D'une façon similaire, les données en résultat de ces fonctions peuvent être converties dans d'autres formats basés sur le XML.

Figure 9.1. Feuille de style XSLT pour convertir du SQL/XML en HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>
```

```
  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:element[@name='row']/

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each
select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.15. Fonctions de manipulation de séquences

Cette section décrit les fonctions opérant sur les *objets de séquence*, aussi appelés générateurs de séquence ou simplement séquences. Ces objets sont des tables spéciales, mono-lignes, créées avec la commande CREATE SEQUENCE(7). Les séquences sont généralement utilisées pour engendrer des identifiants uniques de lignes d'une table. Les fonctions de séquence, listées dans le Tableau 9.39, « Fonctions séquence », fournissent des méthodes simples, et sûres en environnement multi-utilisateurs, d'obtention de valeurs successives à partir d'objets séquence.

Tableau 9.39. Fonctions séquence

Fonction	Type de retour	Description
`currval(regclass)`	bigint	Renvoie la valeur la plus récemment obtenue avec `nextval` pour la séquence indiquée
`lastval()`	bigint	Renvoie la valeur la plus récemment obtenue avec `nextval` pour toute séquence
`nextval(regclass)`	bigint	Incrémente la séquence et renvoie la nouvelle valeur
`setval(regclass, bigint)`	bigint	Positionne la valeur courante de la séquence
`setval(regclass, bigint, boolean)`	bigint	Positionne la valeur courante de la séquence et le drapeau `is_called`

La séquence à traiter par l'appel d'une fonction de traitement de séquences est identifiée par un argument regclass, qui n'est autre que l'OID de la séquence dans le catalogue système pg_class. Il n'est toutefois pas nécessaire de se préoccuper de la recherche de cet OID car le convertisseur de saisie du type de données regclass s'en charge. Il suffit d'écrire le nom de la séquence entre guille-

mets simples, de façon à le faire ressembler à un libellé. Pour obtenir une compatibilité avec la gestion des noms SQL ordinaires, la chaîne est convertie en minuscules, sauf si le nom de la séquence est entouré de guillemets doubles. Du coup :

```
nextval('foo')      opère sur la séquence foo
nextval('FOO')      opère sur la séquence foo
nextval('"Foo"')    opère sur la séquence Foo
```

Le nom de la séquence peut, au besoin, être qualifié du nom du schéma :

```
nextval('mon_schema.foo')     opère sur mon_schema.foo
nextval('"mon_schema".foo')   identique à ci-dessus
nextval('foo')                parcourt le chemin de recherche
pour trouver foo
```

Voir la Section 8.16, « Types identifiant d'objet » pour plus d'informations sur regclass.

Note

Avant la version 8.1 de PostgreSQL™, les arguments des fonctions de traitement de séquences étaient du type text, et non regclass. De ce fait, les conversions précédemment décrites d'une chaîne de caractères en valeur OID se produisaient à chaque appel. Pour des raisons de compatibilité, cette fonctionnalité existe toujours. Mais, en interne, un transtypage implicite est effectué entre text et regclass avant l'appel de la fonction.

Lorsque l'argument d'une fonction de traitement de séquences est écrit comme une simple chaîne de caractères, il devient une constante de type regclass. Puisqu'il ne s'agit que d'un OID, il permet de suivre la séquence originelle même en cas de renommage, changement de schéma... Ce principe de « lien fort » est en général souhaitable lors de références à la séquence dans les vues et valeurs par défaut de colonnes. Un « lien faible » est généralement souhaité lorsque la référence à la séquence est résolue à l'exécution. Ce comportement peut être obtenu en forçant le stockage des constantes sous la forme de constantes text plutôt que regclass :

```
nextval('foo'::text)      foo est recherché à l'exécution
```

Le lien faible est le seul comportement accessible dans les versions de PostgreSQL™ antérieures à 8.1. Il peut donc être nécessaire de le conserver pour maintenir la sémantique d'anciennes applications.

L'argument d'une fonction de traitement de séquences peut être une expression ou une constante. S'il s'agit d'une expression textuelle, le transtypage implicite impose une recherche à l'exécution.

Les fonctions séquence disponibles sont :

`nextval`
: Avance l'objet séquence à sa prochaine valeur et renvoie cette valeur. Ce fonctionnement est atomique : même si de multiples sessions exécutent `nextval` concurrentiellement, chacune obtient sans risque une valeur de séquence distincte.

`currval`
: Renvoie la valeur la plus récemment retournée par `nextval` pour cette séquence dans la session courante. (Une erreur est rapportée si `nextval` n'a jamais été appelée pour cette séquence dans cette session.) Parce qu'elle renvoie une valeur locale à la session, la réponse est prévisible, que d'autres sessions aient exécuté ou non la fonction `nextval` après la session en cours.

`lastval`
: Renvoie la valeur la plus récemment retournée par `nextval` dans la session courante. Cette fonction est identique à `currval`, sauf qu'au lieu de prendre le nom de la séquence comme argument, elle récupère la valeur de la dernière séquence utilisée par `nextval` dans la session en cours. Si `nextval` n'a pas encore été appelée dans la session en cours, un appel à `lastval` produit une erreur.

`setval`
: Réinitialise la valeur du compteur de l'objet séquence. La forme avec deux paramètres initialise le champ `last_value` de la séquence à la valeur précisée et initialise le champ `is_called` à `true`, signifiant que le prochain `nextval` avance la séquence avant de renvoyer une valeur. La valeur renvoyée par `currval` est aussi configuré à la valeur indiquée. Dans la forme à trois paramètres, `is_called` peut être initialisé à `true` ou à `false`. `true` a le même effet que la forme à deux paramètres. Positionné à `false`, le prochain `nextval` retourne exactement la valeur indiquée et l'incrémentation de la séquence commence avec le `nextval` suivant. De plus, la valeur indiquée par `currval` n'est pas modifiée dans ce cas. (Il s'agit d'une modification du comportement des versions antérieures à la 8.3.) Par exemple,

```
SELECT setval('foo', 42);           Le nextval suivant retourne 43
SELECT setval('foo', 42, true);     Comme ci-dessus
SELECT setval('foo', 42, false);    Le nextval suivant retourne 42
```

Le résultat renvoyé par `setval` est la valeur du second argument.

Si un objet séquence a été créé avec les paramètres par défaut, les appels à `nextval` sur celui-ci renvoient des valeurs successives à partir de 1. D'autres comportements peuvent être obtenus en utilisant des paramètres spéciaux de la commande CREATE SEQUENCE(7) ; voir la page de référence de la commande pour plus d'informations.

> **Important**
>
> Pour éviter le blocage de transactions concurrentes qui obtiennent des nombres de la même séquence, une opération `nextval` n'est jamais annulée ; c'est-à-dire qu'une fois la valeur récupérée, elle est considérée utilisée, même si la transaction qui exécute `nextval` avorte par la suite. Cela signifie que les transactions annulées peuvent laisser des « trous » inutilisés dans la séquence des valeurs assignées. Les opérations `setval` ne sont jamais annulées non plus.

9.16. Expressions conditionnelles

Cette section décrit les expressions conditionnelles respectueuses du standard SQL disponibles avec PostgreSQL™.

> **Astuce**
>
> S'il s'avère nécessaire d'aller au-delà des possibilités offertes par les expressions conditionnelles, il faut considérer l'écriture d'une procédure stockée dans un langage de programmation plus expressif.

9.16.1. CASE

L'expression SQL CASE est une expression conditionnelle générique, similaire aux instructions if/else des autres langages de programmation :

```
CASE WHEN condition THEN résultat
     [WHEN ...]
     [ELSE résultat]
END
```

Les clauses CASE peuvent être utilisées partout où une expression est valide. Chaque *`condition`* est une expression qui renvoie un résultat de type boolean. Si le résultat de la condition est vrai, alors la valeur de l'expression CASE est le *`résultat`* qui suit la condition. Si le résultat de la condition n'est pas vrai, toutes les clauses WHEN suivantes sont parcourues de la même façon. Si aucune *`condition`* WHEN n'est vraie, alors la valeur de l'expression CASE est le *`résultat`* de la clause ELSE. Si la clause ELSE est omise et qu'aucune condition ne correspond, alors le résultat est nul.

Un exemple :

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'un'
            WHEN a=2 THEN 'deux'
            ELSE 'autre'
       END
    FROM test;

 a | case
---+-------
 1 | un
 2 | deux
 3 | autre
```

Les types de données de toutes les expressions *`résultat`* doivent être convertibles dans un type de sortie unique. Voir la Section 10.5, « Constructions `UNION`, `CASE` et constructions relatives » pour plus de détails.

L'expression CASE qui suit est une variante de la forme générale ci-dessus :

```
CASE expression
    WHEN valeur THEN
    résultat
    [WHEN ...]
    [ELSE résultat]
END
```

La première *expression* est calculée et comparée à chacune des *valeur* des clauses WHEN jusqu'à en trouver une égale. Si aucune ne correspond, le *résultat* de la clause ELSE (ou une valeur NULL) est renvoyé(e). C'est similaire à l'instruction `switch` du langage C.

L'exemple ci-dessus peut être réécrit en utilisant la syntaxe CASE simple :

```
SELECT a,
       CASE a WHEN 1 THEN 'un'
              WHEN 2 THEN 'deux'
              ELSE 'autre'
       END
    FROM test;

 a | case
---+-------
 1 | un
 2 | deux
 3 | autre
```

Une expression CASE n'évalue pas les sous-expressions qui ne sont pas nécessaires pour déterminer le résultat. Par exemple, une façon possible d'éviter une division par zéro :

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

Note

Comme décrit dans Section 35.6, « Catégories de volatilité des fonctions », les fonctions et les opérateurs marqués `IMMUTABLE` peuvent être évalués quand la requête est planifiée plutôt que quand elle est exécutée. Cela signifie que les parties constantes d'une sous-expression qui n'est pas évaluée durant l'exécution de la requête pourrait toujours être évaluée pendant la planification de la requête.

9.16.2. COALESCE

```
COALESCE(valeur [, ...])
```

La fonction `COALESCE` renvoie le premier de ses arguments qui n'est pas nul. Une valeur NULL n'est renvoyée que si tous les arguments sont nuls. Cette fonction est souvent utile pour substituer une valeur par défaut aux valeurs NULL lorsque la donnée est récupérée pour affichage. Par exemple :

```
SELECT COALESCE(description, description_courte, '(aucune)') ...
```

À l'instar d'une expression CASE, `COALESCE` n'évalue pas les arguments inutiles à la détermination du résultat ; c'est-à-dire que tous les arguments à la droite du premier argument non nul ne sont pas évalués. Cette fonction SQL standard fournit des fonctionnalités similaires à `NVL` et `IFNULL`, qui sont utilisées dans d'autres systèmes de bases de données.

9.16.3. NULLIF

```
NULLIF(valeur1, valeur2)
```

La fonction `NULLIF` renvoie une valeur NULL si *valeur1* et *valeur2* sont égales ; sinon, elle renvoie *valeur1*.

On peut s'en servir pour effectuer l'opération inverse de l'exemple de `COALESCE` donné ci-dessus :

```
SELECT NULLIF(valeur, '(aucune)') ...
```

Dans cet exemple, si `valeur` vaut `(aucune)`, la valeur NULL est renvoyée, sinon la valeur de `valeur` est renvoyée.

9.16.4. GREATEST et LEAST

```
GREATEST(valeur [, ...])
```

```
LEAST(valeur [, ...])
```

Les fonctions GREATEST et LEAST sélectionnent, respectivement, la valeur la plus grande et la valeur la plus petite d'une liste d'expressions. Elles doivent être toutes convertibles en un type de données commun, type du résultat (voir la Section 10.5, « Constructions UNION, CASE et constructions relatives » pour les détails). Les valeurs NULL contenues dans la liste sont ignorées. Le résultat est NULL uniquement si toutes les expressions sont NULL.

GREATEST et LEAST ne sont pas dans le standard SQL mais sont des extensions habituelles. D'autres SGBD leur imposent de retourner NULL si l'un quelconque des arguments est NULL, plutôt que lorsque tous les arguments sont NULL.

9.17. Fonctions et opérateurs de tableaux

Le Tableau 9.40, « Opérateurs pour les tableaux » présente les opérateurs disponibles pour les types tableaux.

Tableau 9.40. Opérateurs pour les tableaux

Opérateur	Description	Exemple	Résultat
=	égal à	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	différent de	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	inférieur à	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	supérieur à	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	inférieur ou égal à	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	supérieur ou égal à	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	contient	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	est contenu par	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	se chevauchent (ont des éléments en commun)	ARRAY[1,4,3] && ARRAY[2,1]	t
||	concaténation de tableaux	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	concaténation de tableaux	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	concaténation d'un élément avec un tableau	3 || ARRAY[4,5,6]	{3,4,5,6}
||	concaténation d'un tableau avec un élément	ARRAY[4,5,6] || 7	{4,5,6,7}

Les comparaisons de tableaux comparent les contenus des tableaux élément par élément, en utilisant la fonction de comparaison par défaut du B-Tree pour le type de données de l'élément. Dans les tableaux multi-dimensionnels, les éléments sont visités dans l'ordre des colonnes (« row-major order », le dernier indice varie le plus rapidement). Si le contenu de deux tableaux est identique mais que les dimensions sont différentes, la première différence dans l'information de dimension détermine l'ordre de tri. (Ce fonctionnement diffère de celui des versions de PostgreSQL™ antérieures à la 8.2 : les anciennes versions indiquent que deux tableaux de même contenu sont identiques même si le nombre de dimensions ou les échelles d'indices diffèrent.)

Voir la Section 8.14, « Tableaux » pour plus de détails sur le comportement des opérateurs.

Le Tableau 9.41, « Fonctions pour les tableaux » présente les fonctions utilisables avec des types tableaux. Voir la Section 8.14, « Tableaux » pour plus d'informations et des exemples d'utilisation de ces fonctions.

Tableau 9.41. Fonctions pour les tableaux

Fonction	Type de retour	Description	Exemple	Résultat
array_append (anyarray, anyelement)	anyarray	ajoute un élément à la fin d'un tableau	array_append(ARRAY[1,2], 3)	{1,2,3}
array_cat (anyarray, anyarray)	anyarray	concatène deux tableaux	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
(anya array_ndimsrray)	int	renvoit le nombre de dimensions du tableau	array_ndims(ARRAY[[1,2,3], [4,5,6]])	2
array_dims (anyarray)	text	renvoie une représentation textuelle des dimensions d'un tableau	array_dims(array[[1,2,3], [4,5,6]])	[1:2][1:3]
(anyel array_fillement, int[], [, int[]])	anyarray	renvoit un tableau initialisé avec une valeur et des dimensions fournies, en option avec des limites basses autre que 1	array_fill(7, ARRAY[3], ARRAY[2])	[2:4]={7,7,7}
(any array_lengtharray, int)	int	renvoit la longueur de la dimension du tableau	array_length(array[1,2,3], 1)	3
array_lower (anyarray, int)	int	renvoie la limite inférieure du tableau donné	array_lower('[0:2]={1,2,3}'::int[], 1)	0
array_prepend (anyelement, anyarray)	anyarray	ajoute un élément au début d'un tableau	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_to_string (anyarray, text)	text	concatène des éléments de tableau en utilisant le délimiteur fourni	array_to_string(array[1, 2, 3], '~^~')	1~^~2~^~3
array_upper (anyarray, int)	int	renvoie la limite supérieure du tableau donné	array_upper(ARRAY[1,2,3,4], 1)	4
string_to_array (text, text)	text[]	divise une chaîne en tableau d'éléments en utilisant le délimiteur fourni	string_to_array('xx~^~yy~^~zz', '~^~')	{xx,yy,zz}
unnest(anyarray)	setof anyelement	étend un tableau à un ensemble de lignes	unnest(ARRAY[1,2])	1 2 (2 rows)

Voir aussi Section 9.18, « Fonctions d'agrégat » à propose la fonction d'agrégat `array_agg` à utiliser avec les tableaux.

9.18. Fonctions d'agrégat

Les *fonctions d'agrégat* calculent une valeur unique à partir d'un ensemble de valeurs en entrée. Les fonctions d'agrégats intégrées sont listées dans Tableau 9.42, « Fonctions d'agrégat générales » et Tableau 9.43, « Fonctions d'agrégats pour les statistiques ». La syntaxe particulière des fonctions d'agrégat est décrite dans la Section 4.2.7, « Expressions d'agrégat ». La Section 2.7, « Fonctions d'agrégat » fournit un supplément d'informations introductives.

Tableau 9.42. Fonctions d'agrégat générales

Fonction	Type d'argument	Type de retour	Description
array_agg(*expression*)	any	tableau du type de l'argument	les valeurs en entrée, pouvant inclure des valeurs NULL, concaténées dans un tableau
avg(*expression*)	smallint, int, bigint, real, double precision, numeric ou interval	numeric pour tout argument de type entier, double precision pour tout argument en virgule flottante, sinon identique au type de données de l'argument	la moyenne arithmétique de toutes les valeurs en entrée
bit_and(*expression*)	smallint, int, bigint ou bit	identique au type de données de l'argument	le AND bit à bit de toutes les valeurs non NULL en entrée ou NULL s'il n'y en a pas
bit_or(*expression*)	smallint, int, bigint ou bit	identique au type de données de l'argument	le OR bit à bit de toutes les valeurs non NULL en entrée ou NULL s'il n'y en a pas
bool_and(*expression*)	bool	bool	true si toutes les valeurs en entrée valent true, false sinon
bool_or(*expression*)	bool	bool	true si au moins une valeur en entrée vaut true, false sinon
count(*)		bigint	nombre de lignes en entrée
count(*expression*)	tout type	bigint	nombre de lignes en entrée pour lesquelles l'*expression* n'est pas NULL
every(*expression*)	bool	bool	équivalent à bool_and
max(*expression*)	tout type array, numeric, string ou date/time	identique au type en argument	valeur maximale de l'*expression* pour toutes les valeurs en entrée
min(*expression*)	tout type array, numeric, string ou date/time	identique au type en argument	valeur minimale de l'*expression* pour toutes les valeurs en entrée
string_agg(*expression*, *delimiter*)	text, text	text	valeurs en entrées concaténées dans une chaîne, séparées par un délimiteur
sum(*expression*)	smallint, int, bigint, real, double precision, numeric ou interval	bigint pour les arguments de type smallint ou int, numeric pour les arguments de type bigint, double precision pour les arguments en virgule flottante, sinon identique au type de données de l'argument	somme de l'*expression* pour toutes les valeurs en entrée
xmlagg(*expression*)	xml	xml	concaténation de valeurs XML (voir aussi Section 9.14.1.7, « xmlagg »)

En dehors de count, ces fonctions renvoient une valeur NULL si aucune ligne n'est sélectionnée. En particulier, une somme (sum) sur aucune ligne renvoie NULL et non zéro, et array_agg renvoit NULL plutôt qu'un tableau vide quand il n'y a pas de lignes en entrée. La fonction coalesce peut être utilisée pour substituer des zéros ou un tableau vide aux valeurs NULL quand cela est nécessaire.

Note

Les agrégats booléens bool_and et bool_or correspondent aux agrégats standard du SQL every et any ou some. Pour any et some, il semble qu'il y a une ambiguïté dans la syntaxe standard :

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Ici, ANY peut être considéré soit comme introduisant une sous-requête soit comme étant une fonction d'agrégat, si

la sous-requête renvoit une ligne avec une valeur booléenne si l'expression de sélection ne renvoie qu'une ligne. Du coup, le nom standard ne peut être donné à ces agrégats.

Note

Les utilisateurs habitués à travailler avec d'autres systèmes de gestion de bases de données SQL peuvent être surpris par les performances de l'agrégat `count` lorsqu'il est appliqué à la table entière. En particulier, une requête identique à

```
SELECT count(*) FROM ma_table;
```

est exécuté par PostgreSQL™ à l'aide d'un parcours séquentiel de la table entière.

Les fonctions d'agrégat `array_agg`, `string_agg` et `xmlagg`, ainsi que d'autres fonctions similaires d'agrégats définies par l'utilisateur, produisent des valeurs de résultats qui ont un sens différents, dépendant de l'ordre des valeurs en entrée. Cet ordre n'est pas précisé par défaut mais peut être contrôlé en ajoutant une clause `ORDER BY` comme indiquée dans Section 4.2.7, « Expressions d'agrégat ». Une alternative revient à fournir les valeurs à partir d'une sous-requête triée fonctionnera généralement. Par exemple :

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Mais cette syntaxe n'est pas autorisée dans le standard SQL et n'est pas portable vers d'autres systèmes de bases de données.

Tableau 9.43, « Fonctions d'agrégats pour les statistiques » présente les fonctions d'agrégat typiquement utilisées dans l'analyse statistique. (Elles sont séparées pour éviter de grossir la liste des agrégats les plus utilisés.) Là où la description mentionne *N*, cela représente le nombre de lignes en entrée pour lesquelles toutes les expressions en entrée sont non NULL. Dans tous les cas, NULL est renvoyé si le calcul n'a pas de signification, par exemple si *N* vaut zéro.

Tableau 9.43. Fonctions d'agrégats pour les statistiques

Fonction	Type de l'argument	Type renvoyé	Description
`corr(Y, X)`	double precision	double precision	coefficient de corrélation
`covar_pop(Y, X)`	double precision	double precision	covariance de population
`covar_samp(Y, X)`	double precision	double precision	covariance exemple
`regr_avgx(Y, X)`	double precision	double precision	moyenne de la variable indépendante (`sum(X)/N`)
`regr_avgy(Y, X)`	double precision	double precision	moyenne de la variable dépendante (sum(*Y*) / *N*)
`regr_count(Y, X)`	double precision	bigint	nombre de lignes dans lesquelles les deux expressions sont non NULL
`regr_intercept(Y, X)`	double precision	double precision	interception de l'axe y pour l'équation linéaire de la méthode des moindres carrés déterminée par les paires (*X*, *Y*)
`regr_r2(Y, X)`	double precision	double precision	carré du coefficient de corrélation
`regr_slope(Y, X)`	double precision	double precision	inclinaison pour l'équation linéaire de la méthode des moindres carrés déterminée par les paires (*X*, *Y*)
`regr_sxx(Y, X)`	double precision	double precision	`sum(X^2) - sum(X)^2 / N` (« somme des carrés » de la variable indépendante)
`regr_sxy(Y, X)`	double precision	double precision	`sum(X*Y) - sum(X) * sum(Y) / N` (« somme des produits » de la variable indépendante multipliée par la variable dépendante)

Fonction	Type de l'argument	Type renvoyé	Description
`regr_syy(`*`Y`*`, `*`X`*`)`	double precision	double precision	`sum(`*`Y`*`^2) - sum(`*`Y`*`)^2 /` *`N`* (« somme des carrés » de la variable dépendante)
`stddev(`*`expression`*`)`	smallint, int, bigint, real, double precision ou numeric	double precision pour les arguments en virgule flottante, numeric sinon	alias historique pour `stddev_samp`
	smallint, int, bigint, real, double precision ou numeric	double precision pour les arguments en virgule flottante, numeric sinon	déviation standard de la population pour les valeurs en entrée

Fonction	Type de l'argument	Type renvoyé	Description
)			
stddev_samp(*expression*)	smallint, int, bigint, real, double precision ou numeric	double precision pour les arguments en virgule flottante, numeric sinon	exemple de déviation standard pour les valeurs en entrée
variance(*expression*)	smallint, int, bigint, real, double precision ou numeric	double precision pour les arguments en virgule flottante, numeric sinon	alias historique de var_samp
var_pop(*expression*)	smallint, int, bigint, real, double precision ou numeric	double precision pour les arguments en virgule flottante, numeric sinon	variance de la population pour les valeurs en entrée (carré de la déviation standard de la population)
var_samp(*expression*)	smallint, int, bigint, real, double precision ou numeric	double precision pour les arguments en virgule flottante, numeric sinon	exemple de la variance des valeurs en entrée (carré de la déviation standard)

9.19. Fonctions Window

Les *fonction Window* fournissent la possibilité de réaliser des calculs au travers d'ensembles de lignes relatifs à la ligne de la requête en cours. Voir Section 3.5, « Fonctions de fenêtrage » pour une introduction à cette fonctionnalité.

Les fonctions window internes sont listées dans Tableau 9.44, « Fonctions Window généralistes ». Notez que ces fonctions *doivent* être appelées en utilisant la syntaxe des fonctions window ; autrement dit, une clause OVER est requise.

En plus de ces fonctions, toute fonction d'agrégat interne ou définie par l'utilisateur peut être utilisée comme une fonction window (voir Section 9.18, « Fonctions d'agrégat » pour une liste des agrégats internes). Les fonctions d'agrégat agissent comme des fonctions window seulement quand une clause OVER suit l'appel ; sinon elles agissent comme des agrégats standards.

Tableau 9.44. Fonctions Window généralistes

Fonction	Type renvoyé	Description
row_number()	bigint	numéro de la ligne en cours de traitement dans sa partition, en comptant à partir de 1
rank()	bigint	rang de la ligne en cours de traitement, avec des trous ; identique row_number pour le premier pair
dense_rank()	bigint	rang de la ligne en cours de traitement, sans trous ; cette fonction compte les groupes de pairs
percent_rank()	double precision	rang relatif de la ligne en cours de traitement ;: (rank - 1) / (nombre total de lignes - 1)
cume_dist()	double precision	rang relatif de la ligne en cours de traitement : (nombre de lignes précédentes, ou pair de la ligne en cours) / (nombre de lignes total)
ntile(*num_buckets* integer)	integer	entier allant de 1 à la valeur de l'argument, divisant la partition aussi équitablement que possible
lag(*value* any [, *offset* integer [, *default* any]])	même type que *value*	renvoit *value* évalué à la ligne qui est *offset* lignes avant la ligne actuelle à l'intérieur de la partition ; s'il n'y a pas de ligne, renvoit à la place *default*. *offset* et *default* sont évalués par rapport à la ligne en cours. Si omis, *offset* a comme valeur par défaut 1 et *default* est NULL

Fonction	Type renvoyé	Description
`lead(value any [, offset integer [, default any ]])`	same type as `value`	renvoit `value` évalué à la ligne qui est `offset` lignes après la ligne actuelle à l'intérieur de la partition ; s'il n'y a pas de ligne, renvoit à la place `default`. `offset` et `default` sont évalués par rapport à la ligne en cours. Si omis, `offset` a comme valeur par défaut 1 et `default` est NULL
`first_value(value any)`	même type que `value`	renvoit `value` évaluée à la ligne qui est la première ligne du frame window
`last_value(value any)`	même type que `value`	renvoit `value` évaluée à la ligne qui est la dernière ligne du frame window
`nth_value(value any, nth integer)`	même type que `value`	renvoit `value` évaluée à la ligne qui est la`nth`-ième ligne de la frame window (en comptant à partir de 1) ; NULL si aucune ligne

Toutes les fonctions listées dans Tableau 9.44, « Fonctions Window généralistes » dépendent du tri indiqué par la clause `ORDER BY` de la définition window associée. Les lignes qui ne sont pas distinctes dans le tri `ORDER BY` sont des *pairs* ; les quatre fonctions de rang sont définies de ce façon à ce qu'elles donnent la même réponse pour toutes les lignes pairs.

Notez que `first_value`, `last_value` et `nth_value` considèrent seulement les lignes à l'intérieur du « frame window » qui contient par défaut les lignes du début de la partition jusqu'au dernier pair de la ligne en cours. Cela risque de donenr des résultats peu intéressants pour `last_value` et quelque fois aussi pour `nth_value`. Vous pouvez redéfinir la frame en ajoutant une spécification convenable de frame (avec `RANGE` ou `ROWS`) dans la clause `OVER`. Voir Section 4.2.8, « Appels de fonction de fenêtrage » pour plus d'informations sur les spécifications de la frame.

Quand une fonction d'agrégat est utilisée comme fonction window, il aggrège les lignes sur le frame window de la ligne en cours de traitement. Pour obtenir un agrégat sur la partition complète, omettez `ORDER BY` ou utilisez `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. Un agrégat utilisé avec `ORDER BY` et la définition de la frame window par défaut produit un comportement de type « somme en cours d'exécution », qui pourrait ou ne pas être souhaité.

Note

Le standard SQL définit une option `RESPECT NULLS` ou `IGNORE NULLS` pour `lead`, `lag`, `first_value`, `last_value` et `nth_value`. Ceci n'est pas implanté dans PostgreSQL™ : le comportement est toujours le même que dans le comportement par défaut du standard, nommément `RESPECT NULLS`. De la même façon, les options `FROM FIRST` ou `FROM LAST` pour `nth_value` ne sont pas implantées : seul le comportement `FROM FIRST` est supporté par défaut. (Vous pouvez obtenir le résultat d'un `FROM LAST` en inversant l'ordre du `ORDER BY`.)

9.20. Expressions de sous-requêtes

Cette section décrit les expressions de sous-requêtes compatibles SQL disponibles sous PostgreSQL™. Toutes les formes d'expressions documentées dans cette section renvoient des résultats booléens (true/false).

9.20.1. EXISTS

```
EXISTS ( sous-requête )
```

L'argument d'EXISTS est une instruction **SELECT** arbitraire ou une *sous-requête*. La sous-requête est évaluée pour déterminer si elle renvoie des lignes. Si elle en renvoie au moins une, le résultat d'EXISTS est vrai (« true ») ; si elle n'en renvoie aucune, le résultat d'EXISTS est faux (« false »).

La sous-requête peut faire référence à des variables de la requête englobante qui agissent comme des constantes à chaque évaluation de la sous-requête.

La sous-requête n'est habituellement pas exécutée plus qu'il n'est nécessaire pour déterminer si au moins une ligne est renvoyée. Elle n'est donc pas forcément exécutée dans son intégralité. Il est de ce fait fortement déconseillé d'écrire une sous-requête qui présente des effets de bord (tels que l'appel de fonctions de séquence) ; il est extrèmement difficile de prédire si ceux-ci se produisent.

Puisque le résultat ne dépend que d'un éventuel retour de lignes, et pas de leur contenu, la liste des champs retournés par la sous-requête n'a normalement aucun intérêt. Une convention de codage habituelle consiste à écrire tous les tests `EXISTS` sous la forme `EXISTS(SELECT 1 WHERE ...)`. Il y a toutefois des exceptions à cette règle, comme les sous-requêtes utilisant INTERSECT.

L'exemple suivant, simpliste, ressemble à une jointure interne sur `col2` mais il sort au plus une ligne pour chaque ligne de `tab1`, même s'il y a plusieurs correspondances dans les lignes de `tab2` :

```
SELECT col1
FROM tab1
WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.20.2. IN

```
expression IN (sous-requête)
```

Le côté droit est une sous-expression entre parenthèses qui ne peut retourner qu'une seule colonne. L'expression de gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête. Le résultat de IN est vrai (« true ») si une ligne équivalente de la sous-requête est trouvée. Le résultat est faux (« false ») si aucune ligne correspondante n'est trouvée (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne).

Si l'expression de gauche est NULL ou s'il n'existe pas de correspondance avec les valeurs du côté droit et qu'au moins une ligne du côté droit est NULL, le résultat de la construction IN est NULL, et non faux. Ceci est en accord avec les règles normales du SQL pour les combinaisons booléennes de valeurs NULL.

Comme avec EXISTS, on ne peut pas assumer que la sous-requête est évaluée complètement.

```
constructeur_ligne IN (sous-requête)
```

Le côté gauche de cette forme de IN est un constructeur de ligne comme décrit dans la Section 4.2.12, « Constructeurs de lignes ». Le côté droit est une sous-requête entre parenthèses qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions dans le côté gauche. Les expressions côté gauche sont évaluées et comparées ligne à ligne au résultat de la sous-requête. Le résultat de IN est vrai (« true ») si une ligne équivalente de la sous-requête est trouvée. Le résultat est faux (« false ») si aucune ligne correspondante n'est trouvée (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne).

Comme d'habitude, les valeurs NULL dans les lignes sont combinées suivant les règles habituelles des expressions booléennes SQL. Deux lignes sont considérées égales si tous leurs membres correspondant sont non nuls et égaux ; les lignes diffèrent si le contenu de leurs membres sont non nuls et différents ; sinon le résultat de la comparaison de la ligne est inconnu, donc nul. Si tous les résultats par lignes sont différents ou nuls, avec au moins un NULL, alors le résultat de IN est nul.

9.20.3. NOT IN

```
expression NOT IN (sous-requête)
```

Le côté droit est une sous-requête entre parenthèses, qui doit retourner exactement une colonne. L'expression de gauche est évalué et comparée à chaque ligne de résultat de la sous-requête. Le résultat de NOT IN n'est « true » que si des lignes différentes de la sous-requête sont trouvées (ce qui inclut le cas spécial de la sous-requête ne retournant pas de ligne). Le résultat est « false » si une ligne égale est trouvée.

Si l'expression de gauche est nulle, ou qu'il n'y a pas de valeur égale à droite et qu'au moins une ligne de droite est nulle, le résultat du NOT IN est nul, pas vrai. Cela concorde avec les règles normales du SQL pour les combinaisons bouléennes de valeurs nulles.

Comme pour EXISTS, on ne peut assumer que la sous-requête est évaluée dans son intégralité.

```
constructeur_ligne NOT IN (sous-requête)
```

Le côté gauche de cette forme de NOT IN est un constructeur de lignes, comme décrit dans la Section 4.2.12, « Constructeurs de lignes ». Le côté droit est une sous-requête entre parenthèses qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions dans la ligne de gauche. Les expressions de gauche sont évaluées et comparée ligne à ligne au résultat de la sous-requête. Le résultat de NOT IN n'est vrai (« true ») que si seules des lignes différentes de la sous-requête sont trouvées (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne). Le résultat est faux (« false ») si une ligne égale est trouvée.

Comme d'habitude, les valeurs nulles des lignes sont combinées en accord avec les règles normales des expressions bouléennes SQL. Deux lignes sont considérées égales si tous leurs membres correspondants sont non-nuls et égaux ; les lignes sont différentes si les membres correspondants sont non-nuls et différents ; dans tous les autres cas, le résultat de cette comparaison de ligne est inconnu (nul). Si tous les résultats par ligne sont différents ou nuls, avec au minimum un nul, alors le résultat du NOT IN est nul.

9.20.4. ANY/SOME

```
expression opérateur ANY (sous-requête)
expression opérateur SOME (sous-requête)
```

Le côté droit est une sous-requête entre parenthèses qui ne doit retourner qu'une seule colonne. L'expression du côté gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête à l'aide de l'*opérateur* indiqué, ce qui doit aboutir à un résultat booléen. Le résultat de ANY est vrai (« true ») si l'un des résultats est vrai. Le résultat est faux (« false ») si aucun résultat vrai n'est trouvé (ce qui inclut le cas spécial de la requête ne retournant aucune ligne).

SOME est un synonyme de ANY. IN est équivalent à `= ANY`.

En l'absence de succès, mais si au moins une ligne du côté droit conduit à NULL avec l'opérateur, le résultat de la construction ANY est nul et non faux. Ceci est en accord avec les règles standard SQL pour les combinaisons booléenne de valeurs NULL.

Comme pour EXISTS, on ne peut assumer que la sous-requête est évaluée entièrement.

```
constructeur_ligne operator ANY (sous-requête)
constructeur_ligne operator SOME (sous-requête)
```

Le côté gauche de cette forme ANY est un constructeur de ligne, tel que décrit dans la Section 4.2.12, « Constructeurs de lignes ». Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions dans la ligne de gauche. Les expressions du côté gauche sont évaluées et comparées ligne à ligne au résultat de la sous-requête à l'aide de l'*opérateur* donné. Le résultat de ANY est « true » si la comparaison renvoie true pour une ligne quelconque de la sous-requête. Le résultat est « false » si la comparaison renvoie false pour chaque ligne de la sous-requête (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne). Le résultat est NULL si la comparaison ne renvoie true pour aucune ligne, et renvoie NULL pour au moins une ligne.

Voir Section 9.21.5, « Comparaison de lignes entières » pour la signification détaillée d'une comparaison ligne à ligne.

9.20.5. ALL

```
expression opérateur ALL
(sous-requête)
```

Le côté droit est une sous-requête entre parenthèses qui ne doit renvoyer qu'une seule colonne. L'expression gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête à l'aide de l'*opérateur*, ce qui doit renvoyer un résultat booléen. Le résultat de ALL est vrai (« true ») si toutes les lignes renvoient true (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne). Le résultat est faux (« false ») si un résultat faux est découvert. Le résultat est NULL si la comparaison ne renvoie false pour aucune ligne, mais NULL pour au moins une ligne.

NOT IN est équivalent à `<> ALL`.

Comme pour EXISTS, on ne peut assumer que la sous-requête est évaluée entièrement.

```
constructeur_ligne opérateur ALL (sous-requête)
```

Le côté gauche de cette forme de ALL est un constructeur de lignes, tel que décrit dans la Section 4.2.12, « Constructeurs de lignes ». Le côté droit est une sous-requête entre parenthèses qui doit renvoyer exactement le même nombre de colonnes qu'il y a d'expressions dans la colonne de gauche. Les expressions du côté gauche sont évaluées et comparées ligne à ligne au résultat de la sous-requête à l'aide de l'*opérateur* donné. Le résultat de ALL est « true » si la comparaison renvoie true pour toutes les lignes de la sous-requête (ce qui inclut le cas spécial de la sous-requête ne retournant aucune ligne). Le résultat est « false » si la comparaison renvoie false pour une ligne quelconque de la sous-requête. Le résultat est NULL si la comparaison ne renvoie false pour aucune ligne de la sous-requête, mais NULL pour au moins une ligne.

Voir Section 9.21.5, « Comparaison de lignes entières » pour la signification détaillée d'une comparaison ligne à ligne.

9.20.6. Comparaison de lignes

```
constructeur_ligne opérateur (sous-requête)
```

Le côté gauche est un constructeur de lignes, tel que décrit dans la Section 4.2.12, « Constructeurs de lignes ». Le côté droit est une sous-requête entre parenthèses qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions du côté gauche. De plus, la sous-requête ne peut pas renvoyer plus d'une ligne. (Si elle ne renvoie aucune ligne, le résultat est considéré nul.) Le côté gauche est évalué et comparé ligne complète avec la ligne de résultat de la sous-requête.

Voir Section 9.21.5, « Comparaison de lignes entières » pour plus de détails sur la signification d'une comparaison ligne à ligne.

9.21. Comparaisons de lignes et de tableaux

Cette section décrit des constructions adaptées aux comparaisons entre groupes de valeurs. Ces formes sont syntaxiquement liées aux formes des sous-requêtes de la section précédente, mais elles n'impliquent pas de sous-requêtes. Les formes qui impliquent des sous-expressions de tableaux sont des extensions de PostgreSQL™ ; le reste est compatible avec SQL. Toutes les formes d'expression documentées dans cette section renvoient des résultats booléens (true/false).

9.21.1. IN

```
expression IN (valeur [, ...])
```

Le côté droit est une liste entre parenthèses d'expressions scalaires. Le résultat est vrai (« true ») si le côté gauche de l'expression est égal à une des expressions du côté droit. C'est une notation raccourcie de

```
expression = valeur1
OR
expression = valeur2
OR
...
```

Si l'expression du côté gauche renvoie NULL, ou s'il n'y a pas de valeur égale du côté droit et qu'au moins une expression du côté droit renvoie NULL, le résultat de la construction IN est NULL et non pas faux. Ceci est en accord avec les règles du standard SQL pour les combinaisons booléennes de valeurs NULL.

9.21.2. NOT IN

```
expression NOT IN (valeur [, ...])
```

Le côté droit est une liste entre parenthèses d'expressions scalaires. Le résultat est vrai (« true ») si le résultat de l'expression du côté gauche est différent de toutes les expressions du côté droit. C'est une notation raccourcie de

```
expression <> valeur1
AND
expression <> valeur2
AND
...
```

Si l'expression du côté gauche renvoie NULL, ou s'il existe des valeurs différentes du côté droit et qu'au moins une expression du côté droit renvoie NULL, le résultat de la construction NOT IN est NULL et non pas vrai. Ceci est en accord avec les règles du standard du SQL pour les combinaisons booléennes de valeurs NULL.

> **Astuce**
>
> `x NOT IN y` est équivalent à `NOT (x IN y)` dans tout les cas. Néanmoins, les valeurs NULL ont plus de chances de surprendre le novice avec NOT IN qu'avec IN. Quand cela est possible, il est préférable d'exprimer la condition de façon positive.

9.21.3. ANY/SOME (array)

```
expression opérateur ANY (expression tableau)
expression opérateur SOME (expression tableau)
```

Le côté droit est une expression entre parenthèses qui doit renvoyer une valeur de type array. L'expression du côté gauche est évaluée et comparée à chaque élément du tableau en utilisant l'*opérateur* donné, qui doit renvoyer un résultat booléen. Le résultat de ANY est vrai (« true ») si un résultat vrai est obtenu. Le résultat est faux (« false ») si aucun résultat vrai n'est trouvé (ce qui inclut le cas spécial du tableau qui ne contient aucun élément).

Si l'expression de tableau ramène un tableau NULL, le résultat de ANY est NULL. Si l'expression du côté gauche retourne NULL, le résultat de ANY est habituellement NULL (bien qu'un opérateur de comparaison non strict puisse conduire à un résultat différent). De plus, si le tableau du côté droit contient des éléments NULL et qu'aucune comparaison vraie n'est obtenue, le résultat de ANY est NULL, et non pas faux (« false ») (là aussi avec l'hypothèse d'un opérateur de comparaison strict). Ceci est en accord avec les règles du standard SQL pour les combinaisons booléennes de valeurs NULL.

SOME est un synonyme de ANY.

9.21.4. ALL (array)

```
expression opérateur ALL (expression tableau)
```

Le côté droit est une expression entre parenthèses qui doit renvoyer une valeur de type tableau. L'expression du côté gauche est évaluée et comparée à chaque élément du tableau à l'aide de l'*opérateur* donné, qui doit renvoyer un résultat booléen. Le résultat de ALL est vrai (« true ») si toutes les comparaisons renvoient vrai (ce qui inclut le cas spécial du tableau qui ne contient aucun élément). Le résultat est faux (« false ») si un résultat faux est trouvé.

Si l'expression de tableau ramène un tableau NULL, le résultat de ALL est NULL. Si l'expression du côté gauche retourne NULL, le résultat de ALL est habituellement NULL (bien qu'un opérateur de comparaison non strict puisse conduire à un résultat différent). De plus, si le tableau du côté droit contient des éléments NULL et qu'aucune comparaison false n'est obtenue, le résultat de ALL est NULL, et non pas true (là aussi avec l'hypothèse d'un opérateur de comparaison strict). Ceci est en accord avec les règles du standard SQL pour les combinaisons booléennes de valeurs NULL.

9.21.5. Comparaison de lignes entières

```
constructeur_ligne opérateur constructeur_ligne
```

Chaque côté est un constructeur de lignes, tel que décrit dans la Section 4.2.12, « Constructeurs de lignes ». Les deux valeurs de lignes doivent avoir le même nombre de lignes. Chaque côté est évalué. Ils sont alors comparés sur toute la ligne. Les comparaisons de lignes sont autorisées quand l'*opérateur* est =, <>, <, <=, >, >=, ou a une sémantique similaire à l'un d'eux. (Pour être précis, un opérateur peut être un opérateur de comparaison de ligne s'il est membre d'une classe d'opérateurs B-Tree ou est le négateur du membre = d'une classe d'opérateurs B-Tree.)

Les cas = et <> fonctionnent légèrement différemment des autres. Les lignes sont considérées égales si leurs membres correspondants sont non-nuls et égaux ; les lignes sont différentes si des membres correspondants sont non-nuls et différents ; autrement, le résultat de la comparaison de ligne est inconnu (NULL).

Pour les cas <, <=, > et >=, les éléments de ligne sont comparés de gauche à droite. La comparaison s'arrête dès qu'une paire d'éléments différents ou NULL est découverte. Si un des éléments de cette paire est NULL, le résultat de la comparaison de la ligne est inconnu, donc NULL ; sinon la comparaison de cette paire d'éléments détermine le résultat. Par exemple, `ROW(1,2,NULL) < ROW(1,3,0)` est vrai, non NULL, car la troisième paire d'éléments n'est pas considérée.

> **Note**
>
> Avant PostgreSQL™ 8.2, les cas <, <=, > et >= n'étaient pas gérés d'après les spécifications SQL. Une comparaison comme `ROW(a,b) < ROW(c,d)` était codée sous la forme `a < c AND b < d` alors que le bon comportement est équivalent à `a < c OR (a = c AND b < d)`.

```
constructeur_ligne IS DISTINCT FROM constructeur_ligne
```

Cette construction est similaire à une comparaison de ligne <>, mais elle ne conduit pas à un résultat NULL pour des entrées NULL. Au lieu de cela, une valeur NULL est considérée différente (distincte) d'une valeur non-NULL et deux valeurs NULL sont considérées égales (non distinctes). Du coup, le résultat est toujours soit true soit false, jamais NULL.

```
constructeur_ligne IS NOT DISTINCT FROM constructeur_ligne
```

Cette construction est similaire à une comparaison de lignes =, mais elle ne conduit pas à un résultat NULL pour des entrées NULL. Au lieu de cela, une valeur NULL est considérée différente (distincte) d'une valeur non NULL et deux valeurs NULL sont considérées identiques (non distinctes). Du coup, le résultat est toujours soit true soit false, jamais NULL.

> **Note**
>
> Le standard SQL requiert qu'une comparaison d'une ligne complète renvoit NULL si le résultat dépend de la comparaison de deux valeurs NULL ou d'une valeur NULL et d'une valeur non NULL. PostgreSQL™ le fait en comparant le résultat de deux constructeurs de lignes ou en comparant un constructeur de ligne avec le résultat d'une sous-requête (comme dans Section 9.20, « Expressions de sous-requêtes »). Dans d'autres contextes où deux valeurs de type composite sont comparées, deux champs NULL sont considérés égaux, et un champ NULL est considéré plus grand qu'un champ non NULL. Ceci est nécessaire pour voir un comportement de tri et d'indexage cohérent pour les types composites.

9.22. Fonctions retournant des ensembles

Cette section décrit des fonctions qui peuvent renvoyer plus d'une ligne. Actuellement, les seules fonctions dans cette classe sont les fonctions de génération de séries, détaillées dans le Tableau 9.45, « Fonctions de génération de séries » et Tableau 9.46, « Fonctions de génération d'indices ».

Tableau 9.45. Fonctions de génération de séries

Fonction	Type d'argument	Type de retour	Description
`generate_series (`*`début`*`, `*`fin`*`)`	int ou bigint	setof int ou setof bigint (même type que l' argument)	Produit une série de valeurs, de *`début`* à *`fin`* avec un incrément de un.
`generate_series (`*`début`*`, `*`fin`*`, `*`pas`*`)`	int ou bigint	setof int ou setof bigint (même type que l'argument)	Produit une série de valeurs, de *`début`* à *`fin`* avec un incrément de *`pas`*.
`generate_series(`*`début`*`, `*`fin`*`, `*`pas`*` interval)`	timestamp or timestamp with time zone	setof timestamp ou setof timestamp with time zone (identique au type de l'argument)	Génère une série de valeurs, allant de *`start`* à *`stop`* avec une taille pour chaque étape de *`pas`*

Quand *`pas`* est positif, aucune ligne n'est renvoyée si *`début`* est supérieur à *`fin`*. À l'inverse, quand *`pas`* est négatif, aucune ligne n'est renvoyée si *`début`* est inférieur à *`fin`*. De même, aucune ligne n'est renvoyée pour les entrées `NULL`. Une erreur est levée si *`pas`* vaut zéro.

Quelques exemples :

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
 2
 3
 4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
 5
 3
 1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

-- cet exemple se base sur l'opérateur date-plus-entier
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
    dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
```

```
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

Tableau 9.46. Fonctions de génération d'indices

Nom	Type de retour	Description
`generate_subscripts(`*`array anyarray, dim int`*`)`	setof int	Génère une série comprenant les indices du tableau donné.
`generate_subscripts(`*`array anyarray, dim int, reverse boolean`*`)`	setof int	Génère une série comprenant les indices du tableau donné. Quand *`reverse`* vaut true, la série est renvoyé en ordre inverse.

`generate_subscripts` est une fonction qui génère un ensemble d'indices valides pour la dimension indiquée du tableau fourni. Aucune ligne n'est renvoyée pour les tableaux qui n'ont pas la dimension requise ou pour les tableaux NULL (mais les indices valides sont renvoyées pour les éléments d'un tableau NULL). Quelques exemples suivent :

```
-- usage basique
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- presenting an array, the subscript and the subscripted
-- value requires a subquery
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- aplatir un tableau 2D
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
postgres=# SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
```

```
       1
       2
       3
       4
(4 rows)
```

9.23. Fonctions d'informations système

Le Tableau 9.47, « Fonctions d'information de session » présente diverses fonctions qui extraient des informations de session et système.

En plus des fonctions listées dans cette section, il existe plusieurs fonctions relatives au système de statistiques qui fournissent aussi des informations système. Voir Section 27.2.2, « Visualiser les statistiques récupérées » pour plus d'informations.

Tableau 9.47. Fonctions d'information de session

Nom	Type de retour	Description
`current_catalog`	name	nom de la base de données en cours (appelée « catalog » dans le standard SQL)
`current_database()`	nom	nom de la base de données courante
`current_schema[()]`	nom	nom du schéma courant
`current_schemas(boolean)`	nom[]	nom des schémas dans le chemin de recherche, avec optionnellement les schémas implicites
`current_user`	nom	nom d'utilisateur du contexte d'exécution courant
`current_query`	text	texte la requête en cours d'exécution, tel qu'elle a été soumise par le client (pourrait contenir plus d'une instruction)
`pg_backend_pid()`	int	Identifiant du processus serveur attaché à la session en cours
`pg_listening_channels()`	setof text	noms de canaux que la session est en train d'écouter
`inet_client_addr()`	inet	adresse de la connexion distante
`inet_client_port()`	int	port de la connexion distante
`inet_server_addr()`	inet	adresse de la connexion locale
`inet_server_port()`	int	port de la connexion locale
`pg_my_temp_schema()`	oid	OID du schéma temporaire de la session, 0 si aucun
`pg_is_other_temp_schema(oid)`	boolean	s'agit-il du schéma temporaire d'une autre session ?
`pg_postmaster_start_time()`	timestamp with time zone	heure de démarrage du serveur
`pg_conf_load_time()`	timestamp with time zone	heure de chargement de la configuration
`session_user`	name	nom de l'utilisateur de session
`user`	name	équivalent à `current_user`
`version()`	text	informations de version de PostgreSQL™

Note

`current_catalog`, `current_schema`, `current_user`, `session_user`, et `user` ont un statut syntaxique spécial en SQL : ils doivent être appelés sans parenthèses à droite (optionnel avec PostgreSQL dans le cas de `current_schema`).

`session_user` est habituellement l'utilisateur qui a initié la connexion à la base de données ; mais les superutilisateurs peuvent modifier ce paramétrage avec SET SESSION AUTHORIZATION(7). `current_user` est l'identifiant de l'utilisateur, utilisable pour les vérifications de permissions. Il est habituellement identique à l'utilisateur de la session, mais il peut être modifié avec SET ROLE(7). Il change aussi pendant l'exécution des fonctions comprenant l'attribut `SECURITY DEFINER`. En langage Unix,

l'utilisateur de la session est le « real user » (NdT : l'utilisateur réel) et l'utilisateur courant est l'« effective user » (NdT : l'utilisateur effectif) .

`current_schema` renvoie le nom du premier schéma dans le chemin de recherche (ou une valeur NULL si ce dernier est vide). C'est le schéma utilisé pour toute création de table ou autre objet nommé sans précision d'un schéma cible. `current_schemas(boolean)` renvoie un tableau qui contient les noms de tous les schémas du chemin de recherche. L'option booléenne indique si les schémas système implicitement inclus, comme `pg_catalog`, doivent être inclus dans le chemin de recherche retourné.

> **Note**
>
> Le chemin de recherche est modifiable à l'exécution. La commande est :
>
> ```
> SET search_path TO schema [, schema, ...]
> ```

`pg_listening_channels` renvoit un ensemble de noms de canaux que la session actuelle écoute. Voir LISTEN(7) pour plus d'informations.

`inet_client_addr` renvoie l'adresse IP du client courant et `inet_client_port` le numéro du port. `inet_server_addr` renvoie l'adresse IP sur laquelle le serveur a accepté la connexion courante et `inet_server_port` le numéro du port. Toutes ces fonctions renvoient NULL si la connexion courante est établie via une socket de domaine Unix.

`pg_my_temp_schema` renvoie l'OID du schéma temporaire de la session courante, ou 0 s'il n'existe pas (parce qu'il n'y a pas eu de création de tables temporaires). `pg_is_other_temp_schema` renvoie true si l'OID donné est l'OID d'un schéma temporaire d'une autre session. (Ceci peut être utile pour exclure les tables temporaires d'autres sessions lors de l'affichage d'un catalogue, par exemple.)

`pg_postmaster_start_time` renvoie la date et l'heure (type timestamp with time zone) de démarrage du serveur.

`pg_conf_load_time` renvoit timestamp with time zone indiquant à quel moment les fichiers de configuration du serveur ont été chargés. (Si la session en cours était déjà là à ce moment, ce sera le moment où la sessions elle-même a relu les fichiers de configurations. Cela veut dire que ce que renvoit cette fonction peut varier un peu suivant les sessions. Sinon, c'est le temps où le processus maître a relu les fichiers de configuration.)

`version` renvoie une chaîne qui décrit la version du serveur PostgreSQL™.

Le Tableau 9.48, « Fonctions de consultation des privilèges d'accès » liste les fonctions qui permettent aux utilisateurs de consulter les privilèges d'accès. Voir la Section 5.6, « Droits » pour plus d'informations sur les privilèges.

Tableau 9.48. Fonctions de consultation des privilèges d'accès

Nom	Type de retour	Description
use `has_any_column_privilege(`*r*`, `*table*`, `*privilege*`)`	boolean	l'utilisateur a-t-il un droit sur une des colonnes de cette table
tab `has_any_column_privilege(`*le*`, `*privilege*`)`	boolean	l'utilisateur actuel a-t-il un droit sur une des colonnes de cette table
`has_column_privilege(`*user*`, `*table*`, `*column*`, `*privilege*`)`	boolean	l'utilisateur a-t-il un droit sur la colonne
`has_column_privilege(`*table*`, `*column*`, `*privilege*`)`	boolean	l'utilisateur actuel a-t-il un droit sur la colonne
`has_database_privilege (`*utilisateur*`, `*base*`, `*privilège*`)`	boolean	*utilisateur* a-t-il le privilège *privilège* sur *base*
`has_database_privilege (`*base*`, `*privilège*`)`	boolean	l'utilisateur courant a-t-il le privilège *privilège* sur *base*
`has_foreign_data_wrapper_privilege(`*user*`, `*fdw*`, `*privilege*`)`	boolean	l'utilisateur a-t-il un droit sur ce wrapper de données distantes
`has_foreign_data_wrapper_privilege(`*fdw*`, `*privilege*`)`	boolean	l'utilisateur actuel a-t-il un droit sur ce wrapper de données distantes
`has_function_privilege (`*utilisateur*`, `*fonction*`, `*privi-*	boolean	*utilisateur* a-t-il le privilège *privilège* sur *fonction*

Nom	Type de retour	Description
lège)		
has_function_privilege (*fonction*, *privilège*)	boolean	l'utilisateur courant a-t-il e privilège *privilège* sur *fonction*
has_language_privilege (*utilisateur*, *langage*, *privilège*)	boolean	*utilisateur* a-t-il le privilège *privilège* sur *langage*
has_language_privilege (*langage*, *droit*)	boolean	l'utilisateur courant a-t-il le privilège *privilège* sur *langage*
has_schema_privilege(*utilisateur*, *schéma*, *privilège*)	boolean	*utilisateur* a-t-il le privilège *privilège* sur *schéma*
has_schema_privilege(*schéma*, *privilège*)	boolean	l'utilisateur courant a-t-il le privilège *privilège* sur *schéma*
has_server_privilege(*user*, *server*, *privilege*)	boolean	l'utilisateur a-t-il un droit sur ce serveur distant
has_server_privilege(*server*, *privilege*)	boolean	l'utilisateur actuel a-t-il un droit sur ce serveur distant
has_sequence_privilege(*user*, *sequence*, *privilege*)	boolean	l'utilisateur actuel a-t-il un droit sur cette séquence
has_sequence_privilege(*sequence*, *privilege*)	boolean	l'utilisateur actuel a-t-il un droit sur cette séquence
has_table_privilege(*utilisateur*, *table*, *privilège*)	boolean	*utilisateur* a-t-il le privilège *privilège* sur *table*
has_table_privilege(*table*, *privilege*)	boolean	l'utilisateur courant a-t-il le privilège *privilège* sur *table*
has_tablespace_privilege (*utilisateur*, *tablespace*, *privilège*)	boolean	*utilisateur* a-t-il le privilège *privilège* sur *tablespace*
has_tablespace_privilege (*tablespace*, *privilège*)	boolean	l'utilisateur courant a-t-il le privilège *privilège* sur *tablespace*
pg_has_role(*utilisateur*, *rôle*, *privilège*)	boolean	*utilisateur* a-t-il le privilège *privilège* sur *rôle*
pg_has_role(*rôle*, *privilège*)	boolean	l'utilisateur courant a-t-il le privilège *privilège* sur *rôle*

has_table_privilege vérifie si l'utilisateur possède un privilège particulier d'accès à une table. L'utilisateur peut être indiqué par son nom ou son OID (pg_authid.oid). Si l'argument est omis, current_user est utilisé. La table peut être indiquée par son nom ou par son OID. (Il existe donc six versions de has_table_privilege qui se distinguent par le nombre et le type de leurs arguments.) Lors de l'indication par nom, il est possible de préciser le schéma. Les privilèges possibles, indiqués sous la forme d'une chaîne de caractères, sont : SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES ou TRIGGER. En option, WITH GRANT OPTION peut être ajouté à un type de droit pour tester si le droit est obtenu avec l'option « grant ». De plus, plusieurs types de droit peuvent être listés, séparés par des virgules, auquel cas le résultat sera true si un des droits listés est obtenu. (la casse des droits n'a pas d'importance et les espaces blancs supplémentaires sont autorisés entre mais pas dans le nom des droits.) Certains exemples :

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

has_sequence_privilege vérifie si un utilisateur peut accéder à une séquence d'une façon ou d'une autre. Les arguments sont analogues à ceux de la fonction has_table_privilege. Le type de droit d'accès doit valoir soit USAGE, soit SELECT soit UPDATE.

has_any_column_privilege vérifie si un utilisateur peut accéder à une colonne d'une table d'une façon particulière. Les possibilités pour que ces arguments correspondent à ceux de has_table_privilege, sauf que le type de droit d'accès désiré doit être évalué à une combinaison de SELECT, INSERT, UPDATE ou REFERENCES. Notez qu'avoir un droit au niveau de la table le donne implicitement pour chaque colonne de la table, donc has_any_column_privilege renverra toujours true si

`has_table_privilege` le fait pour les mêmes arguments. Mais `has_any_column_privilege` réussit aussi s'il y a un droit « grant » sur une colonne pour ce droit.

`has_column_privilege` vérifie si un utilisateur peut accéder à une colonne d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`, avec un supplément : la colonne doit être indiquée soit par nom soit par numéro d'attribut. Le type de droit d'accès désiré doit être une combinaison de `SELECT`, `INSERT`, `UPDATE` ou `REFERENCES`. Notez qu'avoir un de ces droits au niveau table les donne implicitement pour chaque colonne de la table.

`has_database_privilege` vérifie si un utilisateur peut accéder à une base de données d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`. Le type de droit d'accès désiré doit être une combinaison de `CREATE`, `CONNECT`, `TEMPORARY` ou `TEMP` (qui est équivalent à `TEMPORARY`).

`has_function_privilege` vérifie si un utilisateur peut accéder à une fonction d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`. Lors de la spécification d'une fonction par une chaîne texte plutôt que par un OID, l'entrée autorisée est la même que pour le type de données regprocedure (voir Section 8.16, « Types identifiant d'objet »). Le type de droit d'accès désiré doit être `EXECUTE`. Voici un exemple :

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

`has_foreign_data_wrapper_privilege` vérifie si un utilisateur peut accéder à un wrapper de données distantes d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`. Le type de droit d'accès désiré doit être `USAGE`.

`has_language_privilege` vérifie si un utilisateur peut accéder à un langage de procédure d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`. Le type de droit d'accès désiré doit être `USAGE`.

`has_schema_privilege` vérifie si un utilisateur peut accéder à un schéma d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`. Le type de droits d'accès désiré doit être une combinaison de `CREATE` et `USAGE`.

`has_server_privilege` vérifie si un utilisateur peut accéder à un serveur distant d'une façon particulière. Les possibilités pour ses arguments sont analogues à `has_table_privilege`. Le type de droit d'accès désiré doit être `USAGE`.

`has_tablespace_privilege` vérifie si l'utilisateur possède un privilège particulier d'accès à un *tablespace*. Ses arguments sont analogues à `has_table_privilege`. Le seul privilège possible est `CREATE`.

`pg_has_role` vérifie si l'utilisateur possède un privilège particulier d'accès à un rôle. Ses arguments sont analogues à `has_table_privilege`. Le privilège doit être une combinaison de `MEMBER` et `USAGE`. `MEMBER` indique une appartenance directe ou indirecte au rôle (c'est-à-dire le droit d'exécuter `SET ROLE`) alors que `USAGE` indique que les droits du rôle sont immédiatement disponibles sans avoir à exécuter `SET ROLE`.

Le Tableau 9.49, « Fonctions d'interrogation de visibilité dans les schémas » affiche les fonctions qui permettent de savoir si un objet particulier est *visible* dans le chemin de recherche courant. Une table est dite visible si son schéma contenant est dans le chemin de recherche et qu'aucune table de même nom ne la précède dans le chemin de recherche. C'est équivalent au fait que la table peut être référencée par son nom sans qualification explicite de schéma. Par exemple, pour lister les noms de toutes les tables visibles :

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Tableau 9.49. Fonctions d'interrogation de visibilité dans les schémas

Nom	Type de retour	Description
`pg_conversion_is_visible(`*`conversion_oid`*`)`	boolean	la conversion est-elle visible dans le chemin de recherche
`pg_function_is_visible(`*`function_oid`*`)`	boolean	la fonction est-elle visible dans le chemin de recherche
`pg_operator_is_visible(`*`operator_oid`*`)`	boolean	l'opérateur est-il visible dans le chemin de recherche
`pg_opclass_is_visible(`*`opclass_oid`*`)`	boolean	la classe d'opérateur est-elle visible dans le chemin de recherche
`pg_table_is_visible(`*`table_oid`*`)`	boolean	la table est-elle visible dans le chemin de recherche
`pg_ts_config_is_visible(`*`config_oid`*`)`	boolean	la configuration de la recherche textuelle est-elle visible dans le chemin de recherche
`pg_ts_dict_is_visible(`*`dict_oid`*`)`	boolean	le dictionnaire de recherche textuelle est-il visible dans le chemin de recherche
`pg_ts_parser_is_visible(`*`parser_oid`*`)`	boolean	l'analyseur syntaxique de recherche textuelle est-il vi-

Nom	Type de retour	Description
		sible dans le chemin de recherche
pg_ts_template_is_visible(*template_oid*)	boolean	le modèle de recherche textuelle est-il visible dans le chemin de recherche
pg_type_is_visible(*type_oid*)	boolean	le type (ou domaine) est-il visible dans le chemin de recherche

Chaque fonction vérifie la visibilité d'un type d'objet de la base de données. `pg_table_is_visible` peut aussi être utilisée avec des vues, index et séquences, `pg_type_is_visible` avec les domaines. Pour les fonctions et les opérateurs, un objet est visible dans le chemin de recherche si aucun objet de même nom *et prenant des arguments de mêmes types de données* n'est précédemment présent dans le chemin de recherche. Pour les classes d'opérateurs, on considère à la fois le nom et la méthode d'accès à l'index associé.

Toutes ces fonctions nécessitent des OID pour identifier les objets à vérifier. Pour tester un objet par son nom, il est préférable d'utiliser les types d'alias d'OID (regclass, regtype, regprocedure ou regoperator). Par exemple

```
SELECT pg_type_is_visible('mon_schema.widget'::regtype);
```

Il n'est pas très utile de tester ainsi un nom non qualifié -- si le nom peut être reconnu, c'est qu'il est visible.

Le Tableau 9.50, « Fonctions d'information du catalogue système » liste les fonctions qui extraient des informations des catalogues système.

Tableau 9.50. Fonctions d'information du catalogue système

Nom	Type de retour	Description
format_type (*type_oid*, *typemod*)	text	récupère le nom SQL d'un type de données
pg_get_keywords()	setof record	obtient une liste des mots clés SQL et leurs catégories
pg_get_constraintdef(*constraint_oid*)	text	récupère la définition d'une contrainte
pg_get_constraintdef(*constraint_oid*, *pretty_bool*)	text	récupère la définition d'une contrainte
pg_get_expr(*expr_text*, *relation_oid*)	text	décompile la forme interne d'une expression, en supposant que toutes les variables qu'elle contient font référence à la relation indiquée par le second paramètre
pg_get_expr(*expr_text*, *relation_oid*, *pretty_bool*)	text	décompile la forme interne d'une expression, en supposant que toutes les variables qu'elle contient font référence à la relation indiquée par le second paramètre
pg_get_functiondef(*func_oid*)	text	obtient une définition de la fonction
pg_get_function_arguments(*func_oid*)	text	obtient une définition de la liste des arguments de la fonction (avec les valeurs par défaut)
pg_get_function_identity_arguments (*func_oid*)	text	obtient une définition de la liste des arguments de la fonction (sans valeurs par défaut)
pg_get_function_result(*func_oid*)	text	obtient la clause `RETURNS` pour la fonction
pg_get_indexdef(*index_oid*)	text	récupère la commande **CREATE INDEX** de l'index
pg_get_indexdef(*index_oid*, *column_no*, *pretty_bool*)	text	récupère la commande **CREATE INDEX** pour l'index, ou la définition d'une seule colonne d'index quand *column_no* ne vaut pas zéro
pg_get_ruledef(*rule_oid*)	text	récupère la commande **CREATE RULE** pour une règle
pg_get_ruledef(*rule_oid*, *pretty_bool*)	text	récupère la commande **CREATE RULE** de la règle
pg_get_serial_sequence(*table_name*, *column_name*)	text	récupère le nom de la séquence qu'une colonne serial ou bigserial utilise
pg_get_triggerdef(*trigger_oid*)	text	récupère la commande **CREATE [CONSTRAINT] TRIGGER** du trigger
pg_get_triggerdef(*trigger_oid*, *pretty_bool*)	text	récupère la commande **CREATE [CONSTRAINT] TRIGGER** du déclencheur

Nom	Type de retour	Description
`pg_get_userbyid(role_oid)`	name	récupère le nom du rôle possédant cet OID
`pg_get_viewdef(view_name)`	text	récupère la commande **SELECT** sous-jacente à la vue (*obsolète*)
`pg_get_viewdef(view_name, pretty_bool)`	text	récupère la commande **SELECT** sous-jacente à la vue (*obsolète*)
`pg_get_viewdef(view_oid)`	text	récupère la commande **SELECT** sous-jacente à la vue
`pg_get_viewdef(view_oid, pretty_bool)`	text	récupère la commande **SELECT** sous-jacente à la vue
`pg_tablespace_databases(tables-pace_oid)`	setof oid	récupère l'ensemble des OID des bases qui possèdent des objets dans ce tablespace
`pg_typeof(any)`	regtype	obtient le type de données de toute valeur

`format_type` renvoie le nom SQL d'un type de données identifié par son OID de type et éventuellement un modificateur de type. On passe NULL pour le modificateur de type si aucun modificateur spécifique n'est connu.

`pg_get_keywords` renvoit un ensemble d'enregistrements décrivant les mots clés SQL reconnus par le serveur. La colonne *`word`* contient le mot clé. La colonne *`catcode`* contient un code de catégorie : `U` pour non réservé, `C` pour nom de colonne, `T` pour nom d'un type ou d'une fonction et `R` pour réservé. La colonne *`catdesc`* contient une chaîne pouvant être traduite décrivant la catégorie.

`pg_get_constraintdef`, `pg_get_indexdef`, `pg_get_ruledef` et `pg_get_triggerdef` reconstruisent respectivement la commande de création d'une contrainte, d'un index, d'une règle ou d'un déclencheur. (Il s'agit d'une reconstruction décompilée, pas du texte originale de la commande.) `pg_get_expr` décompile la forme interne d'une expression individuelle, comme la valeur par défaut d'une colonne. Cela peut être utile pour examiner le contenu des catalogues système. Si l'expression contient des variables, spécifiez l'OID de la relation à laquelle elles font référence dans le second paramètre ; si aucune variable n'est attendue, zéro est suffisant. `pg_get_viewdef` reconstruit la requête **SELECT** qui définit une vue. La plupart de ces fonctions existent en deux versions, l'une d'elles permettant, optionnellement, d'« afficher joliment » le résultat. Ce format est plus lisible, mais il est probable que les futures versions de PostgreSQL™ continuent d'interprêter le format par défaut actuel de la même façon ; la version « jolie » doit être évitée dans les sauvegardes. Passer `false` pour le paramètre de « jolie » sortie conduit au même résultat que la variante sans ce paramètre.

`pg_get_functiondef` renvoit une instruction **CREATE OR REPLACE FUNCTION** complète pour une fonction. `pg_get_function_arguments` renvoit une liste des arguments d'un fonction, de la façon dont elle apparaîtrait dans **CREATE FUNCTION**. `pg_get_function_result` renvoit de façon similaire la clause `RETURNS` appropriée pour la fonction. `pg_get_function_identity_arguments` renvoit la liste d'arguments nécessaire pour identifier une fonction, dans la forme qu'elle devrait avoir pour faire partie d'un **ALTER FUNCTION**, par exemple. Cette forme omet les valeurs par défaut.

`pg_get_serial_sequence` renvoie le nom de la séquence associée à une colonne ou NULL si aucune séquence n'est associée à la colonne. Le premier argument en entrée est un nom de table, éventuellement qualifié du schéma. Le second paramètre est un nom de colonne. Comme le premier paramètre peut contenir le nom du schéma et de la table, il n'est pas traité comme un identifiant entre guillemets doubles, ce qui signifie qu'il est converti en minuscules par défaut, alors que le second paramètre, simple nom de colonne, est traité comme s'il était entre guillemets doubles et sa casse est préservée. La fonction renvoie une valeur convenablement formatée pour être traitée par les fonctions de traitement des séquences (voir Section 9.15, « Fonctions de manipulation de séquences »). Cette association peut être modifiée ou supprimée avec **ALTER SEQUENCE OWNED BY**. (La fonction aurait probablement dû s'appeler `pg_get_owned_sequence` ; son nom reflète le fait qu'elle est typiquement utilisée avec les colonnes serial et bigserial.)

`pg_get_userbyid` récupère le nom d'un rôle d'après son OID.

`pg_tablespace_databases` autorise l'examen d'un *tablespace*. Il renvoie l'ensemble des OID des bases qui possèdent des objets stockés dans le *tablespace*. Si la fonction renvoie une ligne, le *tablespace* n'est pas vide et ne peut pas être supprimée. Pour afficher les objets spécifiques peuplant le *tablespace*, il est nécessaire de se connecter aux bases identifiées par `pg_tablespace_databases` et de requêter le catalogue pg_class.

`pg_typeof` renvoit l'OID du type de données de la valeur qui lui est passé. Ceci est utile pour dépanner ou pour construire dynamiquement des requêtes SQL. La fonction est déclarée comme renvoyant regtype, qui est une type d'alias d'OID (voir Section 8.16, « Types identifiant d'objet ») ; cela signifie que c'est la même chose qu'un OID pour un bit de comparaison mais que cela s'affiche comme un nom de type. Par exemple :

```
SELECT pg_typeof(33);

 pg_typeof
```

```
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

Les fonctions affichées dans Tableau 9.51, « Fonctions d'informations sur les commentaires » extraient les commentaires stockées précédemment avec la commande COMMENT(7). Une valeur NULL est renvoyée si aucun commentaire ne correspond aux paramètres donnés.

Tableau 9.51. Fonctions d'informations sur les commentaires

Nom	Type de retour	Description
`col_description(`*`table_oid, column_number`*`)`	text	récupère le commentaire d'une colonne de la table
`obj_description (`*`object_oid, catalog_name`*`)`	text	récupère le commentaire d'un objet de la base de données
`obj_description(`*`object_oid`*`)`	text	récupère le commentaire d'un objet de la base de données (*obsolète*)
`shobj_description(`*`object_oid, catalog_name`*`)`	text	récupère le commentaire d'un objet partagé de la base de données

`col_description` renvoie le commentaire d'une colonne de table, la colonne étant précisée par l'OID de la table et son numéro de colonne. `obj_description` ne peut pas être utilisée pour les colonnes de table car les colonnes n'ont pas d'OID propres.

La forme à deux paramètres de `obj_description` renvoie le commentaire d'un objet de la base de données, précisé par son OID et le nom du catalogue système le contenant. Par exemple, `obj_description(123456,'pg_class')` récupère le commentaire pour la table d'OID 123456. La forme à un paramètre de `obj_description` ne requiert que l'OID de l'objet. Elle est maintenant obsolète car il n'existe aucune garantie que les OID soient uniques au travers des différents catalogues système ; un mauvais commentaire peut alors être renvoyé.

`shobj_description` est utilisé comme `obj_description`, mais pour les commentaires des objets partagés. Certains catalogues systèmes sont globaux à toutes les bases de données à l'intérieur de chaque cluster et leurs descriptions sont stockées globalement.

Les fonctions présentées dans Tableau 9.52, « ID de transaction et instantanés » remontent à l'utilisateur des informations de transaction de niveau interne au serveur. L'usage principal de ces fonctions est de déterminer les transactions commitées entre deux instantanés (« snapshots »).

Tableau 9.52. ID de transaction et instantanés

Nom	Type retour	Description
`txid_current()`	bigint	récupère l'ID de transaction courant
`txid_current_snapshot()`	txid_snapshot	récupère l'instantané courant
`txid_snap` `txid_snapshot_xmin(`*`shot`*`)`	bigint	récupère le xmin de l'instantané
`txid_snap` `txid_snapshot_xmax(`*`shot`*`)`	bigint	récupère le xmax de l'instantané
`txid_snaps` `txid_snapshot_xip(`*`hot`*`)`	setof bigint	récupère l'ID de la transaction en cours dans l'instantané
`txid_visible_in_snapshot(`*`bigint, txid_snapshot`*`)`	boolean	l'ID de transaction est-il visible dans l'instantané ? (ne pas utiliser les identifiants de sous-transactions)

Le type interne ID de transaction (xid) est sur 32 bits. Il boucle donc tous les 4 milliards de transactions. Cependant, ces fonctions exportent au format 64 bits, étendu par un compteur « epoch », de façon à éviter tout cycle sur la durée de vie de l'installation. Le

type de données utilisé par ces fonctions, txid_snapshot, stocke l'information de visibilité des ID de transaction à un instant particulier. Ces composants sont décrits dans Tableau 9.53, « Composants de l'instantané ».

Tableau 9.53. Composants de l'instantané

Nom	Description
xmin	ID de transaction (txid) le plus ancien encore actif. Toutes les transactions plus anciennes sont soient commitées et visibles, soient annulées et mortes.
xmax	Premier txid non encore assigné. Tous les txids plus grands ou égals à celui-ci ne sont pas encore démarrés à ce moment de l'instantané, et donc invisibles.
xip_list	Active les identifiants de transactions (txids) au moment de la prise de l'image. La liste inclut seulement les identifiants actifs entre `xmin` et `xmax` ; il pourrait y avoir des identifiants plus gros que `xmax`. Un identifiant qui est `xmin <= txid < xmax` et qui n'est pas dans cette liste est déjà terminé au moment de la prise de l'image, et du coup est soit visible soit mort suivant son statut de validation. La liste n'inclut pas les identifiants de transactions des sous-transactions.

La représentation textuelle du txid_snapshot est *`xmin`*`:`*`xmax`*`:`*`xip_list`*. Ainsi `10:20:10,14,15` signifie `xmin=10, xmax=20, xip_list=10, 14, 15`.

9.24. Fonctions d'administration système

Le Tableau 9.54, « Fonctions agissant sur les paramètres de configuration » affiche les fonctions disponibles pour consulter et modifier les paramètres de configuration en exécution.

Tableau 9.54. Fonctions agissant sur les paramètres de configuration

Nom	Type de retour	Description
`current_setting (`*`nom_paramètre`*`)`	text	valeur courante du paramètre
`set_config (`*`nom_paramètre, nouvelle_valeur, est_local`*`)`	text	configure le paramètre et renvoie la nouvelle valeur

La fonction `current_setting` renvoie la valeur courante du paramètre *`nom_paramètre`*. Elle correspond à la commande SQL **SHOW**. Par exemple :

```
SELECT current_setting('datestyle');

 current_setting
-----------------
ISO, MDY
(1 row)
```

`set_config` positionne le paramètre *`nom_paramètre`* à *`nouvelle_valeur`*. Si *`est_local`* vaut `true`, la nouvelle valeur s'applique uniquement à la transaction en cours. Si la nouvelle valeur doit s'appliquer à la session en cours, on utilise `false`. La fonction correspond à la commande SQL **SET**. Par exemple :

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
off
(1 row)
```

Les fonctions présentées dans le Tableau 9.55, « Fonctions d'envoi de signal au serveur » envoient des signaux de contrôle aux autres processus serveur. L'utilisation de ces fonctions est restreinte aux superutilisateurs.

Tableau 9.55. Fonctions d'envoi de signal au serveur

Nom	Type de retour	Description
pg_cancel_backend (*pid* int)	boolean	Annule une requête en cours sur le serveur
pg_terminate_backend(*pid* int)	boolean	Termine une processus serveur
pg_reload_conf()	boolean	Impose le rechargement des fichiers de configuration par les processus serveur
pg_rotate_logfile()	boolean	Impose une rotation du journal des traces du serveur

Ces fonctions renvoient `true` en cas de succès, `false` en cas d'échec.

`pg_cancel_backend` et `pg_terminate_backend` envoie un signal (respectivement `SIGINT` ou `SIGTERM`) au processus serveur identifié par l'ID du processus. L'identifiant du processus serveur actif peut être trouvé dans la colonne *`procpid`* dans la vue pg_stat_activity ou en listant les processus **postgres** sur le serveur avec ps sur Unix ou le Gestionnaire des tâches sur Windows™.

`pg_reload_conf` envoie un signal `SIGHUP` au serveur, ce qui impose le rechargement des fichiers de configuration par tous les processus serveur.

`pg_rotate_logfile` signale au gestionnaire de journaux de trace de basculer immédiatement vers un nouveau fichier de sortie. Cela ne fonctionne que lorsque le collecteur de traces interne est actif, puisqu'il n'y a pas de sous-processus de gestion des fichiers journaux dans le cas contraire.

Les fonctions présentées dans le Tableau 9.56, « Fonctions de contrôle de la sauvegarde » aident à l'exécution de sauvegardes à chaud. Ces fonctions ne peuvent pas être exécutées lors d'une restauration. L'utilisation des trois premières fonctions est réservée aux superutilisateurs.

Tableau 9.56. Fonctions de contrôle de la sauvegarde

Nom	Type de retour	Description
pg_start_backup (*label* text [, *fast* boolean])	text	Préparation de la sauvegarde à chaud
pg_stop_backup()	text	Arrêt de la sauvegarde à chaud
pg_switch_xlog()	text	Passage forcé à un nouveau journal de transactions
pg_current_xlog_location()	text	Récupération de l'emplacement d'écriture du journal de transactions courant
pg_current_xlog_insert_location()	text	Récupération de l'emplacement d'insertion du journal de transactions courant
pg_xlogfile_name_offset(*location* text)	text, integer	Conversion de la chaîne décrivant l'emplacement du journal de transactions en nom de fichier et décalage en octets dans le fichier
pg_xlogfile_name(*location* text)	text	Conversion de la chaîne décrivant l'emplacement du journal de transactions en nom de fichier

`pg_start_backup` accepte un label utilisateur de la sauvegarde (typiquement, le nom du fichier d'enregistrement de la sauvegarde). La fonction écrit un fichier de label (`backup_label`) dans le répertoire de données du cluster, réalise un point de retournement, et renvoie la position du début de la sauvegarde dans le journal de transactions au format texte. Ce résultat ne nécessite pas qu'on s'y intéresse, mais il est fourni dans cette éventualité.

```
postgres=# select pg_start_backup('le_label_ici');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

Il existe un second paramètre booléen optionnel. Si `true`, il précise l'exécution de `pg_start_backup` aussi rapidement que possible. Cela force un point de retournement immédiat qui causera un pic dans les opérations d'entrées/sorties, ralentissant toutes les requêtes exécutées en parallèle.

`pg_stop_backup` supprime le fichier de label créé par `pg_start_backup` et crée, à la place, un fichier d'historique dans l'aire de stockage des archives des journaux de transactions. Ce fichier contient le label passé à `pg_start_backup`, les emplacements de début et de fin des journaux de transactions correspondant à la sauvegarde et les heures de début et de fin de la sauvegarde. La valeur de retour est l'emplacement du journal de la transaction de fin de sauvegarde (de peu d'intérêt, là encore). Après

notification de l'emplacement de fin, le point d'insertion courant du journal de transactions est automatiquement avancé au prochain journal de transactions, de façon à ce que le journal de transactions de fin de sauvegarde puisse être archivé immédiatement pour terminer la sauvegarde.

`pg_switch_xlog` bascule sur le prochain journal de transactions, ce qui permet d'archiver le journal courant (en supposant que l'archivage continu soit utilisé). La fonction retourne l'emplacement de la transaction finale + 1 dans le journal ainsi terminé. S'il n'y a pas eu d'activité dans les journaux de transactions depuis le dernier changement de journal, `pg_switch_xlog` ne fait rien et renvoie l'emplacement de fin du journal de transactions en cours.

`pg_current_xlog_location` affiche la position d'écriture du journal de transactions en cours dans le même format que celui utilisé dans les fonctions ci-dessus. De façon similaire, `pg_current_xlog_insert_location` affiche le point d'insertion dans le journal de transactions courant. Le point d'insertion est la fin « logique » du journal de transactions à tout instant alors que l'emplacement d'écriture est la fin de ce qui a déjà été écrit à partir des tampons internes du serveur. La position d'écriture est la fin de ce qui peut être examiné extérieurement au serveur. C'est habituellement l'information nécessaire à qui souhaite archiver des journaux de transactions partiels. Le point d'insertion n'est donné principalement que pour des raisons de débogage du serveur. Il s'agit là d'opérations de lecture seule qui ne nécessitent pas de droits superutilisateur.

`pg_xlogfile_name_offset` peut être utilisée pour extraire le nom du journal de transactions correspondant et le décalage en octets à partir du résultat de n'importe quelle fonction ci-dessus. Par exemple :

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000000D |     4039624
(1 row)
```

De façon similaire, `pg_xlogfile_name` n'extrait que le nom du journal de la transaction. Quand la position dans le journal de la transaction donnée est exactement sur une limite de journal, les deux fonctions renvoient le nom du journal précédent. C'est généralement le comportement souhaité pour gérer l'archivage des journaux, car le fichier précédent est le dernier à devoir être archivé.

Pour les détails sur le bon usage de ces fonctions, voir la Section 24.3, « Archivage continu et récupération d'un instantané (PITR) ».

Les fonctions affichées dans Tableau 9.57, « Fonctions d'information sur la restauration » fournissent des informations sur le statut actuel du serveur en attente. Ces fonctions peuvent être utilisées lors d'une restauration mais aussi lors d'un fonctionnement normal.

Tableau 9.57. Fonctions d'information sur la restauration

Nom	Type du retour	Description
`pg_is_in_recovery()`	bool	True si la restauration est en cours.
`pg_last_xlog_receive_location()`	text	Récupère l'emplacement de la dernière transaction reçue et synchronisée sur disque par la réplication en flux. Lorsque cette dernière est en cours d'exécution, l'emplacement aura une progression monotone. Par contre, quand la réplication en flux est redémarrée, elle reviendra à la position de départ de la réplication, typiquement au début du journal de transactions contenant l'emplacement actuel de rejeu. Si la restauration a terminé, elle deviendra statique et aura comme valeur celui du dernier enregistrement de transaction reçu et synchronisé sur disque lors de la restauration. Si la réplication en flux est désactivé ou si elle n'a pas encore commencé, la fonction renvoie NULL.
	text	Récupère l'emplacement du dernier enregistrement WAL rejoué lors de la restauration. Si la restauration est toujours en cours, cela va augmenter progressivement. Si la restauration s'est terminée, alors cette valeur restera statique et dépendera du dernier enregistrement WAL reçu et syn-

Nom	Type du retour	Description
`()`		chronisé sur disque lors de cette restauration. Quand le serveur a été lancé sans restauration de flux, la valeur renvoyée par la fonction sera NULL.

Les fonctions présentées dans le Tableau 9.58, « Fonctions de calcul de la taille des objets de la base de données » calculent l'utilisation de l'espace disque par les objets de la base de données.

Tableau 9.58. Fonctions de calcul de la taille des objets de la base de données

Nom	Code de retour	Description
`pg_column_size(any)`	int	Nombre d'octets utilisés pour stocker une valeur particulière (éventuellement compressée)
`pg_total_relation_size(regclass)`	bigint	Espace disque total utilisé par la table dont l'OID ou le nom est indiqué, en incluant toutes les données TOAST et les index
`pg_table_size(regclass)`	bigint	Espace disque utilisé par la table dont l'OID ou le nom est indiqué, en excluant les index (mais en incluant les données TOAST, la carte des espaces libres et la carte de visibilité)
`pg_indexes_size(regclass)`	bigint	Espace disque total utilisé par les index attachés à la table dont l'OID ou le nom est indiqué
`pg_database_size(oid)`	bigint	Espace disque utilisé par la base de données d'OID indiqué
`pg_database_size(name)`	bigint	Espace disque utilisé par la base de données de nom indiqué
`pg_tablespace_size(oid)`	bigint	Espace disque utilisé par le tablespace ayant cet OID
`pg_tablespace_size(name)`	bigint	Espace disque utilisé par le tablespace ayant ce nom
`pg_relation_size(`*`relation`* `regclass,` *`fork`* `text)`	bigint	Espace disque utilisé par le fork indiqué, `'main'`, `'fsm'` ou `'vm'`, d'une table ou index d'OID ou de nom indiqué.
`pg_relation_size(`*`relation`* `regclass)`	bigint	Raccourci pour `pg_relation_size(..., 'main')`
`pg_size_pretty(bigint)`	text	Convertit une taille en octets en format interprétable par l'utilisateur avec unités

`pg_column_size` affiche l'espace utilisé pour stocker toute valeur individuelle.

`pg_total_relation_size` accepte en argument l'OID ou le nom d'une table ou d'une table TOAST. Elle renvoit l'espace disque total utilisé par cette table, incluant les index associés. Cette fonction est équivalenté à `pg_table_size + pg_indexes_size`.

`pg_table_size` accepte en argument l'OID ou le nom d'une table et renvoie l'espace disque nécessaire pour cette table, à l'exclusion des index (espace des données TOAST, carte des espaces libres et carte de visibilité inclus.)

`pg_indexes_size` accepte en argument l'OID ou le nom d'une table et renvoit l'espace disque total utilisé par tous les index attachés à cette table.

`pg_database_size` et `pg_tablespace_size` acceptent l'OID ou le nom d'une base de données ou d'un *tablespace* et renvoient l'espace disque total utilisé. Pour utiliser `pg_database_size`, vous devez avoir l'attribut `CONNECT` sur la base de données indiquée (qui est donné par défaut). Pour utiliser `pg_tablespace_size`, vous devez avoir l'attribut `CREATE` sur le tablespace indiqué, sauf s'il s'agit du tablespace par défaut de la base courante.

`pg_relation_size` accepte l'OID ou le nom d'une table, d'un index ou d'une table toast et renvoie la taille sur disque en octets. Indiquer `'main'` ou laisser le second argument renvoit la taille du morceau de données principal de la relation. Indiquer `'fsm'` renvoit la taille de la carte des espaces libres (voir Section 54.3, « Carte des espaces libres ») associée à cette relation. Indiquer `'vm'` renvoit la taille de la carte de visibilité (voir Section 54.4, « Carte de visibilité ») associée avec la relation. Notez que cette fonction affiche la taille d'un seul fichier ; dans la plupart des cas, c'est plus simple à utiliser que les fonctions de haut-niveau

comme `pg_total_relation_size` ou `pg_table_size`.

`pg_size_pretty` peut être utilisé pour formater le résultat d'une des autres fonctions de façon interprétable par l'utilisateur, en utilisant kB, MB, GB ou TB suivant le cas.

Les fonctions affichées dans Tableau 9.59, « Fonctions de récupération de l'emplacement des objets de la base de données » facilitent l'identification des fichiers associées aux objets de la base de données.

Tableau 9.59. Fonctions de récupération de l'emplacement des objets de la base de données

Nom	Type en retour	Description
`pg_relation_filenode(`*`relation`* `regclass)`	oid	Numéro filenode de la relation dont l'OID ou le nom est précisé
`pg_relation_filepath(`*`relation`* `regclass)`	text	Chemin et nom du fichier pour la relation dont l'OID ou le nom est précisé

`pg_relation_filenode` accepte l'OID ou le nom d'une table, d'un index, d'une séquence ou d'une table TOAST. Elle renvoit le numéro « filenode » qui lui est affecté. Ce numéro est le composant de base du nom de fichier utilisé par la relation (voir Section 54.1, « Emplacement des fichiers de la base de données » pour plus d'informations). Pour la plupart des tables, le résultat est identique à pg_class.*`relfilenode`* mais pour certains catalogues système, *`relfilenode`* vaut zéro et cette fonction doit être utilisée pour obtenir la bonne valeur. La fonction renvoit NULL si l'objet qui lui est fourni est une relation qui n'a pas de stockage, par exemple une vue.

`pg_relation_filepath` est similaire à `pg_relation_filenode` mais elle renvoit le chemin complet vers le fichier (relatif au répertoire des données de l'instance, `PGDATA`) de la relation.

Les fonctions présentées dans le Tableau 9.60, « Fonctions d'accès générique aux fichiers » fournissent un accès natif aux fichiers situés sur le serveur. Seuls les fichiers contenus dans le répertoire du cluster et ceux du répertoire `log_directory` sont accessibles. On utilise un chemin relatif pour les fichiers contenus dans le répertoire du cluster et un chemin correspondant à la configuration du paramètre `log_directory` pour les journaux de trace. L'utilisation de ces fonctions est restreinte aux superutilisateurs.

Tableau 9.60. Fonctions d'accès générique aux fichiers

Nom	Code de retour	Description
`pg_ls_dir(`*`nom_répertoire`* `text)`	setof text	Liste le contenu d'un répertoire
`pg_read_file(`*`nom_fichier`* `text,` *`décalage`* `bigint,` *`longueur`* `bigint)`	text	Renvoie le contenu d'un fichier texte
`pg_stat_file(`*`nom_fichier`* `text)`	record	Renvoie les informations concernant un fichier

`pg_ls_dir` renvoie tous les noms contenus dans le répertoire indiqué, à l'exception des entrées spéciales « `.` » et « `..` ».

`pg_read_file` renvoie une partie d'un fichier texte, débutant au *`décalage`* indiqué, renvoyant au plus *`longueur`* octets (moins si la fin du fichier est atteinte avant). Si le *`décalage`* est négatif, il est relatif à la fin du fichier.

`pg_stat_file` renvoie un enregistrement contenant la taille du fichier, les date et heure de dernier accès, les date et heure de dernière modification, les date et heure de dernier changement de statut (plateformes Unix seulement), les date et heure de création (Windows seulement) et un booléen indiquant s'il s'agit d'un répertoire. Les usages habituels incluent :

```
SELECT * FROM pg_stat_file('nomfichier');
SELECT (pg_stat_file('nomfichier')).modification;
```

Les fonctions présentées dans Tableau 9.61, « Fonctions de verrous consultatifs » gèrent les verrous consultatifs. Pour les détails sur le bon usage de ces fonctions, voir Section 13.3.4, « Verrous informatifs ».

Tableau 9.61. Fonctions de verrous consultatifs

Nom	Type renvoyé	Description
`pg_advisory_lock(`*`key`* `bigint)`	void	Obtient un verrou consultatif exclusif
`pg_advisory_lock(`*`key1`* `int,` *`key2`* `int)`	void	Obtient un verrou consultatif exclusif
`pg_advisory_lock_shared(`*`key`*	void	Obtient un verrou consultatif partagé

Nom	Type renvoyé	Description
bigint)		
pg_advisory_lock_shared(*key1* int, *key2* int)	void	Obtient un verrou consultatif partagé
pg_try_advisory_lock(*key* bigint)	boolean	Obtient un verrou consultatif exclusif si disponible
pg_try_advisory_lock(*key1* int, *key2* int)	boolean	Obtient un verrou consultatif exclusif si disponible
	boolean	Obtient un verrou consultatif partagé si disponible

Nom	Type renvoyé	Description
key bigint)		
	boolean	Obtient un verrou consultatif partagé si disponible

Nom	Type renvoyé	Description
key1 int, *key2* int)		
pg_advisory_unlock(*key* bigint)	boolean	Relâche un verrou consultatif exclusif
pg_advisory_unlock(*key1* int, *key2* int)	boolean	Relâche un verrou consultatif exclusif
ke pg_advisory_unlock_shared(*y* bigint)	boolean	Relâche un verrou consultatif partagé
ke pg_advisory_unlock_shared(*y1* int, *key2* int)	boolean	Relâche un verrou consultatif partagé
pg_advisory_unlock_all()	void	Relâche tous les verrous consultatifs détenus par la session en cours

pg_advisory_lock verrouille une ressource applicative qui peut être identifiée soit par une valeur de clé sur 64 bits soit par deux valeurs de clé sur 32 bits (les deux espaces de clé ne se surchargent pas). The key type is specified in pg_locks.objid. Si une autre session détient déjà un verrou sur la même ressource, la fonction attend que la ressource devienne disponible. Le verrou est exclusif. Les demandes de verrou s'empilent de sorte que, si une même ressource est verrouillée trois fois, elle doit être déverrouillée trois fois pour être disponible par les autres sessions.

pg_advisory_lock_shared fonctionne de façon identique à pg_advisory_lock sauf que le verrou peut être partagé avec d'autres sessions qui réclament des verrous partagés. Seules les demandes de verrou exclusif sont bloquées.

pg_try_advisory_lock est similaire à pg_advisory_lock sauf que la fonction n'attend pas la disponibilité du verrou. Si le verrou peut être obtenu immédiatement, la fonction renvoie true, sinon, elle renvoie false.

pg_try_advisory_lock_shared fonctionne de la même façon que pg_try_advisory_lock sauf qu'elle tente d'acquérir un verrou partagé au lieu d'un verrou exclusif.

pg_advisory_unlock relâche un verrou exclusif précédemment acquis. Elle retourne true si le verrou est relaché avec succès. Si le verrou n'était pas détenu, false est renvoyé et un message d'avertissement SQL est émis par le serveur.

pg_advisory_unlock_shared fonctionne de la même façon que pg_advisory_unlock mais pour relâcher un verrou partagé.

pg_advisory_unlock_all relâche tous les verrous consultatifs détenus par la session courante. (Cette fonction est appelée implicitement à la fin de la session, même si le client se déconnecte brutalement.)

9.25. Fonctions trigger

Actuellement, PostgreSQL™ fournit une fonction de trigger interne, suppress_redundant_updates_trigger, qui empêchera toute mise à jour qui ne modifie pas réellement les données de cette ligne, en contrate au comportement normal qui réalise toujours une mise à jour, que les données soient réellement changées ou pas. (Ce comportement normal fait que les mises à jour s'exécutent rapidement car aucune vérification n'est nécessaire et c'est aussi utile dans certains cas.)

Idéalement, vous devriez normalement éviter d'exécuter des mises à jour qui ne modifient pas réellement les données de l'enregistrement. Les mise à jour redondantes peuvent coûter considérablement en temps d'exécution, tout spécialement si vous avez beaucoup d'index à modifier, et en espace dans des lignes mortes que vous devrez finir par VACUUMées. Néanmoins, la détection de telles situations dans le code client n'est pas toujours facile, voire même possible, et écrire des expressions pour détecter ce type de cas peut facilement amener des erreyrs. Une alternative est d'utiliser suppress_redundant_updates_trigger, qui ignorera les mises à jour qui ne modifient pas réellement les données. Néanmoins, vous devez être très prudent quant à son utilisation. Le trigger consomme un temps petit, mais à ne pas négliger, pour vérifier que la mise à jour doit se faire. Autrement dit, si la plupart des enregistrements affectés par une mise à jour seront réellement modifiés, utiliser ce trigger rendra la mise à jour bien plus lente.

La fonction suppress_redundant_updates_trigger peut être ajoutée à une table de cette façon :

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

Dans la plupart des cas, vous voudrez déclencher ce tigger en dernier pour chaque ligne. Gardez en tête que les triggers sont dé-

clenchés dans l'ordre alphabétique de leur nom, vous choisirez donc un nom de trigger qui vient apr_s le nom des autres trigegrs que vous pourriez avoir sur la table.

Pour plus d'informations sur la création des trigger, voir CREATE TRIGGER(7).

Chapitre 10. Conversion de types

Le mélange de différents types de données dans la même expression peut être requis, intentionnellement ou pas, par les instructions SQL. PostgreSQL™ possède des fonctionnalités étendues pour évaluer les expressions de type mixte.

Dans la plupart des cas, un utilisateur n'aura pas besoin de comprendre les détails du mécanisme de conversion des types. Cependant, les conversions implicites faites par PostgreSQL™ peuvent affecter le résultat d'une requête. Quand cela est nécessaire, ces résultats peuvent être atteints directement en utilisant la conversion *explicite* de types.

Ce chapitre introduit les mécanismes et les conventions sur les conversions de types dans PostgreSQL™. Référez-vous aux sections appropriées du Chapitre 8, Types de données et du Chapitre 9, Fonctions et opérateurs pour plus d'informations sur les types de données spécifiques, les fonctions et les opérateurs autorisés.

10.1. Aperçu

SQL est un langage fortement typé. C'est-à-dire que chaque élément de données est associé à un type de données qui détermine son comportement et son utilisation permise. PostgreSQL™ a un système de types extensible qui est beaucoup plus général et flexible que les autres implémentations de SQL. Par conséquent, la plupart des comportements de conversion de types dans PostgreSQL™ est régie par des règles générales plutôt que par une heuristique *ad hoc*. Cela permet aux expressions de types mixtes d'être significatives même avec des types définis par l'utilisateur.

L'analyseur de PostgreSQL™ divise les éléments lexicaux en cinq catégories fondamentales : les entiers, les nombres non entiers, les chaînes de caractères, les identifieurs et les mots-clé. Les constantes de la plupart des types non-numériques sont d'abord classifiées comme chaînes de caractères. La définition du langage SQL permet de spécifier le nom des types avec une chaîne et ce mécanisme peut être utilisé dans PostgreSQL™ pour lancer l'analyseur sur le bon chemin. Par exemple, la requête :

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

a deux constantes littérales, de type text et point. Si un type n'est pas spécifié pour une chaîne littérale, alors le type unknown est assigné initialement pour être résolu dans les étapes ultérieures comme décrit plus bas.

Il y a quatre constructions SQL fondamentales qui exigent des règles distinctes de conversion de types dans l'analyseur de PostgreSQL™ :

Les appels de fonctions
: Une grande partie du système de types de PostgreSQL™ est construit autour d'un riche ensemble de fonctions. Les fonctions peuvent avoir un ou plusieurs arguments. Puisque que PostgreSQL™ permet la surcharge des fonctions, le nom seul de la fonction n'identifie pas de manière unique la fonction à appeler ; l'analyseur doit sélectionner la bonne fonction par rapport aux types des arguments fournis.

Les opérateurs
: PostgreSQL™ autorise les expressions avec des opérateurs de préfixe et de suffixe unaires (un argument) aussi bien que binaires (deux arguments). Comme les fonctions, les opérateurs peuvent être surchargés. Du coup, le même problème existe pour sélectionner le bon opérateur.

Le stockage des valeurs
: Les instructions SQL **INSERT** et **UPDATE** placent le résultat des expressions dans une table. Les expressions dans une instruction doivent être en accord avec le type des colonnes cibles et peuvent être converties vers celles-ci.

Les constructions `UNION`, `CASE` et des constructions relatives
: Comme toutes les requêtes issues d'une instruction **SELECT** utilisant une union doivent apparaître dans un ensemble unique de colonnes, les types de résultats de chaque instruction **SELECT** doivent être assortis et convertis en un ensemble uniforme. De façon similaire, les expressions de résultats d'une construction `CASE` doivent être converties vers un type commun de façon à ce que l'ensemble de l'expression `CASE` ait un type de sortie connu. Cela est la même chose pour les constructions avec `ARRAY` et pour les fonctions `GREATEST` et `LEAST`.

Les catalogues systèmes stockent les informations concernant l'existence de conversions entre certains types de données et la façon d'exécuter ces conversions. Les conversions sont appelées *casts* en anglais. Des conversions de types supplémentaires peuvent être ajoutées par l'utilisateur avec la commande CREATE CAST(7) (c'est habituellement réalisé en conjonction avec la définition de nouveaux types de données. L'ensemble des conversions entre les types prédéfinis a été soigneusement choisi et le mieux est de ne pas le modifier).

Une heuristique supplémentaire est fournie dans l'analyseur pour permettre de meilleures estimations sur la bonne conversion de type parmi un groupe de types qui ont des conversions implicites. Les types de données sont divisées en plusieurs *catégories de type* basiques, incluant boolean, numeric, string, bitstring, datetime, timespan, geometric, network et définis par l'utilisateur. (Pour une liste, voir Tableau 45.45, « Codes `typcategory` » ; mais notez qu'il est aussi possible de créer des catégories de type personnalisées.) À l'intérieur de chaque catégorie, il peut y avoir une ou plusieurs *types préférés*, qui sont sélectionnés quand il y a un choix possible de types. Avec une sélection attentive des types préférés et des conversions implicites disponibles, il est possible de s'assurer que les expressions ambigues (celles avec plusieurs solutions candidates) peuvent être résolus d'une façon utile.

Toutes les règles de conversions de types sont écrites en gardant à l'esprit plusieurs principes :

- Les conversions implicites ne doivent jamais avoir de résultats surprenants ou imprévisibles.
- Il n'y aura pas de surcharge depuis l'analyseur ou l'exécuteur si une requête n'a pas besoin d'une conversion implicite de types. C'est-à-dire que si une requête est bien formulée et si les types sont déjà bien distinguables, alors la requête devra s'exécuter sans perte de temps supplémentaire et sans introduire à l'intérieur de celle-ci des appels à des conversions implicites non nécessaires.

  De plus, si une requête nécessite habituellement une conversion implicite pour une fonction et si l'utilisateur définit une nouvelle fonction avec les types des arguments corrects, l'analyseur devrait utiliser cette nouvelle fonction et ne fera plus des conversions implicites en utilisant l'ancienne fonction.

10.2. Opérateurs

L'opérateur spécifique qui est référence par une expression d'opérateur est déterminé par la procédure ci-dessous. Notez que cette procédure est indirectement affectée par l'ordre d'insertion des opérateurs car cela va déterminer les sous-expressions prises en entrée des opérateurs. Voir la Section 4.1.6, « Précédence lexicale » pour plus d'informations.

Procédure 10.1. Résolution de types pour les opérateurs

1. Sélectionner les opérateurs à examiner depuis le catalogue système `pg_operator`. Si un nom non-qualifié d'opérateur était utilisé (le cas habituel), les opérateurs examinés sont ceux avec un nom et un nombre d'arguments corrects et qui sont visibles dans le chemin de recherche courant (voir la Section 5.7.3, « Chemin de parcours des schémas »). Si un nom qualifié d'opérateur a été donné, seuls les opérateurs dans le schéma spécifié sont examinés.

   - Si un chemin de recherche trouve de nombreux opérateurs avec des types d'arguments identiques, seul sera examiné celui apparaissant le plus tôt dans le chemin. Mais les opérateurs avec des types d'arguments différents sont examinés sur une base d'égalité indépendamment de leur position dans le chemin de recherche.

2. Vérifier que l'opérateur accepte le type exact des arguments en entrée. Si un opérateur existe (il peut en avoir uniquement un qui corresponde exactement dans l'ensemble des opérateurs considérés), utiliser cet opérateur.

   - Si un argument lors d'une invocation d'opérateur binaire est de type unknown (NdT : inconnu), alors considérer pour ce contrôle que c'est le même type que l'autre argument. Les invocations impliquant deux entrées de type unknown, ou un opérateur unitaire avec en entrée une donnée de type unknown ne trouveront jamais une correspondance à ce niveau.

3. Rechercher la meilleure correspondance.

   a. Se débarrasser des opérateurs candidats pour lesquels les types en entrée ne correspondent pas et qui ne peuvent pas être convertis (en utilisant une conversion implicite) dans le type correspondant. Le type unknown est supposé être convertible vers tout. Si un candidat reste, l'utiliser, sinon aller à la prochaine étape.

   b. Parcourir tous les candidats et garder ceux avec la correspondance la plus exacte par rapport aux types en entrée (les domaines sont considérés de la même façon que leur type de base pour cette étape). Garder tous les candidats si aucun n'a de correspondance exacte. Si un seul candidat reste, l'utiliser ; sinon, aller à la prochaine étape.

   c. Parcourir tous les candidats et garder ceux qui acceptent les types préférés (de la catégorie des types de données en entrée) aux positions où la conversion de types aurait été requise. Garder tous les candidats si aucun n'accepte les types préférés. Si seulement un candidat reste, l'utiliser ; sinon aller à la prochaine étape.

   d. Si des arguments en entrée sont inconnus, vérifier la catégorie des types acceptés à la position de ces arguments par les candidats restants. À chaque position, sélectionner la catégorie chaîne de caractères si un des candidats accepte cette catégorie (cette préférence vers les chaînes de caractères est appropriée car le terme type-inconnu ressemble à une chaîne de caractères). Dans le cas contraire, si tous les candidats restants acceptent la même catégorie de types, sélectionner

cette catégorie. Dans le cas contraire, échouer car le choix correct ne peut pas être déduit sans plus d'indices. Se débarrasser maintenant des candidats qui n'acceptent pas la catégorie sélectionnée. De plus, si des candidats acceptent un type préféré de cette catégorie, se débarrasser des candidats qui acceptent, pour cet argument, les types qui ne sont pas préférés.

e. Si un seul candidat reste, l'utiliser. Sinon, échouer.

Quelques exemples suivent.

Exemple 10.1. Résolution du type d'opérateur factoriel

Il n'existe qu'un seul opérateur factoriel (`!` postfix) défini dans le catalogue standard. Il prend un argument de type bigint. Le scanner affecte au début le type integer à l'argument dans cette expression :

```
SELECT 40 ! AS "40 factorial";

                   40 factorial
--------------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

L'analyseur fait donc une conversion de types sur l'opérande et la requête est équivalente à

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Exemple 10.2. Résolution de types pour les opérateurs de concaténation de chaînes

La syntaxe d'une chaîne de caractères est utilisée pour travailler avec les types chaînes mais aussi avec les types d'extensions complexes. Les chaînes de caractères avec un type non spécifié sont comparées avec les opérateurs candidats probables.

Un exemple avec un argument non spécifié :

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

Dans ce cas, l'analyseur cherche à voir s'il existe un opérateur prenant text pour ses deux arguments. Comme il y en a, il suppose que le second argument devra être interprété comme un type text.

Voici une concaténation sur des types non spécifiés :

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

Dans ce cas, il n'y a aucune allusion initiale sur quel type utiliser puisqu'aucun type n'est spécifié dans la requête. Donc, l'analyseur regarde pour tous les opérateurs candidats et trouve qu'il existe des candidats acceptant en entrée la catégorie chaîne de caractères (string) et la catégorie morceaux de chaînes (bit-string). Puisque la catégorie chaînes de caractères est préférée quand elle est disponible, cette catégorie est sélectionnée. Le type préféré pour la catégorie chaînes étant text, ce type est utilisé comme le type spécifique pour résoudre les types inconnus.

Exemple 10.3. Résolution de types pour les opérateurs de valeur absolue et de négation

Le catalogue d'opérateurs de PostgreSQL™ a plusieurs entrées pour l'opérateur de préfixe `@`. Ces entrées implémentent toutes des opérations de valeur absolue pour des types de données numériques variées. Une de ces entrées est pour le type float8 (réel) qui

est le type préféré dans la catégorie des numériques. Par conséquent, PostgreSQL™ utilisera cette entrée quand il sera en face d'un argument de type unknown :

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Le système a compris implicitement que le litéral de type unknown est de type float8 (réel) avant d'appliquer l'opérateur choisi. Nous pouvons vérifier que float8, et pas un autre type, a été utilisé :

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

D'un autre côté, l'opérateur préfixe ~ (négation bit par bit) est défini seulement pour les types entiers et non pas pour float8 (réel). Ainsi, si nous essayons un cas similaire avec ~, nous obtenons :

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add explicit
type casts.
```

Ceci se produit parce que le système ne peut pas décider quel opérateur doit être préféré parmi les différents opérateurs ~ possibles. Nous pouvons l'aider avec une conversion explicite :

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

10.3. Fonctions

La fonction spécifique référencée par un appel de fonction est déterminée selon les étapes suivantes.

Procédure 10.2. Résolution de types pour les fonctions

1. Sélectionner les fonctions à examiner depuis le catalogue système `pg_proc`. Si un nom non-qualifié de fonction était utilisé, les fonctions examinées sont celles avec un nom et un nombre d'arguments corrects et qui sont visibles dans le chemin de recherche courant (voir la Section 5.7.3, « Chemin de parcours des schémas »). Si un nom qualifié de fonctions a été donné, seules les fonctions dans le schéma spécifique sont examinées.

   a. Si un chemin de recherche trouve de nombreuses fonctions avec des types d'arguments identiques, seule celle apparaissant le plus tôt dans le chemin sera examinée. Mais les fonctions avec des types d'arguments différents sont examinées sur une base d'égalité indépendamment de leur position dans le chemin de recherche.

   b. Si une fonction est déclarée avec un paramètre `VARIADIC` et que l'appel n'utilise pas le mot clé `VARIADIC`, alors la fonction est traitée comme si le paramètre tableau était remplacé par une ou plusieurs occurrences de son type élémentaire, autant que nécessaire pour correspondre à l'appel. Après cette expansion, la fonction pourrait avoir des types d'arguments identiques à certaines fonctions non variadic. Dans ce cas, la fonction apparaissant plus tôt dans le chemin de recherche est utilisée ou, si les deux fonctions sont dans le même schéma, celle qui n'est pas VARIADIC est préférée.

   c. Les fonctions qui ont des valeurs par défaut pour les paramètres sont considérés comme correspondant à un appel qui omet zéro ou plus des paramètres ayant des valeurs par défaut. Si plus d'une fonction de ce type correspondent à un appel, celui apparaissant en premier dans le chemin des schémas est utilisé. S'il existe deux ou plus de ces fonctions dans le même schémas avec les même types de paramètres pour les paramètres sans valeur par défaut (ce qui est possible s'ils ont des ensembles différents de paramètres par défaut), le système ne sera pas capable de déterminer laquelle sélectionnée, ce qui résultera en une erreur « ambiguous function call ».

2. Vérifier que la fonction accepte le type exact des arguments en entrée. Si une fonction existe (il peut en avoir uniquement une qui correspond exactement dans tout l'ensemble des fonctions considérées), utiliser cette fonction (les cas impliquant le type unknown ne trouveront jamais de correspondance à cette étape).

3. Si aucune correspondance n'est trouvée, vérifier si l'appel à la fonction apparaît être une requête spéciale de conversion de types. Cela arrive si l'appel à la fonction a juste un argument et si le nom de la fonction est le même que le nom (interne) de certains types de données. De plus, l'argument de la fonction doit être soit un type inconnu soit un type qui a une compatibilité binaire avec le type de données nommés, soit un type qui peut être converti dans le type de données indiqué en appliquant les fonctions d'entrées/sorties du type (c'est-à-dire que la conversion est vers ou à partir d'un type standard de chaîne). Quand ces conditions sont rencontrées, l'appel de la fonction est traité sous la forme d'une spécification `CAST`. [1]
4. Regarder pour la meilleure correspondance.
   a. Se débarrasser des fonctions candidates pour lesquelles les types en entrée ne correspondent pas et qui ne peuvent pas être convertis (en utilisant une conversion implicite) pour correspondre. Le type unknown est supposé être convertible vers n'importe quoi. Si un seul candidat reste, utiliser le ; sinon, aller à la prochaine étape.
   b. Parcourir tous les candidats et garder ceux avec la correspondance la plus exacte par rapport aux types en entrée (les domaines sont considérés de la même façon que leur type de base pour cette étape). Garder tous les candidats si aucun n'a de correspondance exacte. Si un seul candidat reste, utiliser le ; sinon, aller à la prochaine étape.
   c. Parcourir tous les candidats et garder ceux qui acceptent les types préférés (de la catégorie des types de données en entrée) aux positions où la conversion de types aurait été requise. Garder tous les candidats si aucun n'accepte les types préférés. Si un seul candidat reste, utiliser le ; sinon, aller à la prochaine étape.
   d. Si des arguments en entrée sont unknown, vérifier les catégories de types acceptées à la position de ces arguments par les candidats restants. À chaque position, sélectionner la catégorie chaîne de caractères si un des candidats accepte cette catégorie (cette préférence envers les chaînes de caractères est appropriée depuis que le terme type-inconnu ressemble à une chaîne de caractères). Dans le cas contraire, si tous les candidats restants acceptent la même catégorie de types, sélectionner cette catégorie. Dans le cas contraire, échouer car le choix correct ne peut pas être déduit sans plus d'indices. Se débarrasser maintenant des candidats qui n'acceptent pas la catégorie sélectionnée. De plus, si des candidats acceptent un type préféré dans cette catégorie, se débarrasser des candidats qui acceptent, pour cet argument, les types qui ne sont pas préférés.
   e. Si un seul candidat reste, utiliser le. Sinon, échouer.

Notez que les règles de « correspondance optimale » sont identiques pour la résolution de types concernant les opérateurs et les fonctions. Quelques exemples suivent.

Exemple 10.4. Résolution de types pour les arguments de la fonction arrondie

Il n'existe qu'une seule fonction `round` avec deux arguments (le premier est de type numeric, le second est de type integer). Ainsi, la requête suivante convertie automatiquement le type du premier argument de integer vers numeric.

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

La requête est en fait transformée par l'analyseur en

```
SELECT round(CAST (4 AS numeric), 4);
```

Puisque le type numeric est initialement assigné aux constantes numériques avec un point décimal, la requête suivante ne requièrera pas une conversion de types et pourra par conséquent être un peu plus efficace :

```
SELECT round(4.0, 4);
```

Exemple 10.5. Résolution de types pour les fonctions retournant un segment de chaîne

Il existe plusieurs fonctions `substr`, une d'entre elles prend les types text et integer. Si cette fonction est appelée avec une constante de chaînes d'un type inconnu, le système choisi la fonction candidate qui accepte un argument issu de la catégorie préférée string (c'est-à-dire de type text).

```
SELECT substr('1234', 3);
```

[1] Elle est simplement nommée suivant le nom du type en sortie et donc il n'est pas nécessaire d'avoir un cas spécial. Pour plus d'informations, voir CREATE CAST(7).

```
 substr
--------
     34
(1 row)
```

Si la chaîne de caractères est déclarée comme étant du type varchar (chaîne de caractères de longueur variable), ce qui peut être le cas si elle vient d'une table, alors l'analyseur essayera de la convertir en text :

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

Ceci est transformé par l'analyseur en

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

Note

L'analyseur apprend depuis le catalogue pg_cast que les types text et varchar ont une compatibilité binaire, ce qui veut dire que l'un peut être passé à une fonction qui accepte l'autre sans avoir à faire aucune conversion physique. Par conséquent, aucun appel de conversion de types n'est réellement inséré dans ce cas.

Et si la fonction est appelée avec un argument de type integer, l'analyseur essaie de le convertir en text :

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

Ceci ne fonctionne pas car integer n'a pas de conversion implicite vers text. Néanmoins, une conversion explicite fonctionnera :

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. Stockage de valeurs

Les valeurs qui doivent être insérées dans une table sont converties vers le type de données de la colonne de destination selon les règles suivantes.

Procédure 10.3. Conversion de types pour le stockage de valeurs

1. Vérifier qu'il y a une correspondance exacte avec la cible.
2. Dans le cas contraire, essayer de convertir l'expression vers le type cible. Cela réussira s'il y a une conversion (cast) enregistrée entre ces deux types. Si une expression est de type inconnu, le contenu de la chaîne littérale sera fourni à l'entrée de la routine de conversion pour le type cible.
3. Vérifier s'il y a une conversion de taille pour le type cible. Une conversion de taille est une conversion d'un type vers lui-même. Si elle est trouvée dans le catalogue pg_cast, appliquez-la à l'expression avant de la stocker dans la colonne de destination. La fonction d'implémentation pour une telle conversion prend toujours un paramètre supplémentaire de type integer, qui reçoit la valeur `atttypmod` de la colonne de +destination (typiquement sa longueur déclarée, bien que l'interprétation +de `atttypmod` varie suivant le type de +données). Elle pourrait accepter un troisième paramètre, de type +booléen, précisant si la conversion est explicite ou implicite. La +fonction de conversion est responsable de l'application de toute +sémantique dépendante de la longueur comme la vérification de la taille +ou une troncature.

Exemple 10.6. Conversion de types pour le stockage de character

Pour une colonne cible déclarée comme character(20), la déclaration suivante montre que la valeur stockée a la taille correcte :

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

Voici ce qui s'est réellement passé ici : les deux types inconnus sont résolus en text par défaut, permettant à l'opérateur `||` de les résoudre comme une concaténation de text. Ensuite, le résultat text de l'opérateur est converti en bpchar (« blank-padded char », le nom interne du type de données character (caractère)) pour correspondre au type de la colonne cible (comme la conversion de text à bpchar est compatible binairement, cette conversion n'insère aucun appel réel à une fonction). Enfin, la fonction de taille `bpchar(bpchar, integer, boolean)` est trouvée dans le catalogue système et appliquée au résultat de l'opérateur et à la longueur de la colonne stockée. Cette fonction de type spécifique effectue le contrôle de la longueur requise et ajoute des espaces pour combler la chaîne.

10.5. Constructions `UNION`, `CASE` et constructions relatives

Les constructions SQL avec des `UNION` doivent potentiellement faire correspondre des types différents pour avoir un ensemble unique dans le résultat. L'algorithme de résolution est appliqué séparément à chaque colonne de sortie d'une requête d'union. Les constructions `INTERSECT` et `EXCEPT` résolvent des types différents de la même manière qu'`UNION`. Les constructions `CASE`, `ARRAY`, `VALUES`, `GREATEST` et `LEAST` utilisent le même algorithme pour faire correspondre les expressions qui les composent et sélectionner un type de résultat.

Procédure 10.4. Résolution des types pour `UNION`, `CASE` et les constructions relatives

1. Si toutes les entrées sont du même type et qu'il ne s'agit pas du type unknown, résoudre comme étant de ce type. Sinon, remplacer tous les types de domaine dans la liste avec les types de base sous-jacents.
2. Si toutes les entrées sont du type unknown, résoudre comme étant du type text (le type préféré de la catégorie chaîne). Dans le cas contraire, les entrées unknown seront ignorées.
3. Si toutes les entrées non-inconnues ne sont pas toutes de la même catégorie, échouer.
4. Choisir la première entrée non-inconnue qui est un type préféré dans sa catégorie, s'il y en a une.
5. Sinon, choisir le dernier type en entrée qui ne soit pas inconnu et qui permet à toutes les entrées précédentes qui ne sont pas inconnues à être implicitement converties. (Il y a toujours un type de ce genre car au moins le premier type dans la liste doit satisfaire cette condition.)
6. Convertir toutes les entrées du type sélectionné. Échoue s'il n'y a pas de conversion à partir de l'entrée donnée vers le type sélectionné.

Quelques exemples suivent.

Exemple 10.7. Résolution de types avec des types sous-spécifiés dans une union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Ici, la chaîne de type inconnu `'b'` sera convertie vers le type text.

Exemple 10.8. Résolution de types dans une union simple

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

Le littéral `1.2` est du type numeric et la valeur `1`, de type integer, peut être convertie implicitement vers un type numeric, donc ce type est utilisé.

Exemple 10.9. Résolution de types dans une union transposée

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Dans cet exemple, le type real (réel) ne peut pas être implicitement converti en integer (entier) mais un integer peut être implicitement converti en real ; le résultat de l'union est résolu comme étant un real.

Chapitre 11. Index

L'utilisation d'index est une façon habituelle d'améliorer les performances d'une base de données. Un index permet au serveur de bases de données de retrouver une ligne spécifique bien plus rapidement. Mais les index ajoutent aussi une surcharge au système de base de données dans son ensemble, si bien qu'ils doivent être utilisés avec discernement.

11.1. Introduction

Soit une table définie ainsi :

```
CREATE TABLE test1 (
    id integer,
    contenu varchar
);
```

et une application qui utilise beaucoup de requêtes de la forme :

```
SELECT contenu FROM test1 WHERE id = constante;
```

Sans préparation, le système doit lire la table test1 dans son intégralité, ligne par ligne, pour trouver toutes les lignes qui correspondent. S'il y a beaucoup de lignes dans test1, et que seules quelques lignes correspondent à la requête (peut-être même zéro ou une seule), alors, clairement, la méthode n'est pas efficace. Mais si le système doit maintenir un index sur la colonne *id*, alors il peut utiliser une manière beaucoup plus efficace pour trouver les lignes recherchées. Il se peut qu'il n'ait ainsi qu'à parcourir quelques niveaux d'un arbre de recherche.

Une approche similaire est utilisée dans la plupart des livres autres que ceux de fiction : les termes et concepts fréquemment recherchés par les lecteurs sont listés par ordre alphabétique à la fin du livre. Le lecteur qui recherche un mot particulier peut facilement parcourir l'index, puis aller directement à la page (ou aux pages) indiquée(s). De la même façon que l'auteur doit anticiper les sujets que les lecteurs risquent de rechercher, il est de la responsabilité du programmeur de prévoir les index qui sont utiles.

La commande suivante peut être utilisé pour créer un index sur la colonne *id* :

```
CREATE INDEX test1_id_index ON test1 (id);
```

Le nom test1_id_index peut être choisi librement mais il est conseillé de choisir un nom qui rappelle le but de l'index.

Pour supprimer l'index, on utilise la commande **DROP INDEX**. Les index peuvent être ajoutés et retirés des tables à tout moment.

Une fois un index créé, aucune intervention supplémentaire n'est nécessaire : le système met à jour l'index lorsque la table est modifiée et utilise l'index dans les requêtes lorsqu'il pense que c'est plus efficace qu'une lecture complète de la table. Il faut néanmoins lancer la commande **ANALYZE** régulièrement pour permettre à l'optimiseur de requêtes de prendre les bonnes décisions. Voir le Chapitre 14, Conseils sur les performances pour comprendre quand et pourquoi l'optimiseur décide d'utiliser ou de ne *pas* utiliser un index.

Les index peuvent aussi bénéficier aux commandes **UPDATE** et **DELETE** à conditions de recherche. De plus, les index peuvent être utilisés dans les jointures. Ainsi, un index défini sur une colonne qui fait partie d'une condition de jointure peut aussi accélérer significativement les requêtes avec jointures.

Créer un index sur une grosse table peut prendre beaucoup de temps. Par défaut, PostgreSQL™ autorise la lecture (**SELECT**) sur la table pendant la création d'un index sur celle-ci, mais interdit les écritures (**INSERT**, **UPDATE**, **DELETE**). Elles sont bloquées jusqu'à la fin de la construction de l'index. Dans des environnements de production, c'est souvent inacceptable. Il est possible d'autoriser les écritures en parallèle de la création d'un index, mais quelques précautions sont à prendre. Pour plus d'informations, voir la section intitulée « Construire des index en parallèle ».

Après la création d'un index, le système doit le maintenir synchronisé avec la table. Cela rend plus lourdes les opérations de manipulation de données. C'est pourquoi les index qui sont peu, voire jamais, utilisés doivent être supprimés.

11.2. Types d'index

PostgreSQL™ propose plusieurs types d'index : B-tree, Hash, GiST et GIN. Chaque type d'index utilise un algorithme différent qui convient à un type particulier de requêtes. Par défaut, la commande **CREATE INDEX** crée un index B-tree, ce qui convient dans la plupart des situations. Les index B-tree savent traiter les requêtes d'égalité et par tranches sur des données qu'il est possible de trier. En particulier, l'optimiseur de requêtes de PostgreSQL™ considère l'utilisation d'un index B-tree lorsqu'une colonne indexée est utilisée dans une comparaison qui utilise un de ces opérateurs :

<

```
<=
=
>=
>
```

Les constructions équivalentes à des combinaisons de ces opérateurs, comme `BETWEEN` et `IN`, peuvent aussi être implantées avec une recherche par index B-tree. Une condition `IS NULL` ou `IS NOT NULL` sur une colonne indexée peut aussi être utilisé avec un index B-tree.

L'optimiseur peut aussi utiliser un index B-tree pour des requêtes qui utilisent les opérateurs de recherche de motif `LIKE` et `~` *si* le motif est une constante et se trouve au début de la chaîne à rechercher -- par exemple, `col LIKE 'foo%'` ou `col ~ '^foo'`, mais pas `col LIKE '%bar'`. Toutefois, si la base de données n'utilise pas la locale C, il est nécessaire de créer l'index avec une classe d'opérateur spéciale pour supporter l'indexation à correspondance de modèles. Voir la Section 11.9, « Classes et familles d'opérateurs » ci-dessous. Il est aussi possible d'utiliser des index B-tree pour `ILIKE` et `~*`, mais seulement si le modèle débute par des caractères non alphabétiques, c'est-à-dire des caractères non affectés par les conversions majuscules/minuscules.

Les index hash ne peuvent gérer que des comparaisons d'égalité simple. Le planificateur de requêtes considère l'utilisation d'un index hash quand une colonne indexée est impliquée dans une comparaison avec l'opérateur `=`. La commande suivante est utilisée pour créer un index hash :

```
CREATE INDEX nom ON table USING hash (column);
```

Attention

Les opérations sur les index de hachage ne sont pas tracées par les journaux de transactions. Il est donc généralement nécessaire de les reconstruire avec **REINDEX** après un arrêt brutal de la base de données si des modifications n'ont pas été écrites. De plus, les modifications dans les index hash ne sont pas répliquées avec la réplication Warm Standby après la sauvegarde de base initiale, donc ces index donneront de mauvaises réponses aux requêtes qui les utilisent. Pour ces raisons, l'utilisation des index hash est actuellement déconseillée.

Les index GiST ne constituent pas un unique type d'index, mais plutôt une infrastructure à l'intérieur de laquelle plusieurs stratégies d'indexage peuvent être implantées. De cette façon, les opérateurs particuliers avec lesquels un index GiST peut être utilisé varient en fonction de la stratégie d'indexage (la *classe d'opérateur*). Par exemple, la distribution standard de PostgreSQL™ inclut des classes d'opérateur GiST pour plusieurs types de données géométriques à deux dimensions, qui supportent des requêtes indexées utilisant ces opérateurs :

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

Voir la Section 9.11, « Fonctions et opérateurs géométriques » pour connaître la signification de ces opérateurs. De plus, une condition `IS NULL` sur une colonne d'index peut être utilisée avec un index GiST. Beaucoup de classes d'opérateur GiST sont disponibles dans l'ensemble des `contrib` ou comme projet séparé. Pour plus d'informations, voir Chapitre 52, Index GiST.

Les index GIN sont des index inversés qui peuvent gérer des valeurs contenant plusieurs clés, les tableaux par exemple. Comme GiST, GIN supporte différentes stratégies d'indexation utilisateur. Les opérateurs particuliers avec lesquels un index GIN peut être utilisé varient selon la stratégie d'indexation. Par exemple, la distribution standard de PostgreSQL™ inclut des classes d'opérateurs GIN pour des tableaux à une dimension qui supportent les requêtes indexées utilisant ces opérateurs :

```
<@
@>
=
&&
```

Voir Section 9.17, « Fonctions et opérateurs de tableaux » pour la signification de ces opérateurs. Beaucoup d'autres classes d'opérateurs GIN sont disponibles dans les modules `contrib` ou dans des projets séparés. Pour plus d'informations, voir Chapitre 53, Index GIN.

11.3. Index multicolonnes

Un index peut porter sur plusieurs colonnes d'une table. Soit, par exemple, une table de la forme :

```
CREATE TABLE test2 (
  majeur int,
  mineur int,
  nom varchar
);
```

(cas d'un utilisateur gardant son répertoire `/dev` dans une base de données...) et que des requêtes comme :

```
SELECT nom FROM test2 WHERE majeur = constante AND mineur = constante;
```

sont fréquemment exécutées. Il peut alors être souhaitable de définir un index qui porte sur les deux colonnes *`majeur`* et *`mineur`*. Ainsi, par exemple :

```
CREATE INDEX test2_mm_idx ON test2 (majeur, mineur);
```

Actuellement, seuls les types d'index B-trees, GiST et GIN supportent les index multicolonnes. 32 colonnes peuvent être précisées, au maximum. Cette limite peut être modifiée à la compilation de PostgreSQL™. Voir le fichier `pg_config_manual.h`.

Un index B-tree multicolonne peut être utilisé avec des conditions de requêtes impliquant un sous-ensemble quelconque de colonnes de l'index. L'index est toutefois plus efficace lorsqu'il y a des contraintes sur les premières colonnes (celles de gauche). La règle exacte est la suivante : les contraintes d'égalité sur les premières colonnes, et toute contrainte d'inégalité sur la première colonne qui ne possède pas de contrainte d'égalité sont utilisées pour limiter la partie parcourue de l'index. Les contraintes sur les colonnes à droite de ces colonnes sont vérifiées dans l'index, et limitent ainsi les visites de la table, mais elles ne réduisent pas la partie de l'index à parcourir.

Par exemple, avec un index sur `(a, b, c)` et une condition de requête `WHERE a = 5 AND b >= 42 AND c < 77`, l'index est parcouru à partir de la première entrée pour laquelle `a` = 5 et `b` = 42 jusqu'à la dernière entrée pour laquelle `a` = 5. Les entrées de l'index avec `c` >= 77 sont sautées, mais elles sont toujours parcourues. En principe, cet index peutt être utilisé pour les requêtes qui ont des contraintes sur `b` et/ou `c` sans contrainte sur `a` -- mais l'index entier doit être parcouru, donc, dans la plupart des cas, le planificateur préfère un parcours séquentiel de la table à l'utilisation de l'index.

Un index GiST multicolonne peut être utilisé avec des conditions de requête qui impliquent un sous-ensemble quelconque de colonnes de l'index. Les conditions sur des colonnes supplémentaires restreignent les entrées renvoyées par l'index, mais la condition sur la première colonne est la plus importante pour déterminer la part de l'index parcourue. Un index GiST est relativement inefficace si sa première colonne n'a que quelques valeurs distinctes, même s'il y a beaucoup de valeurs distinctes dans les colonnes supplémentaires.

Un index multi-colonnes GIN peut être utilisé avec des conditions de requête qui implique tout sous-ensemble des colonnes de l'index. Contrairement à B-tree ou GiST, la qualité de la recherche dans l'index est identique quelque soit les colonnes de l'index que la requête utilise

Chaque colonne doit évidemment être utilisée avec des opérateurs appropriés au type de l'index ; les clauses qui impliquent d'autres opérateurs ne sont pas pris en compte.

Il est préférable d'utiliser les index multicolonnes avec parcimonie. Dans la plupart des cas, un index sur une seule colonne est suffisant et préserve espace et temps. Les index de plus de trois colonnes risquent fort d'être inefficaces, sauf si l'utilisation de cette table est extrêmement stylisée. Voir aussi la Section 11.5, « Combiner des index multiples » pour les discussions sur les mérites des différentes configurations d'index.

11.4. Index et `ORDER BY`

Au delà du simple fait de trouver les lignes à renvoyer à une requête, un index peut les renvoyer dans un ordre spécifique. Cela permet de résoudre une clause `ORDER BY` sans étape de tri séparée. De tous les types d'index actuellement supportés par PostgreSQL™, seuls les B-tree peuvent produire une sortie triée -- les autres types d'index renvoient les lignes correspondantes dans un ordre imprécis, dépendant de l'implantation.

Le planificateur répond à une clause `ORDER BY` soit en parcourant un index disponible qui correspond à la clause, soit en parcourant la table dans l'ordre physique et en réalisant un tri explicite. Pour une requête qui nécessite de parcourir une fraction importante de la table, le tri explicite est probablement plus rapide que le parcours d'un index car il nécessite moins d'entrées/sorties disque, du fait de son accès séquentiel. Les index sont plus utiles lorsqu'il s'agit de ne récupérer que quelques lignes être récupérées. `ORDER BY` combiné à `LIMIT` *`n`* est un cas spécial très important : un tri explicite doit traiter toutes les données pour identifier les *`n`* première lignes, mais s'il y a un index qui correspond à l'`ORDER BY`, alors les *`n`* premières lignes peuvent être récupérées directement sans qu'il soit nécessaires de parcourir les autres.

Par défaut, les index B-tree stockent leurs entrées dans l'ordre ascendant, valeurs NULL en dernier. Cela signifie que le parcours

avant d'un index sur une colonne `x` produit une sortie satisfaisant `ORDER BY x` (ou en plus verbeux `ORDER BY x ASC NULLS LAST`). L'index peut aussi être parcouru en arrière, produisant ainsi une sortie satisfaisant un `ORDER BY x DESC` (ou en plus verbeux `ORDER BY x DESC NULLS FIRST` car `NULLS FIRST` est la valeur par défaut pour un `ORDER BY DESC`).

L'ordre d'un index B-tree peut être défini à la création par l'inclusion des options `ASC`, `DESC`, `NULLS FIRST`, et/ou `NULLS LAST` lors de la création de l'index ; par exemple :

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Un index stocké en ordre ascendant avec les valeurs NULL en premier peut satisfaire soit `ORDER BY x ASC NULLS FIRST` soit `ORDER BY x DESC NULLS LAST` selon la direction du parcours.

On peut s'interroger sur l'intérêt de proposer quatre options, alors que deux options associées à la possibilité d'un parcours inverse semblent suffire à couvrir toutes les variantes d'`ORDER BY`. Dans les index mono-colonne, les options sont en effet redondantes, mais pour un index à plusieurs colonnes, elles sont utiles. Si l'on considère un index à deux colonnes `(x, y)`, il peut satisfaire une clause `ORDER BY x, y` sur un parcours avant, ou `ORDER BY x DESC, y DESC` sur un parcours inverse. Mais il se peut que l'application utilise fréquemment `ORDER BY x ASC, y DESC`. Il n'y a pas moyen d'obtenir cet ordre à partir d'un index plus simple, mais c'est possible si l'index est défini comme `(x ASC, y DESC)` or `(x DESC, y ASC)`.

Les index d'ordre différent de celui par défaut sont visiblement une fonctionnalité très spécialisée, mais ils peuvent parfois être à l'origine d'accélérations spectaculaires des performances sur certaines requêtes. L'intérêt de maintenir un tel index dépend de la fréquence des requêtes qui nécessitent un tri particulier.

11.5. Combiner des index multiples

Un parcours unique d'index ne peut utiliser que les clauses de la requête qui utilisent les colonnes de l'index avec les opérateurs de sa classe d'opérateur et qui sont jointes avec `AND`. Par exemple, étant donné un index sur `(a, b)`, une condition de requête `WHERE a = 5 AND b = 6` peut utiliser l'index, mais une requête `WHERE a = 5 OR b = 6` ne peutt pas l'utiliser directement.

Heureusement, PostgreSQL™ peut combiner plusieurs index (y compris plusieurs utilisations du même index) pour gérer les cas qui ne peuvent pas être résolus par des parcours d'index simples. Le système peut former des conditions `AND` et `OR` sur plusieurs parcours d'index. Par exemple, une requête comme `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` peut être divisée en quatre parcours distincts d'un index sur `x`, chaque parcours utilisant une des clauses de la requête. Les résultats de ces parcours sont alors assemblés par OR pour produire le résultat. Autre exemple, s'il existe des index séparés sur `x` et `y`, une résolution possible d'une requête comme `WHERE x = 5 AND y = 6` consiste à utiliser chaque index avec la clause de la requête appropriée et d'assembler les différents résultats avec un AND pour identifier les lignes résultantes.

Pour combiner plusieurs index, le système parcourt chaque index nécessaire et prépare un *bitmap* en mémoire qui donne l'emplacement des lignes de table qui correspondent aux conditions de l'index. Les bitmaps sont ensuite assemblés avec des opérateurs AND ou OR selon les besoins de la requête. Enfin, les lignes réelles de la table sont visitées et renvoyées. Elles sont visitées dans l'ordre physique parce c'est ainsi que le bitmap est créé ; cela signifie que l'ordre des index originaux est perdu et que, du coup, une étape de tri séparée est nécessaire si la requête comprend une clause `ORDER BY`. Pour cette raison, et parce que chaque parcours d'index supplémentaire ajoute un temps additionnel, le planificateur choisit quelque fois d'utiliser un parcours d'index simple même si des index supplémentaires sont disponibles et peuvent être utilisés.

Le nombre de combinaisons d'index possibles croît parallèlement à la complexité des applications. Il est alors de la responsabilité du développeur de la base de décider des index à fournir. Il est quelques fois préférable de créer des index multi-colonnes, mais il est parfois préférable de créer des index séparés et de s'appuyer sur la fonctionnalité de combinaison des index.

Par exemple, si la charge inclut un mélange de requêtes qui impliquent parfois uniquement la colonne `x`, parfois uniquement la colonne `y` et quelques fois les deux colonnes, on peut choisir deux index séparés sur `x` et `y` et s'appuyer sur la combinaison d'index pour traiter les requêtes qui utilisent les deux colonnes. On peut aussi créer un index multi-colonnes sur `(x, y)`. Cet index est typiquement plus efficace que la combinaison d'index pour les requêtes impliquant les deux colonnes mais, comme discuté dans la Section 11.3, « Index multicolonnes », il est pratiquement inutile pour les requêtes n'impliquant que `y`. Il ne peut donc pas être le seul index. Une combinaison de l'index multi-colonnes et d'un index séparé sur `y` est une solution raisonnable. Pour les requêtes qui n'impliquent que `x`, l'index multi-colonnes peut être utilisé, bien qu'il soit plus large et donc plus lent qu'un index sur `x` seul. La dernière alternative consiste à créer les trois index, mais cette solution n'est raisonnable que si la table est lue bien plus fréquemment qu'elle n'est mise à jour et que les trois types de requête sont communs. Si un des types de requête est bien moins courant que les autres, il est préférable de ne créer que les deux index qui correspondent le mieux aux types communs.

11.6. Index d'unicité

Les index peuvent aussi être utilisés pour garantir l'unicité des valeurs d'une colonne, ou l'unicité des valeurs combinées de plusieurs colonnes.

```
CREATE UNIQUE INDEX nom ON table (colonne [, ...]);
```

À ce jour, seuls les index B-trees peuvent être déclarés uniques.

Lorsqu'un index est déclaré unique, il ne peut exister plusieurs lignes d'une table qui possèdent la même valeur indexée. Les valeurs NULL ne sont pas considérées égales. Un index d'unicité multi-colonnes ne rejette que les cas où toutes les colonnes indexées sont égales dans plusieurs lignes.

PostgreSQL™ crée automatiquement un index d'unicité à la déclaration d'une contrainte d'unicité ou d'une clé primaire sur une table. L'index porte sur les colonnes qui composent la clé primaire ou la contrainte d'unicité (au besoin, il s'agit d'un index multi-colonnes). C'est cet index qui assure le mécanisme de vérification de la contrainte.

Note

La méthode la plus appropriée pour ajouter une contrainte à une table est `ALTER TABLE ... ADD CONSTRAINT`. L'utilisation des index pour vérifier les contraintes d'unicité peut être considérée comme un détail d'implantation qui ne doit pas être utilisé directement. Il n'est pas nécessaire de créer manuellement un index sur les colonnes uniques. Cela duplique l'index créé automatiquement.

11.7. Index d'expressions

Une colonne d'index ne correspond pas nécessairement exactement à une colonne de la table associée, mais peut être une fonction ou une expression scalaire calculée à partir d'une ou plusieurs colonnes de la table. Cette fonctionnalité est utile pour obtenir un accès rapide aux tables en utilisant les résultat de calculs.

Par exemple, une façon classique de faire des comparaisons indépendantes de la casse est d'utiliser la fonction `lower` :

```
SELECT * FROM test1 WHERE lower(col1) = 'valeur';
```

Si un index a été défini sur le résultat de `lower(col1)`, cette requête peut l'utiliser. Un tel index est créé avec la commande :

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Si l'index est déclaré `UNIQUE`, il empêche la création de lignes dont les valeurs de la colonne `col1` ne diffèrent que par la casse, ainsi que celle de lignes dont les valeurs de la colonne `col1` sont identiques. Ainsi, les index d'expressions peuvent être utilisés pour appliquer des contraintes qui ne peuvent être définies avec une simple contrainte d'unicité.

Autre exemple. Lorsque des requêtes comme :

```
SELECT * FROM personnes WHERE (prenom || ' ' || nom) = 'Jean Dupont';
```

sont fréquentes, alors il peut être utile de créer un index comme :

```
CREATE INDEX personnes_noms ON personnes ((prenom || ' ' || nom));
```

La syntaxe de la commande **CREATE INDEX** nécessite normalement de mettre des parenthèses autour de l'expression indexée, comme dans l'exemple précédent. Les parenthèses peuvent être omises quand l'expression est un simple appel de fonction, comme dans le premier exemple.

Les expressions d'index sont relativement coûteuses à calculer car l'expression doit être recalculée à chaque insertion ou mise à jour de ligne. Néanmoins, les expressions d'index ne sont *pas* recalculées lors d'une recherche par index car elles sont déjà stockés dans l'index. Dans les deux exemples ci-dessus, le système voit la requête comme un `WHERE colonne_indexée = 'constante'`. De ce fait, la recherche est aussi rapide que toute autre requête d'index. Ainsi, les index d'expressions sont utiles lorsque la rapidité de recherche est plus importante que la rapidité d'insertion et de mise à jour.

11.8. Index partiels

Un *index partiel* est un index construit sur un sous-ensemble d'une table ; le sous-ensemble est défini par une expression conditionnelle (appelée *prédicat* de l'index partiel). L'index ne contient des entrées que pour les lignes de la table qui satisfont au prédicat. Les index partiels sont une fonctionnalité spécialisée, mais ils trouvent leur utilité dans de nombreuses situations.

Une raison majeure à l'utilisation d'index partiels est d'éviter d'indexer les valeurs courantes. Puisqu'une requête qui recherche une valeur courante (qui correspond à plus de quelques pourcents de toutes les lignes) n'utilise, de toute façon, pas cet index, il ne sert à rien de garder ces lignes dans l'index. Cela réduit la taille de l'index, ce qui accélèrera les requêtes qui l'utilisent. Cela accélère aussi nombre d'opérations de mise à jour de la table, car l'index n'a pas à être mis à jour à chaque fois. L'Exemple 11.1, « Mettre

en place un index partiel pour exclure des valeurs courantes » montre une application possible de cette idée.

Exemple 11.1. Mettre en place un index partiel pour exclure des valeurs courantes

Soit l'enregistrement d'un journal d'accès à un serveur web dans une base de données. La plupart des accès proviennent de classes d'adresses IP internes à l'organisation, mais certaines proviennent de l'extérieur (des employés connectés par modem, par exemple). Si les recherches par adresses IP concernent essentiellement les accès extérieurs, il est inutile d'indexer les classes d'adresses IP qui correspondent au sous-réseau de l'organisation.

Si la table ressemble à :

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Pour créer un index partiel qui corresponde à l'exemple, il faut utiliser une commande comme celle-ci :

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND
               client_ip < inet '192.168.100.255');
```

Une requête typique qui peut utiliser cet index est :

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Une requête qui ne peut pas l'utiliser est :

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

Ce type d'index partiel nécessite que les valeurs courantes soient prédéterminées, de façon à ce que ce type d'index soit mieux utilisé avec une distribution des données qui ne change pas. Les index peuvent être recréés occasionnellement pour s'adapter aux nouvelles distributions de données, mais cela ajoute de la maintenance.

Une autre utilisation possible d'index partiel revient à exclure des valeurs de l'index qui ne correspondent pas aux requêtes courantes ; ceci est montré dans l'Exemple 11.2, « Mettre en place un index partiel pour exclure les valeurs inintéressantes ». Cette méthode donne les mêmes avantages que la précédente mais empêche l'accès par l'index aux valeurs « sans intérêt ». Évidemment, mettre en place des index partiels pour ce genre de scénarios nécessite beaucoup de soin et d'expérimentation.

Exemple 11.2. Mettre en place un index partiel pour exclure les valeurs inintéressantes

Soit une table qui contient des commandes facturées et des commandes non facturées, avec les commandes non facturées qui ne prennent qu'une petite fraction de l'espace dans la table, et qu'elles sont les plus accédées. Il est possible d'améliorer les performances en créant un index limité aux lignes non facturées. La commande pour créer l'index ressemble à :

```
CREATE INDEX index_commandes_nonfacturees ON commandes (no_commande)
    WHERE facturee is not true;
```

La requête suivante utilise cet index :

```
SELECT * FROM commandes WHERE facturee is not true AND no_commande < 10000;
```

Néanmoins, l'index peut aussi être utilisé dans des requêtes qui n'utilisent pas `no_commande`, comme :

```
SELECT * FROM commandes WHERE facturee is not true AND montant > 5000.00;
```

Ceci n'est pas aussi efficace qu'un index partiel sur la colonne `montant`, car le système doit lire l'index en entier. Néanmoins, s'il y a assez peu de commandes non facturées, l'utilisation de cet index partiel pour trouver les commandes non facturées peut être plus efficace.

La requête suivante ne peut pas utiliser cet index :

```
SELECT * FROM commandes WHERE no_commande = 3501;
```

La commande 3501 peut faire partie des commandes facturées ou non facturées.

L'Exemple 11.2, « Mettre en place un index partiel pour exclure les valeurs inintéressantes » illustre aussi le fait que la colonne indexée et la colonne utilisée dans le prédicat ne sont pas nécessairement les mêmes. PostgreSQL™ supporte tous les prédicats sur les index partiels, tant que ceux-ci ne portent que sur des champs de la table indexée. Néanmoins, il faut se rappeler que le prédicat doit correspondre aux conditions utilisées dans les requêtes qui sont supposées profiter de l'index. Pour être précis, un index partiel ne peut être utilisé pour une requête que si le système peut reconnaître que la clause `WHERE` de la requête implique mathématiquement le prédicat de l'index. PostgreSQL™ n'a pas de méthode sophistiquée de démonstration de théorème pour reconnaître que des expressions apparemment différentes sont mathématiquement équivalentes. (Non seulement une telle méthode générale de démonstration serait extrêmement complexe à créer mais, en plus, elle serait probablement trop lente pour être d'une quelconque utilité). Le système peut reconnaître des implications d'inégalités simples, par exemple « x < 1 » implique « x < 2 » ; dans les autres cas, la condition du prédicat doit correspondre exactement à une partie de la clause `WHERE` de la requête, sans quoi l'index ne peut pas être considéré utilisable. La correspondance prend place lors de l'exécution de la planification de la requête, pas lors de l'exécution. À ce titre, les clauses de requêtes à paramètres ne fonctionnent pas avec un index partiel. Par exemple, une requête préparée avec un paramètre peut indiquer « x < ? » qui n'implique jamais « x < 2 » pour toutes les valeurs possibles du paramètre.

Un troisième usage possible des index partiels ne nécessite pas que l'index soit utilisé dans des requêtes. L'idée ici est de créer un index d'unicité sur un sous-ensemble de la table, comme dans l'Exemple 11.3, « Mettre en place un index d'unicité partiel ». Cela permet de mettre en place une unicité parmi le sous-ensemble des lignes de la table qui satisfont au prédicat, sans contraindre les lignes qui n'y satisfont pas.

Exemple 11.3. Mettre en place un index d'unicité partiel

Soit une table qui décrit des résultats de tests. On souhaite s'assurer qu'il n'y a qu'une seule entrée « succès » (succes) pour chaque combinaison de sujet et de résultat, alors qu'il peut y avoir un nombre quelconque d'entrées « echec ». Une façon de procéder :

```
CREATE TABLE tests (
    sujet text,
    resultat text,
    succes boolean,
    ...
);

CREATE UNIQUE INDEX contrainte_tests_reussis ON tests (sujet, resultat)
    WHERE succes;
```

C'est une méthode très efficace quand il y a peu de tests réussis et beaucoup de tests en échec.

Enfin, un index partiel peut aussi être utilisé pour surcharger les choix de plan d'exécution de requête du système. De plus, des jeux de données à distribution particulière peuvent inciter le système à utiliser un index alors qu'il ne devrait pas. Dans ce cas, on peut mettre en place l'index de telle façon qu'il ne soit pas utilisé pour la requête qui pose problème. Normalement, PostgreSQL™ fait des choix d'usage d'index raisonnables. Par exemple, il les évite pour rechercher les valeurs communes, si bien que l'exemple précédent n'économise que la taille de l'index, il n'est pas nécessaire pour éviter l'utilisation de l'index. En fait, les choix de plan d'exécution incorrects doivent être traités comme des bogues, et être transmis à l'équipe de développement.

Mettre en place un index partiel indique une connaissance au moins aussi étendue que celle de l'analyseur de requêtes, en particulier, savoir quand un index peut être profitable. Une telle connaissance nécessite de l'expérience et une bonne compréhension du fonctionnement des index de PostgreSQL™. Dans la plupart des cas, les index partiels ne représentent pas un gros gain par rapport aux index classiques.

Plus d'informations sur les index partiels est disponible dans Stonebraker, M, 1989b, olson93 et Seshardri, 1995.

11.9. Classes et familles d'opérateurs

Une définition d'index peut indiquer une *classe d'opérateurs* pour chaque colonne de l'index.

```
CREATE INDEX nom ON table (colonne classe_operateur  [options de tri][, ...]);
```

La classe d'opérateurs identifie les opérateurs que l'index doit utiliser sur cette colonne. Par exemple, un index B-tree sur une colonne de type int4 utilise la classe `int4_ops`. Cette classe d'opérateurs comprend des fonctions de comparaison pour les valeurs de type int4. En pratique, la classe d'opérateurs par défaut pour le type de données de la colonne est généralement suffisante. Les classes d'opérateurs sont utiles pour certains types de données, pour lesquels il peut y avoir plus d'un comportement utile de l'index. Par exemple, une donnée de type nombre complexe peut être classée par sa valeur absolue, ou par sa partie entière. Cela peut s'obtenir en définissant deux classes d'opérateurs pour ce type de données et en sélectionnant la bonne classe à la création de l'index. La classe d'opérateur détermine l'ordre de tri basique (qui peut ensuite être modifié en ajoutant des options de tri comme `ASC`/`DESC` et/ou `NULLS FIRST`/`NULLS LAST`).

Il y a quelques classes d'opérateurs en plus des classes par défaut :

- Les classes d'opérateurs `text_pattern_ops`, `varchar_pattern_ops` et `bpchar_pattern_ops` supportent les index B-tree sur les types text, varchar et char, respectivement. À la différence des classes d'opérateurs par défaut, les valeurs sont comparées strictement caractère par caractère plutôt que suivant les règles de tri spécifiques à la localisation. Cela rend ces index utilisables pour des requêtes qui utilisent des recherches sur des motifs (`LIKE` ou des expressions régulières POSIX) quand la base de données n'utilise pas la locale standard « C ». Par exemple, on peut indexer une colonne varchar comme ceci :

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Il faut créer un index avec la classe d'opérateurs par défaut pour que les requêtes qui utilisent une comparaison <, <=, > ou >= ordinaire utilisent un index. De telles requêtes ne peuvent pas utiliser les classes d'opérateurs *xxx*`_pattern_ops`. Néanmoins, des comparaisons d'égalité ordinaires peuvent utiliser ces classes d'opérateur. Il est possible de créer plusieurs index sur la même colonne avec différentes classes d'opérateurs. Si la locale C est utilisée, les classes d'opérateur *xxx*`_pattern_ops` ne sont pas nécessaires, car un index avec une classe d'opérateurs par défaut est utilisable pour les requêtes de correspondance de modèles dans la locale C.

Les requêtes suivantes montrent les classes d'opérateurs prédéfinies :

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

Une classe d'opérateurs n'est qu'un sous-ensemble d'une structure plus large appelée *famille d'opérateurs*. Dans les cas où plusieurs types de données ont des comportements similaires, il est fréquemment utile de définir des opérateurs identiques pour plusieurs types de données et d'autoriser leur utilisation avec des index. Pour cela, les classes d'opérateur de chacun de ces types doivent être groupés dans la même famille d'opérateurs. Les opérateurs inter-types sont membres de la famille, mais ne sont pas associés avec une seule classe de la famille.

Cette requête affiche toutes les familles d'opérateurs définies et tous les opérateurs inclus dans chaque famille :

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.10. Examiner l'utilisation des index

Bien que les index de PostgreSQL™ n'aient pas besoin de maintenance ou d'optimisation, il est important de s'assurer que les index sont effectivement utilisés sur un système en production. On vérifie l'utilisation d'un index pour une requête particulière avec la commande EXPLAIN(7). Son utilisation dans notre cas est expliquée dans la Section 14.1, « Utiliser **EXPLAIN** ». Il est aussi possible de rassembler des statistiques globales sur l'utilisation des index sur un serveur en cours de fonctionnement, comme décrit dans la Section 27.2, « Le récupérateur de statistiques ».

Il est difficile de donner une procédure générale pour déterminer les index à créer. Plusieurs cas typiques ont été cités dans les exemples précédents. Une bonne dose d'expérimentation est souvent nécessaire dans de nombreux cas. Le reste de cette section donne quelques pistes.

- La première chose à faire est de lancer ANALYZE(7). Cette commande collecte les informations sur la distribution des valeurs dans la table. Cette information est nécessaire pour estimer le nombre de lignes retournées par une requête. L'optimiseur de requêtes en a besoin pour donner des coûts réalistes aux différents plans de requêtes possibles. En l'absence de statistiques réelles, le système utilise quelques valeurs par défaut, qui ont toutes les chances d'être inadaptées. Examiner l'utilisation des index par une application sans avoir lancé **ANALYZE** au préalable est, de ce fait, peine perdue. Voir Section 23.1.3, « Maintenir les statistiques du planificateur » et Section 23.1.5, « Le démon auto-vacuum » pour plus d'informations.
- Utiliser des données réelles pour l'expérimentation. Utiliser des données de test pour mettre en place des index permet de trouver les index utiles pour les données de test, mais c'est tout.

  Il est particulièrement néfaste d'utiliser des jeux de données très réduits. Alors qu'une requête sélectionnant 1000 lignes parmi

100000 peut utiliser un index, il est peu probable qu'une requête sélectionnant 1 ligne dans une table de 100 le fasse, parce que les 100 lignes tiennent probablement dans une seule page sur le disque, et qu'il n'y a aucun plan d'exécution qui puisse aller plus vite que la lecture d'une seule page.

Être vigilant en créant des données de test. C'est souvent inévitable quand l'application n'est pas encore en production. Des valeurs très similaires, complètement aléatoires, ou insérées déjà triées peuvent modifier la distribution des données et fausser les statistiques.

- Quand les index ne sont pas utilisés, il peut être utile pour les tests de forcer leur utilisation. Certains paramètres d'exécution du serveur peuvent interdire certains types de plans (voir la Section 18.6.1, « Configuration de la méthode du planificateur »). Par exemple, en interdisant les lectures séquentielles de tables `enable_seqscan`) et les jointures à boucles imbriquées (`enable_nestloop`), qui sont les deux plans les plus basiques, on force le système à utiliser un plan différent. Si le système continue néanmoins à choisir une lecture séquentielle ou une jointure à boucles imbriquées, alors il y a probablement une raison plus fondamentale qui empêche l'utilisation de l'index ; la condition peut, par exemple, ne pas correspondre à l'index. (Les sections précédentes expliquent quelles sortes de requêtes peuvent utiliser quelles sortes d'index.)
- Si l'index est effectivement utilisé en forçant son utilisation, alors il y a deux possibilités : soit le système a raison et l'utilisation de l'index est effectivement inappropriée, soit les coûts estimés des plans de requêtes ne reflètent pas la réalité. Il faut alors comparer la durée de la requête avec et sans index. La commande **EXPLAIN ANALYZE** peut être utile pour cela.
- S'il apparaît que les estimations de coûts sont fausses, il y a de nouveau deux possibilités. Le coût total est calculé à partir du coût par ligne de chaque nœud du plan, multiplié par l'estimation de sélectivité du nœud de plan. Le coût estimé des nœuds de plan peut être ajusté avec des paramètres d'exécution (décrits dans la Section 18.6.2, « Constantes de coût du planificateur »). Une estimation de sélectivité inadaptée est due à des statistiques insuffisantes. Il peut être possible de les améliorer en optimisant les paramètres de collecte de statistiques. Voir ALTER TABLE(7).

  Si les coûts ne peuvent être ajustés à une meilleure représentation de la réalité, alors il faut peut-être forcer l'utilisation de l'index explicitement. Il peut aussi s'avérer utile de contacter les développeurs de PostgreSQL™ afin qu'ils examinent le problème.

Chapitre 12. Recherche plein texte

12.1. Introduction

La recherche plein texte (ou plus simplement la *recherche de texte*) permet de sélectionner des *documents* en langage naturel qui satisfont une *requête* et, en option, de les trier par intérêt suivant cette requête. Le type le plus fréquent de recherche concerne la récupération de tous les documents contenant les *termes de recherche* indiqués et de les renvoyer dans un ordre dépendant de leur *similarité* par rapport à la requête. Les notions de `requête` et de `similarité` peuvent beaucoup varier et dépendent de l'application réelle. La recherche la plus simple considère une `requête` comme un ensemble de mots et la `similarité` comme la fréquence des mots de la requête dans le document.

Les opérateurs de recherche plein texte existent depuis longtemps dans les bases de données. PostgreSQL™ dispose des opérateurs `~`, `~*`, `LIKE` et `ILIKE` pour les types de données texte, mais il lui manque un grand nombre de propriétés essentielles requises par les systèmes d'information modernes :

- Aucun support linguistique, même pour l'anglais. Les expressions rationnelles ne sont pas suffisantes car elles ne peuvent pas gérer facilement les mots dérivées, par exemple `satisfait` et `satisfaire`. Vous pouvez laisser passer des documents qui contiennent `satisfait` bien que vous souhaiteriez quand même les trouver avec une recherche sur `satisfaire`. Il est possible d'utiliser `OR` pour rechercher plusieurs formes dérivées mais cela devient complexe et augmente le risque d'erreur (certains mots peuvent avoir des centaines de variantes).
- Ils ne fournissent aucun classement (score) des résultats de la recherche, ce qui les rend inefficaces quand des centaines de documents correspondants sont trouvés.
- Ils ont tendance à être lent car les index sont peu supportés, donc ils doivent traiter tous les documents à chaque recherche.

L'indexage pour la recherche plein texte permet au document d'être *pré-traité* et qu'un index de ce pré-traitement soit sauvegardé pour une recherche ultérieure plus rapide. Le pré-traitement inclut :

Analyse des documents en jetons. Il est utile d'identifier les différentes classes de jetons, c'est-à-dire nombres, mots, mots complexes, adresses email, pour qu'ils puissent être traités différemment. En principe, les classes de jeton dépendent de l'application mais, dans la plupart des cas, utiliser un ensemble prédéfinie de classes est adéquat. PostgreSQL™ utilise un *analyseur* pour réaliser cette étape. Un analyseur standard est fourni, mais des analyseurs personnalisés peuvent être écrits pour des besoins spécifiques.

Conversion des jetons en lexemes. Un lexeme est une chaîne, identique à un jeton, mais elle a été *normalisée* pour que différentes formes du même mot soient découvertes. Par exemple, la normalisation inclut pratiquement toujours le remplacement des majuscules par des minuscules, ainsi que la suppression des suffixes (comme `s` ou `es` en anglais). Ceci permet aux recherches de trouver les variantes du même mot, sans avoir besoin de saisir toutes les variantes possibles. De plus, cette étape élimine typiquement les *termes courants*, qui sont des mots si courants qu'il est inutile de les rechercher. Donc, les jetons sont des fragments bruts du document alors que les lexemes sont des mots supposés utiles pour l'indexage et la recherche. PostgreSQL™ utilise des *dictionnaires* pour réaliser cette étape. Différents dictionnaires standards sont fournis et des dictionnaires personnalisés peuvent être créés pour des besoins spécifiques.

Stockage des documents pré-traités pour optimiser la recherche . Chaque document peut être représenté comme un tableau trié de lexemes normalisés. Avec ces lexemes, il est souvent souhaitable de stocker des informations de position à utiliser pour obtenir un *score de proximité*, pour qu'un document qui contient une région plus « dense » des mots de la requête se voit affecté un score plus important qu'un document qui en a moins.

Les dictionnaires autorisent un contrôle fin de la normalisation des jetons. Avec des dictionnaires appropriés, vous pouvez :

- Définir les termes courants qui ne doivent pas être indexés.
- Établir une liste des synonymes pour un simple mot en utilisant Ispell.
- Établir une correspondance entre des phrases et un simple mot en utilisant un thésaurus.
- Établir une correspondance entre différentes variations d'un mot et une forme canonique en utilisant un dictionnaire Ispell.
- Établir une correspondance entre différentes variations d'un mot et une forme canonique en utilisant les règles du stemmer Snowball.

Un type de données tsvector est fourni pour stocker les documents pré-traités, avec un type tsquery pour représenter les requêtes traitées (Section 8.11, « Types de recherche plein texte »). Il existe beaucoup de fonctions et d'opérateurs disponibles pour ces types de données (Section 9.13, « Fonctions et opérateurs de la recherche plein texte »), le plus important étant l'opérateur de correspondance `@@`, dont nous parlons dans la Section 12.1.2, « Correspondance de base d'un texte ». Les recherches plein texte peuvent être accélérées en utilisant des index (Section 12.9, « Types d'index GiST et GIN »).

12.1.1. Qu'est-ce qu'un document ?

Un *document* est l'unité de recherche dans un système de recherche plein texte, par exemple un article de magazine ou un message email. Le moteur de recherche plein texte doit être capable d'analyser des documents et de stocker les associations de lexemes (mots clés) avec les documents parents. Ensuite, ces associations seront utilisées pour rechercher les documents contenant des mots de la requête.

Pour les recherches dans PostgreSQL™, un document est habituellement un champ texte à l'intérieur d'une ligne d'une table de la base ou une combinaison (concaténation) de champs, parfois stockés dans différentes tables ou obtenus dynamiquement. En d'autres termes, un document peut être construit à partir de différentes parties pour l'indexage et il peut ne pas être stocké quelque part. Par exemple :

```
SELECT titre || ' ' ||  auteur || ' ' ||  resume || ' ' || corps AS document
FROM messages
WHERE mid = 12;

SELECT m.titre || ' ' || m.auteur || ' ' || m.resume || ' ' || d.corps AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

Note

En fait, dans ces exemples de requêtes, `coalesce` devrait être utilisé pour empêcher un résultat `NULL` pour le document entier à cause d'une seule colonne `NULL`.

Une autre possibilité est de stocker les documents dans de simples fichiers texte du système de fichiers. Dans ce cas, la base est utilisée pour stocker l'index de recherche plein texte et pour exécuter les recherches, et un identifiant unique est utilisé pour retrouver le document sur le système de fichiers. Néanmoins, retrouver les fichiers en dehors de la base demande les droits d'un superutilisateur ou le support de fonctions spéciales, donc c'est habituellement moins facile que de conserver les données dans PostgreSQL™. De plus, tout conserver dans la base permet un accès simple aux méta-données du document pour aider l'indexage et l'affichage.

Dans le but de la recherche plein texte, chaque document doit être réduit au format de pré-traitement, tsvector. La recherche et le calcul du score sont réalisés entièrement à partir de la représentation tsvector d'un document -- le texte original n'a besoin d'être retrouvé que lorsque le document a été sélectionné pour être montré à l'utilisateur. Nous utilisons souvent tsvector pour le document mais, bien sûr, il ne s'agit que d'une représentation compacte du document complet.

12.1.2. Correspondance de base d'un texte

La recherche plein texte dans PostgreSQL™ est basée sur l'opérateur de correspondance `@@`, qui renvoie `true` si un tsvector (document) correspond à un tsquery (requête). Peu importe le type de données indiqué en premier :

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

Comme le suggère l'exemple ci-dessus, un tsquery n'est pas un simple texte brut, pas plus qu'un tsvector ne l'est. Un tsquery contient des termes de recherche qui doivent déjà être des lexemes normalisés, et peut combiner plusieurs termes en utilisant les opérateurs AND, OR et NOT. (Pour les détails, voir la Section 8.11, « Types de recherche plein texte ».) Les fonctions `to_tsquery` et `plainto_tsquery` sont utiles pour convertir un texte écrit par un utilisateur dans un tsquery correct, par exemple en normalisant les mots apparaissant dans le texte. De façon similaire, `to_tsvector` est utilisé pour analyser et normaliser un document. Donc, en pratique, une correspondance de recherche ressemblerait plutôt à ceci :

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

Observez que cette correspondance ne réussit pas si elle est écrite ainsi :

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

car ici aucune normalisation du mot `rats` n'interviendra. Les éléments d'un tsvector sont des lexemes, qui sont supposés déjà normalisés, donc `rats` ne correspond pas à `rat`.

L'opérateur `@@` supporte aussi une entrée de type text, permettant l'oubli de conversions explicites de text vers tsvector ou tsquery dans les cas simples. Les variantes disponibles sont :

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

Nous avons déjà vu les deux premières. La forme text `@@` tsquery est équivalent à `to_tsvector(x) @@ y`. La forme text `@@` text est équivalente à `to_tsvector(x) @@ plainto_tsquery(y)`.

12.1.3. Configurations

Les exemples ci-dessus ne sont que des exemples simples de recherche plein texte. Comme mentionné précédemment, la recherche plein texte permet de faire beaucoup plus : ignorer l'indexation de certains mots (termes courants), traiter les synonymes et utiliser une analyse sophistiquée, c'est-à-dire une analyse basée sur plus qu'un espace blanc. Ces fonctionnalités sont contrôlées par les *configurations de recherche plein texte*. PostgreSQL™ arrive avec des configurations prédéfinies pour de nombreux langages et vous pouvez facilement créer vos propres configurations (la commande **\dF** de psql affiche toutes les configurations disponibles).

Lors de l'installation, une configuration appropriée est sélectionnée et default_text_search_config est configuré dans `postgresql.conf` pour qu'elle soit utilisée par défaut. Si vous utilisez la même configuration de recherche plein texte pour le cluster entier, vous pouvez utiliser la valeur de `postgresql.conf`. Pour utiliser différentes configurations dans le cluster mais avec la même configuration pour une base, utilisez **ALTER DATABASE ... SET**. Sinon, vous pouvez configurer `default_text_search_config` dans chaque session.

Chaque fonction de recherche plein texte qui dépend d'une configuration a un argument regconfig en option, pour que la configuration utilisée puisse être précisée explicitement. `default_text_search_config` est seulement utilisé quand cet argument est omis.

Pour rendre plus facile la construction de configurations de recherche plein texte, une configuration est construite à partir d'objets de la base de données. La recherche plein texte de PostgreSQL™ fournit quatre types d'objets relatifs à la configuration :

- Les *analyseurs de recherche plein texte* cassent les documents en jetons et classifient chaque jeton (par exemple, un mot ou un nombre).
- Les *dictionnaires de recherche plein texte* convertissent les jetons en une forme normalisée et rejettent les termes courants.
- Les *modèles de recherche plein texte* fournissent les fonctions nécessaires aux dictionnaires. (Un dictionnaire spécifie uniquement un modèle et un ensemble de paramètres pour ce modèle.)
- Les *configurations de recherche plein texte* sélectionnent un analyseur et un ensemble de dictionnaires à utiliser pour normaliser les jetons produit par l'analyseur.

Les analyseurs de recherche plein texte et les modèles sont construits à partir de fonctions bas niveau écrites en C ; du coup, le développement de nouveaux analyseurs ou modèles nécessite des connaissances en langage C, et les droits superutilisateur pour les installer dans une base de données. (Il y a des exemples d'analyseurs et de modèles en addon dans la partie `contrib/` de la distribution PostgreSQL™.) Comme les dictionnaires et les configurations utilisent des paramètres et se connectent aux analyseurs et modèles, aucun droit spécial n'est nécessaire pour créer un nouveau dictionnaire ou une nouvelle configuration. Les exemples de création de dictionnaires et de configurations personnalisés seront présentés plus tard dans ce chapitre.

12.2. Tables et index

Les exemples de la section précédente illustrent la correspondance plein texte en utilisant des chaînes simples. Cette section montre comment rechercher les données de la table, parfois en utilisant des index.

12.2.1. Rechercher dans une table

Il est possible de faire des recherches plein texte sans index. Une requête qui ne fait qu'afficher le champ title de chaque ligne contenant le mot `friend` dans son champ *`body`* ressemble à ceci :

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

Ceci trouve aussi les mots relatifs comme `friends` et `friendly` car ils ont tous la même racine, le même lexeme normalisé.

La requête ci-dessus spécifie que la configuration `english` doit être utilisée pour analyser et normaliser les chaînes. Nous pouvons aussi omettre les paramètres de configuration :

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

Cette requête utilisera l'ensemble de configuration indiqué par default_text_search_config.

Un exemple plus complexe est de sélectionner les dix documents les plus récents qui contiennent les mots `create` et `table` dans les champs title ou body :

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC LIMIT 10;
```

Pour plus de clareté, nous omettons les appels à la fonction `coalesce` qui est nécessaire pour rechercher les lignes contenant `NULL` dans un des deux champs.

Bien que ces requêtes fonctionnent sans index, la plupart des applications trouvent cette approche trop lente, sauf peut-être pour des recherches occasionnelles. Une utilisation pratique de la recherche plein texte réclame habituellement la création d'un index.

12.2.2. Créer des index

Nous pouvons créer un index GIN (Section 12.9, « Types d'index GiST et GIN ») pour accélérer les recherches plein texte :

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

Notez que la version à deux arguments de `to_tsvector` est utilisée. Seules les fonctions de recherche plein texte qui spécifient un nom de configuration peuvent être utilisées dans les index sur des expressions (Section 11.7, « Index d'expressions »). Ceci est dû au fait que le contenu de l'index ne doit pas être affecté par default_text_search_config. Dans le cas contraire, le contenu de l'index peut devenir incohérent parce que différentes entrées pourraient contenir des tsvector créés avec différentes configurations de recherche plein texte et qu'il ne serait plus possible de deviner à quelle configuration fait référence une entrée. Il serait impossible de sauvegarder et restaurer correctement un tel index.

Comme la version à deux arguments de `to_tsvector` a été utilisée dans l'index ci-dessus, seule une référence de la requête qui utilise la version à deux arguments de `to_tsvector` avec le même nom de configuration utilise cet index. C'est-à-dire que `WHERE to_tsvector('english', body) @@ 'a & b'` peut utiliser l'index, mais `WHERE to_tsvector(body) @@ 'a & b'` ne le peut pas. Ceci nous assure qu'un index est seulement utilisé avec la même configuration que celle utilisée pour créer les entrées de l'index.

Il est possible de configurer des index avec des expressions plus complexes où le nom de la configuration est indiqué dans une autre colonne. Par exemple :

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector(config_name, body));
```

où `config_name` est une colonne de la table `pgweb`. Ceci permet l'utilisation de configuration mixe dans le même index tout en enregistrant la configuration utilisée pour chaque entrée d'index. Ceci est utile dans le cas d'une bibliothèque de documents dans différentes langues. Encore une fois, les requêtes voulant utiliser l'index doivent être écrites pour correspondre à l'index, donc `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Les index peuvent même concaténer des colonnes :

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || ' ' ||
body));
```

Une autre approche revient à créer une colonne tsvector séparée pour contenir le résultat de `to_tsvector`. Cet exemple est une concaténation de `title` et `body`, en utilisant `coalesce` pour s'assurer qu'un champ est toujours indexé même si l'autre vaut `NULL` :

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

Puis nous créons un index GIN pour accélérer la recherche :

```
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
```

Maintenant, nous sommes prêt pour des recherches plein texte rapides :

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Lors de l'utilisation d'une colonne séparée pour stocker la représentation tsvector, il est nécessaire d'ajouter un trigger pour obtenir une colonne tsvector à jour à tout moment suivant les modifications de `title` et `body`. La Section 12.4.3, « Triggers pour les mises à jour automatiques » explique comment le faire.

Un avantage de l'approche de la colonne séparée sur un index par expression est qu'il n'est pas nécessaire de spécifier explicitement la configuration de recherche plein texte dans les requêtes pour utiliser l'index. Comme indiqué dans l'exemple ci-dessus, la requête peut dépendre de `default_text_search_config`. Un autre avantage est que les recherches seront plus rapides car il n'est plus nécessaire de refaire des appels à `to_tsvector` pour vérifier la correspondance de l'index. (Ceci est plus important lors de l'utilisation d'un index GiST par rapport à un index GIN ; voir la Section 12.9, « Types d'index GiST et GIN ».) Néanmoins, l'approche de l'index par expression est plus simple à configurer et elle réclame moins d'espace disque car la représentation tsvector n'est pas réellement stockée.

12.3. Contrôler la recherche plein texte

Pour implémenter la recherche plein texte, une fonction doit permettre la création d'un tsvector à partir d'un document et la création d'un tsquery à partir de la requête d'un utilisateur. De plus, nous avons besoin de renvoyer les résultats dans un ordre utile, donc nous avons besoin d'une fonction de comparaison des documents suivant leur adéquation à la recherche. Il est aussi important de pouvoir afficher joliment les résultats. PostgreSQL™ fournit un support pour toutes ces fonctions.

12.3.1. Analyser des documents

PostgreSQL™ fournit la fonction `to_tsvector` pour convertir un document vers le type de données tsvector.

```
    to_tsvector([ config regconfig, ] document text) returns tsvector
```

`to_tsvector` analyse un document texte et le convertit en jetons, réduit les jetons en des lexemes et renvoie un tsvector qui liste les lexemes avec leur position dans le document. Ce dernier est traité suivant la configuration de recherche plein texte spécifiée ou celle par défaut. Voici un exemple simple :

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                   to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

Dans l'exemple ci-dessus, nous voyons que le tsvector résultant ne contient pas les mots `a`, `on` et `it`, le mot `rats` est devenu `rat` et le signe de ponctuation `-` a été ignoré.

En interne, la fonction `to_tsvector` appelle un analyseur qui casse le texte en jetons et affecte un type à chaque jeton. Pour chaque jeton, une liste de dictionnaires (Section 12.6, « Dictionnaires ») est consultée, liste pouvant varier suivant le type de jeton. Le premier dictionnaire qui *reconnaît* le jeton émet un ou plusieurs *lexemes* pour représenter le jeton. Par exemple, `rats` devient `rat` car un des dictionnaires sait que le mot `rats` est la forme pluriel de `rat`. Certains mots sont reconnus comme des *termes courants* (Section 12.6.1, « Termes courants »), ce qui fait qu'ils sont ignorés car ils surviennent trop fréquemment pour être utile

dans une recherche. Dans notre exemple, il s'agissait de `a`, `on` et `it`. Si aucun dictionnaire de la liste ne reconnaît le jeton, il est aussi ignoré. Dans cet exemple, il s'agit du signe de ponctuation `-` car il n'existe aucun dictionnaire affecté à ce type de jeton (`Space symbols`), ce qui signifie que les jetons espace ne seront jamais indexés. Le choix de l'analyseur, des dictionnaires et des types de jetons à indexer est déterminé par la configuration de recherche plein texte sélectionné (Section 12.7, « Exemple de configuration »). Il est possible d'avoir plusieurs configurations pour la même base, et des configurations prédéfinies sont disponibles pour différentes langues. Dans notre exemple, nous avons utilisé la configuration par défaut, à savoir `english` pour l'anglais.

La fonction `setweight` peut être utilisé pour ajouter un label aux entrées d'un tsvector avec un *poids* donné. Ce poids consiste en une lettre : `A`, `B`, `C` ou `D`. Elle est utilisée typiquement pour marquer les entrées provenant de différentes parties d'un document, comme le titre et le corps. Plus tard, cette information peut être utilisée pour modifier le score des résultats.

Comme `to_tsvector(NULL)` renvoie `NULL`, il est recommandé d'utiliser `coalesce` quand un champ peut être NULL. Voici la méthode recommandée pour créer un tsvector à partir d'un document structuré :

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

Ici nous avons utilisé `setweight` pour ajouter un label au source de chaque lexeme dans le tsvector final, puis assemblé les valeurs tsvector en utilisant l'opérateur de concaténation des tsvector, `||`. (La Section 12.4.1, « Manipuler des documents » donne des détails sur ces opérations.)

12.3.2. Analyser des requêtes

PostgreSQL™ fournit les fonctions `to_tsquery` et `plainto_tsquery` pour convertir une requête dans le type de données tsquery. `to_tsquery` offre un accès à d'autres fonctionnalités que `plainto_tsquery` mais est moins indulgent sur ses arguments.

```
    to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`to_tsquery` crée une valeur tsquery à partir de *`querytext`* qui doit contenir un ensemble de jetons individuels séparés par les opérateurs booléens `&` (AND), `|` (OR) et `!` (NOT). Ces opérateurs peuvent être groupés en utilisant des parenthèses. En d'autres termes, les arguments de `to_tsquery` doivent déjà suivre les règles générales pour un tsquery comme décrit dans la Section 8.11, « Types de recherche plein texte ». La différence est que, alors qu'un tsquery basique prend les jetons bruts, `to_tsquery` normalise chaque jeton en un lexeme en utilisant la configuration spécifiée ou par défaut, et annule tout jeton qui est un terme courant d'après la configuration. Par exemple :

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

Comme une entrée tsquery basique, des poids peuvent être attachés à chaque lexeme à restreindre pour établir une correspondance avec seulement des lexemes tsvector de ces poids. Par exemple :

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

De plus, `*` peut être attaché à un lexeme pour demander la correspondance d'un préfixe :

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Un tel lexeme correspondra à tout mot dans un tsvector qui commence par la chaîne indiquée.

`to_tsquery` peut aussi accepter des phrases avec des guillemets simples. C'est utile quand la configuration inclut un dictionnaire thésaurus qui peut se déclencher sur de telles phrases. Dans l'exemple ci-dessous, un thésaurus contient la règle `supernovae stars : sn`:

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

sans guillemets, `to_tsquery` génère une erreur de syntaxe pour les jetons qui ne sont pas séparés par un opérateur AND ou OR.

```
    plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`plainto_tsquery` transforme le texte non formaté *`querytext`* en tsquery. Le texte est analysé et normalisé un peu comme pour `to_tsvector`, ensuite l'opérateur booléen `&` (AND) est inséré entre les mots restants.

Exemple :

```
 SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Notez que `plainto_tsquery` ne peut pas reconnaître un opérateur booléen, des labels de poids en entrée ou des labels de correspondance de préfixe :

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

Ici, tous les symboles de ponctuation ont été annulés car ce sont des symboles espace.

12.3.3. Ajouter un score aux résultats d'une recherche

Les tentatives de score pour mesurer l'adéquation des documents se font par rapport à une certaine requête. Donc, quand il y a beaucoup de correspondances, les meilleurs doivent être montrés en premier. PostgreSQL™ fournit deux fonctions prédéfinies de score, prennant en compte l'information lexicale, la proximité et la structure ; en fait, elles considèrent le nombre de fois où les termes de la requête apparaissent dans le document, la proximité des termes de la recherche avec ceux de la requête et l'importance du passage du document où se trouvent les termes du document. Néanmoins, le concept d'adéquation pourrait demander plus d'informations pour calculer le score, par exemple la date et l'heure de modification du document. Les fonctions internes de calcul de score sont seulement des exemples. Vous pouvez écrire vos propres fonctions de score et/ou combiner leur résultats avec des facteurs supplémentaires pour remplir un besoin spécifique.

Les deux fonctions de score actuellement disponibles sont :

```
        ts_rank([ weights float4[], ] vector tsvector,
                query tsquery [, normalization integer ]) returns float4
```

Fonction de score standard.

```
        ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [,
normalization integer ]) returns float4
```

Cette fonction calcule le score de la *densité de couverture* pour le vecteur du document et la requête donnés, comme décrit dans l'article de Clarke, Cormack et Tudhope, « Relevance Ranking for One to Three Term Queries », article paru dans le journal « Information Processing and Management » en 1999.

Cette fonction nécessite des informations de position. Du coup, elle ne fonctionne pas sur des valeurs tsvector « strippées » -- elle renvoie toujours zéro.

Pour ces deux fonctions, l'argument optionnel des *`poids`* offre la possibilité d'impacter certains mots plus ou moins suivant la façon dont ils sont marqués. Le tableau de poids indique à quel point chaque catégorie de mots est marquée. Dans l'ordre :

```
{poids-D, poids-C, poids-B, poids-A}
```

Si aucun *poids* n'est fourni, alors ces valeurs par défaut sont utilisées :

```
{0.1, 0.2, 0.4, 1.0}
```

Typiquement, les poids sont utilisés pour marquer les mots compris dans des aires spéciales du document, comme le titre ou le résumé initial, pour qu'ils puissent être traités avec plus ou moins d'importance que les mots dans le corps du document.

Comme un document plus long a plus de chance de contenir un terme de la requête, il est raisonnable de prendre en compte la taille du document, par exemple un document de cent mots contenant cinq fois un mot de la requête est probablement plus intéressant qu'un document de mille mots contenant lui-aussi cinq fois un mot de la requête. Les deux fonctions de score prennent une option *normalization*, de type integer, qui précise si la longueur du document doit impacter son score. L'option contrôle plusieurs comportements, donc il s'agit d'un masque de bits : vous pouvez spécifier un ou plusieurs comportements en utilisant `|` (par exemple, `2|4`).

- 0 (valeur par défaut) ignore la longueur du document
- 1 divise le score par 1 + le logarithme de la longueur du document
- 2 divise le score par la longueur du document
- 4 divise le score par "mean harmonic distance between extents" (ceci est implémenté seulement par `ts_rank_cd`)
- 8 divise le score par le nombre de mots uniques dans le document
- 16 divise le score par 1 + le logarithme du nombre de mots uniques dans le document
- 32 divise le score par lui-même + 1

Si plus d'un bit de drapeau est indiqué, les transformations sont appliquées dans l'ordre indiqué.

Il est important de noter que les fonctions de score n'utilisent aucune information globale donc il est impossible de produire une normalisation de 1% ou 100%, comme c'est parfois demandé. L'option de normalisation 32 (`score/(score+1)`) peut s'appliquer pour échelonner tous les scores dans une échelle de zéro à un mais, bien sûr, c'est une petite modification cosmétique, donc l'ordre des résultats ne changera pas.

Voici un exemple qui sélectionne seulement les dix correspondances de meilleur score :

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

Voici le même exemple en utilisant un score normalisé :

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
```

```
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

Le calcul du score peut consommer beaucoup de ressources car il demande de consulter le tsvector de chaque document correspondant, ce qui est très consommateur en entrées/sorties et du coup lent. Malheureusement, c'est presque impossible à éviter car les requêtes intéressantes ont un grand nombre de correspondances.

12.3.4. Surligner les résultats

Pour présenter les résultats d'une recherche, il est préférable d'afficher une partie de chaque document et en quoi cette partie concerne la requête. Habituellement, les moteurs de recherche affichent des fragments du document avec des marques pour les termes recherchés. PostgreSQL™ fournit une fonction `ts_headline` qui implémente cette fonctionnalité.

```
    ts_headline([ config regconfig, ] document text, query tsquery [, options text ])
returns text
```

`ts_headline` accepte un document avec une requête et renvoie un résumé du document. Les termes de la requête sont surlignés dans les extractions. La configuration à utiliser pour analyser le document peut être précisée par *`config`* ; si *`config`* est omis, le paramètre `default_text_search_config` est utilisé.

Si une chaîne *`options`* est spécifiée, elle doit consister en une liste de une ou plusieurs paires *`option=valeur`* séparées par des virgules. Les options disponibles sont :

- `StartSel`, `StopSel` : les chaînes qui permettent de délimiter les mots de la requête parmi le reste des mots. Vous devez mettre ces chaînes entre guillemets doubles si elles contiennent des espaces ou des virgules.
- `MaxWords`, `MinWords` : ces nombres déterminent les limites minimum et maximum des résumés à afficher.
- `ShortWord` : les mots de cette longueur et les mots plus petits seront supprimés au début et à la fin d'un résumé. La valeur par défaut est de trois pour éliminer les articles anglais communs.
- `HighlightAll` : booléen ; si `true`, le document complet sera utilisé pour le surlignage, en ignorant les trois paramètres précédents.
- `MaxFragments` : nombre maximum d'extraits ou de fragments de texte à afficher. La valeur par défaut, 0, sélectionne une méthode de génération d'extraits qui n'utilise pas les fragments. Une valeur positive et non nulle sélectionne la génération d'extraits basée sur les fragments. Cette méthode trouve les fragments de texte avec autant de mots de la requête que possible et restreint ces fragments autour des mots de la requête. Du coup, les mots de la requête se trouvent au milieu de chaque fragment et ont des mots de chaque côté. Chaque fragment sera au plus de `MaxWords` et les mots auront une longueur maximum de `ShortWord`. Si tous les mots de la requête ne sont pas trouvés dans le document, alors un seul fragment de `MinWords` sera affiché.
- `FragmentDelimiter` : quand plus d'un fragment est affiché, alors les fragments seront séparés par ce délimiteur.

Toute option omise recevra une valeur par défaut :

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

Par exemple :

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms
 and return them in order of their <b>similarity</b> to the
 <b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
```

```
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
                      ts_headline
-------------------------------------------------------
 containing given <query> terms
 and return them in order of their <similarity> to the
 <query>.
```

`ts_headline` utilise le document original, pas un résumé tsvector, donc elle peut être lente et doit être utilisée avec parcimonie et attention. Une erreur typique est d'appeler `ts_headline` pour *chaque* document correspondant quand seuls dix documents sont à afficher. Les sous-requêtes SQL peuvent aider ; voici un exemple :

```
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
      FROM apod, to_tsquery('stars') q
      WHERE ti @@ q
      ORDER BY rank DESC
      LIMIT 10) AS foo;
```

12.4. Fonctionnalités supplémentaires

Cette section décrit des fonctions et opérateurs supplémentaires qui sont utiles en relation avec la recherche plein texte.

12.4.1. Manipuler des documents

La Section 12.3.1, « Analyser des documents » a montré comment des documents en texte brut peuvent être convertis en valeurs tsvector. PostgreSQL™ fournit aussi des fonctions et des opérateurs pouvant être utilisés pour manipuler des documents qui sont déjà au format tsvector.

```
      tsvector || tsvector
```

L'opérateur de concaténation tsvector renvoie un vecteur qui combine les lexemes et des informations de position pour les deux vecteurs donnés en argument. Les positions et les labels de poids sont conservés lors de la concaténation. Les positions apparaissant dans le vecteur droit sont décalés par la position la plus large mentionnée dans le vecteur gauche, pour que le résultat soit pratiquement équivalent au résultat du traitement de `to_tsvector` sur la concaténation des deux documents originaux. (L'équivalence n'est pas exacte car tout terme courant supprimé de la fin de l'argument gauche n'affectera pas le résultat alors qu'ils auraient affecté les positions des lexemes dans l'argument droit si la concaténation de texte avait été utilisée.)

Un avantage de l'utilisation de la concaténation au format vecteur, plutôt que la concaténation de texte avant d'appliquer `to_tsvector`, est que vous pouvez utiliser différentes configurations pour analyser les différentes sections du document. De plus, comme la fonction `setweight` marque tous les lexemes du secteur donné de la même façon, il est nécessaire d'analyser le texte et de lancer `setweight` avant la concaténation si vous voulez des labels de poids différents sur les différentes parties du document.

```
      setweight(vector tsvector, weight "char") returns tsvector
```

Cette fonction renvoie une copie du vecteur en entrée pour chaque position de poids *`weight`*, soit `A`, soit `B`, soit `C` soit `D`. (`D` est la valeur par défaut pour les nouveaux vecteurs et, du coup, n'est pas affiché en sortie.) Ces labels sont conservés quand les vecteurs sont concaténés, permettant aux mots des différentes parties d'un document de se voir attribuer un poids différent par les fonctions de score.

Notez que les labels de poids s'appliquent seulement aux *positions*, pas aux *lexemes*. Si le vecteur en entrée se voit supprimer les positions, alors `setweight` ne pourra rien faire.

```
      length(vector tsvector) returns integer
```

Renvoie le nombre de lexemes enregistrés dans le vecteur.

```
strip(vector tsvector) returns tsvector
```

Renvoie un vecteur qui liste les mêmes lexemes que le vecteur donné mais il manquera les informations de position et de poids. Alors que le vecteur renvoyé est bien moins utile qu'un vecteur normal pour calculer le score, il est habituellement bien plus petit.

12.4.2. Manipuler des requêtes

La Section 12.3.2, « Analyser des requêtes » a montré comment des requêtes texte peuvent être converties en valeurs de type tsquery. PostgreSQL™ fournit aussi des fonctions et des opérateurs pouvant être utilisés pour manipuler des requêtes qui sont déjà de la forme tsquery.

```
tsquery && tsquery
```

Renvoie une combinaison AND des deux requêtes données.

```
tsquery || tsquery
```

Renvoie une combinaison OR des deux requêtes données.

```
!! tsquery
```

Renvoie la négation (NOT) de la requête donnée.

```
numnode(query tsquery) returns integer
```

Renvoie le nombre de nœuds (lexemes et opérateurs) dans un tsquery. Cette fonction est utile pour déterminer si la requête (*query*) a un sens (auquel cas elle renvoie > 0) ou s'il ne contient que des termes courants (auquel cas elle renvoie 0). Exemples :

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

```
querytree(query tsquery) returns text
```

Renvoie la portion d'un tsquery qui peut être utilisé pour rechercher dans un index.Cette fonction est utile pour détecter les requêtes qui ne peuvent pas utiliser un index, par exemple celles qui contiennent des termes courants ou seulement des négations de termes. Par exemple :

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

12.4.2.1. Ré-écriture des requêtes

La famille de fonctions `ts_rewrite` cherche dans un tsquery donné les occurrences d'une sous-requête cible et remplace chaque occurrence avec une autre sous-requête de substitution. En fait, cette opération est une version spécifique à tsquery d'un remplacement de sous-chaîne. Une combinaison cible et substitut peut être vu comme une *règle de ré-écriture de la requête*. Un ensemble de règles de ré-écriture peut être une aide puissante à la recherche. Par exemple, vous pouvez étendre la recherche en utilisant des synonymes (`new york`, `big apple`, `nyc`, `gotham`) ou restreindre la recherche pour diriger l'utilisateur vers des thèmes en vogue. Cette fonctionnalité n'est pas sans rapport avec les thésaurus (Section 12.6.4, « Dictionnaire thésaurus »). Néanmoins, vous pouvez modifier un ensemble de règles de ré-écriture directement, sans ré-indexer, alors que la mise à jour d'un thésaurus nécessite un ré-indexage pour être pris en compte.

```
ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery
```

Cette forme de `ts_rewrite` applique simplement une seule règle de ré-écriture : *`target`* est remplacé par *`substitute`* partout où il apparaît dans *`query`*. Par exemple :

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

```
ts_rewrite (query tsquery, select text) returns tsquery
```

Cette forme de `ts_rewrite` accepte une *`query`* de début et une commande SQL *`select`*, qui est fournie comme une chaîne de caractères. *`select`* doit renvoyer deux colonnes de type tsquery. Pour chaque ligne de résultats du *`select`*, les occurrences de la valeur de la première colonne (la cible) sont remplacées par la valeur de la deuxième colonne (le substitut) dans la valeur actuelle de *`query`*. Par exemple :

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Notez que, quand plusieurs règles de ré-écriture sont appliquées de cette façon, l'ordre d'application peut être important ; donc, en pratique, vous voudrez que la requête source utilise `ORDER BY` avec un ordre précis.

Considérons un exemple réel pour l'astronomie. Nous étendons la requête `supernovae` en utilisant les règles de ré-écriture par la table :

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

Nous pouvons modifier les règles de ré-écriture simplement en mettant à jour la table :

```
UPDATE aliases SET s = to_tsquery('supernovae|sn & !nebulae') WHERE t =
to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
---------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

La ré-écriture peut être lente quand il y a beaucoup de règles de ré-écriture car elle vérifie l'intérêt de chaque règle. Pour filtrer les règles qui ne sont pas candidates de façon évidente, nous pouvons utiliser les opérateurs de contenant pour le type tsquery. Dans

l'exemple ci-dessous, nous sélectionnons seulement les règles qui peuvent correspondre avec la requête originale :

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. Triggers pour les mises à jour automatiques

Lors de l'utilisation d'une colonne séparée pour stocker la représentation tsvector de vos documents, il est nécessaire de créer un trigger pour mettre à jour la colonne tsvector quand le contenu des colonnes document change. Deux fonctions trigger intégrées sont disponibles pour cela, mais vous pouvez aussi écrire la vôtre.

```
    tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ...
])
    tsvector_update_trigger_column(tsvector_column_name, config_column_name,
text_column_name [, ... ])
```

Ces fonctions trigger calculent automatiquement une colonne tsvector à partir d'une ou plusieurs colonnes texte sous le contrôle des paramètres spécifiés dans la commande **CREATE TRIGGER**. Voici un exemple de leur utilisation :

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

Après avoir créé ce trigger, toute modification dans *title* ou *body* sera automatiquement reflétée dans *tsv*, sans que l'application n'ait à s'en soucier.

Le premier argument du trigger doit être le nom de la colonne tsvector à mettre à jour. Le second argument spécifie la configuration de recherche plein texte à utiliser pour réaliser la conversion. Pour `tsvector_update_trigger`, le nom de la configuration est donné en deuxième argument du trigger. Il doit être qualifié du nom du schéma comme indiqué ci-dessus pour que le comportement du trigger ne change pas avec les modifications de `search_path`. Pour `tsvector_update_trigger_column`, le deuxième argument du trigger est le nom d'une autre colonne de table qui doit être du type regconfig. Ceci permet une sélection par ligne de la configuration à faire. Les arguments restant sont les noms des colonnes texte (de type text, varchar ou char). Elles sont inclus dans le document suivant l'ordre donné. Les valeurs NULL sont ignorées (mais les autres colonnes sont toujours indexées).

Une limitation des triggers internes est qu'ils traitent les colonnes de façon identique. Pour traiter les colonnes différemment -- par exemple pour donner un poids plus important au titre qu'au corps -- il est nécessaire d'écrire un trigger personnalisé. Voici un exemple utilisant PL/pgSQL comme langage du trigger :

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
```

```
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

Gardez en tête qu'il est important de spécifier explicitement le nom de la configuration lors de la création de valeurs tsvector dans des triggers, pour que le contenu de la colonne ne soit pas affecté par des modifications de `default_text_search_config`. Dans le cas contraire, des problèmes surviendront comme des résultats de recherche changeant après une sauvegarde/restauration.

12.4.4. Récupérer des statistiques sur les documents

La fonction `ts_stat` est utile pour vérifier votre configuration et pour trouver des candidats pour les termes courants.

```
    ts_stat(sqlquery text, [ weights text, ]
            OUT word text, OUT ndoc integer,
            OUT nentry integer) returns setof record
```

`sqlquery` est une valeur de type texte contenant une requête SQL qui doit renvoyer une seule colonne tsvector. `ts_stat` exécute la requête et renvoie des statistiques sur chaque lexeme (mot) contenu dans les données tsvector. Les colonnes renvoyées sont :

- *`word`* text -- la valeur d'un lexeme
- *`ndoc`* integer -- le nombre de documents (tsvector) où le mot se trouve
- *`nentry`* integer -- le nombre total d'occurrences du mot

Si *`weights`* est précisé, seules les occurrences d'un de ces poids sont comptabilisées.

Par exemple, pour trouver les dix mots les plus fréquents dans un ensemble de document :

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

De la même façon, mais en ne comptant que les occurrences de poids `A` ou `B` :

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. Analyseurs

Les analyseurs de recherche plein texte sont responsable du découpage d'un document brut en *jetons* et d'identifier le type des jetons. L'ensemble des types possibles est défini par l'analyseur lui-même. Notez qu'un analyseur ne modifie pas le texte -- il identifie les limites plausibles des mots. Comme son domaine est limité, il est moins important de pouvoir construire des analyseurs personnalisés pour une application. Actuellement, PostgreSQL™ fournit un seul analyseur interne qui s'est révélé utile pour un ensemble varié d'applications.

L'analyseur interne est nommé `pg_catalog.default`. Il reconnait 23 types de jeton :

Tableau 12.1. Types de jeton de l'analyseur par défaut

Alias	Description	Exemple
`asciiword`	Mot, toute lettre ASCII	`elephant`
`word`	Mot, toute lettre	`mañana`
`numword`	Mot, lettres et chiffres	`beta1`
`asciihword`	Mot composé, en ASCII	`up-to-date`
`hword`	Mot composé, toutes les lettres	`lógico-matemática`
`numhword`	Mot composé, lettre et chiffre	`postgresql-beta1`

Alias	Description	Exemple
hword_asciipart	Partie d'un mot composé, en ASCII	postgresql dans le contexte postgresql-beta1
hword_part	Partie d'un mot composé, toutes les lettres	lógico ou matemática dans le contexte lógico-matemática
hword_numpart	Partie d'un mot composé, lettres et chiffres	beta1 dans le contexte postgresql-beta1
email	Adresse email	foo@example.com
protocol	En-tête de protocole	http://
url	URL	example.com/stuff/index.html
host	Hôte	example.com
url_path	Chemin URL	/stuff/index.html, in the context of a URL
file	Fichier ou chemin	/usr/local/foo.txt, if not within a URL
sfloat	Notation scientifique	-1.234e56
float	Notation décimale	-1.234
int	Entier signé	-1234
uint	Entier non signé	1234
version	Numéro de version	8.3.0
tag	Balise XML	<a href="dictionaries.html">
entity	Entité XML	&
blank	Symboles espaces	(tout espace blanc, ou signe de ponctuation non reconnu autrement)

Note

La notion de l'analyseur d'une « lettre » est déterminée par la configuration de la locale sur la base de données, spécifiquement par lc_ctype. Les mots contenant seulement des lettres ASCII basiques sont reportés comme un type de jeton séparé car il est parfois utile de les distinguer. Dans la plupart des langues européennes, les types de jeton word et asciiword doivent toujours être traités de la même façon.

email ne supporte pas tous les caractères email valides tels qu'ils sont définis par la RFC 5322. Spécifiquement, les seuls caractères non-alphanumériques supportés sont le point, le tiret et le tiret bas.

Il est possible que l'analyseur produise des jetons qui coïncident à partir du même texte. Comme exemple, un mot composé peut être reporté à la fois comme un mot entier et pour chaque composante :

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

Ce comportement est souhaitable car il autorise le bon fonctionnement de la recherche sur le mot composé et sur les composants. Voici un autre exemple instructif :

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+-----------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
```

```
host     | Host          | example.com
url_path | URL path      | /stuff/index.html
```

12.6. Dictionnaires

Les dictionnaires sont utilisés pour éliminer des mots qui ne devraient pas être considérés dans une recherche (*termes courants*), et pour *normaliser* des mots pour que des formes dérivées de ce même mot établissent une correspondance. Un mot normalisé avec succès est appelé un *lexeme*. En dehors d'améliorer la qualité de la recherche, la normalisation et la suppression des termes courants réduisent la taille de la représentation d'un document en tsvector, et donc améliorent les performances. La normalisation n'a pas toujours une signification linguistique et dépend habituellement de la sémantique de l'application.

Quelques exemples de normalisation :

- Linguistique - les dictionnaires ispell tentent de réduire les mots en entrée en une forme normalisée ; les dictionnaires stemmer suppriment la fin des mots
- Les URL peuvent être réduites pour établir certaines correspondance :
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Les noms de couleur peuvent être remplacés par leur valeur hexadécimale, par exemple `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- En cas d'indexation de nombre, nous pouvons supprimer certains chiffres à fraction pour réduire les nombres possibles, donc par exemple *3.14*159265359, *3.14*15926, *3.14* seront identiques après normalisation si seuls deux chiffres sont conservés après le point décimal.

Un dictionnaire est un programme qui accepte un jeton en entrée et renvoie :

- un tableau de lexemes si le jeton en entrée est connu dans le dictionnaire (notez qu'un jeton peut produire plusieurs lexemes)
- un unique lexeme avec le drapeau `TSL_FILTER` configuré, pour remplacer le jeton original avec un nouveau jeton à passer aux dictionnaires suivants (un dictionnaire de ce type est appelé un *dictionnaire filtrant*)
- un tableau vide si le dictionnaire connaît le jeton mais que ce dernier est un terme courant
- `NULL` si le dictionnaire n'a pas reconnu le jeton en entrée

PostgreSQL™ fournit des dictionnaires prédéfinis pour de nombreuses langues. Il existe aussi plusieurs modèles prédéfinis qui peuvent être utilisés pour créer de nouveaux dictionnaires avec des paramètres personnalisés. Chaque modèle prédéfini de dictionnaire est décrit ci-dessous. Si aucun modèle ne convient, il est possible d'en créer de nouveaux ; voir le répertoire `contrib/` de PostgreSQL™ pour des exemples.

Une configuration de recherche plein texte lie un analyseur avec un ensemble de dictionnaires pour traiter les jetons en sortie de l'analyseur. Pour chaque type de jeton que l'analyseur peut renvoyer, une liste séparée de dictionnaires est indiquée par la configuration. Quand un jeton de ce type est trouvée par l'analyseur, chaque dictionnaire de la liste est consulté jusqu'à ce qu'un dictionnaire le reconnaisse comme un mot connu. S'il est identifié comme un terme courant ou si aucun dictionnaire ne le reconnait, il sera ignoré et non indexé. Normalement, le premier dictionnaire qui renvoie une sortie non `NULL` détermine le résultat et tout dictionnaire restant n'est pas consulté ; par contre, un dictionnaire filtrant peut remplacer le mot donné avec un autre mot qui est ensuite passé aux dictionnaires suivants.

La règle générale pour la configuration de la liste des dictionnaires est de placer en premier les dictionnaires les plus précis, les plus spécifiques, puis les dictionnaires généralistes, en finissant avec un dictionnaire le plus général possible, comme par exemple un stemmer Snowball ou `simple`, qui reconnait tout. Par exemple, pour une recherche en astronomie (configuration `astro_en`), vous pouvez lier le type de jeton asciiword (mot ASCII) vers un dictionnaire des synonymes des termes de l'astronomie, un dictionnaire anglais généraliste et un stemmer Snowball anglais :

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

Un dictionnaire filtrant peut être placé n'importe où dans la liste. Cependant, le placer à la fin n'a aucun intérêt. Les dictionnaires filtrants sont utiles pour normaliser partiellement les mots, ce qui permet de simplifier la tâche aux dictionnaires suivants. Par exemple, un dictionnaire filtrant peut être utilisé pour supprimer les accents des lettres accentués. C'est ce que fait le module contrib `contrib/unaccent`.

12.6.1. Termes courants

Les termes courants sont des mots très courants, apparaissant dans pratiquement chaque document et n'ont donc pas de valeur dis-

criminatoire. Du coup, ils peuvent être ignorés dans le contexte de la recherche plein texte. Par exemple, tous les textes anglais contiennent des mots comme `a` et `the`, donc il est inutile de les stocker dans un index. Néanmoins, les termes courants n'affectent pas les positions dans tsvector, ce qui affecte le score :

```
SELECT to_tsvector('english','in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

Les positions 1, 2, 4 manquantes sont dûes aux termes courants. Les scores calculés pour les documents avec et sans termes courants sont vraiment différents :

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list
& stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list &
stop'));
 ts_rank_cd
------------
        0.1
```

C'est au dictionnaire de savoir comment traiter les mots courants. Par exemple, les dictionnaires `Ispell` normalisent tout d'abord les mots puis cherchent les termes courants alors que les stemmers `Snowball` vérifient d'abord leur liste de termes courants. La raison de leur comportement différent est qu'ils tentent de réduire le bruit.

12.6.2. Dictionnaire simple

Le modèle du dictionnaire `simple` opère en convertissant le jeton en entrée en minuscule puis en vérifiant s'il fait partie de la liste des mots courants qu'il a sur fichier. S'il est trouvé dans ce fichier, un tableau vide est renvoyé. Le jeton sera alors ignoré. Dans le cas contraire, la forme minuscule du mot est renvoyé en tant que lexeme normalisé. Autrement, le dictionnaire peut être configuré pour rapporter les termes courants comme étant non reconnus, ce qui permet de les passer au prochain dictionnaire de la liste.

Voici un exemple d'une définition de dictionnaire utilisant le modèle `simple` :

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Dans ce cas, `english` est le nom de base du fichier contenant les termes courants. Le nom complet du fichier sera donc `$SHAREDIR/tsearch_data/english.stop`, où `$SHAREDIR` est le répertoire des données partagées de l'installation de PostgreSQL™ (souvent `/usr/local/share/postgresql` mais utilisez **pg_config --sharedir** pour vous en assurer). Le format du fichier est une simple liste de mots, un mot par ligne. Les lignes vides et les espaces en fin de mot sont ignorés. Les mots en majuscule sont basculés en minuscule, mais aucun autre traitement n'est réalisé sur le contenu de ce fichier.

Maintenant, nous pouvons tester notre dictionnaire :

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

Nous pouvons aussi choisir de renvoyer `NULL` à la place du mot en minuscule s'il n'est pas trouvé dans le fichier des termes courants. Ce comportement est sélectionné en configurant le paramètre `Accept` du dictionnaire à `false`. En continuant l'exemple :

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );
```

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

Avec le paramètrage par défaut d'`Accept` (à savoir, `true`), il est préférable de placer un dictionnaire `simple` à la fin de la liste des dictionnaires. `Accept` = `false` est seulement utile quand il y a au moins un dictionnaire après celui-ci.

> **Attention**
>
> La plupart des types de dictionnaires se basent sur des fichiers de configuration, comme les fichiers de termes courants. Ces fichiers *doivent* être dans l'encodage UTF-8. Ils seront traduit vers l'encodage actuelle de la base de données, si elle est différente, quand ils seront lus.

> **Attention**
>
> Habituellement, une session lira un fichier de configuration du dictionnaire une seule fois, lors de la première utilisation. Si vous modifiez un fichier de configuration et que vous voulez forcer la prise en compte des modifications par les sessions en cours, exécutez une commande **ALTER TEXT SEARCH DICTIONARY** sur le dictionnaire. Cela peut être une mise à jour « à vide », donc sans réellement modifier des valeurs.

12.6.3. Dictionnaire des synonymes

Ce modèle de dictionnaire est utilisé pour créer des dictionnaires qui remplacent un mot par un synonyme. Les phrases ne sont pas supportées (utilisez le modèle thésaurus pour cela, Section 12.6.4, « Dictionnaire thésaurus »). Un dictionnaire des synonyme peut être utilisé pour contourner des problèmes linguistiques, par exemple pour empêcher un dictionnaire stemmer anglais de réduire le mot « Paris » en 'pari'. Il suffit d'avoir une ligne `Paris paris` dans le dictionnaire des synonymes et de le placer avant le dictionnaire `english_stem`. Par exemple :

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description    | token |  dictionaries  |  dictionary  | lexemes
-----------+------------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description    | token |       dictionaries        | dictionary | lexemes
-----------+------------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

Le seul paramètre requis par le modèle `synonym` est `SYNONYMS`, qui est le nom de base de son fichier de configuration -- `my_synonyms` dans l'exemple ci-dessus. Le nom complet du fichier sera `$SHAREDIR/tsearch_data/my_synonyms.syn` (où `$SHAREDIR` correspond au répertoire des données partagées de l'installation de PostgreSQL™). Le format du fichier est une ligne par mot à substituer, avec le mot suivi par son synonyme séparé par un espace blanc. Les lignes vierges et les espaces après les mots sont ignorés, les lettres majuscules sont mises en minuscules.

Le modèle `synonym` a aussi un paramètre optionnel, appelé `CaseSensitive`, qui vaut par défaut `false`. Quand `CaseSensitive` vaut `false`, les mots dans le fichier des synonymes sont mis en minuscule, comme les jetons en entrée. Quand il vaut vrai, les mots et les jetons ne sont plus mis en minuscule, mais comparés tels quels..

Un astérisque (`*`) peut être placé à la fin d'un synonyme dans le fichier de configuration. Ceci indique que le synonyme est un pré-

fixe. L'astérisque est ignoré quand l'entrée est utilisée dans `to_tsvector()`, mais quand il est utilisé dans `to_tsquery()`, le résultat sera un élément de la requête avec le marqueur de correspondance du préfixe (voir Section 12.3.2, « Analyser des requêtes »). Par exemple, supposons que nous avons ces entrées dans `$SHAREDIR/tsearch_data/synonym_sample.syn` :

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

Alors nous obtiendrons les résultats suivants :

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. Dictionnaire thésaurus

Un dictionnaire thésaurus (parfois abrévié en TZ) est un ensemble de mots qui incluent des informations sur les relations des mots et des phrases, par exemple des termes plus lointains (BT), plus proches (NT), des termes préférés, des termes non aimés, des termes en relation, etc.

De façon simple, un dictionnaire thésaurus remplace tous les termes par des termes préférés et, en option, conserve les termes originaux pour l'indexage. L'implémentation actuelle du dictionnaire thésaurus par PostgreSQL™ est une extension du dictionnaire des synonymes avec un support additionnel des *phrases*. Un dictionnaire thésaurus nécessite un fichier de configuration au format suivant :

```
# ceci est un commentaire
mots(s) : mot(s) indexé(s)
d'autre(s) mot(s) : d'autre(s) mot(s) indexé(s)
...
```

où le deux-points (:) agit comme un délimiteur entre une phrase et son remplacement.

Un dictionnaire thésaurus utilise un *sous-dictionnaire* (qui est spécifié dans la configuration du dictionnaire) pour normaliser le texte en entrée avant la vérification des correspondances de phrases. Un seul sous-dictionnaire est sélectionnable. Une erreur est renvoyée si le sous-dictionnaire échoue dans la reconnaissance d'un mot. Dans ce cas, vous devez supprimer l'utilisation du mot ou le faire connaître au sous-dictionnaire. Vous pouvez placer une astérisque (*) devant un mot indexé pour ignorer l'utilisation du

sous-dictionnaire mais tous les mots *doivent* être connus du sous-dictionnaire.

Le dictionnaire thésaurus choisit la plus grande correspondance s'il existe plusieurs phrases correspondant à l'entrée.

Les mots spécifiques reconnus par le sous-dictionnaire ne peuvent pas être précisés ; à la place, utilisez `?` pour marquer tout emplacement où un terme courant peut apparaître. Par exemple, en supposant que `a` et `the` sont des termes courants d'après le sous-dictionnaire :

```
? one ? two : swsw
```

correspond à `a one the two` et à `the one a two`. Les deux pourraient être remplacés par `swsw`.

Comme un dictionnaire thésaurus a la possibilité de reconnaître des phrases, il doit se rappeler son état et interagir avec l'analyseur. Un dictionnaire thésaurus utilise ces assignements pour vérifier s'il doit gérer le mot suivant ou arrêter l'accumulation. Le dictionnaire thésaurus doit être configuré avec attention. Par exemple, si le dictionnaire thésaurus s'occupe seulement du type de jeton `asciiword`, alors une définition du dictionnaire thésaurus comme `one 7` ne fonctionnera pas car le type de jeton `uint` n'est pas affecté au dictionnaire thésaurus.

> **Attention**
>
> Les thésaurus sont utilisés lors des indexages pour que toute modification dans les paramètres du dictionnaire thésaurus *nécessite* un réindexage. Pour la plupart des autres types de dictionnaire, de petites modifications comme l'ajout ou la suppression de termes courants ne demandent pas un réindexage.

12.6.4.1. Configuration du thésaurus

Pour définir un nouveau dictionnaire thésaurus, utilisez le modèle `thesaurus`. Par exemple :

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Dans ce cas :

- `thesaurus_simple` est le nom du nouveau dictionnaire
- `mythesaurus` est le nom de base du fichier de configuration du thésaurus. (Son nom complet est `$SHAREDIR/tsearch_data/mythesaurus.ths`, où `$SHAREDIR` est remplacé par le répertoire des données partagées de l'installation.)
- `pg_catalog.english_stem` est le sous-dictionnaire (ici un stemmer Snowball anglais) à utiliser pour la normalisation du thésaurus. Notez que le sous-dictionnaire aura sa propre configuration (par exemple, les termes courants) qui n'est pas affichée ici.

Maintenant, il est possible de lier le dictionnaire du thésaurus `thesaurus_simple` aux types de jeton désirés dans une configuration, par exemple :

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart WITH thesaurus_simple;
```

12.6.4.2. Exemple de thésaurus

Considérez un thésaurus d'astronomie `thesaurus_astro`, contenant quelques combinaisons de mots d'astronomie :

```
supernovae stars : sn
crab nebulae : crab
```

Ci-dessous, nous créons un dictionnaire et lions certains types de jeton à un thésaurus d'astronomie et à un stemmer anglais :

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);
```

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart WITH thesaurus_astro,
english_stem;
```

Maintenant, nous pouvons voir comment cela fonctionne. `ts_lexize` n'est pas très utile pour tester un thésaurus car elle traite l'entrée en tant que simple jeton. À la place, nous pouvons utiliser `plainto_tsquery` et `to_tsvector` qui cassera les chaînes en entrée en plusieurs jetons :

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

En principe, il es possible d'utiliser `to_tsquery` si vous placez l'argument entre guillemets :

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

Notez que `supernova star` établit une correspondance avec `supernovae stars` dans `thesaurus_astro` car nous avions indiqué le stemmer `english_stem` dans la définition du thésaurus. Le stemmer a supprimé `e` et `s`.

Pour indexer la phrase originale ainsi que son substitut, incluez-le dans la partie droite de la définition :

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Dictionnaire Ispell

Le modèle de dictionnaire Ispell ajoute le support des *dictionnaires morphologiques* qui peuvent normaliser plusieurs formes linguisitiques différentes d'un mot en un même lexeme. Par exemple, un dictionnaire Ispell anglais peut établir une correspondance avec toutes les déclinaisons et conjugaisons du terme `bank`, c'est-à-dire `banking`, `banked`, `banks`, `banks'` et `bank's`.

La distribution standard de PostgreSQL™ n'inclut aucun des fichiers de configuration Ispell. Les dictionnaires sont disponibles pour un grand nombre de langues à partir du *site web Ispell*. De plus, certains formats de fichiers dictionnaires plus modernes sont supportés -- *MySpell* (OO < 2.0.1) et *Hunspell* (OO >= 2.0.2). Une large liste de dictionnaires est disponible sur le *Wiki d'OpenOffice* .

Pour créer un dictionnaire Ispell, utilisez le modèle interne `Ispell` et précisez plusieurs paramètres :

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Ici, `DictFile`, `AffFile` et `StopWords` indiquent les noms de base des fichiers dictionnaire, affixes et termes courants. Le fichier des termes courants a le même format qu'indiqué ci-dessus pour le type de dictionnaire `simple`. Le format des autres fichiers n'est pas indiqué ici mais est disponible sur les sites web mentionnés ci-dessus.

Les dictionnaires Ispell reconnaissent habituellement un ensemble limité de mots, pour qu'ils puissent être suivis par un dictionnaire encore plus généraliste ; par exemple un dictionnaire Snowball qui reconnaît tout.

Les dictionnaires Ispell supportent la séparation des mots composés, une fonctionnalité intéressante. Notez que le fichier d'affixes doit indiquer une option spéciale qui marque les mots du dictionnaire qui peuvent participer à une formation composée :

```
compoundwords  controlled z
```

Voici quelques exemples en norvégien :

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

Note

MySpell ne supporte pas les mots composés. Hunspell a un support sophistiqué des mots composés. Actuellement, PostgreSQL™ implémente seulement les opérations basiques de Hunspell pour les mots composés.

12.6.6. Dictionnaire Snowball

Le modèle de dictionnaire Snowball est basé sur le projet de Martin Porter, inventeur du populaire algorithme stemming de Porter pour l'anglais. Snowball propose maintenant des algorithmes stemming pour un grand nombre de langues (voir le *site Snowball* pour plus d'informations). Chaque algorithme sait comment réduire les variantes standard d'un mot vers une base, ou stem, en rapport avec la langue. Un dictionnaire Snowball réclame un paramètre `langue` pour identifier le stemmer à utiliser et, en option, un nom de fichier des `termes courants` donnant une liste de mots à éliminer. (Les listes de termes courants au standard PostgreSQL™ sont aussi fournies par le projet Snowball.) Par exemple, il existe un équivalent de la définition interne en

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

Le format du fichier des termes courants est identique à celui déjà expliqué.

Un dictionnaire Snowball reconnaît tout, qu'il soit ou non capable de simplifier le mot, donc il doit être placé en fin de la liste des dictionnaires. Il est inutile de l'avoir avant tout autre dictionnaire car un jeton ne passera jamais au prochain dictionnaire.

12.7. Exemple de configuration

Une configuration de recherche plein texte précise toutes les options nécessaires pour transformer un document en un tsvector : le planificateur à utiliser pour diviser le texte en jetons, et les dictionnaires à utiliser pour transformer chaque jeton en un lexeme. Chaque appel à `to_tsvector` ou `to_tsquery` a besoin d'une configuration de recherche plein texte pour réaliser le traitement. Le paramètre de configuration default_text_search_config indique le nom de la configuration par défaut, celle utilisée par les fonctions de recherche plein texte si un paramètre explicite de configuration est oublié. Il se configure soit dans `postgresql.conf` soit dans une session individuelle en utilisant la commande **SET**.

Plusieurs configurations de recherche plein texte prédéfinies sont disponibles et vous pouvez créer des versions personnalisées facilement. Pour faciliter la gestion des objets de recherche plein texte, un ensemble de commandes SQL est disponible, et il existe plusieurs commandes psql affichant des informations sur les objets de la recherche plein texte (Section 12.10, « Support de psql »).

Comme exemple, nous allons créer une configuration `pg` en commençant à partir d'une duplication de la configuration `english`.

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

Nous allons utiliser une liste de synonymes spécifique à PostgreSQL et nous allons la stocker dans `$SHAREDIR/tsearch_data/pg_dict.syn`. Le contenu du fichier ressemble à ceci :

```
postgres    pg
pgsql       pg
postgresql  pg
```

Nous définissons le dictionnaire des synonymes ainsi :

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
```

```
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Ensuite, nous enregistrons le dictionnaire Ispell™ `english_ispell` qui a ses propres fichiers de configuration :

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Maintenant, nous configurons la correspondance des mots dans la configuration `pg` :

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

Nous choisissons de ne pas indexer certains types de jeton que la configuration par défaut peut gérer :

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Maintenant, nous pouvons tester notre configuration :

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

La prochaine étape est d'initialiser la session pour utiliser la nouvelle configuration qui était créée dans le schéma `public` :

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. Tester et déboguer la recherche plein texte

Le comportement d'une configuration personnalisée de recherche plein texte peut facilement devenir confuse. Les fonctions décrites dans cette section sont utiles pour tester les objets de recherche plein texte. Vous pouvez tester une configuration complète ou tester séparément analyseurs et dictionnaires.

12.8.1. Test d'une configuration

La fonction `ts_debug` permet un test facile d'une configuration de recherche plein texte.

```
   ts_debug([ config regconfig, ] document text,
            OUT alias text,
            OUT description text,
            OUT token text,
            OUT dictionaries regdictionary[],
```

```
            OUT dictionary regdictionary,
            OUT lexemes text[])
            returns setof record
```

`ts_debug` affiche des informations sur chaque jeton d'un *`document`* tel qu'il est produit par l'analyseur et traité par les dictionnaires configurés. Elle utilise la configuration indiquée par *`config`*, ou `default_text_search_config` si cet argument est omis.

`ts_debug` renvoie une ligne pour chaque jeton identifié dans le texte par l'analyseur. Les colonnes renvoyées sont :

- *`alias`* text -- nom court du type de jeton
- *`description`* text -- description du type de jeton
- *`token`* text -- texte du jeton
- *`dictionaries`* regdictionary[] -- les dictionnaires sélectionnés par la configuration pour ce type de jeton
- *`dictionary`* regdictionary -- le dictionnaire qui a reconnu le jeton, ou `NULL` dans le cas contraire
- *`lexemes`* text[] -- le ou les lexemes produit par le dictionnaire qui a reconnu le jeton, ou `NULL` dans le cas contraire ; un tableau vide (`{}`) signifie qu'il a été reconnu comme un terme courant

Voici un exemple simple :

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

Pour une démonstration plus importante, nous créons tout d'abord une configuration `public.english` et un dictionnaire ispell pour l'anglais :

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |
```

```
dictionary   |   lexemes
-----------+-----------------+-------------+-------------------------------+----------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} |
english_ispell | {}
 blank     | Space symbols   |             | {}                            |
|
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} |
english_ispell | {bright}
 blank     | Space symbols   |             | {}                            |
|
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} |
english_stem   | {supernova}
```

Dans cet exemple, le mot `Brightest` a été reconnu par l'analyseur comme un `mot ASCII` (alias `asciiword`). Pour ce type de jeton, la liste de dictionnaire est `english_ispell` et `english_stem`. Le mot a été reconnu par `english_ispell`, qui l'a réduit avec le mot `bright`. Le mot `supernovaes` est inconnu dans le dictionnaire `english_ispell` donc il est passé au dictionnaire suivant et, heureusement, est reconnu (en fait, `english_stem` est un dictionnaire Snowball qui reconnaît tout ; c'est pourquoi il est placé en dernier dans la liste des dictionnaires).

Le mot `The` est reconnu par le dictionnaire `english_ispell` comme étant un terme courant (Section 12.6.1, « Termes courants ») et n'est donc pas indexé. Les espaces sont aussi ignorés car la configuration ne fournit aucun dictionnaire pour eux.

Vous pouvez réduire le volume en sortie en spécifiant explicitement les colonnes que vous voulez voir :

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. Test de l'analyseur

Les fonctions suivantes permettent un test direct d'un analyseur de recherche plein texte.

```
   ts_parse(parser_name text, document text, OUT tokid integer, OUT token text) returns
setof record
   ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text) returns
setof record
```

`ts_parse` analyse le *`document`* indiqué et renvoie une série d'enregistrements, un pour chaque jeton produit par l'analyse. Chaque enregistrement inclut un `tokid` montrant le type de jeton affecté et un jeton (`token`) qui est le texte dudit jeton. Par exemple :

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
   ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description
text) returns setof record
   ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description
text) returns setof record
```

`ts_token_type` renvoie une table qui décrit chaque type de jeton que l'analyseur indiqué peut reconnaître. Pour chaque type de jeton, la table donne l'entier `tokid` que l'analyseur utilise pour labeliser un jeton de ce type, l'`alias` qui nomme le type de jeton dans les commandes de configuration et une courte `description`. Par exemple :

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+-------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. Test des dictionnaires

La fonction `ts_lexize` facilite le test des dictionnaires.

```
    ts_lexize(dict regdictionary, token text) returns text[]
```

`ts_lexize` renvoie un tableau de lexemes si le jeton (*`token`*) en entrée est connu du dictionnaire ou un tableau vide si le jeton est connu du dictionnaire en tant que terme courant, ou enfin `NULL` si le mot n'est pas connu.

Exemples :

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

Note

La fonction `ts_lexize` attend un seul jeton, pas du texte. Voici un cas où cela peut devenir source de confusion :

```
SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
 ?column?
----------
 t
```

Le dictionnaire thésaurus `thesaurus_astro` connaît la phrase `supernovae stars` mais `ts_lexize` échoue car il ne peut pas analyser le texte en entrée mais le traite bien en tant que simple jeton. Utilisez `plain-`

`to_tsquery` ou `to_tsvector` pour tester les dictionnaires thésaurus. Par exemple :

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

12.9. Types d'index GiST et GIN

Il existe deux types d'index qui peuvent être utilisés pour accélérer les recherches plein texte. Notez que les index ne sont pas obligatoires pour la recherche plein texte mais, dans les cas où une colonne est utilisée fréquemment dans une recherche, un index sera suffisamment intéressant.

```
CREATE INDEX nom ON table USING gist(colonne);
```

Crée un index GiST (Generalized Search Tree). La *`colonne`* peut être de type tsvector ou tsquery.

```
CREATE INDEX nom ON table USING gin(colonne);
```

Crée un index GIN (Generalized Inverted Index). La *`colonne`* doit être de type tsvector.

Il y a des différences de performances substantielles entre les deux types d'index, donc il est important de comprendre leurs caractéristiques.

Un index GiST est *à perte*, signifiant que l'index peut produire des faux positifs, et il est nécessaire de vérifier la ligne de la table pour les éliminer. PostgreSQL™ le fait automatiquement si nécessire. Les index GiST sont à perte car chaque document est représenté dans l'index par une signature à longueur fixe. La signature est générée par le hachage de chaque mot en un bit aléatoire dans une chaîne à n bit, tous ces bits étant assemblés dans une opération OR qui produit une signature du document sur n bits. Quand deux hachages de mots sont identiques, nous avons un faux positif. Si tous les mots de la requête ont une correspondance (vraie ou fausse), alors la ligne de la table doit être récupérée pour voir si la correspondance est correcte.

La perte implique une dégradation des performances à cause de récupérations inutiles d'enregistrements de la table qui s'avèrent être de fausses correspondances. Comme les accès aléatoire aux enregistrements de la table sont lents, ceci limite l'utilité des index GiST. La probabilité de faux positifs dépends de plusieurs facteurs, en particulier le nombre de mots uniques, donc l'utilisation de dictionnaires qui réduisent ce nombre est recommandée.

Les index GIN ne sont pas à perte pour les requêtes standards mais leur performance dépend de façon logarithmique au nombre de mots uniques. (Néanmoins, les index GIN enregistrent seulement les mots (lexemes) des valeurs de type tsvector, et non pas les labels de poids. Donc, la re-vérification d'une ligne de table est nécessaire quand vous utilisez une requête qui indique des poids.

Dans le choix du type d'index à utiliser, GiST ou GIN, pensez à ces différences de performances :

- Les recherches par index GIN sont environ trois fois plus rapides que celles par index GiST.
- Les index GIN prennent trois fois plus de temps à se contruire que les index GiST.
- Les index GIN sont un peu plus lents à mettre à jour que les index GiST, mais dix fois plus lent si le support de la mise à jour rapide a été désactivé (voir Section 53.3.1, « Technique GIN de mise à jour rapide » pour les détails)
- Les index GIN sont entre deux et trois fois plus gros que les index GiST.

En règle générale, les index GIN sont meilleurs pour des données statiques car les recherches sont plus rapides. Pour des données dynamiques, les index GiST sont plus rapides à mettre à jour. Autrement dit, les index GiST sont très bons pour les données dynamiques et rapides si le nombre de mots uniques (lexemes) est inférieur à 100000 alors que les index GIN gèreront plus de 100000 lexemes plus facilement mais sont plus lents à mettre à jour.

Notez que le temps de construction de l'index GIN peut souvent être amélioré en augmentant maintenance_work_mem alors qu'un index GiST n'est pas sensible à ce paramètre.

Le partitionnement de gros ensembles et l'utilisation intelligente des index GIN et GiST autorise l'implémentation de recherches très rapides avec une mise à jour en ligne. Le partitionnement peut se faire au niveau de la base en utilisant l'héritage, ou en distribuant les documents sur des serveurs et en récupérant les résultats de la recherche en utilisant le module `contrib/dblink`. Ce dernier est possible car les fonctions de score utilisent les informations locales.

12.10. Support de psql

Des informations sur les objets de configuration de la recherche plein texte peuvent être obtenues dans psql en utilisant l'ensemble de commandes :

```
\dF{d,p,t}[+] [MODÈLE]
```

Un + supplémentaire affiche plus de détails.

Le paramètre `MODÈLE` doit être le nom d'un objet de la recherche plein texte, pouvant être qualifié du nom du schéma. Si `MODÈLE` est omis, alors l'information sur tous les objets visibles est affichée. `MODÈLE` peut être une expression rationnelle et peut fournir des modèles *séparés* pour les noms du schéma et de l'objet. Les exemples suivants illustrent ceci :

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |
```

```
=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+----------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

Les commandes suivantes sont :

```
\dF[+] [MODÈLE]
```

Liste les configurations de recherche plein texte (ajouter + pour plus de détails).

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-------------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

```
\dFd[+] [MODÈLE]
```

Liste les dictionnaires de recherche plein texte (ajouter + pour plus de détails).

```
=> \dFd
                              List of text search dictionaries
   Schema   |      Name       |                        Description
------------+-----------------+-----------------------------------------------------------
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
 pg_catalog | english_stem    | snowball stemmer for english language
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | simple          | simple dictionary: just lower case and check for
stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
```

```
\dFp[+] [MODÈLE]
```

Liste les analyseurs de recherche plein texte (ajouter + pour plus de détails).

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | HTML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
```

```
 version         | Version number
 word            | Word, all letters
(23 rows)
```

```
\dFt[+] [MODÈLE]
```

Liste les modèles de recherche plein texte (ajouter `+` pour plus de détails).

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+-----------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. Limites

Les limites actuelles de la recherche plein texte de PostgreSQL™ sont :

- La longueur de chaque lexeme doit être inférieure à 2 Ko
- La longueur d'un tsvector (lexemes + positions) doit être inférieure à 1 Mo
- Le nombre de lexemes doit être inférieur à 2^{64}
- Les valeurs de position dans un tsvector doivent être supérieures à 0 et inférieures ou égales à 16383
- Pas plus de 256 positions par lexeme
- Le nombre de nœuds (lexemes + opérateurs) dans un tsquery doit être inférieur à 32768

Pour comparaison, la documentation de PostgreSQL™ 8.1 contient 10441 mots uniques, un total de 335420 mots, et le mot le plus fréquent, « postgresql », est mentionné 6127 fois dans 655 documents.

Un autre exemple -- les archives de la liste de discussion de PostgreSQL™ contenait 910989 mots uniques avec 57491343 lexemes dans 461020 messages.

12.12. Migration à partir d'une recherche plein texte antérieure à 8.3

Les applications qui ont utilisé le module contrib `contrib/tsearch2` pour la recherche plein texte auront besoin de quelques ajustements pour fonctionner avec la version interne :

- Certaines fonctions ont été renommées ou ont profité de petits ajustements dans leur listes d'arguments. Elles sont toutes dans le schéma `pg_catalog` alors que, dans une installation précédente, elles auraient fait partie de `public` ou d'un autre schéma utilisateur. Il existe une nouvelle version de `contrib/tsearch2` (voir la Section F.38, « tsearch2 ») qui fournit une couche de compatibilité permettant de résoudre la majorité des problèmes connus.
- Les anciennes fonctions et les autres objets de `contrib/tsearch2` *doivent* être supprimés lors du chargement d'une sauvegarde pg_dump provenant d'une version antérieure à la 8.3. Bien que beaucoup des objets ne sont pas chargés de toute façon, certains le sont et peuvent causer des problèmes. La façon la plus simple de gérer ceci est de charger seulement le module `contrib/tsearch2` avant la restauration de la sauvegarde ; cela bloquera la restauration des anciens objets.
- Le paramétrage de la configuration de la recherche plein texte est complètement différent maintenant. Au lieu d'insérer manuellement des lignes dans les tables de configuration, la recherche se configure avec des commandes SQL spécialisées indiquées dans tout ce chapitre. Il n'existe pas de support automatisé pour convertir une configuration personnalisée existante pour la 8.3. Vous devez vous en occuper manuellement.
- Le plupart des types de dictionnaires repose sur certains fichiers de configuration en dehors de la base de données. Ils sont largement compatibles pour une utilisation pre-8.3, mais notez malgré tout les différences qui suivent :
  - Les fichiers de configuration doivent être placés dans le répertoire `$SHAREDIR/tsearch_data`, et doivent avoir une extension spécifique dépendant du type de fichier, comme indiqué précédemment dans les descriptions des différents types de dictionnaires. Cette restriction a été ajoutée pour éviter des problèmes de sécurité.
  - Les fichiers de configuration doivent être encodés en UTF-8, quelque soit l'encodage utilisé par la base de données.
  - Dans les fichiers de configuration du thésaurus, les termes courants doivent être marqués avec `?`.

Chapitre 13. Contrôle d'accès simultané

Ce chapitre décrit le comportement de PostgreSQL™ lorsque deux sessions, ou plus, essaient d'accéder aux mêmes données au même moment. Le but dans cette situation est de permettre un accès efficace pour toutes les sessions tout en maintenant une intégrité stricte des données. Chaque développeur d'applications utilisant des bases de données doit avoir une bonne compréhension des thèmes couverts dans ce chapitre.

13.1. Introduction

PostgreSQL™ fournit un ensemble d'outils pour les développeurs qui souhaitent gérer des accès simulatnés aux données. En interne, la cohérence des données est obtenue avec l'utilisation d'un modèle multiversion (Multiversion Concurrency Control, MVCC). Ceci signifie que, lors de l'envoi d'une requête à la base de données, chaque transaction voit une image des données (une *version de la base de données*) telle qu'elles étaient quelque temps auparavant, quel que soit l'état actuel des données sous-jacentes. Ceci protège la transaction de données incohérentes, causées par des mises à jour effectuées par une (autre) transaction simultanée sur les mêmes lignes de données, fournissant ainsi une *isolation des transactions* pour chaque session de la base de données. MVCC, en évitant les méthodes des verrous explicites des systèmes de bases de données traditionnels, minimise la durée des verrous pour permettre des performances raisonnables dans des environnements multiutilisateurs.

Le principal avantage de l'utilisation du modèle MVCC pour le contrôle des accès simultanés, contrairement au verrouillage, est que, dans les verrous acquis par MVCC pour récupérer (en lecture) des données, aucun conflit n'intervient avec les verrous acquis pour écrire des données. Du coup, lire ne bloque jamais l'écriture et écrire ne bloque jamais la lecture.

Des possibilités de verrouillage des tables ou des lignes sont aussi disponibles dans PostgreSQL™ pour les applications ne pouvant pas s'adapter facilement au comportement de MVCC. Néanmoins, un bon usage de MVCC fournira généralement de meilleures performances que les verrous. De plus, les verrous informatifs définis par l'utilisateur fournissent un mécanisme d'acquisition de verrous qui n'est pas lié à une transaction.

13.2. Isolation des transactions

Le standard SQL définit quatre niveaux d'isolation de transaction pour empêcher trois phénomènes de se produire lors de transactions simultanées. Ces phénomènes indésirables sont :

lecture sale
: Une transaction lit des données écrites par une transaction concurrente non validée.

lecture non reproductible
: Une transaction relit des données qu'elle a lu précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale).

lecture fantôme
: Une transaction ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée.

Les quatre niveaux d'isolation de transaction et les comportements correspondants sont décrits dans le Tableau 13.1, « Niveaux d'isolation des transactions SQL ».

Tableau 13.1. Niveaux d'isolation des transactions SQL

Niveau d'isolation	Lecture sale	Lecture non reproductible	Lecture fantôme
Uncommited Read (en français, « Lecture de données non validées »)	Possible	Possible	Possible
Commited Read (en français, « Lecture de données validées »)	Impossible	Possible	Possible
Repeatable Read (en français, « Lecture répétée »)	Impossible	Impossible	Possible
Serializable (en français, « Sérialisable »)	Impossible	Impossible	Impossible

Dans PostgreSQL™, vous pouvez demander un des quatre niveaux standards d'isolation de transaction. Mais, en interne, il existe seulement deux niveaux distincts d'isolation, qui correspondent aux niveaux Read Committed et Serializable. Lorsque vous sélectionnez le niveau Read Uncommitted, vous obtenez réellement Read Committed, et quand vous sélectionnez Repeatable Read, vous obtenez réellement Serializable, donc le niveau d'isolation actuel pourrait être plus strict que ce que vous sélectionnez. Ceci est permis par le standard SQL. Les quatre niveaux d'isolation définissent seulement quel phénomène ne doit pas

survenir, ils ne définissent pas ce qui doit arriver. La raison pour laquelle PostgreSQL™ fournit seulement deux niveaux d'isolation est qu'il s'agit de la seule façon raisonnable de faire correspondre les niveaux d'isolation standards avec l'architecture de contrôle des accès simultanés multiversion. Le comportement des niveaux standards d'isolation est détaillé dans les sous-sections suivantes.

Pour initialiser le niveau d'isolation d'une transaction, utilisez la commande SET TRANSACTION(7).

13.2.1. Niveau d'isolation Read committed (lecture uniquement des données validées)

Read Committed est le niveau d'isolation par défaut dans PostgreSQL™. Quand une transaction utilise ce niveau d'isolation, une requête **SELECT** (sans clause `FOR UPDATE/SHARE`) voit seulement les données validées avant le début de la requête ; il ne voit jamais les données non validées et les modifications validées pendant l'exécution de la requête par des transactions exécutées en parallèle. En effet, une requête **SELECT** voit une image de la base de données datant du moment où l'exécution de la requête commence. Néanmoins, **SELECT** voit les effets de mises à jour précédentes exécutées dans sa propre transaction, même si celles-ci n'ont pas encore été validées. De plus, notez que deux commandes **SELECT** successives peuvent voir des données différentes, même si elles sont exécutées dans la même transaction si d'autres transactions valident des modifications pendant l'exécution du premier **SELECT**.

Les commandes **UPDATE**, **DELETE**, **SELECT FOR UPDATE** et **SELECT FOR SHARE** se comportent de la même façon que **SELECT** en ce qui concerne la recherche des lignes cibles : elles ne trouveront que les lignes cibles qui ont été validées avant le début de la commande. Néanmoins, une telle ligne cible pourrait avoir déjà été mise à jour (ou supprimée ou verrouillée) par une autre transaction concurrente au moment où elle est découverte. Dans ce cas, le processus de mise à jour attendra que la première transaction soit validée ou annulée (si elle est toujours en cours). Si la première mise à jour est annulée, alors ses effets sont niés et le deuxième processus peut exécuter la mise à jour des lignes originellement trouvées. Si la première mise à jour est validée, la deuxième mise à jour ignorera la ligne si la première mise à jour l'a supprimée, sinon elle essaiera d'appliquer son opération à la version mise à jour de la ligne. La condition de la recherche de la commande (la clause `WHERE`) est ré-évaluée pour savoir si la version mise à jour de la ligne correspond toujours à la condition de recherche. Dans ce cas, la deuxième mise à jour continue son opération en utilisant la version mise à jour de la ligne. Dans le cas des commandes **SELECT FOR UPDATE** et **SELECT FOR SHARE**, cela signifie que la version mise à jour de la ligne est verrouillée et renvoyée au client.

À cause de la règle ci-dessus, une commande de mise à jour a la possibilité de voir une image non cohérente : elle peut voir les effets de commandes de mises à jour concurrentes sur les mêmes lignes que celles qu'elle essaie de mettre à jour mais elle ne voit pas les effets de ces commandes sur les autres lignes de la base de données. Ce comportement rend le mode de lecture validée non convenable pour les commandes qui impliquent des conditions de recherche complexes ; néanmoins, il est intéressant pour les cas simples. Par exemple, considérons la mise à jour de balances de banque avec des transactions comme :

```
BEGIN;
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte = 12345;
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte = 7534;
COMMIT;
```

Si deux transactions comme celle-ci essaient de modifier en même temps la balance du compte 12345, nous voulons clairement que la deuxième transaction commence à partir de la version mise à jour de la ligne du compte. Comme chaque commande n'affecte qu'une ligne prédéterminée, la laisser voir la version mise à jour de la ligne ne crée pas de soucis de cohérence.

Des utilisations plus complexes peuvent produire des résultats non désirés dans le mode Read Committed. Par exemple, considérez une commande **DELETE** opérant sur des données qui sont à la fois ajoutées et supprimées du critère de restriction par une autre commande. Supposons que `website` est une table sur deux lignes avec `website.hits` valant `9` et `10` :

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- exécuté par une autre session :  DELETE FROM website WHERE hits = 10;
COMMIT;
```

La commande **DELETE** n'aura pas d'effet même s'il existe une ligne `website.hits = 10` avant et après la commande **UPDATE**. Cela survient parce que la valeur `9` de la ligne avant mise à jour est ignorée et que lorsque l'**UPDATE** termine et que **DELETE** obtient un verrou, la nouvelle valeur de la ligne n'est plus `10`, mais `11`, ce qui ne correspond plus au critère.

Comme le mode Read Committed commence chaque commande avec une nouvelle image qui inclut toutes les transactions validées jusqu'à cet instant, les commandes suivantes dans la même transaction verront les effets de la transaction validée en parallèle dans tous les cas. Le problème en question est de savoir si une *seule* commande voit une vue absolument cohérente ou non de la base de données.

L'isolation partielle des transactions fournie par le mode Read Committed est adéquate pour de nombreuses applications, et ce mode est rapide et simple à utiliser. Néanmoins, il n'est pas suffisant dans tous les cas. Les applications qui exécutent des requêtes et des mises à jour complexes pourraient avoir besoin d'une vue plus rigouresement cohérente de la base de données, une vue que

le mode Read Committed ne fournit pas.

13.2.2. Niveau d'isolation sérialisable

Le niveau *sérialisable* est le niveau d'isolation de transaction le plus strict. Ce niveau émule l'exécution de la transaction en série, comme si les transactions avaient été exécutées l'une après l'autre, en série plutôt que parallèlement. Néanmoins, les applications utilisant ce niveau doivent être préparées à tenter de nouveau les transactions suite aux échecs de la sérialisation.

Quand une transaction utilise le niveau sérialisable, une requête **SELECT** voit seulement les données validées avant le début de la transaction ; elle ne voit jamais les données non validées et les modifications validées lors de l'exécution de la transaction par des transactions concurrentes (néanmoins, la requête voit bien les effets des mises à jour précédentes exécutées à l'intérieur de sa propre transaction même si elles ne sont pas encore validées). C'est différent du niveau Read Commited dans la mesure où une requête, dans une transaction sérialisable, voit une image du début de la transaction et non pas du début de la requête en cours à l'intérieur de la transaction. Du coup, les commandes **SELECT** successives à l'intérieur d'une *seule* transaction voient toujours les mêmes données, c'est-à-dire qu'elles ne voient jamais les modifications faites par les autres transactions qui ont validé après le début de leur propre transaction (ce comportement peut être idéal pour les applications de création de rapports.)

Les commandes **UPDATE**, **DELETE**, **SELECT FOR UPDATE** et **SELECT FOR SHARE** se comportent de la même façon que **SELECT** en ce qui concerne la recherche de lignes cibles : elles trouveront seulement les lignes cibles qui ont été validées avant le début de la transaction. Néanmoins, une telle ligne cible pourrait avoir été mise à jour (ou supprimée ou verrouillée) par une autre transaction concurrente au moment où elle est utilisée. Dans ce cas, la transaction sérialisable attendra que la première transaction de mise à jour soit validée ou annulée (si celle-ci est toujours en cours). Si la première mise à jour est annulée, les effets sont inversés et la transaction sérialisable peut continuer avec la mise à jour de la ligne trouvée à l'origine. Mais si la mise à jour est validée (et que la ligne est mise à jour ou supprimée, pas simplement verrouillée), alors la transaction sérialisable sera annulée avec le message

```
ERROR:  could not serialize access due to concurrent update
```

parce qu'une transaction sérialisable ne peut pas modifier ou verrouiller les lignes changées par d'autres transactions après que la transaction sérialisable ait commencé.

Quand une application reçoit ce message d'erreurs, elle devrait annuler la transaction actuelle et ré-essayer la transaction complète. La seconde fois, la transaction voit les modifications déjà validées comme faisant partie de sa vue initiale de la base de données, donc il n'y a pas de conflit logique en utilisant la nouvelle version de la ligne comme point de départ pour la mise à jour de la nouvelle transaction.

Notez que seules les transactions de modifications ont besoin d'être tentées de nouveau ; les transactions en lecture seule n'auront jamais de conflits de sérialisation.

Le mode sérialisable fournit une garantie rigoureuse que chaque transaction accède à une vue totalement cohérente de la base de données. Néanmoins, l'application doit être prête à tenter de nouvelles transactions lorsque des mises à jour concurrentes rendent impossibles de soutenir l'illusion d'une exécution en série. Comme le coût de la ré-exécution de transactions complexes pourrait être significatif, le mode sérialisable est seulement recommandé lors de mise à jour contenant une logique suffisamment complexe pour donner de mauvaises réponses dans le mode de lecture validée. Plus communément, le mode sérialisable est nécessaire quand une transaction exécute plusieurs commandes successives qui doivent avoir des vues identiques de la base de données.

13.2.2.1. Isolation sérialisable et vraie sérialisation

La signification intuitive (et la définition mathématique) de l'exécution « sérialisable » est que toute paire de transactions concurrentes validée avec succès apparaîtra comme ayant été exécutée en série, l'une après l'autre -- bien que celle survenant en premier n'est pas prévisible. Il est important de réaliser qu'interdire les comportements indésirables listés dans le Tableau 13.1, « Niveaux d'isolation des transactions SQL » n'est pas suffisant pour garantir une vraie exécution en série et, en fait, le mode sérialisable de PostgreSQL™ *ne garantit pas une exécution en série dans ce sens*. Comme exemple, considérez la table ma_table, contenant initialement

```
 classe | valeur
--------+-------
      1 |    10
      1 |    20
      2 |   100
      2 |   200
```

Supposons que la transaction sérialisable A traite

```
SELECT SUM(valeur) FROM ma_table WHERE classe = 1;
```

puis insère le résultat (30) comme `valeur` dans une nouvelle ligne avec `classe = 2`. Simultanément, la transaction serialisable B traite

```
SELECT SUM(valeur) FROM ma_table WHERE classe = 2;
```

et obtient le résultat 300, qu'il insère dans une nouvelle ligne avec `classe = 1`. Donc, les deux transactions valident. Aucun des comportements indiqués comme non désirables n'est survenu, pourtant nous avons un résultat qui n'aurait pû arriver dans tout ordre sériel. Si A a été exécuté avant B, B aurait trouvé la somme 330, et non pas 300. De façon similaire, l'autre ordre aurait eu comme résultat une somme différente pour le calcul par A.

Pour garantir une vraie sérialisation mathématique, il est nécessaire que le système de bases de données force le *verrouillage du prédicat*, ce qui signifie qu'une transaction ne peut pas insérer ou modifier une ligne qui aurait correspondue à la condition `WHERE` d'une requête dans une autre transaction concurrente. Par exemple, une fois que la transaction A a exécuté la requête `SELECT ... WHERE classe = 1`, un système à verrouillage de prédicat aurait interdit l'insertion de toute ligne de classe 1 par la transaction B jusqu'à la validation de la transaction A. [1] Un tel système de verrous est complexe à implémenter et extrêmement coûteux à l'exécution car toutes les sessions doivent être conscientes des détails de chaque requête exécutée par chaque transaction concurrente. Et cette grande dépense est pratiquement complètement perdue car, en pratique, la plupart des applications ne posent pas ce genre de problèmes (l'exemple ci-dessus est assez petit et a peu de chance de se présenter avec de vrais logiciels). Pour ces raisons, PostgreSQL™ n'implémente pas le verrouillage de prédicat.

Dans ces cas où la possibilité d'une exécution non sérialisable est un vrai hasard, les problèmes peuvent être prévenus par l'utilisation appropriée d'un verrou explicite. Les sections suivantes contiennent plus d'informations sur ce sujet.

13.3. Verrouillage explicite

PostgreSQL™ fournit de nombreux modes de verrous pour contrôler les accès simultanés aux données des tables. Ces modes peuvent être utilisés pour contrôler le verrouillage par l'application dans des situations où MVCC n'a pas le comportement désiré. De plus, la plupart des commandes PostgreSQL™ acquièrent automatiquement des verrous avec les modes appropriés pour s'assurer que les tables référencées ne sont pas supprimées ou modifiées de façon incompatible lorsque la commande s'exécute (par exemple, **ALTER TABLE** ne peut pas être exécuté de façon sûr en même temps que d'autres opérations sur la même table, donc il obtient un verrou exclusif sur la table pour s'assurer d'une bonne exécution).

Pour examiner une liste des verrous en cours, utilisez la vue système pg_locks. Pour plus d'informations sur la surveillance du statut du sous-système de gestion des verrous, référez-vous au Chapitre 27, Surveiller l'activité de la base de données.

13.3.1. Verrous de niveau table

La liste ci-dessous affiche les modes de verrous disponibles et les contextes dans lesquels ils sont automatiquement utilisés par PostgreSQL™. Vous pouvez aussi acquérir explicitement n'importe lequel de ces verrous avec la commande LOCK(7). Rappelez-vous que tous ces modes de verrous sont des verrous au niveau table, même si le nom contient le mot « row » (NdT : ligne) ; les noms des modes de verrous sont historiques. Dans une certaine mesure, les noms reflètent l'utilisation typique de chaque mode de verrou -- mais la sémantique est identique. La seule vraie différence entre un mode verrou et un autre est l'ensemble des modes verrous avec lesquels ils rentrent en conflit (voir Tableau 13.2, « Modes de verrou conflictuels »). Deux transactions ne peuvent pas conserver des verrous de modes en conflit sur la même table au même moment (néanmoins, une transaction n'entre jamais en conflit avec elle-même. Par exemple, elle pourrait acquérir un verrou `ACCESS EXCLUSIVE` et acquérir plus tard un verrou `ACCESS SHARE` sur la même table). Des modes de verrou sans conflit peuvent être détenus en même temps par plusieurs transactions. Notez, en particulier, que certains modes de verrous sont en conflit avec eux-même (par exemple, un verrou `ACCESS EXCLUSIVE` ne peut pas être détenu par plus d'une transaction à la fois) alors que d'autres n'entrent pas en conflit avec eux-même (par exemple, un verrou `ACCESS SHARE` peut être détenu par plusieurs transactions).

Modes de verrous au niveau table

`ACCESS SHARE`
: En conflit avec le mode verrou `ACCESS EXCLUSIVE`.

  Les commandes **SELECT** acquièrent un verrou de ce mode avec les tables référencées. En général, tout requête *lisant* seulement une table et ne la modifiant pas obtient ce mode de verrou.

`ROW SHARE`
: En conflit avec les modes de verrous `EXCLUSIVE` et `ACCESS EXCLUSIVE`.

  La commande **SELECT FOR UPDATE** et **SELECT FOR SHARE** acquièrent un verrou de ce mode avec la table cible (en plus des verrous `ACCESS SHARE` des autres tables référencées mais pas sélectionnées `FOR UPDATE/FOR SHARE`).

`ROW EXCLUSIVE`
: En conflit avec les modes de verrous `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`.

[1] En fait, un système de verrouillage de prédicat empêche les lectures fantômes en restreignant ce qui est écrit alors que MVCC les empêche de restreindre ce qui est lu.

Les commandes **UPDATE**, **DELETE** et **INSERT** acquièrent ce mode de verrou sur la table cible (en plus des verrous `ACCESS SHARE` sur toutes les autres tables référencées). En général, ce mode de verrouillage sera acquis par toute commande *modifiant* des données de la table.

`SHARE UPDATE EXCLUSIVE`
En conflit avec les modes de verrous `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode protège une table contre les modifications simultanées de schéma et l'exécution d'un **VACUUM**.

Acquis par **VACUUM** (sans `FULL`), **ANALYZE** et **CREATE INDEX CONCURRENTLY**.

`SHARE`
En conflit avec les modes de verrous `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode protège une table contre les modifications simultanées des données.

Acquis par **CREATE INDEX** (sans `CONCURRENTLY`).

`SHARE ROW EXCLUSIVE`
En conflit avec les modes de verrous `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`.

Ce mode de verrouillage n'est automatiquement acquis par aucune commande PostgreSQL™.

`EXCLUSIVE`
En conflit avec les modes de verrous `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode autorise uniquement les verrous `ACCESS SHARE` concurrents, c'est-à-dire que seules les lectures à partir de la table peuvent être effectuées en parallèle avec une transaction contenant ce mode de verrouillage.

Ce mode de verrouillage n'est pas automatiquement acquis sur les tables utilisateur par une commande PostgreSQL™. Néanmoins, il est utilisé sur certains catalogues systèmes pour certaines opérations.

`ACCESS EXCLUSIVE`
Entre en conflit avec tous les modes (`ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`). Ce mode garantit que le détenteur est la seule transaction à accéder à la table de quelque façon que ce soit.

Acquis par les commandes **ALTER TABLE**, **DROP TABLE**, **TRUNCATE**, **REINDEX**, **CLUSTER** et **VACUUM FULL**. C'est aussi le mode de verrou par défaut des instructions **LOCK TABLE** qui ne spécifient pas explicitement de mode de verrouillage.

Astuce

Seul un verrou `ACCESS EXCLUSIVE` bloque une instruction **SELECT** (sans `FOR UPDATE/SHARE`).

Une fois acquis, un verrou est normalement détenu jusqu'à la fin de la transaction. Mais si un verrou est acquis après l'établissement d'un point de sauvegarde, le verrou est relâché immédiatement si le point de sauvegarde est annulé. Ceci est cohérent avec le principe du **ROLLBACK** annulant tous les effets des commandes depuis le dernier point de sauvegarde. Il se passe la même chose pour les verrous acquis à l'intérieur d'un bloc d'exception PL/pgSQL : un échappement d'erreur à partir du bloc lâche les verrous acquis dans le bloc.

Tableau 13.2. Modes de verrou conflictuels

Verrou demandé	**Verrou déjà détenu**							
	ACCESS SHARE	**ROW SHARE**	**ROW EXCLUSIVE**	**SHARE UPDATE EXCLUSIVE**	**SHARE**	**SHARE ROW EXCLUSIVE**	**EXCLUSIVE**	**ACCESS EXCLUSIVE**
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UP-				X	X	X	X	X

Verrou demandé	Verrou déjà détenu							
	ACCESS SHARE	**ROW SHARE**	**ROW EXCLUSIVE**	**SHARE UPDATE EXCLUSIVE**	**SHARE**	**SHARE ROW EXCLUSIVE**	**EXCLUSIVE**	**ACCESS EXCLUSIVE**
DATE EXCLUSIVE								
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. Verrous au niveau ligne

En plus des verrous au niveau table, il existe des verrous au niveau ligne, qui peuvent être des verrous exclusifs ou partagés. Un verrou exclusif sur une ligne spécifique est automatiquement acquis lorsque la ligne est mise à jour ou supprimée. Le verrou est détenu jusqu'à la fin de la transaction, que ce soit une validation ou une annulation, de la même façon que les verrous de niveau table. Les verrous au niveau ligne n'affectent pas les requêtes sur les données ; ils bloquent seulement les *modifieurs d'une même ligne*.

Pour acquérir un verrou exclusif au niveau ligne sans modifier réellement la ligne, sélectionnez la ligne avec **SELECT FOR UPDATE**. Notez qu'une fois le verrou au niveau ligne acquis, la transaction pourrait mettre à jour la ligne plusieurs fois sans peur des conflits.

Pour acquérir un verrou partagé niveau ligne sur une ligne spécifique, sélectionnez la ligne avec **SELECT FOR SHARE**. Un verrou partagé n'empêche pas les autres transactions d'obtenir le même verrou partagé. Néanmoins, aucune transaction n'est autorisée à mettre à jour, supprimer ou verrouiller exclusivement une ligne dont une autre transaction a obtenu un verrou partagé. Toute tentative de le faire bloque tant que les verrous partagés n'ont pas été enlevés.

PostgreSQL™ ne garde en mémoire aucune information sur les lignes modifiées, il n'y a donc aucune limite sur le nombre de lignes verrouillées à un moment donné. Néanmoins, verrouiller une ligne peut causer une écriture disque ; ainsi, par exemple, **SELECT FOR UPDATE** modifie les lignes sélectionnées pour les marquer verrouillées et cela aboutit à des écritures disques.

En plus des verrous tables et lignes, les verrous partagés/exclusifs sur les pages sont utilisés pour contrôler la lecture et l'écriture des pages de table dans l'ensemble des tampons partagées. Ces verrous sont immédiatement relâchés une fois la ligne récupérée ou mise à jour. Les développeurs d'applications ne sont normalement pas concernés par les verrous au niveau page mais nous les mentionnons dans un souci d'exhaustivité.

13.3.3. Verrous morts (blocage)

L'utilisation de verrous explicites accroît le risque de *verrous morts* lorsque deux transactions (voire plus) détiennent chacune un verrou que l'autre convoite. Par exemple, si la transaction 1 a acquis un verrou exclusif sur la table A puis essaie d'acquérir un verrou exclusif sur la table B alors que la transaction 2 possède déjà un verrou exclusif sur la table B et souhaite maintenant un verrou exclusif sur la table A, alors aucun des deux ne peut continuer. PostgreSQL™ détecte automatiquement ces situations de blocage et les résout en annulant une des transactions impliquées, permettant ainsi à l'autre (aux autres) de se terminer (quelle est exactement la transaction annulée est difficile à prévoir mais vous ne devriez pas vous en préoccuper).

Notez que les verrous morts peuvent aussi se produire en conséquence à des verrous de niveau ligne (et du coup, ils peuvent se produire même si le verrouillage explicite n'est pas utilisé). Considérons le cas où il existe deux transactions concurrentes modifiant une table. La première transaction exécute :

```
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte = 11111;
```

Elle acquiert un verrou au niveau ligne sur la ligne spécifiée par le numéro de compte (no_compte). Ensuite, la deuxième transaction exécute :

```
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte = 22222;
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte = 11111;
```

La première instruction **UPDATE** acquiert avec succès un verrou au niveau ligne sur la ligne spécifiée, donc elle réussit à mettre à

jour la ligne. Néanmoins, la deuxième instruction **UPDATE** trouve que la ligne qu'elle essaie de mettre à jour a déjà été verrouillée, alors elle attend la fin de la transaction ayant acquis le verrou. Maintenant, la première transaction exécute :

```
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte = 22222;
```

La première transaction essaie d'acquérir un verrou au niveau ligne sur la ligne spécifiée mais ne le peut pas : la deuxième transaction détient déjà un verrou. Donc, elle attend la fin de la transaction deux. Du coup, la première transaction est bloquée par la deuxième et la deuxième est bloquée par la première : une condition de blocage, un verrou mort. PostgreSQL™ détectera cette situation et annulera une des transactions.

La meilleure défense contre les verrous morts est généralement de les éviter en s'assurant que toutes les applications utilisant une base de données acquièrent des verrous sur des objets multiples dans un ordre cohérent. Dans l'exemple ci-dessus, si les deux transactions avaient mis à jour les lignes dans le même ordre, aucun blocage n'aurait eu lieu. Vous devez vous assurer que le premier verrou acquis sur un objet dans une transaction est dans le mode le plus restrictif pour cet objet. S'il n'est pas possible de vérifier ceci à l'avance, alors les blocages doivent être gérés à l'exécution en ré-essayant les transactions annulées à cause de blocage.

Tant qu'aucune situation de blocage n'est détectée, une transaction cherchant soit un verrou de niveau table soit un verrou de niveau ligne attend indéfiniment que les verrous en conflit soient relâchés. Ceci signifie que maintenir des transactions ouvertes sur une longue période de temps (par exemple en attendant une saisie de l'utilisateur) est parfois une mauvaise idée.

13.3.4. Verrous informatifs

PostgreSQL™ fournit un moyen pour créer des verrous qui ont une signification définie par l'application. Ils sont qualifiés d'*informatifs* car le système ne force pas leur utilisation -- c'est à l'application de les utiliser correctement. Les verrous informatifs peuvent être utiles pour des manières d'utiliser le verrouillage qui ne sont pas en phase avec le modèle MVCC. Une fois acquis, un verrou informatif est détenu jusqu'à ce qu'il soit relâché ou que la session se termine. Contrairement aux verrous standards, les verrous informatifs n'honorent pas la sémantique des transactions : un verrou informatif acquis lors d'une transaction qui est par la suite annulée est toujours détenu après l'annulation. Et de la même façon un déverrouillage sera toujours effectif même si la transaction appelante échoue ultérieurement. Le même verrou peut être acquis plusieurs fois par le processus qui le détient : à chaque demande de verrou doit correspondre une demande de déverrouillage afin que le verrou soit réellement supprimé. (Si une session détient déjà un verrou donné, les demandes suivantes réussissent même si d'autres sessions attendent le verrou.) Comme tous les verrous de PostgreSQL™, une liste complète des verrous informatifs détenus actuellement par une session est visible dans la vue système pg_locks.

Les verrous informatifs sont alloués à partir d'une portion de mémoire partagée dont la taille est définie par les variables de configuration max_locks_per_transaction et max_connections. Attention à ne pas vider cette mémoire, sinon le serveur ne serait plus capable d'accorder des verrous. Ceci impose une limite supérieure au nombre de verrous informatifs que le serveur peut accorder, typiquement entre des dizaines et des centaines de milliers suivant la façon dont le serveur est configuré.

Une utilisation commune des verrous informatifs est d'émuler des stratégies de verrou pessimiste, typiques des systèmes de gestion de données sur « fichier plat ». Alors qu'une option stockée dans une table pourrait être utilisée dans le même but, les verrous informatifs sont plus rapides, évitent le grossissement de MVCC et sont automatiquement nettoyés par le serveur à la fin d'une session. Dans certains cas qui utilisent cette méthode, tout spécialement les requêtes impliquant un tri explicite et des clauses `LIMIT`, une grande attention doit être portée au contrôle des verrous acquis, à cause de l'ordre dans lequel les expressions SQL sont évaluées. Par exemple :

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger !
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

Dans les requêtes ci-dessus, la deuxième forme est dangereuse parce qu'il n'est pas garanti que l'application de `LIMIT` ait lieu avant que la fonction du verrou soit exécutée. Ceci pourrait entraîner l'acquisition de certains verrous que l'application n'attendait pas, donc qu'elle ne pourrait, du coup, pas relâcher (sauf à la fin de la session). Du point de vue de l'application, de tels verrous sont en attente, bien qu'ils soient visibles dans pg_locks.

Les fonctions fournies pour manipuler les verrous informatifs sont décrites dans Tableau 9.61, « Fonctions de verrous consultatifs ».

13.4. Vérification de cohérence des données au niveau de l'application

Quel que soit le niveau d'isolation, les lecteurs ne verrouillent pas les données avec PostgreSQL™. Les données lues par une transaction peuvent donc être écrasées par une autre transaction concurrente. En d'autres termes, si une ligne est renvoyée par **SELECT**, cela ne signifie pas que la ligne est toujours d'actualité au moment où elle est renvoyée (c'est-à-dire, quelque temps après

que la requête **SELECT** ait commencé). La ligne pourrait avoir été modifiée ou supprimée par une transaction validée entretemps. Même si la ligne est encore valide « maintenant », elle pourrait être modifiée ou supprimée avant que notre transaction ne soit validée ou annulée.

Une autre façon d'aborder ceci est que chaque transaction voit un instantané du contenu de la base de données. Des transactions concurrentes en cours d'exécution pourraient très bien voir des instantanés différents. Le concept entier de « maintenant » est donc quelque peu mal défini. Habituellement, ceci n'est pas un gros problème si les applications clientes sont isolées les unes des autres. Mais si les clients peuvent communiquer via des canaux externes à la base de données, de sérieuses confusions pourraient survenir.

Pour s'assurer de la validité actuelle d'une ligne et pour la protéger contre les modifications concurrentes, vous devez utiliser **SELECT FOR UPDATE**, **SELECT FOR SHARE** ou une instruction **LOCK TABLE** appropriée (**SELECT FOR UPDATE** et **SELECT FOR SHARE** verrouillent uniquement les lignes renvoyées contre les mises à jours concurrentes alors que **LOCK TABLE** verrouille la table entière). Ceci doit être pris en compte lors du portage d'applications vers PostgreSQL™ depuis d'autres environnements.

Des vérifications globales de validité requièrent une réflexion supplémentaire avec MVCC. Par exemple, une application bancaire pourrait souhaiter vérifier que la somme de tous les crédits d'une table est équivalente à la somme des débits d'une autre table lorsque les deux tables sont activement mises à jour. Comparer les résultats de deux commandes `SELECT sum(...)` successives ne fonctionnera pas correctement avec le mode de lecture validée car la deuxième requête a des chances d'inclure les résultats de transactions non comptabilisées dans la première. Faire les deux sommes dans une seule transaction sérialisée donnera une image plus précise des effets des transactions validées avant le début de la transaction sérialisable -- mais vous pourriez vous demander si la réponse est toujours juste au moment où elle est fournie. Si la transaction sérialisable a elle-même appliqué des modifications avant de tenter la vérification de cohérence, l'utilité de la vérification devient bien plus discutable car, maintenant, elle inclut des modifications apparues après le début de la transaction mais pas toutes. Dans de tels cas, une personne attentionnée pourrait souhaiter verrouiller toutes les tables nécessaires à la vérification pour obtenir une image indiscutable de la réalité. Un verrou de mode `SHARE` (ou plus important) garantit qu'il n'y a aucune modification non validée dans la table verrouillée, autres que celles de la transaction.

Il est à noter que si vous voulez vous reposer sur les verrous explicites pour éviter les changements concurrents, vous devriez utiliser le mode Read Commited, ou alors, dans le mode sérialisable, être attentif à l'obtention des verrous avant d'effectuer des requêtes. Un verrou obtenu par une transaction sérialisable garantit qu'aucune autre transaction modifiant la table n'est en cours d'exécution mais si l'image vue par la transaction est antérieure à l'obtention du verrou, elle pourrait être antérieure aux quelques modifications maintenant validées dans la table. Une image de la transaction sérialisable est généralement gelée au début de la première requête ou de la première commande de modification de données (`SELECT`, `INSERT`, `UPDATE` ou `DELETE`). Du coup, il est possible d'obtenir des verrous de façon explicite avant que l'image ne soit gelée.

13.5. Verrouillage et index

Bien que PostgreSQL™ fournisse des accès non bloquants en lecture/écriture aux données de la table, un accès non bloquant en lecture/écriture n'est pas fourni pour chaque méthode d'accès aux index implémentée dans PostgreSQL™. Les différents types d'index sont gérés ainsi :

Index B-tree et GiST
: Les verrous partagés/exclusifs de court terme au niveau page sont utilisés pour les accès en lecture/écriture. Les verrous sont immédiatement relâchés après le parcours ou l'insertion de chaque ligne d'index. Ces types d'index fournissent le plus haut niveau de concurrence sans conditions de blocage.

Index hachés
: Les verrous partagés/exclusifs au niveau hash-bucket sont utilisés pour des accès en lecture/écriture. Les verrous sont relâchés après la fin des traitements sur le bucket. Les verrous au niveau bucket fournissent un meilleur accès concurrent que les verrous au niveau index mais sont sensibles aux interblocages car les verrous sont détenus plus longtemps que pour une opération sur un index.

Index GIN
: Les verrous partagés/exclusifs à court terme et de niveau page sont utilisés pour les accès en lecture/écriture. Les verrous sont relâchés immédiatement après chaque récupération ou insertion de ligne d'index. Mais notez que l'insertion d'une valeur indexée par GIN produit habituellement plusieurs de clés d'index par ligne, donc GIN peut faire un travail substantiel pour l'insertion d'une simple valeur.

Actuellement, les index B-tree offrent la meilleure performance pour les applications concurrentes ; comme ils ont aussi plus de fonctionnalités que les index hachés, ils constituent le type d'index recommandé pour les applications concurrentes nécessitant des index sur des données scalaires. Pour les données non scalaires, les index B-trees ne sont pas utiles et les index GIN et GiST devraient être utilisés à leur place.

Chapitre 14. Conseils sur les performances

La performance des requêtes peut être affectée par un grand nombre d'éléments. Certains peuvent être contrôlés par l'utilisateur, d'autres sont fondamentaux au concept sous-jacent du système. Ce chapitre fournit des conseils sur la compréhension et sur la configuration fine des performances de PostgreSQL™.

14.1. Utiliser EXPLAIN

PostgreSQL™ réalise un *plan de requête* pour chaque requête qu'il reçoit. Choisir le bon plan correspondant à la structure de la requête et aux propriétés des données est absolument critique pour de bonnes performances, donc le système inclut un *planificateur* complexe qui tente de choisir les bons plans. Vous pouvez utiliser la commande EXPLAIN(7) pour voir quel plan de requête le planificateur crée pour une requête particulière. La lecture du plan est un art qui mérite un tutoriel complet, ce que vous n'aurez pas là ; ici ne se trouvent que des informations de base.

La structure d'un plan de requête est un arbre de *nœuds de plan*. Les nœuds de bas niveau sont les nœuds de parcours de tables : ils renvoient les lignes brutes d'une table. Il existe différents types de nœuds de parcours pour les différentes méthodes d'accès aux tables : parcours séquentiel, parcours d'index et parcours d'index bitmap. Si la requête requiert des jointures, agrégations, tris ou d'autres opérations sur les lignes brites, ce seront des nœuds supplémentaires au-dessus des nœuds de parcours pour réaliser ces opérations. Encore une fois, il existe plus d'une façon de réaliser ces opérations, donc différents types de nœuds peuvent aussi apparaître ici. La sortie d'**EXPLAIN** comprend une ligne pour chaque nœud dans l'arbre du plan, montrant le type de nœud basique avec les estimations de coût que le planificateur a fait pour l'exécution de ce nœud du plan. La première ligne (le nœud tout en haut) comprend le coût d'exécution total estimé pour le plan ; c'est ce nombre que le planificateur cherche à minimiser.

Voici un exemple trivial, juste pour montrer à quoi ressemble l'affichage. [1]

```
EXPLAIN SELECT * FROM tenk1;

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Les nombres donnés par **EXPLAIN** sont (de gauche à droite) :

- Coût estimé du lancement (temps passé avant que l'affichage de la sortie ne commence, c'est-à-dire pour faire le tri dans un nœud de tri) ;
- Coût total estimé (si toutes les lignes doivent être récupérées, ce qui pourrait ne pas être le cas : par exemple une requête avec une clause `LIMIT` ne paiera pas le coût total du nœud d'entrée du nœud du plan `Limit`) ;
- Nombre de lignes estimé en sortie par ce nœud de plan (encore une fois, seulement si exécuté jusqu'au bout) ;
- Largeur moyenne estimée (en octets) des lignes en sortie par ce nœud de plan.

Les coûts sont mesurés en unités arbitraires déterminées par les paramètres de coût du planificateur (voir Section 18.6.2, « Constantes de coût du planificateur »). La pratique habituelle est de mesurer les coûts en unité de récupération de pages disque ; autrement dit, seq_page_cost est initialisé à `1.0` par convention et les autres paramètres de coût sont relatifs à cette valeur. Les exemples de cette section sont exécutés avec les paramètres de coût par défaut.

Il est important de noter que le coût d'un nœud de haut niveau inclut le coût de tous les nœuds fils. Il est aussi important de réaliser que le coût reflète seulement les éléments d'importance pour le planificateur. En particulier, le coût ne considère pas le temps dépensé dans la transmission des lignes de résultat au client, ce qui pourrait être un facteur important dans le temps réel passé ; mais le planificateur l'ignore parce qu'il ne peut pas le changer en modifiant le plan (chaque plan correct sortira le même ensemble de lignes).

La valeur `rows` est un peu difficile car il ne s'agit *pas* du nombre de lignes traitées ou parcourues par le plan de nœuds. C'est habituellement moins, reflétant la sélectivité estimée des conditions de la clause `WHERE` qui sont appliquées au nœud. Idéalement, les estimations des lignes de haut niveau sera une approximation des nombres de lignes déjà renvoyées, mises à jour, supprimées par la requête.

Pour revenir à notre exemple :

```
EXPLAIN SELECT * FROM tenk1;
```

[1] Les exemples dans cette section sont récupérés de la base de données des tests de régression après avoir lancé un **VACUUM ANALYZE**, en utilisant les sources de la version 8.2. Vous devriez être capable d'obtenir des résultats similaires si vous essayez vous-même les exemples mais vos coûts estimés et les nombres de lignes varieront probablement légèrement car les statistiques d'**ANALYZE** se font à partir de valeurs prises au hasard.

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

C'est aussi direct que ce que nous obtenons. Si vous faîtes :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

vous trouverez que `tenk1` a 358 pages disque et 10000 lignes. Le coût estimé est calculé avec (nombre de pages lues * seq_page_cost) + (lignes parcourues * cpu_tuple_cost). Par défaut, `seq_page_cost` vaut 1.0 et `cpu_tuple_cost` vaut 0.01. Donc le coût estimé est de (358 * 1.0) + (10000 * 0.01), soit 458.

Maintenant, modifions la requête originale pour ajouter une condition `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7033 width=244)
   Filter: (unique1 < 7000)
```

Notez que l'affichage d'**EXPLAIN** montre la clause `WHERE` appliquée comme une condition de « filtre » ; ceci signifie que le nœud de plan vérifie la condition pour chaque ligne qu'il parcourt et ne conserve que celles qui satisfont la condition. L'estimation des lignes en sortie a baissé à cause de la clause `WHERE`. Néanmoins, le parcours devra toujours visiter les 10000 lignes, donc le coût n'a pas baissé ; en fait, il a un peu augmenté (par 10000 * cpu_operator_cost pour être exact) dans le but de refléter le temps CPU supplémentaire dépensé pour vérifier la condition `WHERE`.

Le nombre réel de lignes que cette requête sélectionnera est 7000 mais l'estimation `rows` est approximative. Si vous tentez de dupliquer cette expérience, vous obtiendrez probablement une estimation légèrement différente ; de plus, elle changera après chaque commande **ANALYZE** parce que les statistiques produites par **ANALYZE** sont prises à partir d'un extrait au hasard de la table.

Maintenant, rendons la condition plus restrictive :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=2.37..232.35 rows=106 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
         Index Cond: (unique1 < 100)
```

Ici, le planificateur a décidé d'utiliser un plan en deux étapes : le nœud en bas du plan visite un index pour trouver l'emplacement des lignes correspondant à la condition de l'index, puis le nœud du plan du dessus récupère réellement ces lignes de la table. Récupérer séparément les lignes est bien plus coûteux que de les lire séquentiellement mais comme toutes les pages de la table n'ont pas à être visitées, cela revient toujours moins cher qu'un parcours séquentiel (la raison de l'utilisation d'un plan à deux niveaux est que le nœud du plan du dessus trie les emplacements des lignes identifiés par l'index dans l'ordre physique avant de les lire pour minimiser les coûts des récupérations séparés. Le « bitmap » mentionné dans les noms de nœuds est le mécanisme qui s'occupe du tri).

Si la condition `WHERE` est assez sélective, le planificateur pourrait basculer vers un plan de parcours d'index « simple » :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..10.00 rows=2 width=244)
   Index Cond: (unique1 < 3)
```

Dans ce cas, les lignes de la table sont récupérées dans l'ordre de l'index, ce qui les rend encore plus coûteuses à lire mais elles sont si peu nombreuses que le coût supplémentaire de triage des emplacements de lignes ne vaut pas le coup. Vous verrez plus fréquemment ce type de plan pour les requêtes qui récupèrent une seule ligne et pour les requêtes qui ont une condition `ORDER BY` correspondant à l'ordre de l'index.

Ajoutez une autre condition à la clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3 AND stringu1 = 'xxx';

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..10.01 rows=1 width=244)
```

```
   Index Cond: (unique1 < 3)
   Filter: (stringu1 = 'xxx'::name)
```

La condition ajoutée `stringu1 = 'xxx'` réduit l'estimation du nombre de lignes en sortie mais pas le coût car nous devons toujours visiter le même ensemble de lignes. Notez que la clause `stringu1` ne peut pas être appliqué à une condition d'index (car cet index est seulement sur la colonne `unique1`). À la place, il est appliqué comme un filtre sur les lignes récupérées par l'index. Du coup, le coût a un peu augmenté pour refléter cette vérification supplémentaire.

S'il existe des index sur plusieurs colonnes référencées dans la condition `WHERE`, le planificateur pourrait choisir d'utiliser une combinaison AND ou OR des index :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=11.27..49.11 rows=11 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=11.27..11.27 rows=11 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..8.65 rows=1042 width=0)
               Index Cond: (unique2 > 9000)
```

Mais ceci requiert de visiter plusieurs index, donc ce n'est pas nécessaire un gain comparé à l'utilisation d'un seul index et au traitement de l'autre condition par un filtre. Si vous variez les échelles de valeurs impliquées, vous vous apercevrez que le plan change en accord.

Maintenant, essayons de joindre deux tables, en utilisant les colonnes dont nous avons discuté :

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244)
         Index Cond: (t2.unique2 = t1.unique2)
```

Dans cette jointure en boucle imbriquée, le parcours externe utilise le même parcours de bitmap que celui vu précédemment et donc son coût et le nombre de lignes sont les mêmes parce que nous appliquons la clause `WHERE unique1 < 100` à ce nœud. La clause `t1.unique2 = t2.unique2` n'a pas encore d'intérêt donc elle n'affecte pas le nombre de lignes du parcours externe. Pour le parcours interne, la valeur `unique2` de la ligne courante du parcours externe est connectée dans le parcours d'index interne pour produire une condition d'index identique à `t2.unique2 =` *`constante`*. Donc, nous obtenons le même plan de parcours interne et les coûts que nous obtenons de, disons, `EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42`. Les coûts du nœud correspondant à la boucle sont ensuite initialisés sur la base du coût du parcours externe, avec une répétition du parcours interne pour chaque ligne externe (ici, 106 * 3.01), plus un petit temps CPU pour traiter la jointure.

Dans cet exemple, le nombre de lignes en sortie de la jointure est identique aux nombres de lignes des deux parcours mais ce n'est pas vrai en règle générale car vous pouvez avoir des clauses `WHERE` mentionnant les deux tables et qui, donc, peuvent seulement être appliquées au point de jointure, et non pas aux parcours d'index. Par exemple, si nous avions ajouté `WHERE ... AND t1.hundred < t2.hundred`, cela aurait diminué le nombre de lignes en sortie du nœud de jointure mais cela n'aurait pas changé les parcours d'index.

Une façon de rechercher des plans différents est de forcer le planificateur à oublier certaines stratégies qu'il aurait trouvé moins coûteuses en utilisant les options d'activation (enable)/désactivation (disable) décrites dans la Section 18.6.1, « Configuration de la méthode du planificateur » (c'est un outil complexe mais utile ; voir aussi la Section 14.3, « Contrôler le planificateur avec des clauses `JOIN` explicites »).

```
SET enable_nestloop = off;
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                         QUERY PLAN
-----------------------------------------------------------------------------------------
```

```
 Hash Join  (cost=232.61..741.67 rows=106 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..458.00 rows=10000 width=244)
   ->  Hash  (cost=232.35..232.35 rows=106 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106
width=0)
                     Index Cond: (unique1 < 100)
```

Ce plan propose d'extraire les 100 lignes intéressantes de `tenk1` en utilisant le même parcours d'index, de les placer dans une table de hachage en mémoire puis de faire un parcours séquentiel de `tenk2`, en cherchant dans la table de hachage des correspondances possibles de la ligne `t1.unique2 = t2.unique2` pour chaque `tenk2`. Le coût pour lire `tenk1` et pour initialiser la table de hachage correspond au coût de lancement complet pour la jointure hachée car nous n'obtiendrons pas de lignes jusqu'à avoir lu `tenk2`. Le temps total estimé pour la jointure inclut aussi une charge importante du temps CPU pour requêter la table de hachage 10000 fois. Néanmoins, notez que nous ne chargeons *pas* 10000 fois 232,35 ; la configuration de la table de hachage n'est exécutée qu'une fois dans ce type de plan.

Il est possible de vérifier la précision des coûts estimés par le planificateur en utilisant **EXPLAIN ANALYZE**. Cette commande exécute réellement la requête puis affiche le vrai temps d'exécution accumulé par chaque nœud du plan, avec les mêmes coûts estimés que ceux affichés par un simple **EXPLAIN**. Par exemple, nous pourrions obtenir un résultat comme celui-ci :

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..12.700
rows=100 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244) (actual
time=0.878..2.367 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
(actual time=0.546..0.546 rows=100 loops=1)
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244)
(actual time=0.067..0.078 rows=1 loops=100)
         Index Cond: (t2.unique2 = t1.unique2)
 Total runtime: 14.452 ms
```

Notez que les valeurs « temps réel » sont en millisecondes alors que les estimations de « coût » sont exprimées dans des unités arbitraires ; donc il y a peu de chances qu'elles correspondent. L'important est de vérifier si les ratios temps réel et coûts estimés correspondent.

Dans certains plans de requête, il est possible qu'un nœud de sous-plan soit exécuté plus d'une fois. Par exemple, le parcours d'index interne est exécuté une fois par ligne externe dans le plan de boucle imbriquée ci-dessus. Dans de tels cas, la valeur `loops` renvoie le nombre total d'exécution du nœud, et le temps réel et les valeurs des lignes affichées sont une moyenne par exécution. Ceci est fait pour que les nombres soient comparables avec la façon dont les estimations de coûts sont affichées. Multipliez par la valeur de `loops` pour obtenir le temps total réellement passé dans le nœud.

Le `Total runtime` (temps total d'exécution) affiché par **EXPLAIN ANALYZE** inclut les temps de lancement et d'arrêt de l'exécuteur, mais pas le temps passé pour l'analyse, la réécriture ou la planification. Pour les commandes **INSERT**, **UPDATE** et **DELETE**, le temps passé à appliquer les modifications est comptabilisé sur le nœud de haut-niveau Insert, Update ou Delete. (Les nœuds du plan sous ce nœud représentent le travail de recherche des anciennes lignes et/ou de calcul des anciennes.) Le temps passé à exécuter les triggers `BEFORE` (s'ils sont présents) est comptabilisé dans le nœud relatif (Insert, Update ou Delete). Par contre, ce n'est pas le cas pour les triggers `AFTER`. Le temps passé dans chaque trigger (`BEFORE` ou `AFTER`) est aussi affiché séparément et est inclus dans le temps d'exécution total. Notez cependant que les triggers de contraintes différés ne seront pas exécutés avant la fin de la transaction. Du coup, ils ne sont pas affichés par **EXPLAIN ANALYZE**.

Il existe deux raisons importantes pour lesquelles les temps d'exécution mesurés par **EXPLAIN ANALYZE** peuvent dévier de l'exécution normale de la même requête. Tout d'abord, comme aucune ligne n'est réellement envoyée au client, les coûts de transmission réseau et les coûts de formatage des entrées/sorties ne sont pas inclus. Ensuite, la surcharge généré par la commande **EXPLAIN ANALYZE** peut être importante, tout spécialement sur les machines dont les appels à la fonction `gettimeofday()` sont particulièrement lents.

Il est bon de noter que les résultats de **EXPLAIN** ne devraient pas être extrapolés pour des situations autres que celles de vos tests en cours ; par exemple, les résultats sur une petite table ne peuvent être appliqués à des tables bien plus importantes. Les estimations de coût du planificateur ne sont pas linéaires et, du coup, il pourrait bien choisir un plan différent pour une table plus petite

ou plus grande. Un exemple extrême est celui d'une table occupant une page disque. Vous obtiendrez pratiquement toujours un parcours séquentiel que des index soient disponibles ou non. Le planificateur réalise que cela va nécessiter la lecture d'une seule page disque pour traiter la table dans ce cas, il n'y a donc pas d'intérêt à étendre des lectures de pages supplémentaires pour un index.

14.2. Statistiques utilisées par le planificateur

Comme nous avons vu dans la section précédente, le planificateur de requêtes a besoin d'estimer le nombre de lignes récupérées par une requête pour faire les bons choix dans ses plans de requêtes. Cette section fournit un aperçu rapide sur les statistiques que le système utilise pour ces estimations.

Un élément des statistiques est le nombre total d'entrées dans chaque table et index, ainsi que le nombre de blocs disque occupés par chaque table et index. Cette information est conservée dans la table pg_class sur les colonnes `reltuples` et `relpages`. Nous pouvons la regarder avec des requêtes comme celle-ci :

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Ici, nous pouvons voir que tenk1 contient 10000 lignes, comme pour ses index, mais que les index sont bien plus petits que la table (ce qui n'est pas surprenant).

Pour des raisons d'efficacité, `reltuples` et `relpages` ne sont pas mis à jour en temps réel, et du coup, elles contiennent habituellement des valeurs un peu obsolètes. Elles sont mises à jour par les commandes **VACUUM**, **ANALYZE** et quelques commandes DDL comme **CREATE INDEX**. Un **ANALYZE** seul, donc ne faisant pas partie d'un **VACUUM**, génère une valeur approximative de `reltuples` car il ne lit pas chaque ligne de la table. Le planificateur mettra à l'échelle les valeurs qu'il aura trouver dans pg_class pour correspondre à la taille physique de la table, obtenant ainsi une approximation plus proche de la réalité.

La plupart des requêtes ne récupère qu'une fraction des lignes dans une table à cause de clauses `WHERE` qui restreignent les lignes à examiner. Du coup, le planificateur a besoin d'une estimation de la *sélectivité* des clauses `WHERE`, c'est-à-dire la fraction des lignes qui correspondent à chaque condition de la clause `WHERE`. L'information utilisée pour cette tâche est stockée dans le catalogue système pg_statistic. Les entrées de pg_statistic sont mises à jour par les commandes **ANALYZE** et **VACUUM ANALYZE** et sont toujours approximatives même si elles ont été mises à jour récemment.

Plutôt que de regarder directement dans pg_statistic, il est mieux de visualiser sa vue pg_stats lors de l'examen manuel des statistiques. pg_stats est conçu pour être plus facilement lisible. De plus, pg_stats est lisible par tous alors que pg_statistic n'est lisible que par un superutilisateur (ceci empêche les utilisateurs non privilégiés d'apprendre certains choses sur le contenu des tables appartenant à d'autres personnes à partir des statistiques. La vue pg_stats est restreinte pour afficher seulement les lignes des tables lisibles par l'utilisateur courant). Par exemple, nous pourrions lancer :

```
SELECT attname, alled, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | alled | n_distinct |          most_common_vals
---------+-----------+------------+-----------------------------------
 name    | f         |  -0.363388 | I- 580                        Ramp+
         |           |            | I- 880                        Ramp+
         |           |            | Sp Railroad                       +
         |           |            | I- 580                            +
         |           |            | I- 680                        Ramp
 name    | t         |  -0.284859 | I- 880                        Ramp+
         |           |            | I- 580                        Ramp+
         |           |            | I- 680                        Ramp+
         |           |            | I- 580                            +
         |           |            | State Hwy 13                  Ramp
(2 rows)
```

Notez que deux lignes sont affichées pour la même colonne, une correspondant à la hiérarchie d'héritage complète commençant à

la table `road` (`alled=t`), et une autre incluant seulement la table `road` elle-même (`alled=f`).

Le nombre d'informations stockées dans pg_statistic par **ANALYZE**, en particulier le nombre maximum d'éléments dans les tableaux *`most_common_vals`* et *`histogram_bounds`* pour chaque colonne, peut être initialisé sur une base colonne-par-colonne en utilisant la commande **ALTER TABLE SET STATISTICS** ou globalement en initialisant la variable de configuration default_statistics_target. La limite par défaut est actuellement de cent entrées. Augmenter la limite pourrait permettre des estimations plus précises du planificateur, en particulier pour les colonnes ayant des distributions de données irrégulières, au prix d'un plus grand espace consommé dans pg_statistic et en un temps plus long pour calculer les estimations. En revanche, une limite plus basse pourrait être suffisante pour les colonnes à distributions de données simples.

Le Chapitre 56, Comment le planificateur utilise les statistiques donne plus de détails sur l'utilisation des statistiques par le planificateur.

14.3. Contrôler le planificateur avec des clauses `JOIN` explicites

Il est possible de contrôler le planificateur de requêtes à un certain point en utilisant une syntaxe `JOIN` explicite. Pour voir en quoi ceci est important, nous avons besoin de quelques connaissances.

Dans une simple requête de jointure, telle que :

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

le planificateur est libre de joindre les tables données dans n'importe quel ordre. Par exemple, il pourrait générer un plan de requête qui joint A à B en utilisant la condition `WHERE a.id = b.id`, puis joint C à cette nouvelle table jointe en utilisant l'autre condition `WHERE`. Ou il pourrait joindre B à C, puis A au résultat de cette jointure précédente. Ou il pourrait joindre A à C puis les joindre avec B mais cela pourrait ne pas être efficace car le produit cartésien complet de A et C devra être formé alors qu'il n'y a pas de condition applicable dans la clause `WHERE` pour permettre une optimisation de la jointure (toutes les jointures dans l'exécuteur PostgreSQL™ arrivent entre deux tables en entrées donc il est nécessaire de construire le résultat de l'une ou de l'autre de ces façons). Le point important est que ces différentes possibilités de jointures donnent des résultats sémantiquement équivalents mais pourraient avoir des coûts d'exécution grandement différents. Du coup, le planificateur va toutes les explorer pour trouver le plan de requête le plus efficace.

Quand une requête implique seulement deux ou trois tables, il y a peu d'ordres de jointures à préparer. Mais le nombre d'ordres de jointures possibles grandit de façon exponentielle au fur et à mesure que le nombre de tables augmente. Au-delà de dix tables en entrée, il n'est plus possible de faire une recherche exhaustive de toutes les possibilités et même la planification de six ou sept tables pourrait prendre beaucoup de temps. Quand il y a trop de tables en entrée, le planificateur PostgreSQL™ basculera d'une recherche exhaustive à une recherche *génétique* probabiliste via un nombre limité de possibilités (la limite de bascule est initialisée par le paramètre en exécution geqo_threshold). La recherche génétique prend moins de temps mais elle ne trouvera pas nécessairement le meilleur plan possible.

Quand la requête implique des jointures externes, le planificateur est moins libre qu'il ne l'est lors de jointures internes. Par exemple, considérez :

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Bien que les restrictions de cette requête semblent superficiellement similaires à l'exemple précédent, les sémantiques sont différentes car une ligne doit être émise pour chaque ligne de A qui n'a pas de ligne correspondante dans la jointure entre B et C. Du coup, le planificateur n'a pas de choix dans l'ordre de la jointure ici : il doit joindre B à C puis joindre A à ce résultat. Du coup, cette requête prend moins de temps à planifier que la requête précédente. Dans d'autres cas, le planificateur pourrait être capable de déterminer que plus d'un ordre de jointure est sûr. Par exemple, étant donné :

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

il est valide de joindre A à soit B soit C en premier. Actuellement, seul un `FULL JOIN` contraint complètement l'ordre de jointure. La plupart des cas pratiques impliquant un `LEFT JOIN` ou un `RIGHT JOIN` peuvent être arrangés jusqu'à un certain degré.

La syntaxe de jointure interne explicite (`INNER JOIN`, `CROSS JOIN` ou `JOIN`) est sémantiquement identique à lister les relations en entrées du `FROM`, donc il ne contraint pas l'ordre de la jointure.

Même si la plupart des types de `JOIN` ne contraignent pas complètement l'ordre de jointure, il est possible d'instruire le planificateur de requête de PostgreSQL™ pour qu'il traite toutes les clauses `JOIN` de façon à contraindre quand même l'ordre de jointure. Par exemple, ces trois requêtes sont logiquement équivalentes :

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Mais si nous disons au planificateur d'honorer l'ordre des `JOIN`, la deuxième et la troisième prendront moins de temps à planifier que la première. Cet effet n'est pas inquiétant pour seulement trois tables mais cela pourrait bien nous aider avec un nombre im-

portant de tables.

Pour forcer le planificateur à suivre l'ordre de jointure demandé par les `JOIN` explicites, initialisez le paramètre en exécution join_collapse_limit à 1 (d'autres valeurs possibles sont discutées plus bas).

Vous n'avez pas besoin de restreindre l'ordre de jointure pour diminuer le temps de recherche car il est bien d'utiliser les opérateurs `JOIN` dans les éléments d'une liste `FROM`. Par exemple, considérez :

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Avec `join_collapse_limit` = 1, ceci force le planificateur à joindre A à B avant de les joindre aux autres tables mais sans restreindre ses choix. Dans cet exemple, le nombre d'ordres de jointures possibles est réduit par un facteur de cinq.

Restreindre la recherche du planificateur de cette façon est une technique utile pour réduire les temps de planification et pour diriger le planificateur vers un bon plan de requêtes. Si le planificateur choisit un mauvais ordre de jointure par défaut, vous pouvez le forcer à choisir un meilleur ordre via la syntaxe `JOIN` -- en supposant que vous connaissiez un meilleur ordre. Une expérimentation est recommandée.

Un problème très proche et affectant le temps de planification est le regroupement de sous-requêtes dans leurs requêtes parents. Par exemple, considérez :

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE quelquechose) AS ss
WHERE quelquechosedautre;
```

Cette requête pourrait survenir suite à l'utilisation d'une vue contenant une jointure ; la règle `SELECT` de la vue sera insérée à la place de la référence de la vue, demande une requête plutôt identique à celle ci-dessus. Normalement, le planificateur essaiera de regrouper la sous-requête avec son parent, donnant :

```
SELECT * FROM x, y, a, b, c WHERE quelquechose AND quelquechosedautre;
```

Ceci résulte habituellement en un meilleur plan que de planifier séparément la sous-requête (par exemple, les conditions `WHERE` externes pourraient être telles que joindre X à A élimine en premier lieu un bon nombre de lignes de A, évitant ainsi le besoin de former la sortie complète de la sous-requête). Mais en même temps, nous avons accru le temps de planification ; ici, nous avons une problème de jointure à cinq tables remplaçant un problème de deux jointures séparées à trois tables. À cause de l'augmentation exponentielle du nombre de possibilités, ceci fait une grande différence. Le planificateur essaie d'éviter de se retrouver coincé dans des problèmes de recherche de grosses jointures en ne regroupant pas une sous-requête sur plus de `from_collapse_limit` éléments sont la résultante de la requête parent. Vous pouvez comparer le temps de planification avec la qualité du plan en ajustant ce paramètre en exécution.

from_collapse_limit et join_collapse_limit sont nommés de façon similaire parce qu'ils font pratiquement la même chose : l'un d'eux contrôle le moment où le planificateur « aplatira » les sous-requêtes et l'autre contrôle s'il y a aplatissement des jointures explicites. Typiquement, vous initialiserez `join_collapse_limit` comme `from_collapse_limit` (de façon à ce que les jointures explicites et les sous-requêtes agissent de la même façon) ou vous initialiserez `join_collapse_limit` à 1 (si vous voulez contrôler l'ordre de jointure des jointures explicites). Mais vous pourriez les initialiser différemment si vous tentez de configurer finement la relation entre le temps de planification et le temps d'exécution.

14.4. Remplir une base de données

Vous pourriez avoir besoin d'insérer un grand nombre de données pour remplir une base de données au tout début. Cette section contient quelques suggestions pour réaliser cela de la façon la plus efficace.

14.4.1. Désactivez la validation automatique (autocommit)

Lors d'**INSERT** multiples, désactivez la validation automatique et faites une seule validation à la fin (en SQL, ceci signifie de lancer **BEGIN** au début et **COMMIT** à la fin. Quelques bibliothèques client pourraient le faire derrière votre dos auquel cas vous devez vous assurer que la bibliothèque le fait quand vous le voulez). Si vous permettez à chaque insertion d'être validée séparément, PostgreSQL™ fait un gros travail pour chaque ligne ajoutée. Un bénéfice supplémentaire de réaliser toutes les insertions dans une seule transaction est que si l'insertion d'une ligne échoue alors les lignes insérées jusqu'à maintenant seront annulées. Vous ne serez donc pas bloqué avec des données partiellement chargées.

14.4.2. Utilisez COPY

Utilisez COPY(7) pour charger toutes les lignes en une seule commande, plutôt que d'utiliser une série de commandes **INSERT**. La commande **COPY** est optimisée pour charger un grand nombre de lignes ; elle est moins flexible que **INSERT** mais introduit significativement moins de surcharge lors du chargement de grosses quantités de données. Comme **COPY** est une seule commande, il n'y a pas besoin de désactiver la validation automatique (autocommit) si vous utilisez cette méthode pour remplir une table.

Si vous ne pouvez pas utiliser **COPY**, utiliser PREPARE(7) pourrait vous aider à créer une instruction préparée **INSERT**, puis utilisez **EXECUTE** autant de fois que nécessaire. Ceci évite certaines surcharges lors d'une analyse et d'une planification répétées de commandes **INSERT**. Différentes interfaces fournissent cette fonctionnalité de plusieurs façons ; recherchez « instructions préparées » dans la documentation de l'interface.

Notez que charger un grand nombre de lignes en utilisant **COPY** est pratiquement toujours plus rapide que d'utiliser **INSERT**, même si **PREPARE ... INSERT** est utilisé lorsque de nombreuses insertions sont groupées en une seule transaction.

COPY est plus rapide quand il est utilisé dans la même transaction que la commande **CREATE TABLE** ou **TRUNCATE** précédente. Dans ce cas, les journaux de transactions ne sont pas impactés car, en cas d'erreur, les fichiers contenant les données nouvellement chargées seront supprimés de toute façon. Néanmoins, cette considération ne s'applique que quand wal_level vaut `minimal`, car toutes les commandes doivent écrire dans les journaux de transaction dans ce cas.

14.4.3. Supprimez les index

Si vous chargez une table tout juste créée, la méthode la plus rapide est de créer la table, de charger en lot les données de cette table en utilisant **COPY**, puis de créer tous les index nécessaires pour la table. Créer un index sur des données déjà existantes est plus rapide que de mettre à jour de façon incrémentale à chaque ligne ajoutée.

Si vous ajoutez beaucoup de données à une table existante, il pourrait être avantageux de supprimer les index, de charger la table, puis de recréer les index. Bien sûr, les performances de la base de données pour les autres utilisateurs pourraient souffrir tout le temps où les index seront manquant. Vous devez aussi y penser à deux fois avant de supprimer des index uniques car la vérification d'erreur apportée par la contrainte unique sera perdue tout le temps où l'index est manquant.

14.4.4. Suppression des contraintes de clés étrangères

Comme avec les index, une contrainte de clé étrangère peut être vérifiée « en gros volume » plus efficacement que ligne par ligne. Donc, il pourrait être utile de supprimer les contraintes de clés étrangères, de charger les données et de créer de nouveau les contraintes. De nouveau, il y a un compromis entre la vitesse de chargement des données et la perte de la vérification des erreurs lorsque la contrainte manque.

What's more, when you load data into a table with existing foreign key constraints, each new row requires an entry in the server's list of pending trigger events (since it is the firing of a trigger that checks the row's foreign key constraint). Loading many millions of rows can cause the trigger event queue to overflow available memory, leading to intolerable swapping or even outright failure of the command. Therefore it may be *necessary*, not just desirable, to drop and re-apply foreign keys when loading large amounts of data. If temporarily removing the constraint isn't acceptable, the only other recourse may be to split up the load operation into smaller transactions.

14.4.5. Augmentez `maintenance_work_mem`

Augmentez temporairement la variable maintenance_work_mem lors du chargement de grosses quantités de données peut amener une amélioration des performances. Ceci aidera à l'accélération des commandes **CREATE INDEX** et **ALTER TABLE ADD FOREIGN KEY**. Cela ne changera pas grand chose pour la commande **COPY**. Donc, ce conseil est seulement utile quand vous utilisez une des deux ou les deux techniques ci-dessus.

14.4.6. Augmentez `checkpoint_segments`

Augmenter temporairement la variable de configuration checkpoint_segments peut aussi aider à un chargement rapide de grosses quantités de données. Ceci est dû au fait que charger une grosse quantité de données dans PostgreSQL™ causera la venue trop fréquente de points de vérification (la fréquence de ces points de vérification est spécifiée par la variable de configuration `checkpoint_timeout`). Quand survient un point de vérification, toutes les pages modifiées sont écrites sur le disque. En augmentant `checkpoint_segments` temporairement lors du chargement des données, le nombre de points de vérification requis peut être diminué.

14.4.7. Désactiver l'archivage des journaux de transactions et la réplication en flux

Lors du chargement de grosse quantité de données dans une instance qui utilise l'archivage des journaux de transactions ou la réplication en flux, il pourrait être plus rapide de prendre une nouvelle sauvegarde de base après que le chargement ait terminé, plutôt que de traiter une grosse quantité de données incrémentales dans les journaux de transactions. Pour empêcher un accroissement de la journalisation des transactions lors du chargement, vous pouvez désactiver l'archivage et la réplication en flux lors du chargement en configurant wal_level à `minimal`, archive_mode à `off` et max_wal_senders à zéro). Mais notez que le changement de ces paramètres requiert un redémarrage du serveur.

En dehors d'éviter le temps de traitement des données des journaux de transactions par l'archiveur ou l'émetteur des journaux de

transactions, le faire rendrait certaines commandes plus rapides parce qu'elles sont conçues pour ne pas écrire du tout dans les journaux de transactions si `wal_level` vaut `minimal`. (Elles peuvent garantir la sûreté des données de façon moins coûteuse en exécutant un `fsync` à la fin plutôt qu'en écrivant les journaux de transactions :

- **CREATE TABLE AS SELECT**
- **CREATE INDEX** (et les variantes telles que **ALTER TABLE ADD PRIMARY KEY**)
- **ALTER TABLE SET TABLESPACE**
- **CLUSTER**
- **COPY FROM**, quand la table cible vient d'être créée ou vidée auparavant dans la transaction

14.4.8. Lancez ANALYZE après

Quand vous avez changé significativement la distribution des données à l'intérieur d'une table, lancer ANALYZE(7) est fortement recommandée. Ceci inclut le chargement de grosses quantités de données dans la table. Lancer **ANALYZE** (ou **VACUUM ANALYZE**) vous assure que le planificateur dispose de statistiques à jour sur la table. Sans statistiques ou avec des statistiques obsolètes, le planificateur pourrait prendre de mauvaises décisions lors de la planification de la requête, amenant des performances pauvres sur toutes les tables sans statistiques ou avec des statistiques inexactes. Notez que si le démon autovacuum est désactivée, il pourrait exécuter **ANALYZE** automatiquement ; voir Section 23.1.3, « Maintenir les statistiques du planificateur » et Section 23.1.5, « Le démon auto-vacuum » pour plus d'informations.

14.4.9. Quelques notes sur pg_dump

Les scripts de sauvegarde générés par pg_dump appliquent automatiquement plusieurs des indications ci-dessus, mais pas toutes. Pour recharger une sauvegarde pg_dump aussi rapidement que possible, vous avez besoin de faire quelques étapes supplémentaires manuellement (notez que ces points s'appliquent lors de la *restauration* d'une sauvegarde, et non pas lors de sa *création*. Les mêmes points s'appliquent soit lors de la restauration d'une sauvegarde texte avec psql soit lors de l'utilisation de pg_restore pour charger un fichier de sauvegarde pg_dump).

Par défaut, pg_dump utilise **COPY** et, lorsqu'il génère une sauvegarde complexe, schéma et données, il est préférable de charger les données avant de créer les index et les clés étrangères. Donc, dans ce cas, plusieurs lignes de conduite sont gérées automatiquement. Ce qui vous reste à faire est de :

- Configurez des valeurs appropriées (c'est-à-dire plus importante que la normale) pour `maintenance_work_mem` et `checkpoint_segments`.
- Si vous utilisez l'archivage des journaux de transactions ou la réplication en flux, considérez leur désactivation lors de la restauration. Pour faire cela, configurez `archive_mode` à `off`, `wal_level` à `minimal` et `max_wal_senders` à zéro avant de charger le script de sauvegarde. Après coup, remettez les anciennes valeurs et effectuez une nouvelle sauvegarde de base.
- Demandez-vous si la sauvegarde complète doit être restaurée dans une seule transaction. Pour cela, passez l'option `-1` ou `--single-transaction` à psql pi pg_restore. Lors de l'utilisation de ce mode, même les erreurs les plus petites annuleront la restauration complète, peut-être en annulant des heures de traitement. Suivant à quel point les données sont en relation, il peut être préférable de faire un nettoyage manuel. Les commandes **COPY** s'exécuteront plus rapidement si vous utilisez une transaction simple et que vous avez désactivé l'archivage des journaux de transaction.
- Si plusieurs processeurs sont disponibles sur le serveur, pensez à utiliser l'option `--jobs` de pg_restore. Cela permet la parallélisation du chargement des données et de la création des index.
- Exécutez **ANALYZE** après coup.

Une sauvegarde des données seules utilise toujours **COPY** mais elle ne supprime ni ne recrée les index et elle ne touche généralement pas les clés étrangères. [2] Donc, lorsque vous chargez une sauvegarde ne contenant que les données, c'est à vous de supprimer et recréer les index et clés étrangères si vous souhaitez utiliser ces techniques. Il est toujours utile d'augmenter `checkpoint_segments` lors du chargement des données mais ne vous embêtez pas à augmenter `maintenance_work_mem` ; en fait, vous le ferez lors d'une nouvelle création manuelle des index et des clés étrangères. Et n'oubliez pas **ANALYZE** une fois que vous avez terminé ; voir Section 23.1.3, « Maintenir les statistiques du planificateur » et Section 23.1.5, « Le démon auto-vacuum » pour plus d'informations.

[2] Vous pouvez obtenir l'effet de désactivation des clés étrangères en utilisant l'option `--disable-triggers` -- mais réalisez que cela élimine, plutôt que repousse, la validation des clés étrangères et qu'il est du coup possible d'insérer des données mauvaises si vous l'utilisez.

14.5. Configuration avec une perte acceptée

La durabilité est une fonctionnalité des serveurs de bases de données permettant de garantir l'enregistrement des transactions validées même si le serveur s'arrête brutalement, par exemple en cas de coupure électrique. Néanmoins, la durabilité ajoute une surcharge significative. Si votre base de données n'a pas besoin de cette ganratie, PostgreSQL™ peut être configuré pour fonctionner bien plus rapidement. Voici des modifications de configuration que vous pouvez faire pour améliorer les performances dans ce cas ; elles n'invalident pas les garanties sur la validité des transactions en cas d'arrêt brutal, seulement dans le cas où le système d'exploitation est stoppé brutalement, sauf cas contraire mentionné ci-dessous :

- Placer le répertoire des données dans un système de fichiers en mémoire (par exemple un disque RAM). Ceci élimine toutes les entrées/sorties disque de la base de données. Cela limite aussi la quantité de mémoire disponible (et peut-être aussi du swap).
- Désactiver fsync ; il n'est pas nécessaire d'écrire les données sur disque.
- Désactiver full_page_writes ; il n'est pas nécessaire de se prémunir contre les écritures de pages partielles.
- Augmenter checkpoint_segments et checkpoint_timeout ; cela réduit les fréquences des CHECKPOINT mais augmente l'espace disque nécessaire dans `pg_xlog`.
- Désactiver synchronous_commit ; il n'est pas forcément nécessaire d'écrire les journaux de transactions (WAL) à chaque validation de transactions. Ceci affecte la durabilité des transactions suite à un arrêt brutal de la base.

Partie III. Administration du serveur

Cette partie couvre des thèmes de grand intérêt pour un administrateur de bases de données PostgreSQL™, à savoir l'installation du logiciel, la mise en place et la configuration du serveur, la gestion des utilisateurs et des bases de données et la maintenance. Tout administrateur d'un serveur PostgreSQL™, même pour un usage personnel, mais plus particulièrement en production, doit être familier des sujets abordés dans cette partie.

Les informations sont ordonnées de telle sorte qu'un nouvel utilisateur puisse les lire linéairement du début à la fin. Cependant les chapitres sont indépendants et peuvent être lus séparément. L'information est présentée dans un style narratif, regroupée en unités thématiques. Les lecteurs qui recherchent une description complète d'une commande particulière peuvent se référer à la Partie VI, « Référence ».

Les premiers chapitres peuvent être compris sans connaissances préalables. Ainsi, de nouveaux utilisateurs installant leur propre serveur peuvent commencer leur exploration avec cette partie.

Le reste du chapitre concerne l'optimisation (tuning) et la gestion. Le lecteur doit être familier avec l'utilisation générale du système de bases de données PostgreSQL™. Les lecteurs sont encouragés à regarder la Partie I, « Tutoriel » et la Partie II, « Langage SQL » pour obtenir des informations complémentaires.

Chapitre 15. Procédure d'installation de PostgreSQL™ du code source

Ce document chapitre décrit l'installation de PostgreSQL™ en utilisant le code source. (Ce document chapitre peut être ignoré lors de l'installation d'une distribution pré-empaquetée, paquet RPM ou Debian, par exemple. Il est alors plus utile de lire les instruction du mainteneur du paquet.)

15.1. Version courte

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

Le reste du document chapitre est la version longue.

15.2. Prérequis

En général, les plateformes style unix modernes sont capables d'exécuter PostgreSQL™. Les plateformes sur lesquelles des tests ont été effectués sont listées dans la Section 15.9, « Plateformes supportées » ci-après. Dans le répertoire `doc` de la distribution, il y a plusieurs FAQ spécifiques à des plateformes particulières à consulter en cas de difficultés.

Les logiciels suivants sont nécessaires pour compiler PostgreSQL™ :

- GNU make est nécessaire ; les autres programmes make *ne* fonctionnent *pas*. GNU make est souvent installé sous le nom `gmake` ; ce document y fait toujours référence sous ce nom (sur certains système, GNU make est l'outil par défaut et est nommé `make`). Pour connaître la version utilisée, saisir

  ```
  gmake --version
  ```

  Il est recommandé d'avoir une version postérieure à la version 3.79.1.
- Il est nécessaire d'avoir un compilateur C ISO/ANSI (au minimum compatible avec C89). Une version récente de GCC™ est recommandée mais PostgreSQL™ est connu pour être compilable avec de nombreux compilateurs de divers vendeurs.
- tar est requis pour déballer la distribution des sources, associé à gzip ou bzip2.
- La bibliothèque GNU Readline™ est utilisée par défaut. Elle permet à psql (l'interpréteur de ligne de commandes SQL de PostgreSQL) de se souvenir de chaque commande saisie, et permet d'utiliser les touches de flêches pour rappeler et éditer les commandes précédentes. C'est très pratique et fortement recommandé. Pour ne pas l'utiliser, il faut préciser `--without-readline` au moment de l'exécution de la commande `configure`. Une alternative possible est l'utilisation de la bibliothèqe `libedit` sous license BSD, développée au début sur NetBSD™. La bibliothèque `libedit` est compatible GNU Readline™ et est utilisée si cette dernière n'est pas trouvée ou si `--with-libedit-preferred` est utilisé sur la ligne de commande de `configure`. Lorsqu'une distribution Linux à base de paquets est utilisée, si les paquets `readline` et `readline-devel` sont séparés, il faut impérativement installer les deux.
- La bibliothèque de compression zlib™ est utilisée par défaut. Pour ne pas l'utiliser, il faut préciser `--without-zlib` à `configure`. Cela a pour conséquence de désactiver le support des archives compressées dans pg_dump et pg_restore.

Les paquets suivants sont optionnels. S'ils ne sont pas obligatoires lors d'une compilation par défaut de PostgreSQL™, ils le deviennent lorsque certaines options sont utilisées, comme cela est expliqué par la suite.

- Pour installer le langage procédural PL/Perl, une installation complète de Perl™, comprenant la bibliothèque `libperl` et les fichiers d'en-tête, est nécessaire.

  Comme PL/Perl est une bibliothèque partagée, la bibliothèque `libperl` doit aussi être partagée sur la plupart des plateformes. C'est désormais le choix par défaut dans les versions récentes de Perl™, mais ce n'était pas le cas dans les versions

plus anciennes. Dans tous les cas, c'est du ressort de celui qui installe Perl. Si vous avez l'intention d'avoir plus qu'une utilisation occasionnelle de PL/Perl, vous devez vous assurer que l'installation de Perl™ a été faite avec l'option `usemultiplicity` activée (`perl -V` vous indiquera si c'est le cas).

Si la bibliothèque partagée, nécessaire, n'est pas présente, un message d'avertissement tel que celui qui suit apparaît à la compilation :

```
*** Cannot build PL/Perl because libperl is not a shared library.
*** You might have to rebuild your Perl installation.  Refer to
*** the documentation for details.
```

(Si la sortie écran est ignorée, on peut constater que la bibliothèque `plperl.so` de PL/Perl, ou similaire, n'est pas installée.) Si ce message est affiché, il faut recompiler et réinstaller Perl™ manuellement pour pouvoir compiler PL/Perl. Lors de la phase de configuration de Perl™, il faut demander une bibliothèque partagée.

- Pour compiler le langage de programmation serveur PL/Python, il faut que Python™ soit installé avec les fichiers d'en-tête et le module distutils. La version requise minimum est Python™ 2.2. Python 3™ est supporté s'il s'agit d'une version 3.1 ou ultérieure ; voir la documentation de PL/Python Section 42.1, « Python 2 et Python 3 » lors de l'utilisation de Python 3.

  Puisque PL/Python doit être une bibliothèque partagée, la bibliothèque `libpython` doit l'être aussi sur la plupart des plateformes. Ce n'est pas le cas des installations par défaut de Python. Si, après la compilation et l'installation de PostgreSQL™, il existe un fichier nommé `plpython.so` (des extensions différentes sont possibles), tout va bien. Sinon, il est fort probable qu'un avertissement semblable à celui qui suit soit apparu :

  ```
  *** Cannot build PL/Python because libpython is not a shared library.
  *** You might have to rebuild your Python installation.  Refer to
  *** the documentation for details.
  ```

  Ce qui signifie qu'il faut recompiler (une partie de) Python™ pour créer cette bibliothèque partagée.

  En cas de problèmes, on peut exécuter le configure de Python™ 2.3 ou ultérieur en utilisant le commutateur `--enable-shared`. Sur certains systèmes d'exploitation, il n'est pas nécessaire de construire une bibliothèque partagée, mais il faut en convaincre le système de construction de PostgreSQL™. Le fichier `Makefile` du répertoire `src/pl/plpython` donne des détails complémentaires.

- Pour construire le langage procédural PL/Tcl, Tcl™ doit être installé. Si une version antérieure à la version 8.4 de Tcl™, est utilisée, on s'assurera qu'il a été construit sans le support du multi-thread.

- Pour activer le support de langage natif (NLS), qui permet d'afficher les messages d'un programme dans une langue autre que l'anglais, une implantation de l'API Gettext est nécessaire. Certains systèmes d'exploitation l'intégrent (par exemple, `Linux`, `NetBSD`, `Solaris`). Pour les autres systèmes, un paquet additionnel peut être téléchargé sur *http://www.gnu.org/software/gettext/*. Pour utiliser l'implantation Gettext des bibliothèques C GNU, certains utilitaires nécessitent le paquet GNU Gettext™. Il n'est pas nécessaire dans les autres implantations.

- Vous avez besoin de Kerberos, OpenSSL™, OpenLDAP™ ou PAM pour bénéficier de l'authentification ou du chiffrement en utilisant ces services.

En cas de compilation à partir d'une arborescence Git et non d'un paquet de sources publié, ou pour faire du développement au niveau serveur, les paquets suivants seront également nécessaires :

- GNU Flex et Bison sont nécessaires pour compiler à partir d'un export du Git ou lorsque les fichiers de définition de l'analyseur ou du « scanner » sont modifiés. Les versions nécessaires sont Flex 2.5.31 ou ultérieure et Bison 1.875 ou ultérieure. Les autres programmes lex et yacc ne peuvent pas être utilisés.

- Perl 5.8 ou ultérieur est aussi nécessaire pour construire les sources du Git, ou lorsque les fichiers en entrée pour n'importe laquelle des étapes de construction qui utilisent des scripts Perl ont été modifiés. Sous Windows, Perl est nécessaire dans tous les cas.

Si d'autres paquets GNU sont nécessaires, ils peuvent être récupérés sur un site miroir de GNU (voir *http://www.gnu.org/order/ftp.html* pour la liste) ou sur *ftp://ftp.gnu.org/gnu/*.

Il est important de vérifier qu'il y a suffisamment d'espace disque disponible. 100 Mo sont nécessaires pour la compilation et 20 Mo pour le répertoire d'installation. Un groupe de bases de données vide nécessite 35 Mo ; les fichiers des bases prennent cinq fois plus d'espace que des fichiers texte contenant les mêmes données. Si des tests de régression sont prévus, 150 Mo supplémentaires sont temporairement nécessaires. On peut utiliser la commande **df** pour vérifier l'espace disque disponible.

15.3. Obtenir les sources

Les sources de PostgreSQL™ 9.0.18 peuvent être obtenues par ftp anonyme sur *ftp://ftp.postgresql.org/pub/source/v9.0.18/postgresql-9.0.18.tar.gz*. D'autres options de téléchargement sont possibles à partir du site web : *http://www.postgresql.org/download/*. Après avoir obtenu le fichier, on le décompresse :

```
gunzip postgresql-9.0.18.tar.gz
tar xf postgresql-9.0.18.tar
```

Cela crée un répertoire `postgresql-9.0.18` contenant les sources de PostgreSQL™ dans le répertoire courant. Le reste de la procédure d'installation s'effectue depuis ce répertoire.

Les sources peuvent également être obtenues directement à partir du système de contrôle de version. Pour plus d'informations, voir Annexe H, Dépôt du code source.

15.4. Mise à jour

Ces instructions supposent que l'installation existante se trouve dans le répertoire `/usr/local/pgsql`, et que l'aire de données est dans `/usr/local/pgsql/data`. Il convient de modifier ces chemins en fonction de l'installation actuelle.

Le format de stockage interne des données change typiquement à chaque version majeure de PostgreSQL™. C'est pourquoi, lors de la mise à jour d'une installation qui n'est pas du numéro de version « 9.0.x », il faut sauvegarder et restaurer ses données. Lors d'une mise à jour à partir de PostgreSQL™ « 9.0.x », la nouvelle version peut utiliser les fichiers de données actuels ; les étapes de sauvegarde et de restauration ci-dessous, inutiles, peuvent être ignorées.

1. Lors de la sauvegarde, il est préférable de s'assurer que la base de données n'est pas actualisée. Non que cela affecte l'intégrité de la sauvegarde, mais parce que les données modifiées ne sont pas incluses. Il peut, dans ce cas, être nécessaire d'éditer les permissions dans le fichier `/usr/local/pgsql/data/pg_hba.conf` (ou équivalent) pour interdire les accès autres que celui utile à la sauvegarde.

   Pour sauvegarder la base, saisir :

   ```
   pg_dumpall > fichierdesortie
   ```

   Pour conserver les identifiants d'objets (OID) (lorsqu'ils sont utilisés comme clé étrangère), utiliser l'option `-o` lors de l'exécution de pg_dumpall.

   Pour effectuer la sauvegarde, il est possible d'utiliser la commande pg_dumpall incluse dans la distribution de la version actuelle. Cependant, pour de meilleurs résultats, il est préférable d'utiliser la commande **pg_dumpall** contenue dans PostgreSQL™ 9.0.18 puisqu'elle corrige les erreurs des versions précédentes. Bien que ce conseil puisse paraître absurde, tant que la nouvelle version n'est pas installée, il convient de le suivre si l'objectif est d'installer la nouvelle version en parallèle de l'ancienne. Dans ce cas, il est possible de terminer normalement l'installation et de transférer les données dans un second temps, ce qui réduit l'indisponibilité.

2. Arrêter l'ancien serveur :

   ```
   pg_ctl stop
   ```

   Sur les systèmes qui lancent PostgreSQL™ au démarrage, il y a probablement un fichier de démarrage qui peut faire la même chose. Par exemple, sur un système `Red Hat Linux`, cette commande doit fonctionner :

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

3. En cas de restauration d'une sauvegarde, renommer ou supprimer l'ancien répertoire d'installation. Il est préférable de renommer le répertoire plutôt que de le supprimer, parce qu'en cas de problème, on peut avoir besoin d'y revenir. Le répertoire peut prendre beaucoup d'espace disque. Pour renommer le répertoire, utiliser une commande comme :

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

4. Installer la nouvelle version de PostgreSQL™ comme indiquée dans la prochaine section Section 15.5, « Procédure d'installation ».

5. Créer un nouveau cluster de bases de données. Ces commandes doivent être exécutées sous le compte superutilisateur initial (qui existe déjà dans le cas d'une mise à jour).

   ```
   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

6. Restaurer le `pg_hba.conf` précédent et toutes les modifications de `postgresql.conf`.

7. Lancer le serveur de bases de données, toujours à partir du compte superutilisateur :

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8. Enfin, restaurer les données à partir de la sauvegarde avec :

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

en utilisant le *nouveau* psql.

Des informations supplémentaires sont disponibles dans la documentation Section 24.4, « Migration entre versions ». Cette documentation contient notmamment des instructions sur la manière d'exécuter les deux versions en parallèle.

15.5. Procédure d'installation

1. **Configuration**

   La première étape de la procédure d'installation est de configurer l'arborescence système et de choisir les options intéressantes. Ce qui est fait en exécutant le script `configure`. Pour une installation par défaut, entrer simplement

   ```
   ./configure
   ```

   Ce script exécutera de nombreux tests afin de déterminer les valeurs de certaines variables dépendantes du système et de détecter certains aléas relatifs au système d'exploitation. Il créera divers fichiers dans l'arborescence de compilation pour enregistrer ce qui a été trouvé. `configure` peut aussi être exécuté à partir d'un répertoire hors de l'arborescence des sources pour conserver l'arborescence de compilation séparé. Cette procédure est aussi appelé une construction a *VPATH* build. Voici comment la faire :

   ```
   mkdir build_dir
   cd build_dir
   /path/to/source/tree/configure [les options vont ici]
   gmake
   ```

   La configuration par défaut compilera le serveur et les utilitaires, aussi bien que toutes les applications clientes et interfaces qui requièrent seulement un compilateur C. Tous les fichiers seront installés par défaut sous `/usr/local/pgsql`.

   Les processus de compilation et d'installation peuvent être personnalisés par l'utilisation d'une ou plusieurs options sur la ligne de commande après `configure` :

   `--prefix=`*`PREFIX`*
   Installe tous les fichiers dans le répertoire *`PREFIX`* au lieu du répertoire `/usr/local/pgsql`. Les fichiers actuels seront installés dans divers sous-répertoires ; aucun fichier ne sera directement installé sous *`PREFIX`*.

   Pour satisfaire des besoins spécifiques, les sous-répertoires peuvent être personnalisés à l'aide des options qui suivent. Toutefois, en laissant les options par défaut, l'installation est déplaçable, ce qui signifie que le réperoire peut être déplacé après installation. (Cela n'affecte pas les emplacements de `man` et `doc`.)

   Pour les installations déplaçables, on peut utiliser l'option `--disable-rpath` de `configure`. De plus, il faut indiquer au système d'exploitation comment trouver les bibliothèques partagées.

   `--exec-prefix=`*`EXEC-PREFIX`*
   Les fichiers qui dépendent de l'architecture peuvent être installés dans un répertoire différent, *`EXEC-PREFIX`*, de celui donné par *`PREFIX`*. Ce qui peut être utile pour partager les fichiers dépendant de l'architecture entre plusieurs machines. S'il est omis, *`EXEC-PREFIX`* est égal à *`PREFIX`* et les fichiers dépendant seront installés sous la même arborescence que les fichiers indépendants de l'architecture, ce qui est probablement le but recherché.

   `--bindir=`*`REPERTOIRE`*
   Précise le répertoire des exécutables. Par défaut, il s'agit de *`EXEC-PREFIX`*`/bin`, ce qui signifie `/usr/local/pgsql/bin`.

   `--sysconfdir=`*`REPERTOIRE`*
   Précise le répertoire de divers fichiers de configuration. Par défaut, il s'agit de *`PREFIX`*`/etc`.

`--libdir=REPERTOIRE`
Précise le répertoire d'installation des bibliothèques et des modules chargeables dynamiquement. Par défaut, il s'agit de *EXEC-PREFIX*/`lib`.

`--includedir=REPERTOIRE`
Précise le répertoire d'installation des en-têtes C et C++. Par défaut, il s'agit de *PREFIX*/`include`.

`--datarootdir=REPERTOIRE`
Indique le répertoire racine de différents types de fichiers de données en lecture seule. Cela ne sert qu'à paramétrer des valeurs par défaut pour certaines des options suivantes. La valeur par défaut est *PREFIX*/`share`.

`--datadir=REPERTOIRE`
Indique le répertoire pour les fichiers de données en lecture seule utilisés par les programmes installés. La valeur par défaut est *DATAROOTDIR*. Cela n'a aucun rapport avec l'endroit où les fichiers de base de données seront placés.

`--localedir=REPERTOIRE`
Indique le répertoire pour installer les données locales, en particulier les fichiers catalogues de traductions de messages. La valeur par défaut est *DATAROOTDIR*/`locale`.

`--mandir=REPERTOIRE`
Les pages man fournies avec PostgreSQL™ seront installées sous ce répertoire, dans leur sous-répertoire `man`*x* respectif. Par défaut, il s'agit de *DATAROOTDIR*/`man`.

`--docdir=RÉPERTOIRE`
Configure le répertoire racine pour installer les fichiers de documentation, sauf les pages « man ». Ceci ne positionne la valeur par défaut que pour les options suivantes. La valeur par défaut pour cette option est *DATAROOTDIR*/`doc/postgresql`.

`--htmldir=RÉPERTOIRE`
La documentation formatée en HTML pour PostgreSQL™ sera installée dans ce répertoire. La valeur par défaut est *DATAROOTDIR*.

Note

Une attention toute particulière a été prise afin de rendre possible l'installation de PostgreSQL™ dans des répertoires partagés (comme `/usr/local/include`) sans interférer avec des noms de fichiers relatifs au reste du système. En premier lieu, le mot « `/postgresql` » est automatiquement ajouté aux répertoires `datadir`, `sysconfdir` et `docdir`, à moins que le nom du répertoire à partir de la racine contienne déjà le mot « `postgres` » où « `pgsql` ». Par exemple, si `/usr/local` est choisi comme préfixe, la documentation sera installée dans `/usr/local/doc/postgresql`, mais si le préfixe est `/opt/postgres`, alors il sera dans `/opt/postgres/doc`. Les fichiers d'en-têtes publiques C de l'interface cliente seront installés sous `includedir` et sont indépendants des noms de fichiers relatifs au reste du système. Les fichiers d'en-têtes privés et les fichiers d'en-têtes du serveur sont installés dans des répertoires privés sous `includedir`. Voir la documentation de chaque interface pour savoir comment obtenir ces fichiers d'en-tête. Enfin, un répertoire privé sera aussi créé si nécessaire sous `libdir` pour les modules chargeables dynamiquement.

`--with-includes=REPERTOIRES`
REPERTOIRES est une liste de répertoires séparés par des caractères deux points (:) qui sera ajoutée à la liste de recherche des fichiers d'en-tête. Si vous avez des paquetages optionnels (tels que Readline GNU) installés dans des répertoires non conventionnels, vous pouvez utiliser cette option et certainement l'option `--with-libraries` correspondante.

Exemple : `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=REPERTOIRES`
REPERTOIRES est une liste de recherche de répertoires de bibliothèques séparés par des caractères deux points (:). Si des paquets sont installés dans des répertoires non conventionnels, il peut s'avérer nécessaire d'utiliser cette option (et l'option correspondante `--with-includes`).

Exemple : `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-nls[=LANGUES]`
Permet de mettre en place le support des langues natives (NLS). C'est la possibilité d'afficher les messages des programmes dans une langue autre que l'anglais. *LANGUES* est une liste, optionnelle, des codes des langues que vous voulez supporter séparés par un espace. Par exemple, `--enable-nls='de fr'` (l'intersection entre la liste et l'ensemble des langues tra-

duites actuellement sera calculée automatiquement). En l'absence de liste, toutes les traductions disponibles seront installées.

Pour utiliser cette option, une implantation de l'API Gettext est nécessaire ; voir après.

`--with-pgport=NUMERO`
Positionne *NUMERO* comme numéro de port par défaut pour le serveur et les clients. La valeur par défaut est 5432. Le port peut toujours être changé ultérieurement mais, précisé ici, les exécutables du serveur et des clients auront la même valeur par défaut, ce qui est vraiment très pratique. Habituellement, la seule bonne raison de choisir une valeur autre que celle par défaut est l'exécution de plusieurs serveurs PostgreSQL™ sur la même machine.

`--with-perl`
Permet l'utilisation du langage de procédures PL/Perl côté serveur.

`--with-python`
Permet la compilation du langage de procédures PL/Python.

`--with-tcl`
Permet la compilation du langage de procédures PL/Tcl.

`--with-tclconfig=REPERTOIRE`
Tcl installe les fichiers `tclConfig.sh`, contenant certaines informations de configuration nécessaires pour compiler le module d'interfaçage avec Tcl. Ce fichier est trouvé automatiquement mais, si pour utiliser une version différente de Tcl, il faut indiquer le répertoire où le trouver.

`--with-gssapi`
Construire avec le support de l'authentification GSSAPI. Sur de nombreux systèmes, GSSAPI (qui fait habituellement partie d'une installation Kerberos) n'est pas installé dans un emplacement recherché par défaut (c'est-à-dire `/usr/include`, `/usr/lib`), donc vous devez utiliser les options `--with-includes` et `--with-libraries` en plus de cette option. `configure` vérifiera les fichiers d'en-têtes nécessaires et les bibliothèques pour s'assurer que votre installation GSSAPI est suffisante avant de continuer.

`--with-krb5`
Compile le support d'authentification de Kerberos 5. Sur beaucoup de systèmes, le système Kerberos n'est pas installé à un emplacement recherché par défaut (c'est-à-dire `/usr/include`, `/usr/lib`), donc vous devez utiliser les options `--with-includes` et `--with-libraries` en plus de cette option. `configure` vérifiera les fichiers d'en-tête et les bibliothèques requis pour s'assurer que votre installation Kerberos est suffisante avant de continuer

`--with-krb-srvnam=NOM`
Le nom par défaut du service principal de Kerberos (aussi utilisé par GSSAPI). `postgres` est pris par défaut. Il n'y a habituellement pas de raison de le changer sauf dans le cas d'un environnement Windows, auquel cas il doit être mis en majuscule, `POSTGRES`.

`--with-openssl`

Compile le support de connexion SSL (chiffrement). Le paquetage OpenSSL™ doit être installé. `configure` vérifiera que les fichiers d'en-tête et les bibliothèques soient installés pour s'assurer que votre installation d'OpenSSL™ est suffisante avant de continuer.

`--with-pam`
Compile le support PAM (*Modules d'Authentification Pluggable*).

`--with-ldap`
Demande l'ajout du support de LDAP pour l'authentification et la recherche des paramètres de connexion (voir la documentation sur l'authentification des clients et libpqSection 31.16, « Recherches LDAP des paramètres de connexion » et Section 19.3.7, « Authentification LDAP »). Sur Unix, cela requiert l'installation du paquet OpenLDAP™. Sur Windows, la bibliothèque WinLDAP™ est utilisée par défaut. `configure` vérifiera l'existence des fichiers d'en-tête et des bibliothèques requis pour s'assurer que votre installation d'OpenLDAP™ est suffisante avant de continuer.

`--without-readline`
Évite l'utilisation de la bibliothèque Readline (et de celle de libedit). Cela désactive l'édition de la ligne de commande et l'historique dans psql, ce n'est donc pas recommandé.

`--with-libedit-preferred`
Favorise l'utilisation de la bibliothèque libedit (sous licence BSD) plutôt que Readline (GPL). Cette option a seulement un sens si vous avez installé les deux bibliothèques ; dans ce cas, par défaut, Readline est utilisé.

`--with-bonjour`
Compile le support de Bonjour. Ceci requiert le support de Bonjour dans votre système d'exploitation. Recommandé sur Mac OS X.

`--with-ossp-uuid`
Utilise la *bibliothèque OSSP UUID* lors de la construction du module `contrib/uuid-ossp`. Cette bibliothèque fournit des fonctions pour générer les UUID.

`--with-libxml`
Construit avec libxml (active le support SQL/XML). Une version 2.6.23 ou ultérieure de libxml est requise pour cette fonctionnalité.

Libxml installe un programme **xml2-config** qui est utilisé pour détecter les options du compilateur et de l'éditeur de liens. PostgreSQL l'utilisera automatiquement si elle est trouvée. Pour indiquer une installation de libxml dans un emplacement inhabituel, vous pouvez soit configurer la variable d'environnement `XML2_CONFIG` pour pointer vers le programme **xml2-config** appartenant à l'installation, ou utiliser les options `--with-includes` et `--with-libraries`.

`--with-libxslt`
Utilise libxslt pour construire `contrib/xml2`. Le module `contrib/xml2` se base sur cette bibliothèque pour réaliser les transformations XSL du XML.

`--disable-integer-datetimes`
Désactive le support pour le stockage des intervalles et horodatages en entier 64 bits, et stocke les valeurs de type date/temps en temps que nombre à virgule flottante à la place. Le stockage à virgule flottante des dates/temps était la valeur par défaut dans les versions de PostgreSQL™ antérieures à la 8.4, mais est maintenant obsolète parce qu'il ne permet pas une précision à la microseconde pour toute l'étendue des valeurs timestamp. Toutefois, le stockage des dates/temps à base d'entiers nécessite un type entier de 64 bits. Par conséquent, cette option peut être utilisée quand ce type de données n'est pas disponible, ou pour maintenir la compatibilité avec des applications écrites pour des versions antérieures de PostgreSQL™. Voir la documentation à propos des types dates/temps Section 8.5, « Types date/heure » pour plus d'informations.

`--disable-float4-byval`
Désactive le passage « par valeur » des valeurs float4, entraînant leur passage « par référence » à la place. Cette option a un coût en performance, mais peut être nécessaire pour maintenir la compatibilité avec des anciennes fonctions créées par l'utilisateur qui sont écrites en C et utilisent la convention d'appel « version 0 ». Une meilleure solution à long terme est de mettre à jour toutes ces fonctions pour utiliser la convention d'appel « version 1 ».

`--disable-float8-byval`
Désactive le passage « par valeur » des valeurs float8, entraînant leur passage « par référence » à la place. Cette option a un coût en performance, mais peut être nécessaire pour maintenir la compatibilité avec des anciennes fonctions créées par l'utilisateur qui sont écrites en C et utilisent la convention d'appel « version 0 ». Une meilleure solution à long terme est de mettre à jour toutes ces fonctions pour utiliser la convention d'appel « version 1 ». Notez que cette option n'affecte pas que float8, mais aussi int8 et quelques types apparentés comme timestamp. Sur les plateformes 32 bits, `--disable-float8-byval` est la valeur par défaut, et il n'est pas permis de sélectionner `--enable-float8-byval`.

`--with-segsize=`*`TAILLESEG`*
Indique la *taille d'un segment*, en gigaoctets. Les grandes tables sont divisées en plusieurs fichiers du système d'exploitation, chacun de taille égale à la taille de segment. Cela évite les problèmes avec les limites de tailles de fichiers qui existent sur de nombreuses plateformes. Si votre système d'exploitation supporte les fichiers de grande taille (« largefile »), ce qui est le cas de la plupart d'entre eux de nos jours, vous pouvez utiliser une plus grande taille de segment. Cela peut être utile pour réduire le nombre de descripteurs de fichiers qui peuvent être utilisés lors de travail sur des très grandes tables. Attention à ne pas sélectionner une valeur plus grande que ce qui est supporté par votre plateforme et le(s) système(s) de fichiers que vous prévoyez d'utiliser. D'autres outils que vous pourriez vouloir utiliser, tels que tar, pourraient aussi limiter la taille maximum utilisable pour un fichier. Il est recommandé, même si pas vraiment nécessaire, que cette valeur soit un multiple de 2. Notez que changer cette valeur impose de faire un initdb.

`--with-blocksize=`*`TAILLEBLOC`*
Indique la *taille d'un bloc*, en kilooctets. C'est l'unité de stockage et d'entrée/sortie dans les tables. La valeur par défaut, 8 kilooctets, est appropriée pour la plupart des cas ; mais d'autres valeurs peuvent être utilises dans des cas spéciaux. Cette valeur doit être une puissance de 2 entre 1 et 32 (kilooctets). Notez que changer cette valeur impose de faire un initdb.

`--with-wal-segsize=`*`TAILLESEG`*
Indique la *taille d'un segment WAL*, en mégaoctets. C'est la taille de chaque fichier individuel dans les journaux de transactions. Il peut être utile d'ajuster cette taille pour contrôler la granularité du transport de journaux de transations. La valeur par défaut est de 16 mégaoctets. La valeur doit être une puissance de 2 entre 1 et 6 (mégaoctets). Notez que changer cette valeur impose de faire un initdb.

`--with-wal-blocksize=`*`TAILLEBLOC`*
Indique la *taille d'un bloc WAL*, en kilooctets. C'est l'unité de stockage et d'entrée/sortie dans le journal des transactions. La valeur par défaut, 8 kilooctets, est appropriée pour la plupart des cas ; mais d'autres valeurs peuvent être utilises dans des cas

spéciaux. La valeur doit être une puissance de 2 entre 1 et 64 (kilooctets).

`--disable-spinlocks`
Autorise le succès de la construction y compris lorsque PostgreSQL™ n'a pas le support spinlock du CPU pour la plateforme. Ce manque de support résultera en des performances faibles ; du coup, cette option devra seulement être utilisée si la construction échoue et vous informe du manque de support de spinlock sur votre plateforme. Si cette option est requise pour construire PostgreSQL™ sur votre plateforme, merci de rapporter le problème aux développeurs de PostgreSQL™.

`--disable-thread-safety`
Désactive la sûreté des threads pour les bibliothèques clients. Ceci empêche les threads concurrents dans les programmes libpq et ECPG de contrôler avec sûreté leur pointeurs de connexion privés.

`--with-system-tzdata=`*`RÉPERTOIRE`*

PostgreSQL™ inclut sa propre base de données des fuseaux horaires, nécessaire pour les opérations sur les dates et les heures. Cette base de données est en fait compatible avec la base de fuseaux horaires « zoneinfo » fournie par de nombreux systèmes d'exploitation comme FreeBSD, Linux et Solaris, donc ce serait redondant de l'installer une nouvelle fois. Quand cette option est utilisée, la base des fuseaux horaires, fournie par le système, dans *`RÉPERTOIRE`* est utilisée à la place de celle inclus dans la distribution des sources de PostgreSQL. *`RÉPERTOIRE`* doit être indiqué avec un chemin absolu. `/usr/share/zoneinfo` est un répertoire très probable sur certains systèmes d'exploitation. Notez que la routine d'installation ne détectera pas les données de fuseau horaire différentes ou erronées. Si vous utilisez cette option, il vous est conseillé de lancer les tests de régression pour vérifier que les données de fuseau horaire que vous pointez fonctionnent correctement avec PostgreSQL™.

Cette option a pour cible les distributeurs de paquets binaires qui connaissent leur système d'exploitation. Le principal avantage d'utiliser cette option est que le package PostgreSQL n'aura pas besoin d'être mis à jour à chaque fois que les règles des fuseaux horaires changent. Un autre avantage est que PostgreSQL peut être cross-compilé plus simplement si les fichiers des fuseaux horaires n'ont pas besoin d'être construit lors de l'installation.

`--without-zlib`
Évite l'utilisation de la bibliothèque Zlib. Cela désactive le support des archives compressées dans pg_dump et pg_restore. Cette option est seulement là pour les rares systèmes qui ne disposent pas de cette bibliothèque.

`--enable-debug`
Compile tous les programmes et bibliothèques en mode de débogage. Cela signifie que vous pouvez exécuter les programmes via un débogueur pour analyser les problèmes. Cela grossit considérablement la taille des exécutables et, avec des compilateurs autres que GCC, habituellement, cela désactive les optimisations du compilateur, provoquant des ralentissements. Cependant, mettre ce mode en place est extrêmement utile pour repérer les problèmes. Actuellement, cette option est recommandée pour les installations en production seulement si vous utilisez GCC. Néanmoins, vous devriez l'utiliser si vous développez ou si vous utilisez une version béta.

`--enable-coverage`
Si vous utilisez GCC, les programmes et bibliothèques sont compilés avec de l'instrumentation de test de couverture de code. Quand ils sont exécutés, ils génèrent des fichiers dans le répertoire de compilation avec des métriques de couverture de code. Voir Section 30.4, « Examen de la couverture du test » pour davantage d'informations. Cette option ne doit être utilisée qu'avec GCC et uniquement en phase de développement.

`--enable-profiling`
En cas d'utilisation de GCC, tous les programmes et bibliothèques sont compilés pour qu'elles puissent être profilées. À la sortie du processus serveur, un sous-répertoire sera créé pour contenir le fichier `gmon.out` à utiliser pour le profilage. Cette option est à utiliser seulement avec GCC lors d'un développement.

`--enable-cassert`
Permet la vérification des *assertions* par le serveur qui teste de nombreux cas de conditions « impossibles ». Ce qui est inestimable dans le cas de développement, mais les tests peuvent ralentir sensiblement le système. Activer cette option n'influe pas sur la stabilité de votre serveur ! Les assertions vérifiées ne sont pas classées par ordre de sévérité et il se peut qu'un bogue anodin fasse redémarrer le serveur s'il y a un échec de vérification. Cette option n'est pas recommandée dans un environnement de production mais vous devriez l'utiliser lors de développement ou pour les versions béta.

`--enable-depend`
Active la recherche automatique des dépendances. Avec cette option, les fichiers makefile sont appelés pour recompiler les fichiers objet dès qu'un fichier d'en-tête est modifié. C'est pratique si vous faites du développement, mais inutile si vous ne voulez que compiler une fois et installer. Pour le moment, cette option ne fonctionne qu'avec GCC.

`--enable-dtrace`
Compile PostgreSQL™ avec le support de l'outil de trace dynamique, DTrace. Voir Section 27.4, « Traces dynamiques » pour plus d'informations.

Pour pointer vers le programme **dtrace**, la variable d'environnement `DTRACE` doit être configurée. Ceci sera souvent nécessaire car **dtrace** est typiquement installé sous `/usr/sbin`, qui pourrait ne pas être dans le chemin.

Des options supplémentaires en ligne de commande peuvent être indiquées dans la variable d'environnement `DTRACEFLAGS` pour le programme **dtrace**. Sur Solaris, pour inclure le support de DTrace dans un exécutable 64-bit, ajoutez l'option `DTRACEFLAGS="-64"` pour configure. Par exemple, en utilisant le compilateur GCC :

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

En utilisant le compilateur de Sun :

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace
DTRACEFLAGS='-64' ...
```

Si vous préférez utiliser un compilateur C différent de ceux listés par `configure`, positionnez la variable d'environnement `CC` pour qu'elle pointe sur le compilateur de votre choix. Par défaut, `configure` pointe sur `gcc` s'il est disponible, sinon il utilise celui par défaut de la plateforme (habituellement `cc`). De façon similaire, vous pouvez repositionner les options par défaut du compilateur à l'aide de la variable `CFLAGS`.

Les variables d'environnement peuvent être indiquées sur la ligne de commande `configure`, par exemple :

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

Voici une liste des variables importantes qui sont configurables de cete façon :

`BISON`
programme Bison

`CC`
compilateur C

`CFLAGS`
options à passer au compilateur C

`CPP`
préprocesseur C

`CPPFLAGS`
options à passer au préprocesseur C

`DTRACE`
emplacement du programme **dtrace**

`DTRACEFLAGS`
options à passer au programme **dtrace**

`FLEX`
programme Flex

`LDFLAGS`
options à utiliser lors de l'édition des liens des exécutables et des bibliothèques partagées

`LDFLAGS_EX`
options supplémentaires valables uniquement lors de l'édition des liens des exécutables

`LDFLAGS_SL`
options supplémentaires valables uniquement lors de l'édition des liens des bibliothèques partagées

`MSGFMT`
programme **msgfmt** pour le support des langues

`PERL`
chemin complet vers l'interpréteur Perl. Il sera utilisé pour déterminer les dépendances pour la construction de PL/Perl.

`PYTHON`
chemin complet vers l'interpréteur Python. Il sera utilisé pour déterminer les dépendances pour la construction de PL/Python. De plus, si Python 2 ou 3 est spécifié ici (ou implicitement choisi), il détermine la variante de PL/Python qui devient disponible. Voir la documentation PL/Python Section 42.1, « Python 2 et Python 3 » pour plus d'informations.

`TCLSH`
chemin complet vers l'interpréteur Tcl. Il sera utilisé pour déterminer les dépendances pour la construction de PL/Tcl, et il sera substitué dans des scripts Tcl.

`XML2_CONFIG`
programme **xml2-config** utilisé pour localiser l'installation de libxml.

2. **Compilation**

   Pour démarrer la compilation, saisissez

   ```
   gmake
   ```

   (Rappelez-vous d'utiliser GNU make). La compilation prendra quelques minutes, suivant de votre matériel. La dernière ligne affichée devrait être

   ```
   All of PostgreSQL is successfully made. Ready to install.
   ```

   Si vous voulez construire tout ce qui peut être construit, ceci incluant la documentation (HTML et pages man) et les modules supplémentaires (`contrib`), saisissez à la place :

   ```
   gmake world
   ```

   La dernière ligne affichée doit être :

   ```
   PostgreSQL, contrib and HTML documentation successfully made. Ready to install.
   ```

3. **Tests de régression**

   Si vous souhaitez tester le serveur nouvellement compilé avant de l'installer, vous pouvez exécuter les tests de régression à ce moment. Les tests de régression sont une suite de tests qui vérifient que PostgreSQL™ fonctionne sur votre machine tel que les développeurs l'espèrent. Saisissez

   ```
   gmake check
   ```

   (cela ne fonctionne pas en tant que root ; faites-le en tant qu'utilisateur sans droits). Le fichier `src/test/regress/README` et la documentation contiennentLe Chapitre 30, Tests de régression contient des détails sur l'interprétation des résultats de ces tests. Vous pouvez les répéter autant de fois que vous le voulez en utilisant la même commande.

4. **Installer les fichiers**

   > **Note**
   >
   > Si vous mettez à jour une version existante et que vous placez les fichiers au même endroit que les anciens, assurez-vous d'avoir sauvegardé vos données et arrêté l'ancien serveur avant de continuer, comme l'explique la Section 15.4, « Mise à jour » ci-après.

   Pour installer PostgreSQL™, saisissez

   ```
   gmake install
   ```

   Cela installera les fichiers dans les répertoires spécifiés dans l'Étape 1. Assurez-vous d'avoir les droits appropriés pour écrire dans ces répertoires. Normalement, vous avez besoin d'être superutilisateur pour cette étape. Une alternative consiste à créer les répertoires cibles à l'avance et à leur donner les droits appropriées.

   Pour installer la documentation (HTML et pages man), saisissez :

   ```
   gmake install-docs
   ```

   Si vous construisez tout, saisissez ceci à la place :

   ```
   gmake install-world
   ```

   Cela installe aussi la documentation.

Vous pouvez utiliser `gmake install-strip` en lieu et place de `gmake install` pour dépouiller l'installation des exécutables et des bibliothèques. Cela économise un peu d'espace disque. Si vous avez effectué la compilation en mode de débogage, ce dépouillage l'enlèvera, donc ce n'est à faire seulement si ce mode n'est plus nécessaire. `install-strip` essaie d'être raisonnable en sauvegardant de l'espace disque mais il n'a pas une connaissance parfaite de la façon de dépouiller un exécutable de tous les octets inutiles. Ainsi, si vous voulez sauvegarder le maximum d'espace disque, vous devrez faire le travail à la main.

L'installation standard fournit seulement les fichiers en-têtes nécessaires pour le développement d'applications clientes ainsi que pour le développement de programmes côté serveur comme des fonction personnelles ou des types de données écrits en C (avant PostgreSQL™ 8.0, une commande `gmake install-all-headers` séparée était nécessaire pour ce dernier point mais cette étape a été intégrée à l'installation standard).

Installation du client uniquement : Si vous voulez uniquement installer les applications clientes et les bibliothèques d'interface, alors vous pouvez utilisez ces commandes :

```
gmake -C src/bin install
gmake -C src/include install
gmake -C src/interfaces install
gmake -C doc install
```

`src/bin` comprend quelques exécutables utilisés seulement par le serveur mais ils sont petits.

Enregistrer eventlog sur `Windows` : Pour enregistrer une bibliothèque eventlog sur `Windows` grâce au système d'exploitation, exécutez cette commande après l'installation :

```
regsvr32 pgsql_library_directory/pgevent.dll
```

Ceci crée les entrées du registre utilisées par le visualisateur d'événements.

Désinstallation : Pour désinstaller, utilisez la commande **gmake uninstall**. Cependant, cela ne supprimera pas les répertoires créés.

Nettoyage : Après l'installation, vous pouvez libérer de l'espace en supprimant les fichiers issus de la compilation des répertoires sources à l'aide de la commande **gmake clean**. Cela conservera les fichiers créés par la commande **configure**, ainsi vous pourrez tout recompiler ultérieurement avec **gmake**. Pour remettre l'arborescence source dans l'état initial, utilisez **gmake distclean**. Si vous voulez effectuer la compilation pour diverses plateformes à partir des mêmes sources vous devrez d'abord refaire la configuration à chaque fois (autrement, utilisez un répertoire de construction séparé pour chaque plateforme, de façon à ce que le répertoire des sources reste inchangé).

Si vous avez compilé et que vous vous êtes rendu compte que les options de **configure** sont fausses ou si vous changez quoique ce soit que **configure** prenne en compte (par exemple, la mise à jour d'applications), alors faire un **gmake distclean** avant de reconfigurer et recompiler est une bonne idée. Sans ça, vos changements dans la configuration ne seront pas répercutés partout où il faut.

15.6. Initialisation post-installation

15.6.1. Bibliothèques partagées

Sur certains systèmes qui utilisent les bibliothèques partagées (ce que font de nombreux systèmes), vous avez besoin de leurs spécifier comment trouver les nouvelles bibliothèques partagées. Les systèmes sur lesquels ce *n'est* pas nécessaire comprennent `BSD/OS`, `FreeBSD`, `HP-UX`, `IRIX`, `Linux`, `NetBSD`, `OpenBSD`, `Tru64 UNIX` (auparavant `Digital UNIX`) et `Solaris`.

La méthode pour le faire varie selon la plateforme, mais la méthode la plus répandue consiste à positionner des variables d'environnement comme `LD_LIBRARY_PATH` : avec les shells Bourne (**sh**, **ksh**, **bash**, **zsh**) :

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

ou en **csh** ou **tcsh** :

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Remplacez `/usr/local/pgsql/lib` par la valeur donnée à `--libdir` dans l'Étape 1. Vous pouvez mettre ces commandes dans un script de démarrage tel que `/etc/profile` ou `~/.bash_profile`. Certaines informations pertinentes au sujet de mises en garde associées à cette méthode peuvent être trouvées sur *http://xahlee.org/UnixResource_dir/_/ldpath.html*.

Sur certains systèmes, il peut être préférable de renseigner la variable d'environnement `LD_RUN_PATH` *avant* la compilation.

Avec `Cygwin`, placez le répertoire des bibliothèques dans la variable `PATH` ou déplacez les fichiers `.dll` dans le répertoire `bin`.

En cas de doute, référez-vous aux pages de man de votre système (peut-être **ld.so** ou **rld**). Si vous avez ultérieurement un message tel que

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

alors cette étape est vraiment nécessaire. Faites-y attention.

Si votre système d'exploitation est `BSD/OS`, `Linux` ou `SunOS 4` et que vous avez les accès de superutilisateur, vous pouvez exécuter :

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(ou le répertoire équivalent) après l'installation pour permettre à l'éditeur de liens de trouver les bibliothèques partagées plus rapidement. Référez-vous aux pages man portant sur **ldconfig** pour plus d'informations. Pour les systèmes d'exploitation `FreeBSD`, `NetBSD` et `OpenBSD`, la commande est :

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

Les autres systèmes d'exploitation ne sont pas connus pour avoir de commande équivalente.

15.6.2. Variables d'environnement

Si l'installation a été réalisée dans `/usr/local/pgsql` ou à un autre endroit qui n'est pas dans les répertoires contenant les exécutables par défaut, vous devez ajouter `/usr/local/pgsql/bin` (ou le répertoire fourni à `--bindir` au moment de l'Étape 1) dans votre `PATH`. Techniquement, ce n'est pas une obligation mais cela rendra l'utilisation de PostgreSQL™ plus confortable.

Pour ce faire, ajoutez ce qui suit dans le fichier d'initialisation de votre shell, par exemple `~/.bash_profile` (ou `/etc/profile`, si vous voulez que tous les utilisateurs l'aient) :

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Si vous utilisez le **csh** ou le **tcsh**, alors utilisez la commande :

```
set path = ( /usr/local/pgsql/bin $path )
```

Pour que votre système trouve la documentation man, il vous faut ajouter des lignes telles que celles qui suivent à votre fichier d'initialisation du shell, à moins que vous installiez ces pages dans un répertoire où elles sont mises normalement :

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

Les variables d'environnement `PGHOST` et `PGPORT` indiquent aux applications clientes l'hôte et le port du serveur de base. Elles surchargent les valeurs utilisées lors de la compilation. Si vous exécutez des applications clientes à distance, alors c'est plus pratique si tous les utilisateurs peuvent paramétrer `PGHOST`. Ce n'est pas une obligation, cependant, la configuration peut être communiquée via les options de lignes de commande à la plupart des programmes clients.

15.7. Démarrer

La suite est un résumé rapide de la façon de faire fonctionner PostgreSQL™ une fois l'installation terminée. La documentation principale contient plus d'informations.

1. Créer un compte utilisateur pour le serveur PostgreSQL™. C'est cet utilisateur qui fera démarrer le serveur. Pour un usage en production, vous devez créer un compte sans droits (« postgres » est habituellement utilisé). Si vous n'avez pas les accès superutilisateur ou si vous voulez juste regarder, votre propre compte utilisateur est suffisant. Mais, utiliser le compte superutilisateur pour démarrer le serveur est risqué (au point de vue sécurité) et ne fonctionnera pas.

   ```
   adduser postgres
   ```

2. Faire l'installation de la base de données avec la commande **initdb**. Pour exécuter **initdb**, vous devez être connecté sur votre serveur avec le compte PostgreSQL™. Cela ne fonctionnera pas avec le compte superutilisateur.

   ```
   root# mkdir /usr/local/pgsql/data
   root# chown postgres /usr/local/pgsql/data
   root# su - postgres
   ```

```
postgres$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

L'option `-D` spécifie le répertoire où les données seront stockées. Vous pouvez utiliser le chemin que vous voulez, il n'a pas à être sous le répertoire de l'installation. Avant de lancer **initdb**, assurez-vous que le compte serveur peut écrire dans ce répertoire (ou le créer s'il n'existe pas), comme c'est montré ici.

3. À ce moment, si vous n'utilisez pas l'option `-A` de **initdb**, vous devez modifier le fichier `pg_hba.conf` pour contrôler les accès en local du serveur avant de le lancer. La valeur par défaut est de faire confiance à tous les utilisateurs locaux.
4. L'étape **initdb** précédente vous a indiqué comment démarrer le serveur de base. Maintenant, faites-le. La commande doit ressembler à :

   ```
   /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
   ```

   Cela démarrera le serveur en avant-plan. Pour le mettre en arrière plan faites quelque chose comme :

   ```
   nohup /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data \
       </dev/null >>server.log 2>&1 </dev/null &
   ```

   Pour arrêter le serveur fonctionnant en arrière-plan, vous pouvez saisir :

   ```
   kill `cat /usr/local/pgsql/data/postmaster.pid`
   ```

5. Créer une base de données :

   ```
   createdb testdb
   ```

   Ensuite, entrez

   ```
   psql testdb
   ```

   pour vous connecter à la base. À l'invite, vous pouvez saisir des commandes SQL et commencer l'expérimentation.

15.8. Et maintenant ?

- La distribution de PostgreSQL™ comprend un document compréhensible que vous devriez lire de temps en temps. Après l'installation, le document peut être lu en faisant pointer votre navigateur internet sur `/usr/local/pgsql/doc/html/index.html`, excepté si vous avez changé les répertoires d'installation.

  Le premier chapitre de la documentation est un tutoriel qui devrait être votre première lecture si vous êtes nouveau dans le monde des bases de données SQL. Si vous êtes familier avec les concepts des bases de données, alors vous voudrez commencer avec la partie administration du serveur qui contient des informations sur la façon de mettre en place le serveur de base, les bases des utilisateurs et l'authentification.
- Normalement, vous voudrez faire en sorte que le serveur de base démarre automatiquement au boot de la machine. Pour ce faire, quelques suggestions se trouvent dans la documentation.
- Faire les tests de régression sur le serveur installé (en utilisant **gmake installcheck**). Si vous ne l'avez pas fait auparavant, vous devriez définitivement le faire maintenant. C'est aussi expliqué dans la documentation.
- Par défaut, PostgreSQL™ est configuré pour fonctionner sur un matériel mimimal. Ceci lui permet de fonctionner sur pratiquement toutes les configurations matérielles. Néanmoins, la configuration par défaut n'est pas conçue pour des performances optimales. Pour disposer des meilleures performances, plusieurs paramètres serveurs doivent être ajustés, les deux plus communs étant `shared_buffers` et `work_mem`. Les autres paramètres mentionnés dans la documentation affectent aussi les performances.

15.9. Plateformes supportées

Une platforme (c'est-à-dire une combinaison d'un processeur et d'un système d'exploitation) est considérée supportée par la communauté de développeur de PostgreSQL™ si le code permet le fonctionnement sur cette plateforme et que la construction et les tests de regréssion ont été récemment vérifiés sur cette plateforme. Actuellement, la plupart des tests de compatibilité de plateforme se fait automatiquement par des machines de tests dans la *ferme de construction de PostgreSQL*. Si vous êtes intéressés par l'utilisation de PostgreSQL™ sur une plateforme qui n'est pas représentée dans la ferme de construction, mais pour laquelle le code fonctionne ou peut fonctionner, nous vous suggérons fortement de construction une machine qui sera membre de la ferme pour que la compatibilité puisse être assurée dans la durée.

En général, PostgreSQL™ doit fonctionner sur les architectures processeur suivantes : x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, Alpha, ARM, MIPS, MIPSEL, M68K et PA-RISC. Un support du code existe pour M32R, NS32K et VAX mais ces architectures n'ont pas été testées récemment à notre connaissance. Il est souvent possible de construire PostgreSQL™ sur un type de processeur non supporté en précisant `--disable-spinlocks`. Cependant, les performance en souffriront.

PostgreSQL™ doit fonctionner sur les systèmes d'exploitation suivants : Linux (toutes les distributions récentes), Windows (Win2000 SP4 et ultérieure), FreeBSD, OpenBSD, NetBSD, Mac OS X, AIX, HP/UX, IRIX, Solaris, Tru64 Unix et UnixWare. D'autres systèmes style Unix peuvent aussi fonctionner mais n'ont pas été récemment testés. Dans la plupart des cas, toutes les architectures processeurs supportées par un système d'exploitation donné fonctionneront. Cherchez dans le répertoire Section 15.10, « Notes spécifiques à des plateformes » ci-dessous pour voir s'il y a des informations spécifiques à votre système d'exploitation, tout particulièrement dans le cas d'un vieux système.

Si vous avez des problèmes d'installation sur une plateforme qui est connue comme étant supportée d'après les récents résultats de la ferme de construction, merci de rapporter cette information à `<pgsql-bugs@postgresql.org>`. Si vous êtes intéressé pour porter PostgreSQL™ sur une nouvelle plateforme, `<pgsql-hackers@postgresql.org>` est l'endroit approprié pour en discuter.

15.10. Notes spécifiques à des plateformes

Cette section documente des problèmes spécifiques additionnels liés à des plateformes, en ce qui concerne l'installation et le paramétrage de PostgreSQL. Assurez-vous de lire aussi les instructions d'installation, et en particulier Section 15.2, « Prérequis ». Par ailleurs, consultez le fichier `src/test/regress/README` et la documentation Chapitre 30, Tests de régression à propos de l'interprétation des tests de régression.

Les plateformes qui ne sont pas traitées ici n'ont pas de problèmes d'installation spécifiques connus.

15.10.1. AIX

PostgreSQL fonctionne sur AIX, mais réussir à l'installer correctement peut s'avérer difficile. Les versions AIX de la 4.3.3 à la 6.1 sont considérées comme supportées en théorie. Vous pouvez utiliser GCC ou le compilateur natif IBM xlc. En général, utiliser des versions récentes d'AIX et PostgreSQL rend la tâche plus simple. Vérifiez la ferme de compilation pour avoir des informations à jour sur les versions d'AIX connues pour être compatibles.

Les niveaux minimums recommandés de correctifs pour les versions supportées de `AIX` sont :

AIX 4.3.3
: Maintenance Level 11 + post ML11 bundle

AIX 5.1
: Maintenance Level 9 + post ML9 bundle

AIX 5.2
: Technology Level 10 Service Pack 3

AIX 5.3
: Technology Level 7

AIX 6.1
: Base Level

Pour vérifier votre niveau de correctif, utilisez **oslevel -r** de AIX 4.3.3 à AIX 5.2 ML 7, et **oslevel -s** pour les versions ultérieures.

Utilisez les options de **configure** en plus de vos propres options si vous avez installé Readline ou libz dans `/usr/local` : `--with-includes=/usr/local/include --with-libraries=/usr/local/lib`.

15.10.1.1. Problèmes avec GCC

Sur AIX 5.3, il y a des problèmes pour compiler et faire fonctionner PostgreSQL avec GCC.

Vous voudrez utiliser une version de GCC supérieure à 3.3.2, en particulier si vous utilisez une version pré-packagée. Nous avons eu de bons résultats avec la 4.0.1. Les problèmes avec les versions précédentes semblent être davantages liées à la façon dont IBM a packagé GCC qu'à des problèmes réels, avec GCC, ce qui fait que si vous compilez GCC vous même, vous pourriez réussir avec une version plus ancienne de GCC.

15.10.1.2. Sockets du domaine Unix inutilisables

Dans AIX 5.3, la structure sockaddr_storage n'est pas définie avec une taille suffisante. En version 5.3, IBM a augmenté la taille de sockaddr_un, la structure d'adresse pour une socket de domaine Unix, mais n'a pas augmenté en conséquence la taille de sockaddr_storage. La conséquence est que les tentatives d'utiliser une socket de domaine Unix avec PostgreSQL amènent libpq à dépasser la taille de la structure de données. Les connexions TCP/IP fonctionnent, mais pas les sockets de domaine Unix, ce qui empêche les tests de régression de fonctionner.

Le problème a été rapporté à IBM, et est enregistré en tant que rapport de bogue PMR29657. Si vous mettez à jour vers le niveau de maintenance 5300-03 et ultérieur, le correctif sera inclus. Une résolution immédiate est de corriger _SS_MAXSIZE à 1025 dans `/usr/include/sys/socket.h`. Dans tous les cas, recompilez PostgreSQL une fois que vous avez l'en-tête corrigé.

15.10.1.3. Problèmes avec les adresses internet

PostgreSQL se base sur la fonction système `getaddrinfo` pour analyser les adresses IP dans `listen_addresses` et dans `pg_hba.conf`, etc. Les anciennes versions d'AIX ont quelques bogues dans cette fonction. Si vous avez des problèmes relatifs à ces paramètres, la mise à jour vers le niveau de correctif AIX approprié indiqué ci-dessus pourrait se charger de cela.

Un utilisateur a rapporté :

Lors de l'implémentation de PostgreSQL version 8.1 sur AIX 5.3, nous tombions périodiquement sur des problèmes quand le collecteur de statistiques ne voulait « mystérieusement » pas démarrer. Cela se trouvait être le résultat d'un comportement inattendu dans l'implémentation d'IPv6. Il semble que PostgreSQL et IPv6 ne fonctionnent pas bien ensemble sur AIX 5.3.

Chacune des actions suivantes « corrigent » le problème.

- Supprimer l'adresse IPv6 pour localhost :

  ```
  (as root)
  # ifconfig lo0 inet6 ::1/0 delete
  ```

- Supprimer IPv6 des services réseau. Le fichier `/etc/netsvc.conf` sur AIX est en gros équivalent à `/etc/nsswitch.conf` sur Solaris/Linux. La valeur par défaut, sur AIX, est donc :

  ```
  hosts=local,bind
  ```

  Remplacez ceci avec :

  ```
  hosts=local4,bind4
  ```

  pour désactiver la recherche des adresses IPv6.

> **Avertissement**
>
> Ceci est en réalité un contournement des problèmes relatifs à l'immaturité du support IPv6, qui a amélioré la visibilité pour les versions 5.3 d'AIX. Cela a fonctionné avec les versions 5.3 d'AIX mais n'est fait pas pour autant une solution élégante à ce problème. Certaines personnes ont indiqué que ce contournement est non seulement inutile, mais pose aussi des problèmes sur AIX 6.1, où le support IPv6 est beaucoup plus mature.

15.10.1.4. Gestion de la mémoire

AIX est particulier dans la façon dont il gère la mémoire. Vous pouvez avoir un serveur avec des gigaoctets de mémoire libre, mais malgré tout avoir des erreurs de mémoire insuffisante ou des erreurs d'espace d'adressage quand vous lancez des applications. Un exemple est **createlang** qui échoue avec des erreurs inhabituelles. Par exemple, en exécutant en tant que propriétaire de l'installation PostgreSQL :

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library
"/opt/dbs/pgsql748/lib/plperl.so": A memory address is not in the address space for the
process.
```

En l'exécutant en tant que non-propriétaire, mais dans le groupe propriétaire de l'installation PostgreSQL :

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library
"/opt/dbs/pgsql748/lib/plperl.so": Bad address
```

On a un autre exemple avec les erreurs 'out of memory' dans les traces du serveur PostgreSQL, avec toute allocation de mémoire proche ou supérieure 256 Mo qui échoue.

La cause générale de ces problèmes est le nombre de bits et le modèle mémoire utilisé par le processus serveur. Par défaut, tous les binaires compilés sur AIX sont 32-bits. Cela ne dépend pas du matériel ou du noyau en cours d'utilisation. Ces processus 32-bits sont limités à 4 Go de mémoire présentée en segments de 256 Mo utilisant un parmi quelques modèles. Le modèle par défaut permet moins de 256 Mo dans le tas, comme il partage un seul segment avec la pile.

Dans le cas de l'exemple **createlang** ci-dessus, vérifiez votre umask et les droits des binaires de l'installation PostgreSQL. Les binaires de l'exemple étaient 32-bits et installés en mode 750 au lieu de 755. En raison des droits, seul le propriétaire ou un membre du groupe propriétaire peut charger la bibliothèque. Puisqu'il n'est pas lisible par tout le mode, le chargeur place l'objet dans le tas du processus au lieu d'un segment de mémoire de bibliothèque où il aurait été sinon placé.

La solution « idéale » pour ceci est d'utiliser une version 64-bits de PostgreSQL, mais ce n'est pas toujours pratique, parce que les systèmes équipés de processeurs 32-bits peuvent compiler mais ne peuvent pas exécuter de binaires 64-bits.

Si un binaire 32-bits est souhaité, positionnez LDR_CNTRL à `MAXDATA=0x`*n*`0000000`, où 1<=n <= 8 avant de démarrer le serveur PostgreSQL, et essayez différentes valeurs et paramètres de `postgresql.conf` pour trouver une configuration qui fonctionne de façon satisfaisante. Cette utilisation de LDR_CNTRL notifie à AIX que vous voulez que le serveur réserve MAXDATA octets pour le tas, alloués en segments de 256 Mo. Quand vous avez trouvé une configuration utilisable, **ldedit** peut être utilisé pour modifier les binaires pour qu'ils utilisent par défaut la taille de tas désirée. PostgreSQL peut aussi être recompilé, en passant à `configure LDFLAGS="-Wl,-bmaxdata:0x`*n*`0000000"` pour obtenir le même résultat.

Pour une compilation 64-bits, positionnez `OBJECT_MODE` à 64 et passez `CC="gcc -maix64"` et `LDFLAGS="-Wl,-bbigtoc"` à **configure**. (Les options pour **xlc** pourraient différer.) Si vous omettez les exports de `OBJECT_MODE`, votre compilation échouera avec des erreurs de l'éditeur de liens. Quand `OBJECT_MODE` est positionné, il indique aux outils de compilation d'AIX comme **ar**, **as** et **ld** quel types de fichiers manipuler par défaut.

Par défaut, de la surallocation d'espace de pagination peut se produire. Bien que nous ne l'ayons jamais constaté, AIX tuera des processus quand il se trouvera à court de mémoire et que la zone surallouée est accédée. Le comportement le plus proche de ceci que nous ayons constaté est l'échec de fork parce que le système avait décidé qu'il n'y avait plus de suffisamment de mémoire disponible pour un nouveau processus. Comme beaucoup d'autres parties d'AIX, la méthode d'allocation de l'espace de pagination et le « out-of-memory kill » sont configurables soit pour le système soit pour un processus, si cela devient un problème.

Références et ressources

« *Large Program Support* ». *AIX Documentation: General Programming Concepts: Writing and Debugging Programs*.

« *Program Address Space Overview* ». *AIX Documentation: General Programming Concepts: Writing and Debugging Programs*.

« *Performance Overview of the Virtual Memory Manager (VMM)* ». *AIX Documentation: Performance Management Guide*.

« *Page Space Allocation* ». *AIX Documentation: Performance Management Guide*.

« *Paging-space thresholds tuning* ». *AIX Documentation: Performance Management Guide*.

Developing and Porting C and C++ Applications on AIX. IBM Redbook.

15.10.2. Cygwin

PostgreSQL peut être construit avec Cygwin, un environnement similaire à Linux pour Windows, mais cette méthode est inférieure à la version native Windows (voir Chapitre 16, Installation à partir du code source sur Windows™) et n'est plus recommandée.

Quand vous compilez à partir des sources, suivant la procédure normale d'installation (c'est-à-dire `./configure; make`; etc...), notez les différences suivantes spécifiques à Cygwin :

- Positionnez le path pour utiliser le répertoire binaire Cygwin avant celui des utilitaires Windows. Cela permettra d'éviter des problèmes avec la compilation.
- La commande make GNU est appelée **make**, pas **gmake**.
- La commande **adduser** n'est pas supportée ; utilisez les outils appropriés de gestion d'utilisateurs sous Windows NT, 2000 ou XP. Sinon, sautez cette étape.
- La commande **su** n'est pas supportée ; utilisez ssh pour simuler la commande **su** sous Windows NT, 2000 ou XP. Sinon, sautez cette étape.

- OpenSSL n'est pas supporté.
- Démarrez **cygserver** pour le support de la mémoire partagée. Pour cela, entrez la commande `/usr/sbin/cygserver &`. Ce programme doit fonctionner à chaque fois que vous démarrez le serveur PostgreSQL ou que vous initialisez un cluster de bases de données (**initdb**). La configuration par défaut de **cygserver** pourrait nécessiter des changements (par exemple, augmenter SEMMNS) pour éviter à PostgreSQL d'échouer en raison d'un manque de ressources système.
- Les tests de régression en parallèle (`make check`) peuvent générer des échecs de tests de régression aléatoires en raison d'un dépassement de capacité de la file d'attente de `listen()` qui cause des erreurs de connexion refusée ou des blocages. Vous pouvez limiter le nombre de connexion en utilisant la variable de make `MAX_CONNECTIONS` comme ceci :

  ```
  make MAX_CONNECTIONS=5 check
  ```

  (Sur certains systèmes, vous pouvez avoir jusqu'à 10 connexions simultanées).

Il est possible d'installer **cygserver** et le serveur PostgreSQL en tant que services Windows NT. Pour plus d'informations sur comment le faire, veuillez vous référer au document `README` inclus avec le package binaire PostgreSQL sur Cygwin. Il est installé dans le répertoire `/usr/share/doc/Cygwin`.

15.10.3. HP-UX

PostgreSQL 7.3 et plus devraient fonctionner sur les machines PA-RISC Séries 700/800 sous HP-UX 10.X ou 11.X, si les correctifs appropriés sur le système et les outils de compilation sont bien appliqués. Au moins un développeur teste de façon régulière sur HP-UX 10.20, et nous avons des rapports d'installations réussies sur HP-UX 11.00 et 11.11.

En plus de la distribution source de PostgreSQL, vous aurez besoin de GNU make (le make HP ne fonctionnera pas) et soit GCC soit le compilateur ANSI HP. Si vous avez l'intention de compiler à partir des sources Git plutôt que d'une distribution tar, vous aurez aussi besoin de Flex (les GNU) et Bison (yacc GNU). Nous vous recommandons aussi de vous assurer que vous êtes assez à jour sur les correctifs HP. Au minimum, si vous compilez des binaires 64 bits sur HP-UX 11.11, vous aurez probablement besoin de PHSS_30966 (11.11) ou d'un correctif supérieur, sinon **initdb** pourrait bloquer :

```
PHSS_30966  s700_800 ld(1) and linker tools cumulative patch
```

De façon générale, vous devriez être à jour sur les correctifs libc et ld/dld, ainsi que sur les correctifs du compilateur si vous utilisez le compilateur C de HP. Voir les sites de support HP comme *http://itrc.hp.com* et *ftp://us-ffs.external.hp.com/* pour télécharger gratuitement leurs derniers correctifs.

Si vous compilez sur une machine PA-RISC 2.0 et que vous voulez avoir des binaires 64 bits en utilisant GCC, vous devez utiliser la version 64 bits de GCC. Des binaires GCC pour HP-UX PA-RISC et Itanium sont disponibles sur *http://www.hp.com/go/gcc*. N'oubliez pas de récupérer et d'installer les binutils en même temps.

Si vous compilez sur une machine PA-RISC 2.0 et que vous voulez que les binaires compilés fonctionnent sur une machine PA-RISC 1.1, vous devez spécifier `+DAportable` comme `CFLAGS`.

Si vous compilez sur une machine HP-UX Itanium, vous aurez besoin du dernier compilateur C ANSI HP avec les correctifs qui en dépendent :

```
PHSS_30848  s700_800 HP C Compiler (A.05.57)
PHSS_30849  s700_800 u2comp/be/plugin library Patch
```

Si vous avez à la fois le compilateur C HP et celui de GCC, vous voudrez peut être spécifier explicitement le compilateur à utiliser quand vous exécuterez **configure** :

```
./configure CC=cc
```

pour le compilateur HP, ou

```
./configure CC=gcc
```

pour GCC. Si vous omettez ce paramètre, configure choisira **gcc** s'il en a la possibilité.

Le répertoire par défaut d'installation est `/usr/local/pgsql`, que vous voudrez peut être remplacer par quelque chose dans `/opt`. Si c'est le cas, utilisez l'option `--prefix` de **configure**.

Dans les tests de régression, il pourrait y avoir des différences dans les chiffres les moins significatifs des tests de géométrie, qui varient suivant les versions de compilateur et de bibliothèque mathématique utilisées. Toute autre erreur est suspecte.

15.10.4. IRIX

PostgreSQL a été rapporté comme fonctionnant correctement sur les processeurs MIPS r8000, r10000 (à la fois ip25 et ip27) et r120000 (ip35), sur IRIX 6.5.5m, 6.5.12, 6.5.13, et 6.5.26 avec les compilateurs MIPSPro de versions 7.30, 7.3.1.2m, 7.3, et 7.4.4m.

Vous aurez besoin du compilateur C ANSI MIPSPro. Il y a des problèmes à la compilation avec GCC. C'est dû à un bogue GCC connu (non corrigé en version 3.0), lié à l'utilisation de fonctions qui retournent certains types de structures. Ce bogue affecte des fonctions telles que `inet_ntoa`, `inet_lnaof`, `inet_netof`, `inet_makeaddr`, et `semctl`. Il semblerait qu'on puisse résoudre le problème en forçant les fonctions à l'éditeur de liens avec libgcc, mais ceci n'a pas encore été testé.

Il est connu que la version 7.4.1m du compilateur MIPSPro génère du code incorrect. Le symptôme est « invalid primary checkpoint record » quand on tente de démarrer la base. La version 7.4.4m est OK ; le statut des versions intermédiaires est inconnu.

Il pourrait y avoir un problème de compilation comme celui-ci :

```
cc-1020 cc: ERROR File = pqcomm.c, Line = 427
  The identifier "TCP_NODELAY" is undefined.

                if (setsockopt(port->sock, IPPROTO_TCP, TCP_NODELAY,
```

Certaines versions incluent les définitions TCP dans `sys/xti.h`, il est alors nécessaire d'ajouter `#include <sys/xti.h>` dans `src/backend/libpq/pqcomm.c` et dans `src/interfaces/libpq/fe-connect.c`. Si vous rencontrez ce problème, merci de nous le faire savoir, afin que nous puissions développer un correctif approprié.

Dans les tests de régression, il pourrait y avoir des différences dans les chiffres les moins significatifs des tests de géométrie, suivant le FPU que vous utilisez. Toute autre erreur est suspecte.

15.10.5. MinGW/Windows Natif

PostgreSQL pour Windows peut être compilé en utilisant MinGW, un environnement de compilation similaire à Unix pour les systèmes d'exploitation Microsoft, ou en utilisant la suite de compilation Visual C++™ de Microsoft. La variante de compilation MinGW utilise le système de compilation normal décrit dans ce chapitre ; la compilation via Visual C++ fonctionne de façon totalement différente et est décrite dans la documentationChapitre 16, Installation à partir du code source sur Windows™. C'est une compilation totalement native qui n'utilise aucun logiciel supplémentaire comme MinGW. Un installeur est disponible sur le serveur web officiel de PostgreSQL.

Le port natif Windows requiert un système Microsoft 200 ou ultérieurs, 32 bits ou 64 bits. Les systèmes plus anciens n'ont pas l'infrastructure nécessaire (mais Cygwin peut être utilisé pour ceux-ci). MinGW, le système de compilation similaire à Unix, et MSYS, une suite d'outils Unix nécessaires pour exécuter des scripts shell tels que **configure**, peuvent être téléchargés de *http://www.mingw.org/*. Aucun de ces outils n'est nécessaire pour exécuter les binaires générés ; ils ne sont nécessaires que pour créer les binaires.

Après que vous ayez tout installé, il est suggéré que vous lanciez psql dans **CMD.EXE**, car la console MSYS a des problèmes de tampons.

15.10.6. SCO OpenServer et SCO UnixWare

PostgreSQL peut être compilé sur SCO UnixWare 7 et SCO OpenServer 5. Sur OpenServer, vous pouvez utiliser soit l'OpenServer Development Kit soit l'Universal Development Kit. Toutefois, quelques ajustements peuvent être nécessaires, comme décrit ci-dessous.

15.10.6.1. Skunkware

Vous aurez besoin de votre copie du CD SCO Skunkware. Le CD Skunkware est inclus avec UnixWare 7 et les versions actuelles d'OpenServer 5. Skunkware inclut des versions prêtes à l'installation de nombreux programmes populaires qui sont disponibles sur Internet. Par exemple, sont inclus gzip, gunzip, GNU Make, Flex et Bison. Pour UnixWare 7.1, ce CD est maintenant appelé « Open License Software Supplement ». Si vous n'avez pas ce CD, les logiciels qu'il contient sont disponibles sur le serveur FTP anonyme *http://www.sco.com/skunkware/*.

Les versions de Skunkware sont différentes entre UnixWare et OpenServer. Faites attention à installer la version correcte pour votre système d'exploitation, sauf pour les cas notifiés ci-dessous.

Sous UnixWare 7.1.3 et supérieur, le compilateur GCC est inclus sur le CD UDK, ainsi que GNU Make.

15.10.6.2. GNU Make

Vous devez utiliser le programme GNU Make, qui est inclus sur le CD Skunkware. Par défaut, il s'installe en tant que `/`

`usr/local/bin/make`. Pour éviter la confusion avec le programme `make` SCO, vous pouvez renommer GNU `make` en `gmake`.

À partir d'UnixWare 7.1.3, le programme GNU Make est dans la portion OSTK du CD UDK, et est dans `/usr/gnu/bin/gmake`.

15.10.6.3. Readline

La bibliothèque Readline est disponible sur le CD Skunkware, mais pas sur le CD Skunkware d'UnixWare 7.1. Si vous avez UnixWare 7.0.0 ou 7.0.1, vous pouvez installer à partir du CD, sinon essayez *http://www.sco.com/skunkware/*.

Par défaut, Readline s'installe dans `/usr/local/lib` et `/usr/local/include`. Toutefois, le programme **configure** de PostgreSQL ne la trouvera pas là sans aide. Si vous avez installé Readline, alors utilisez les options suivantes avec **configure** :

```
./configure --with-libraries=/usr/local/lib --with-includes=/usr/local/include
```

15.10.6.4. Utilisation de l'UDK avec OpenServer

Si vous utilisez le nouveau compilateur Universal Development Kit (UDK) avec OpenServer, vous devez spécifier l'emplacement des bibliothèques UDK :

```
./configure --with-libraries=/udk/usr/lib --with-includes=/udk/usr/include
```

Ajouté aux options Readline précédentes, cela donne :

```
./configure --with-libraries="/udk/usr/lib /usr/local/lib"
--with-includes="/udk/usr/include /usr/local/include"
```

15.10.6.5. Lire les man pages de PostgreSQL

Par défaut, les man pages PostgreSQL sont installées dans `/usr/local/pgsql/man`. Par défaut, UnixWare ne recherche pas de man pages à cet endroit. Pour pouvoir les lire, vous devez modifier la variable `MANPATH` pour y inclure `/etc/default/man`, par exemple :

```
MANPATH=/usr/lib/scohelp/%L/man:/usr/dt/man:/usr/man:/usr/share/man:scohelp:/usr/local/man:/
```

Sur OpenServer, un effort supplémentaire devra être fait pour rendre les man pages utilisables, parce que le système man est un peu différent de celui des autres plateformes. À l'heure actuelle, PostgreSQL ne les installera pas du tout.

15.10.6.6. Problèmes C99 avec le 7.1.1b Feature Supplement

Pour les compilateurs antérieurs à celui fourni avec OpenUNIX 8.0.0 (UnixWare 7.1.2), dont celui du 7.1.1b Feature Supplement, vous pourriez avoir besoin de spécifier `-Xb` dans `CFLAGS` ou la variable d'environnement `CC`. Ce qui l'annonce est une erreur dans la compilation de `tuplesort.c`, sur les fonctions inline. Apparemment, il y a eu un changement dans le compilateur 7.1.2 (8.0.0) et les suivants.

15.10.6.7. Threads avec UnixWare

Pour les threads, vous *devez* utiliser `-Kpthread` sur *tous* les programmes utilisant libpq. libpq utilise des appels `pthread_*`, qui ne sont disponibles qu'avec l'option `-Kpthread`/`-Kthread`.

15.10.7. Solaris

PostgreSQL est bien supporté sous Solaris. Plus le système d'exploitation est à jour, moins de problèmes vous aurez ; les détails sont ci-dessous.

Notez que PostgreSQL est fourni avec Solaris 10 (à partir de l'update 2). Des packages officiels sont aussi disponibles sur *http://pgfoundry.org/projects/solarispackages/*. Des packages pour des versions plus anciennes de Solaris peuvent aussi être obtenus sur *http://www.sunfreeware.com/* ou *http://www.blastwave.org/*.

15.10.7.1. Outils requis

Vous pouvez compiler soit avec GCC, soit avec le compilateur de Sun. Pour une meilleure optimisation du code, le compilateur de Sun est fortement recommandé sur l'architecture SPARC. Il y a eu des problèmes rapportés à l'utilisation de GCC 2.95.1 ; des versions de gcc 2.95.3 ou supérieure sont recommandées. Si vous utilisez le compilateur de Sun, attention à ne pas sélectionner `/`

`usr/ucb/cc` ; utilisez `/opt/SUNWspro/bin/cc`.

Vous pouvez télécharger Sun Studio sur *http://developers.sun.com/sunstudio/downloads/*. De nombreux outils GNU sont intégrés dans Solaris 10, ou sont présents sur le Solaris companion CD. Si vous voulez des packages pour des plus anciennes versions de Solaris, vous pouvez trouver ces outils sur *http://www.sunfreeware.com* ou *http://www.blastwave.org*. Si vous préférez les sources, allez sur *http://www.gnu.org/order/ftp.html*.

15.10.7.2. Problèmes avec OpenSSL

Quand vous compilez PostgreSQL avec le support OpenSSL, vous pourriez rencontrer des erreurs de compilation dans les fichiers suivants :

- `src/backend/libpq/crypt.c`
- `src/backend/libpq/password.c`
- `src/interfaces/libpq/fe-auth.c`
- `src/interfaces/libpq/fe-connect.c`

C'est en raison d'un conflit d'espace de nom entre l'en-tête standard `/usr/include/crypt.h` et les fichiers d'en-tête fournis par OpenSSL.

La mise à jour de l'installation d'OpenSSL vers la version 0.9.6a résout ce problème. Solaris 9 et supérieurs ont une version plus récente d'OpenSSL.

15.10.7.3. configure se plaint d'un programme de test en échec

Si **configure** se plaint d'un programme de test en échec, c'est probablement un cas de l'éditeur de lien à l'exécution qui ne trouve pas une bibliothèque, probablement libz, libreadline ou une autre bibliothèque non standard telle que libssl. Pour l'amener au bon endroit, positionnez la variable d'environnement `LDFLAGS` sur la ligne de commande de **configure**, par exemple,

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

Voir la man page de ld(1) pour plus d'informations.

15.10.7.4. La compilation 64-bit plante parfois

Dans Solaris 7 et précédentes, la version 64 bits de la libc a une routine `vsnprintf` boguée, qui génère des « core dumps » aléatoires dans PostgreSQL. Le contournement le plus simple connu est de forcer PostgreSQL à utiliser sa propre version de `vsnprintf` plutôt que celle de la bibliothèque. Pour faire ceci, après avoir exécuté **configure**, éditez un des fichiers produits par **configure**. Dans `src/Makefile.global`, modifiez la ligne

```
LIBOBJS =
```

par

```
LIBOBJS = snprintf.o
```

(Il pourrait y avoir d'autres fichiers déjà listés dans cette variable. L'ordre est sans importance.) Puis compilez comme d'habitude.

15.10.7.5. Compiler pour des performances optimales

Sur l'architecture SPARC, Sun Studio est fortement recommandé pour la compilation. Essayez d'utiliser l'option d'optimisation `-xO5` pour générer des binaires sensiblement plus rapides. N'utilisez pas d'options qui modifient le comportement des opérations à virgule flottante et le traitement de `errno` (par exemple, `-fast`). Ces options pourraient amener des comportements PostgreSQL non standard, par exemple dans le calcul des dates/temps.

Si vous n'avez pas de raison d'utiliser des binaires 64 bits sur SPARC, préférez la version 32 bits. Les opérations et les binaires 64 bits sont plus lents que les variantes 32 bits. D'un autre côté, le code 32 bits sur un processeur de la famille AMD64 n'est pas natif, ce qui fait que le code 32 bits est significativement plus lent sur cette famille de processeurs.

Des astuces pour optimiser les performances de PostgreSQL sur Solaris peuvent être trouvées sur *http://www.sun.com/servers/coolthreads/tnb/applications_postgresql.jsp*. Cet article se focalise principalement sur la plateforme T2000, mais beaucoup des recommandations sont aussi utiles avec d'autres plateformes sous Solaris.

15.10.7.6. Utiliser DTrace pour tracer PostgreSQL

Oui, l'utilisation de DTrace est possible. Voir la documentation Section 27.4, « Traces dynamiques » pour davantage

d'informations. Vous pouvez aussi trouver plus d'informations dans cet article : *http://blogs.sun.com/robertlor/entry/user_level_dtrace_probes_in*.

Si vous voyez l'édition de liens de l'exécutable **postgres** échouer avec un message d'erreur similaire à :

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
gmake: *** [postgres] Error 1
```

l'installation DTrace est trop ancienne pour gérer les sondes dans les fonctions statiques. Solaris 10u4 ou plus récent est nécessaire.

Chapitre 16. Installation à partir du code source sur Windows™

Il est recommandé que la plupart des utilisateurs téléchargent la distribution binaire pour Windows, disponible sous la forme d'un package d'installation one-click™ à partir du site web de PostgreSQL™. Construire à partir des sources a pour seule cible les personnes qui développent PostgreSQL™ ou des extensions.

Il existe différentes façons de construire PostgreSQL sur Windows™. La façon la plus simple de le faire est d'utiliser les outils Microsoft. Pour cela, il faut installer une version supportée du Microsoft Platform SDK™ et d'utiliser le compilateur inclus. Il est aussi possible de construire PostgreSQL avec Microsoft Visual C++ 2005 ou 2008™. Dans certains cas, il faut installer le Platform SDK™ en plus du compilateur.

Il est aussi possible de construire PostgreSQL en utilisant les outils de compilation GNU fournis par MinGW™ ou en utilisant Cygwin™ pour les anciennes versions de Windows™.

Enfin, la bibliothèque d'accès pour les clients (libpq) peut être construit en utilisant Visual C++ 7.1™ ou Borland C++™ pour la compatibilité avec des applications liées statiquement en utilisant ces outils.

La construction par MinGW™ ou Cygwin™ utilise le système habituel de construction, voir Chapitre 15, Procédure d'installation de PostgreSQL™ du code source et les notes spécifiques dans Section 15.10.5, « MinGW/Windows Natif » et Section 15.10.2, « Cygwin ». Ces constructions ne peuvent pas générer des exécutables 64 bits. Cygwin™ n'est pas recommandé, et devrait seulement être utilisé pour les anciennes versions de Windows™ où la construction native ne fonctionne pas, comme par exemple Windows 98™. MinGW™ est seulement recommandé si vous construisez les autres modules avec. Les exécutables officiels sont construits avec Visual Studio™.

16.1. Construire avec Visual C++™ ou le Platform SDK™

PostgreSQL peut être construit en utilisant la suite de compilation Visual C++ de Microsoft. Ces compilateurs peuvent être soit Visual Studio™, soit Visual Studio Express™ soit certaines versions du Platform SDK™. Si vous n'avez pas déjà un environnement Visual Studio™ configuré, la façon la plus simple est d'utiliser les compilateurs du Platform SDK™, téléchargement libre chez Microsoft.

PostgreSQL supporte les compilateurs de Visual Studio 2005™ et Visual Studio 2008™. Lors de l'utilisation du Platform SDK seul ou lors de la construction pour Windows 64 bits, seul Visual Studio 2008™ est supporté. Visual Studio 2010™ n'est pas encore supporté.

Lors d'une compilation avec le Platform SDK™, les versions 6.0 à 7.0 du SDK sont supportées. Les versions plus anciennes ou plus récentes ne fonctionneront pas. En particulier, les versions 7.0a et ultérieures ne fonctionneront pas car elles incluent les compilateurs de Visual Studio 2010™.

Les outils pour construire en utilisant Visual C++™ se trouvent dans le répertoire `src/tools/msvc`. Lors de la construction, assurez-vous qu'il n'y a pas d'outils provenant de MinGW™ ou Cygwin™ dans le chemin (PATH) de votre environnement. Dans Visual Studio™, lancez l'invite de Visual Studio. Dans le Platform SDK™, lancez le CMD shell listé sous le répertoire SDL du menu de démarrage. Si vous souhaitez construire une version 64-bits, vous devez utiliser la version 64-bit de la commande, et vice versa. Toutes les commandes devraient être exécutées à partir du répertoire `src\tools\msvc`.

Avant de lancer la construction, vous aurez besoin d'éditer le fichier `config.pl` pour y modifier toutes les options de configuration nécessaires, ainsi que les chemins utilisés par les bibliothèques de tierces parties. La configuration complète est déterminée tout d'abord en lisant et en analysant le fichier `config_default.pl`, puis en appliquant les modifications provenant du fichier `config.pl`. Par exemple, pour indiquer l'emplacement de votre installation de Python™, placez la ligne suivante dans `config.pl` :

```
$config->{python} = 'c:\python26';
```

Vous avez seulement besoin de spécifier les paramètres qui sont différents de la configuration par défaut, spécifiée par le fichier `config_default.pl`.

Si vous avez besoin de configurer d'autres variables d'environnement, créez un fichier appelé `buildenv.pl` et placez-y les commandes souhaitées. Par exemple, pour ajouter le chemin vers bison s'il ne se trouve pas dans le PATH, créez un fichier contenant :

```
$ENV{PATH}=$ENV{PATH} . ';c:\chemin\vers\bison\bin';
```

16.1.1. Pré-requis

Les outils supplémentaires suivants sont requis pour construire PostgreSQL™. Utilisez le fichier `config.pl` pour indiquer les répertoires où se trouvent les bibliothèques.

Microsoft Platform SDK™
: Il est recommandé de mettre à jour avec la dernière version disponible du Microsoft Platform SDK™, téléchargeable sur *http://www.microsoft.com/downloads/*.

  Vous devez toujours inclure la partie Windows Headers and Libraries du SDK. Si vous installez le Platform SDK™ incluant les compilateurs (Visual C++ Compilers), vous n'avez pas de Visual Studio™.

ActiveState Perl™
: ActiveState Perl est requis pour exécuter les scripts de construction. Le Perl de MinGW et de Cygwin ne fonctionnera pas. Il doit aussi être présent dans le PATH. Les binaires de cet outil sont téléchargeables à partir de *http://www.activestate.com* (Note : la version 5.8 au minimum est requise, la distribution standard libre est suffisante).

Les produits suivants ne sont pas nécessaires pour commencer, mais sont requis pour installer la distribution complète. Utilisez le fichier `config.pl` pour indiquer les répertoires où sont placées les bibliothèques.

ActiveState TCL™
: Requis pour construire PL/TCL (Note : la version 8.4 est requise, la distribution standard libre est suffisante).

Bison™ et Flex™
: Bison et Flex sont requis pour construire à partir d'une extraction du Git, mais ils ne sont pas nécessaires si vous utilisez une version packagée. Notez que seul Bison 1.875 ou les versions 2.2 et ultérieures fonctionneront. De plus, Flex version 2.5.31 ou une version ultérieure est requise. Bison est téléchargeable à partir de *http://gnuwin32.sourceforge.net*. Flex est disponible sur *http://www.postgresql.org/ftp/misc/winflex/*.

Note

La distribution Bison provenant de GnuWin32 contient un bug qui pose problème lorsque Bison est installé dans un répertoire dont le nom contient des espaces (ce qui est le cas par défaut sur les installations anglaises, `C:\Program Files\GnuWin32`). Pensez plutôt à l'installer dans `C:\GnuWin32` pour éviter ce bug.

Diff™
: Diff est nécessaire pour exécuter les tests de régression, et peut être téléchargé à partir de *http://gnuwin32.sourceforge.net*.

Gettext™
: Gettext est requis pour construire le support NLS, et peut être téléchargé à partir de *http://gnuwin32.sourceforge.net*. Notez que les binaires, dépendances et fichiers développeurs sont tous nécessaires.

MIT Kerberos™
: Requis pour le support de l'authentification Kerberos. MIT Kerberos est téléchargeable sur *http://web.mit.edu/Kerberos/dist/index.html*.

libxml2™ et libxslt™
: Requis pour le support du XML. Les binaires sont disponibles sur *http://zlatkovic.com/pub/libxml* et les sources sur *http://xmlsoft.org*. Notez que libxml2 nécessite iconv, qui est disponible sur le même site web.

openssl™
: Requis pour le support de SSL. Les binaires peuvent être téléchargés à partir de *http://www.slproweb.com/products/Win32OpenSSL.html* alors que les sources sont disponibles sur *http://www.openssl.org*.

ossp-uuid™
: Requis pour le support d'UUID-OSSP (seulement en contrib). Les sources peuvent être récupérées sur le *site ossp.org*.

Python™
: Requis pour la construction de PL/Python. Les binaires sont téléchargeables sur *http://www.python.org*.

zlib™
: Requis pour le support de la compression dans pg_dump et pg_restore. Les binaires sont disponibles à partir de *http://www.zlib.net*.

16.1.2. Considérations spéciales pour Windows 64-bits

PostgreSQL ne peut être compilé pour l'architecture x64 que sur Windows 64-bits, il n'y a pas de support pour les processeurs Itanium.

Mixer des versions 32-bits et des versions 64-bits dans le même répertoire de construction n'est pas supporté. Le système de compilation détectera automatiquement si l'environnement est 32-bits ou 64-bits, et construira PostgreSQL en accord. Pour cette raison, il est important de commencer avec la bonne invite de commande avant de lancer la compilation.

Pour utiliser une bibliothèque de tierce partie côté serveur comme python™ ou openssl™, cette bibliothèque *doit* aussi être en 64-bits. Il n'y a pas de support pour le chargement d'une bibliothèque 32-bits sur un serveur 64-bits. Plusieurs bibliothèques de tierce partie que PostgreSQL supporte ne sont disponibles qu'en version 32-bits, auquel cas elles ne peuvent pas être utilisées avec un PostgreSQL 64-bits.

16.1.3. Construction

Pour construire tout PostgreSQL dans la configuration par défaut, exécutez la commande :

```
build
```

Pour construire tout PostgreSQL dans la configuration de débogage, exécutez la commande :

```
build DEBUG
```

Pour construire un seul projet, par exemple psql, exécutez les commandes :

```
build psql

build DEBUG psql
```

Pour modifier la configuration de construction par défaut, placez ce qui suit dans le fichier `buildenv.pl` :

```

```

Il est aussi possible de construire à partir de l'interface de Visual Studio. Dans ce cas, vous devez exécuter :

```
perl mkvcbuild.pl
```

à partir de l'invite, puis ouvrir le fichier `pgsql.sln` généré (dans le répertoire racine des sources) dans Visual Studio.

16.1.4. Nettoyage et installation

La plupart du temps, la récupération automatique des dépendances dans Visual Studio prendra en charge les fichiers modifiés. Mais, s'il y a eu trop de modifications, vous pouvez avoir besoin de nettoyer l'installation. Pour cela, exécutez simplement la commande `clean.bat`, qui nettoiera automatiquement les fichiers générés. Vous pouvez aussi l'exécuter avec le paramètre *`dist`*, auquel cas il se comporte comme **`make distclean`** et supprime les fichiers flex/bison en sortie.

Par défaut, tous les fichiers sont écrits dans un sous-répertoire de `debug` ou `release`. Pour installer ces fichiers en utilisant les emplacements standards et pour générer aussi les fichiers requis pour initialiser et utiliser la base de données, exécutez la commande :

```
install c:\destination\directory
```

16.1.5. Exécuter les tests de régression

Pour exécuter les tests de régression, assurez-vous que vous avez terminé la construction de toutes les parties requises. Ensuite, assurez-vous que les DLL nécessaires au chargement de toutes les parties du système (comme les DLL Perl et Python pour les langages de procédure) sont présentes dans le chemin système. Dans le cas contraire, configurez-les dans le fichier `buildenv.pl`. Pour lancer les tests, exécutez une des commandes suivantes à partir du répertoire `src\tools\msvc` :

```
vcregress check

vcregress installcheck

vcregress plcheck

vcregress contribcheck
```

Pour modifier la planification utilisée (en parallèle par défaut), ajoutez-la à la ligne de commande, comme :

```
vcregress check serial
```

Pour plus d'informations sur les tests de régression, voir Chapitre 30, Tests de régression.

16.1.6. Construire la documentation

Construire la documentation PostgreSQL au format HTML nécessite plusieurs outils et fichiers. Créez un répertoire racine pour tous ces fichiers et stockez-les dans des sous-répertoires conformément à la liste ci-dessous.

OpenJade 1.3.1-2
: À télécharger à partir de *http://sourceforge.net/projects/openjade/files/openjade/1.3.1/openjade-1_3_1-2-bin.zip/download* et à décompresser dans le sous-répertoire `openjade-1.3.1`.

DocBook DTD 4.2
: À télécharger à partir de *http://www.oasis-open.org/docbook/sgml/4.2/docbook-4.2.zip* et à décompresser dans le sous-répertoire `docbook`.

DocBook DSSSL 1.79
: À télécharger à partir de *http://sourceforge.net/projects/docbook/files/docbook-dsssl/1.79/docbook-dsssl-1.79.zip/download* et à décompresser dans le sous-répertoire `docbook-dsssl-1.79`.

ISO character entities
: À télécharger à partir de *http://www.oasis-open.org/cover/ISOEnts.zip* et à décompresser dans le sous-répertoire `docbook`.

Modifiez le fichier `buildenv.pl` et ajoutez une variable pour l'emplacement du répertoire racine, par exemple :

```
$ENV{DOCROOT}='c:\docbook';
```

Pour construire la documentation, exécutez la commande `builddoc.bat`. Notez que ceci exécutera la construction une deuxième fois, pour générer les index. Les fichiers HTML générés seront dans le répertoire `doc\src\sgml`.

16.2. Construire libpq avec Visual C++™ ou Borland C++™

Utiliser Visual C++ 7.1-9.0™ ou Borland C++™ pour construire libpq est seulement recommandé si vous avez besoin d'une version contenant des drapeaux débogage/version finale, ou si vous avez besoin d'une bibliothèque statique que vous lierez à une application. Pour une utilisation normale, MinGW™ et Visual Studio 2005™ or Platform SDK™ method sont recommandés.

Pour construire la bibliothèque client libpq en utilisant Visual Studio 7.1™ (ou ultérieur), allez dans le répertoire `src` et exécutez

la commande :

```
nmake /f win32.mak
```

Pour construire une version 64-bit de la bibliothèque client libpq en utilisant Visual Studio 8.0™ (ou ultérieur), allez dans le répertoire `src` et exécutez la commande :

```
nmake /f win32.mak CPU=AMD64
```

Voir le fichier `win32.mak` pour plus de détails sur les variables supportées.

Pour construire la bibliothèque client libpq en utilisant Borland C++™, allez dans le répertoire `src` et exécutez la commande :

```
make -N -DCFG=Release /f bcc32.mak
```

16.2.1. Fichiers générés

Les fichiers suivants seront produits :

`interfaces\libpq\Release\libpq.dll`
: la bibliothèque client ;

`interfaces\libpq\Release\libpqdll.lib`
: la bibliothèque d'import nécessaire à l'édition de liens avec `libpq.dll`

`interfaces\libpq\Release\libpq.lib`
: la version statique de la bibliothèque d'interface client ;

Habituellement, vous n'avez pas besoin d'installer les fichiers client. Vous devez placer le fichier `libpq.dll` dans le même répertoire que vos applications. N'installez pas `libpq.dll` dans votre répertoire `Windows`, `System` or `System32`, sauf en cas d'absolue nécessité. S'il est installé par un programme, ce dernier doit en contrôler au préalable la ressource VERSIONINFO afin d'éviter l'écrasement d'une version plus récente.

Si l'on prévoit de développer sur cette machine une application qui utilise libpq, il faut ajouter les sous-répertoires `src\include` et `src\interfaces\libpq` dans le chemin d'inclusion des sources de votre compilateur.

Pour utiliser la bibliothèque, il faut ajouter `libpqdll.lib` au projet (sous Visual C++, clic droit sur le projet et choisir ajouter).

Chapitre 17. Configuration du serveur et mise en place

Ce chapitre discute de la configuration, du lancement du serveur de bases de données et de ses interactions avec le système d'exploitation.

17.1. Compte utilisateur PostgreSQL™

Comme avec tout démon serveur accessible au monde externe, il est conseillé de lancer PostgreSQL™ sous un compte utilisateur séparé. Ce compte devrait seulement être le propriétaire des données gérées par le serveur et ne devrait pas être partagé avec d'autres démons (par exemple, utiliser l'utilisateur `nobody` est une mauvaise idée). Il n'est pas conseillé de changer le propriétaire des exécutables par cet utilisateur car les systèmes compromis pourraient alors se voir modifier leur propres binaires.

Pour ajouter un compte utilisateur Unix, jetez un œil à la commande **useradd** ou **adduser** de votre système. Le nom de l'utilisateur `postgres` est souvent utilisé et l'est sur tout le livre, mais vous pouvez utiliser un autre nom si vous le souhaitez.

17.2. Créer un groupe de base de données

Avant de faire quoi que ce soit, vous devez initialiser un emplacement de stockage pour la base de données. Nous appelons ceci un *groupe de bases de données* (sql utilise le terme de groupe de catalogues). Un groupe de bases de données est une collection de bases données et est géré par une seule instance d'un serveur de bases de données en cours d'exécution. Après initialisation, un groupe de bases de données contiendra une base de données nommée `postgres`, qui a pour but d'être la base de données par défaut utilisée par les outils, les utilisateurs et les applications tiers. Le serveur de la base de données lui-même ne requiert pas la présence de la base de données `postgres` mais beaucoup d'outils supposent son existence. Une autre base de données est créée à l'intérieur de chaque groupe lors de l'initialisation. Elle est appelée `template1`. Comme le nom le suggère, elle sera utilisée comme modèle pour les bases de données créées après ; elle ne devrait pas être utilisée pour un vrai travail (voir le Chapitre 21, Administration des bases de données pour des informations sur la création de nouvelles bases de données dans le groupe).

En terme de système de fichiers, un groupe de bases de données sera un simple répertoire sous lequel les données seront stockées. Nous l'appelons le *répertoire de données* ou l'*emplacement des données*. Le choix de cet emplacement vous appartient complètement. Il n'existe pas de valeur par défaut bien que les emplacements tels que `/usr/local/pgsql/data` ou `/var/lib/pgsql/data` sont populaires. Pour initialiser un groupe de bases de données, utilisez la commande initdb(1), installée avec PostgreSQL™. L'emplacement désiré sur le groupe de fichier est indiqué par l'option `-d`, par exemple

```
$ initdb -D /usr/local/pgsql/data
```

Notez que vous devez exécuter cette commande en étant connecté sous le compte de l'utilisateur PostgreSQL™ décrit dans la section précédente.

> **Astuce**
>
> Comme alternative à l'option `-d`, vous pouvez initialiser la variable d'environnement `pgdata`.

Autrement, vous pouvez exécuter **initdb** via le programme pg_ctl(1) ainsi :

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

C'est peut-être plus intuitif si vous utilisez déjà **pg_ctl** pour démarrer et arrêter le serveur (voir Section 17.3, « Lancer le serveur de bases de données » pour les détails). Un gros intérêt est de ne connaître que cette seule commande pour gérer l'instance du serveur de bases de données.

initdb tentera de créer le répertoire que vous avez spécifié si celui-ci n'existe pas déjà. Il est possible qu'il n'ait pas le droit de le faire (si vous avez suivi notre conseil et créé un compte sans droits). Dans ce cas, vous devez créer le répertoire vous-même (en tant que root) et modifier le propriétaire pour qu'il corresponde à l'utilisateur PostgreSQL™. Voici comment réaliser ceci :

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

initdb refusera de s'exécuter si le répertoire des données semble être déjà initialisé.

Comme le répertoire des données contient toutes les données stockées par le système de bases de données, il est essentiel qu'il soit sécurisé par rapport à des accès non autorisés. Du coup, **initdb** supprimera les droits d'accès à tout le monde sauf l'utilisateur PostgreSQL™.

Néanmoins, bien que le contenu du répertoire soit sécurisé, la configuration d'authentification du client par défaut permet à tout utilisateur local de se connecter à la base de données et même à devenir le super-utilisateur de la base de données. Si vous ne faites pas confiance aux utilisateurs locaux, nous vous recommandons d'utiliser une des options `-w` ou `--pwprompt` de la commande **initdb** pour affecter un mot de passe au super-utilisateur de la base de données . De plus, spécifiez `-a md5` ou `-a mot_de_passe` de façon à ce que la méthode d'authentification `trust` par défaut ne soit pas utilisée ; ou modifiez le fichier `pg_hba.conf` généré après l'exécution d'**initdb** (d'autres approches raisonnables incluent l'utilisation de l'authentification `ident` ou les droits du système de fichiers pour restreindre les connexions. Voir le Chapitre 19, Authentification du client pour plus d'informations).

initdb initialise aussi la locale par défaut du groupe de bases de données. Normalement, elle prends seulement le paramétrage local dans l'environnement et l'applique à la base de données initialisée. Il est possible de spécifier une locale différente pour la base de données ; la Section 22.1, « Support des locales » propose plus d'informations là-dessus. L'ordre de tri utilisé par défaut pour ce cluster de bases de données est initialisé par **initdb** et, bien que vous pouvez créer de nouvelles bases de données en utilisant des ordres de tris différents, l'ordre utilisé dans les bases de données modèle que initdb a créé ne peut pas être modifié sans les supprimer et les re-créer. Cela a aussi un impact sur les performances pour l'utilisation de locales autres que `c` ou `posix`. Du coup, il est important de faire ce choix correctement la première fois.

initdb configure aussi le codage par défaut de l'ensemble de caractères pour le groupe de bases de données. Normalement, cela doit être choisi pour correspondre au paramétrage de la locale. Pour les détails, voir la Section 22.2, « Support des jeux de caractères ».

17.2.1. Systèmes de fichiers réseaux

Beaucoup d'installations créent les clusters de bases de données sur des systèmes de fichiers réseau. Parfois, cela utilise directement par NFS. Cela peut aussi passer par un NAS (acronyme de *Network Attached Storage*), périphérique qui utilise NFS en interne. PostgreSQL™ ne fait rien de particulier avec les systèmes de fichiers NFS, ceci signifiant que PostgreSQL™ suppose que NFS se comporte exactement comme les lecteurs connectés en local (DAS, acronyme de *Direct Attached Storage*). Si les implémentations du client et du serveur NFS ont une sémantique non standard, cela peut poser des problèmes de fiabilité (voir *http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html*). En particulier, des écritures asynchrones (décalées dans le temps) sur le serveur NFS peuvent poser des soucis de fiabilité. Si possible, montez les systèmes de fichiers NFS en synchrone (autrement dit sans cache) pour éviter cela. De même, le montage NFS n'est pas recommandé. Les SAN utilisent un protocole de communication bas-niveau plutôt que NFS.

17.3. Lancer le serveur de bases de données

Avant qu'une personne ait accès à la base de données, vous devez démarrer le serveur de bases de données. Le programme serveur est appelé **postgres**. Le programme **postgres** doit savoir où trouver les données qu'il est supposé utiliser. Ceci se fait avec l'option `-d`. Du coup, la façon la plus simple de lancer le serveur est :

```
$ postgres -D /usr/local/pgsql/data
```

qui laissera le serveur s'exécuter en avant plan. Pour cela, vous devez être connecté en utilisant le compte de l'utilisateur PostgreSQL™. Sans l'option `-d`, le serveur essaiera d'utiliser le répertoire de données nommé par la variable d'environnement `pgdata`. Si cette variable ne le fournit pas non plus, le lancement échouera.

Habituellement, il est préférable de lancer **postgres** en tâche de fond. Pour cela, utilisez la syntaxe shell Unix habituelle :

```
$ postgres -D /usr/local/pgsql/data >journaux_trace 2>&1 &
```

Il est important de sauvegarder les sorties `stdout` et `stderr` du serveur quelque part, comme montré ci-dessus. Cela vous aidera dans des buts d'audits ou pour diagnostiquer des problèmes (voir la Section 23.3, « Maintenance du fichier de traces » pour une discussion plus détaillée de la gestion de journaux de trace).

Le programme **postgres** prend aussi un certain nombre d'autres options en ligne de commande. Pour plus d'informations, voir la page de référence postmaster(1) ainsi que le Chapitre 18, Configuration du serveur ci-dessous.

Cette syntaxe shell peut rapidement devenir ennuyante. Donc, le programme d'emballage pg_ctl(1) est fourni pour simplifier certaines tâches. Par exemple :

```
pg_ctl start -l journaux_trace
```

lancera le serveur en tâche de fond et placera les sorties dans le journal de trace indiqué. L'option `-d` a la même signification ici que pour **postgres**. **pg_ctl** est aussi capable d'arrêter le serveur.

Normalement, vous lancerez le serveur de bases de données lors du démarrage de l'ordinateur . Les scripts de lancement automa-

tique sont spécifiques au système d'exploitation. Certains sont distribués avec PostgreSQL™ dans le répertoire `contrib/start-scripts`. En installer un demandera les droits de root.

Différents systèmes ont différentes conventions pour lancer les démons au démarrage. La plupart des systèmes ont un fichier `/etc/rc.local` ou `/etc/rc.d/rc.local`. D'autres utilisent les répertoires `rc.d`. Quoi que vous fassiez, le serveur doit être exécuté par le compte utilisateur PostgreSQL™ *et non pas par root* ou tout autre utilisateur. Donc, vous devriez probablement former vos commandes en utilisant `su postgres -c '...'` . Par exemple :

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Voici quelques suggestions supplémentaires par système d'exploitation (dans chaque cas, assurez-vous d'utiliser le bon répertoire d'installation et le bon nom de l'utilisateur où nous montrons des valeurs génériques).

- Pour freebsd™, regardez le fichier `contrib/start-scripts/freebsd` du répertoire des sources de PostgreSQL™.
- Sur openbsd™, ajoutez les lignes suivantes à votre fichier `/etc/rc.local` :

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
      su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log
  -D /usr/local/pgsql/data'
      echo -n ' PostgreSQL'
  fi
  ```

- Sur les systèmes linux™, soit vous ajoutez

  ```
  /usr/local/pgsql/bin/pg_ctl start -l journaux_trace -D /usr/local/pgsql/data
  ```

  à `/etc/rc.d/rc.local` soit vous jetez un œil à `contrib/start-scripts/linux` dans le répertoire des sources de PostgreSQL™.
- Sur netbsd™, vous pouvez utiliser les scripts de lancement de freebsd™ ou de linux™ suivant vos préférences.
- Sur solaris™, créez un fichier appelé `/etc/init.d/PostgreSQL` et contenant la ligne suivante :

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l journaux_trace -D
  /usr/local/pgsql/data"
  ```

  Puis, créez un lien symbolique vers lui dans `/etc/rc3.d` de nom `s99PostgreSQL`.

Tant que le serveur est lancé, son pid est stocké dans le fichier `postmaster.pid` du répertoire de données. C'est utilisé pour empêcher plusieurs instances du serveur d'être exécutées dans le même répertoire de données et peut aussi être utilisé pour arrêter le processus le serveur.

17.3.1. Échecs de lancement

Il existe de nombreuses raisons habituelles pour lesquelles le serveur échouerait au lancement. Vérifiez le journal des traces du serveur ou lancez-le manuellement (sans redirection des sorties standard et d'erreur) et regardez les messages d'erreurs qui apparaissent. Nous en expliquons certains ci-dessous parmi les messages d'erreurs les plus communs.

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds
and retry.
FATAL:  could not create TCP/IP listen socket
```

Ceci signifie seulement ce que cela suggère : vous avez essayé de lancer un autre serveur sur le même port où un autre est en cours d'exécution. Néanmoins, si le message d'erreur du noyau n'est pas `address already in use` ou une quelconque variante, il pourrait y avoir un autre problème. Par exemple, essayer de lancer un serveur sur un numéro de port réservé pourrait avoir ce résultat :

```
$ postgres -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds
and retry.
FATAL:  could not create TCP/IP listen socket
```

Un message du type

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

signifie probablement que les limites de votre noyau sur la taille de la mémoire partagée est plus petite que l'aire de fonctionne-

ment que PostgreSQL™ essaie de créer (4011376640 octets dans cet exemple). Ou il pourrait signifier que vous n'avez pas du tout configuré le support de la mémoire partagée de type System-V dans votre noyau. Comme contournement temporaire, vous pouvez essayer de lancer le serveur avec un nombre de tampons plus petit que la normale (shared_buffers). Éventuellement, vous pouvez reconfigurer votre noyau pour accroître la taille de mémoire partagée autorisée. Vous pourriez voir aussi ce message en essayant d'exécuter plusieurs serveurs sur la même machine si le total de l'espace qu'ils requièrent dépasse la limite du noyau.

Une erreur du type

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

ne signifie *pas* qu'il vous manque de l'espace disque. Elle signifie que la limite de votre noyau sur le nombre de sémaphores `system v` est inférieure au nombre que PostgreSQL™ veut créer. Comme ci-dessus, vous pouvez contourner le problème en lançant le serveur avec un nombre réduit de connexions autorisées (max_connections) mais vous voudrez éventuellement augmenter la limite du noyau.

Si vous obtenez une erreur « illegal system call », il est probable que la mémoire partagée ou les sémaphores ne sont pas du tout supportés par votre noyau. Dans ce cas, votre seule option est de reconfigurer le noyau pour activer ces fonctionnalités.

Des détails sur la configuration des capacités ipc `System V` sont donnés dans la Section 17.4.1, « Mémoire partagée et sémaphore ».

17.3.2. Problèmes de connexion du client

Bien que les conditions d'erreurs possibles du côté client sont assez variées et dépendantes de l'application, certaines pourraient être en relation direct avec la façon dont le serveur a été lancé. Les conditions autres que celles montrées ici devraient être documentées avec l'application client respective.

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

Ceci est l'échec générique « je n'ai pas trouvé de serveur à qui parler ». Cela ressemble au message ci-dessus lorsqu'une connexion TCP/IP est tentée. Une erreur commune est d'oublier de configurer le serveur pour qu'il autorise les connexions TCP/IP.

Autrement, vous obtiendrez ceci en essayant une communication de type socket de domaine Unix vers un serveur local :

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

La dernière ligne est utile pour vérifier si le client essaie de se connecter au bon endroit. Si aucun serveur n'est exécuté ici, le message d'erreur du noyau sera typiquement soit `connection refused` soit `no such file or directory`, comme ce qui est illustré (il est important de réaliser que `connection refused`, dans ce contexte, ne signifie *pas* que le serveur a obtenu une demande de connexion et l'a refusé. Ce cas produira un message différent comme indiqué dans la Section 19.4, « Problèmes d'authentification »). D'autres messages d'erreurs tel que `connection timed out` pourraient indiquer des problèmes plus fondamentaux comme un manque de connexion réseau.

17.4. Gérer les ressources du noyau

Une installation importante de PostgreSQL™ peut rapidement épuiser les limites des ressources du système d'exploitation (Sur certains systèmes, les valeurs par défaut sont trop basses que vous n'avez même pas besoin d'une installation « importante ».). Si vous avez rencontré ce type de problème, continuez votre lecture.

17.4.1. Mémoire partagée et sémaphore

La mémoire partagée et les sémaphores sont nommés collectivement « ipc `system v` » (ensemble avec les queues de messages, qui n'ont pas d'importance pour PostgreSQL™). Pratiquement, tous les systèmes d'exploitation modernes fournissent ces fonctionnalités mais, parmi elles, toutes ne sont pas activées ou dimensionnées suffisamment par défaut, car la mémoire disponible et la demande des applications augmente. (Sur `Windows`, PostgreSQL™ fournit sa propre implémentation de remplacement de ces fonctionnalités, du coup, ce qui suit peut être ignoré).

Le manque complet de fonctionnalités est généralement manifesté par une erreur illegal system call au lancement du serveur. Dans ce cas, il n'y a rien à faire à part reconfigurer votre noyau. PostgreSQL™ ne fonctionnera pas sans. Néanmoins, cette situation est rare parmi les systèmes d'exploitation modernes.

Quand PostgreSQL™ dépasse une des nombreuses limites ipc, le serveur refusera de s'exécuter et lèvera un message d'erreur instructif décrivant le problème rencontré et que faire avec (voir aussi la Section 17.3.1, « Échecs de lancement »). Les paramètres

adéquats du noyau sont nommés de façon cohérente parmi les différents systèmes ; le Tableau 17.1, « Paramètres `system v` ipc » donne un aperçu. Néanmoins, les méthodes pour les obtenir varient. Les suggestions pour quelques plateformes sont données ci-dessous.

Tableau 17.1. Paramètres `system v` ipc

Nom	Description	Valeurs raisonnables
`shmmax`	taille maximum du segment de mémoire partagée (octets)	au moins plusieurs mo (voir texte)
`shmmin`	taille minimum du segment de mémoire partagée (octets)	1
`shmall`	total de la mémoire partagée disponible (octets ou pages)	si octets, identique à `shmmax` ; si pages, `ceil(shmmax/page_size)`
`shmseg`	nombre maximum de segments de mémoire partagée par processus	seul un segment est nécessaire mais la valeur par défaut est bien plus importante
`shmmni`	nombre maximum de segments de mémoire partagée pour tout le système	comme `shmseg` plus la place pour les autres applications
`semmni`	nombre maximum d'identifiants de sémaphores (c'est-à-dire d'ensembles)	au moins `ceil((max_connections + autovacuum_max_workers + 4) / 16)`
`semmns`	nombre maximum de sémaphores répartis dans le système	`ceil((max_connections + autovacuum_max_workers + 4) / 16) * 17` plus la place pour les autres applications
`semmsl`	nombre maximum de sémaphores par ensemble	au moins 17
`semmap`	nombre d'entrées dans la carte des sémaphores	voir le texte
`semvmx`	valeur maximum d'un sémaphore	au moins 1000 (vaut souvent par défaut 32767, ne pas changer sauf si vous êtes forcé.)

le paramètre de mémoire partagé le plus important est `shmmax`, la taille maximum, en octets, d'un segment de mémoire partagée. Si vous obtenez un message d'erreur à partir de `shmget` comme « invalid argument », il est possible que cette limite soit dépassée. La taille du segment de mémoire partagée requis dépend de plusieurs paramètres de configuration de PostgreSQL™, comme indiqué dans le Tableau 17.2, « Usage de la mémoire partagée PostgreSQL™ » (tout message d'erreur obtenu incluera la taille exacte utilisée dans la requête d'allocation qui a échoué). Temporairement, vous pouvez baisser certains de ces paramètres pour éviter un échec. Alors qu'il est possible d'obtenir de PostgreSQL™ qu'il fonctionne avec un `shmmax` de 2 Mo, vous avez besoin de bien plus pour obtenir des performances acceptables. Les paramètrages désirables sont plutôt de l'ordre de centaines de Mo à quelques Go.

Certains systèmes ont aussi une limite sur le nombre total de mémoire partagée dans le système (`shmall`). Assurez-vous que cela soit suffisamment important pour PostgreSQL™ et quelque autres applications utilisant des segments de mémoire partagée (notez que `shmall` est mesuré en pages plutôt qu'en octets sur beaucoup de systèmes).

La taille minimum des segments de mémoire partagée (`shmmin`) est moins sensible aux problèmes. Elle devrait être au plus à environ 500 Ko pour PostgreSQL™ (il est habituellement à 1). Le nombre maximum de segments au travers du système (`shmmni`) ou par processus (`shmseg`) a peu de chances de causer un problème sauf s'ils sont configurés à zéro sur votre système.

PostgreSQL™ utilise un sémaphore par connexion autorisée (max_connections) et par processus autovacuum autorisé (autovacuum_max_workers), le tout par ensemble de 16. Chacun de ces ensembles contiendra aussi un 17è sémaphore qui contient un « nombre magique » pour détecter la collision avec des ensembles de sémaphore utilisés par les autres applications. Le nombre maximum de sémaphores dans le système est initialisé par `semmns`, qui en conséquence doit être au moins aussi haut que `max_connections` plus `autovacuum_max_workers` plus un extra de chacune des 16 connexions autorisées et des processus autovacuum (voir la formule dans le Tableau 17.1, « Paramètres `system v` ipc »). Le paramètre `semmni` détermine la limite sur le nombre d'ensembles de sémaphores qui peuvent exister sur le système à un instant précis. Donc, ce paramètre doit être au moins égal à `ceil((max_connections + autovacuum_max_workers + 4) / 16)`. Baisser le nombre de connexions autorisées est un contournement temporaire pour les échecs qui sont habituellement indiqués par le message « no space left on device », à partir de la fonction `semget`.

Dans certains cas, il pourrait être nécessaire d'augmenter `semmap` pour être au moins dans l'ordre de `semmns`. Ce paramètre définit la taille de la carte de ressources de sémaphores, dans laquelle chaque bloc contigü de sémaphores disponibles ont besoin d'une entrée. Lorsqu'un ensemble de sémaphores est libéré ou qu'il est enregistré sous une nouvelle entrée de carte. Si la carte est pleine, les sémaphores libérés sont perdus (jusqu'au redémarrage). La fragmentation de l'espace des sémaphores pourrait amener dans le temps à moins de sémaphores disponibles.

La paramètre `semmsl`, qui détermine le nombre de sémaphores dans un ensemble, pourrait valoir au moins 17 pour

™.

D'autres paramètres en relation avec l'« annulation de sémaphores », tels que `semmnu` et `semume`, n'affectent pas PostgreSQL™.

Tableau 17.2. Usage de la mémoire partagée PostgreSQL™

Usage	Nombre d'octets approximatifs pour la mémoire partagée (en 8.3)
Connexions	(1800 + 270 * max_locks_per_transaction) * max_connections
Processus travailleurs de l'autovacuum	(1800 + 270 * max_locks_per_transaction) * autovacuum_max_workers
Transactions préparées	(770 + 270 * max_locks_per_transaction) * max_prepared_transactions
Tampons disque partagés	(block_size + 208) * shared_buffers
Tampons WAL	(wal_block_size + 8) * wal_buffers
Espace fixe requis	770 kB

`AIX`

À partir de la version 5.1, il ne doit plus être nécessaire de faire une configuration spéciale pour les paramètres tels que `SHMMAX`, car c'est configuré de façon à ce que toute la mémoire puisse être utilisée en tant que mémoire partagée. C'est le type de configuration habituellement utilisée pour d'autres bases de données comme DB/2.

Néanmoins, il pourrait être nécessaire de modifier l'information globale **ulimit** dans `/etc/security/limits` car les limites en dur par défaut pour les tailles de fichiers (`fsize`) et les nombres de fichiers (`nofiles`) pourraient être trop bas.

`bsd/os`

Mémoire partagée. Par défaut, seulement 4 Mo de mémoire partagée est supportée. Gardez en tête que la mémoire partagée n'est pas paginable ; elle est verrouillée en RAM. Pour accroître la mémoire partagée supportée par votre système, ajoutez ce qui suit à la configuration de votre noyau. Une valeur de 1024 pour `shmall` représente 4 mo de mémoire partagée. Pour argumenter la mémoire partagée supportée par votre système, ajoutez quelque chose comme ceci à votre configuration du noyau :

```
options "SHMALL=8192"
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

`shmall` est mesuré en pages de 4 Ko, donc une valeur de 1024 représente 4 Mo de mémoire partagée. Du coup, la configuration ci-dessus augmente l'aire de mémoire partagée à 32 Mo. Pour ceux utilisant une version 4.3 ou ultérieure, vous aurez probablement besoin d'augmenter `kernel_virtual_mb` au-dessus de la valeur par défaut, `248`. Une fois tous les changements effectués, recompilez le noyau et redémarrez.

Sémaphores. Vous voudrez probablement aussi augmenter le nombre de sémaphores ; la somme totale par défaut du système (60) n'autorisera seulement que 50 connexions PostgreSQL™. Initialisez les valeurs que vous souhaitez dans le fichier de configuration du noyau :

```
options "SEMMNI=40"
options "SEMMNS=240"
```

`freebsd`

Les paramètres par défaut sont seulement acceptables pour de petites installations (par exemple, la valeur par défaut de `shmmax` est de 32 mo). Les modifications se font via les interfaces **sysctl** ou **loader**. Les paramètres suivants peuvent être configurés en utilisant **sysctl** :

```
$ sysctl -w kern.ipc.shmall=32768
$ sysctl -w kern.ipc.shmmax=134217728
$ sysctl -w kern.ipc.semmap=256
```

Pour que ces paramètres persistent après les redémarrages, modifiez `/etc/sysctl.conf`.

Les paramètres restant, concernant les sémaphores, sont en lecture seule en ce qui concerne **sysctl** mais peuvent être modifiés avant le redémarrage en utilisant l'invite **loader** :

```
(loader) set kern.ipc.semmni=256
(loader) set kern.ipc.semmns=512
(loader) set kern.ipc.semmnu=256
```

De façon similaire, ils peuvent être sauvegardés entre les redémarrages dans `/boot/loader.conf`.

Vous pourriez aussi vouloir configurer votre noyau pour verrouiller la mémoire partagée en RAM et l'empêcher d'être envoyé dans la swap. Ceci s'accomplit en utilisant le paramètre `kern.ipc.shm_use_phys` de **sysctl**.

En cas d'exécution dans une cage FreeBSD en activant `security.jail.sysvipc_allowed` de sysctl, les postmaster exécutés dans différentes cages devront être exécutés par différents utilisateurs du système d'exploitation. Ceci améliore la sécurité car cela empêche les utilisateurs non root d'interférer avec la mémoire partagée ou les sémaphores d'une cage différente et cela permet au code de nettoyage des IPC PostgreSQL de fonctionner correctement (dans FreeBSD 6.0 et ultérieurs, le code de nettoyage IPC ne détecte pas proprement les processus des autres cages, empêchant les postmaster en cours d'exécution d'utiliser le même port dans différentes cages).

Les `freebsd`, avant la 4.0, fonctionnent comme `OpenBSD` (voir ci-dessous).

`NetBSD`

Avec `NetBSD` 5.0 et ultérieur, les paramètres IPC peuvent être ajustés en utilisant **sysctl**. Par exemple :

```
$ sysctl -w kern.ipc.shmmax=16777216
```

Pour que ce paramètrage persiste après un redémarrage, modifiez le fichier `/etc/sysctl.conf`.

Vous pourriez aussi vouloir configurer votre noyau pour verrouiller la mémoire partagée en RAM et l'empêcher d'être mise dans le swap. Cela peut se faire en utilisant le paramètre `kern.ipc.shm_use_phys` de **sysctl**.

Les versions de `NetBSD` antérieures à la 5.0 fonctionnent comme `OpenBSD` (voir ci-dessous), sauf que les paramètres doivent être configurés avec le mot clé `options`, et non pas `option`.

`OpenBSD`

Les options `sysvshm` et `sysvsem` doivent être activées à la compilation du noyau (ils le sont par défaut). La taille maximum de mémoire partagée est déterminée par l'option `shmmaxpgs` (en pages). Ce qui suit montre un exemple de l'initialisation des différents paramètres :

```
option        SYSVSHM
option        SHMMAXPGS=4096
option        SHMSEG=256

option        SYSVSEM
option        SEMMNI=256
option        SEMMNS=512
option        SEMMNU=256
option        SEMMAP=256
```

Vous pourriez aussi vouloir configurer votre noyau pour verrouiller la mémoire partagée en RAM et l'empêcher d'être paginée en swap. Ceci se fait en utilisant le paramètre `kern.ipc.shm_use_phys` de **sysctl**.

`hp-ux`

Les paramètres par défaut tendent à suffire pour des installations normales. Sur hp-ux™ 10, la valeur par défaut de `semmns` est 128, qui pourrait être trop basse pour de gros sites de bases de données.

Les paramètres ipc peuvent être initialisés dans system administration manager (sam) sous kernel configuration → configurable Parameters. Allez sur create a new kernel une fois terminée.

`linux`

La taille maximale du segment par défaut est de 32 Mo, ce qui n'est adéquat que pour les très petites installations de PostgreSQL™. La taille totale maximale par défaut est de 2097152 pages. Une page équivaut pratiquement toujours à 4096 octets sauf pour certaines configurations inhabituelles du noyau comme « huge pages » (utilisez `getconf PAGE_SIZE` pour vérifier). Cela donne une limite par défaut de 8 Go, ce qui est souvent suffisant.

La configuration de la taille de mémoire partagée peut être modifiée avec l'interface proposée par la commande **sysctl**. Par exemple, pour permettre l'utilisation de 16 Go :

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

De plus, ces paramètres peuvent être préservés entre des redémarrages dans le fichier `/etc/sysctl.conf`. Il est recommandé de le faire.

Les anciennes distributions pourraient ne pas avoir le programme **sysctl** mais des modifications équivalentes peuvent se faire en manipulant le système de fichiers `/proc` :

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

Les valeurs par défaut restantes sont taillées de façon assez généreuses pour ne pas nécessiter de modifications.

`macos x`

La méthode recommandée pour configurer la mémoire partagée sous OS X est de créer un fichier nommé `/etc/sysctl.conf` contenant des affectations de variables comme :

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Notez que, dans certaines versions d'OS X, *les cinq* paramètres de mémoire partagée doivent être configurés dans `/etc/sysctl.conf`, sinon les valeurs seront ignorées.

Attention au fait que les versions récentes d'OS X ignorent les tentatives de configuration de `SHMMAX` à une valeur qui n'est pas un multiple exact de 4096.

`SHMALL` est mesuré en page de 4 Ko sur cette plateforme.

Dans les anciennes versions d'OS X, vous aurez besoin de redémarrer pour que les modifications de la mémoire partagée soient prises en considération. À partir de la version 10.5, il est possible de tous les modifier en ligne sauf `SHMMNI`, grâce à sysctl. Mais il est toujours préférable de configurer vos valeurs préférées dans `/etc/sysctl.conf`, pour que les nouvelles valeurs soient conservées après un redémarrage.

Le fichier `/etc/sysctl.conf` est seulement honoré à partir de la version 1.0.3.9 de OS X. Si vous utilisez une version antérieure, vous devez modifier le fichier `/etc/rc` et changer les valeurs dans les commandes suivantes :

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

Notez que `/etc/rc` est habituellement écrasé lors de mises à jour systèmes d'OS X, donc vous devez vous attendre à les modifier manuellement après chaque mise à jour.

En 10.2 et avant cette version, modifiez ces commandes dans le fichier `/System/Library/StartupItems/SystemTuning/SystemTuning`.

`sco openserver`

Dans la configuration par défaut, seuls 512 Ko de mémoire partagée par segment est autorisé. Pour augmenter ce paramétrage, allez tout d'abord dans le répertoire `/etc/conf/cf.d`. Pour afficher la valeur courante de `shmmax`, lancez :

```
./configure -y SHMMAX
```

Pour configurer une nouvelle valeur de `shmmax`, lancez :

```
./configure SHMMAX=valeur
```

où *`value`* est la nouvelle valeur que vous voulez utiliser (en octets). Après avoir configuré `shmmax`, reconstruisez le noyau :

```
./link_unix
```

et redémarrez.

`solaris`

Au moins dans la version 2.6, la taille maximum par défaut d'un segment de mémoire partagée est trop basse pour PostgreSQL™. Le paramétrage adéquat peut être modifié dans `/etc/system`, par exemple :

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
```

```
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

Vous avez besoin de redémarrer pour que les modifications prennent effet.

Voir aussi *pour des informations sur la mémoire partagée sous solaris™*.

17.4.2. Limites de ressources

Les systèmes d'exploitation style Unix renforcent différents types de limites de ressources qui pourraient interférer avec les opérations de votre serveur PostgreSQL™. Les limites sur le nombre de processus par utilisateur, le nombre de fichiers ouverts par un processus et la taille mémoire disponible pour chaque processus sont d'une grande importance. Chacun d'entre elles ont une limite « dure » et une limite « souple ». La limite souple est réellement ce qui compte mais cela pourrait être changé par l'utilisateur jusqu'à la limite dure. La limite dure pourrait seulement être modifiée par l'utilisateur root. L'appel système `setrlimit` est responsable de l'initialisation de ces paramètres. La commande interne du shell **ulimit** (shells Bourne) ou **limit** (csh) est utilisé pour contrôler les limites de ressource à partir de la ligne de commande. Sur les systèmes dérivés BSD, le fichier `/etc/login.conf` contrôle les différentes limites de ressource initialisées à la connexion. Voir la documentation du système d'exploitation pour les détails. Les paramètres en question sont `maxproc`, `openfiles` et `datasize`. par exemple :

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` est la limite douce. Ajoutez `-max` pour configurer la limite dure.)

Les noyaux peuvent aussi avoir des limites sur le système complet pour certaines ressources.

- Sur linux™, `/proc/sys/fs/file-max` détermine le nombre maximum de fichiers ouverts que le noyau supportera. Ce nombre est modifiable en écrivant un autre nombre dans le fichier ou en ajoutant une affectation dans `/etc/sysctl.conf`. La limite des fichiers par processus est fixée lors de la compilation du noyau ; voir `/usr/src/linux/documentation/proc.txt` pour plus d'informations.

Le serveur PostgreSQL™ utilise un processus par connexion de façon à ce que vous puissiez fournir au moins autant de processus que de connexions autorisées, en plus de ce dont vous avez besoin pour le reste de votre système. Ceci n'est habituellement pas un problème mais si vous exécutez plusieurs serveurs sur une seule machine, cela pourrait devenir étroit.

La limite par défaut des fichiers ouverts est souvent initialisée pour être « amicalement sociale », pour permettre à de nombreux utilisateurs de coexister sur une machine sans utiliser une fraction inappropriée des ressources du système. Si vous lancez un grand nombre de serveurs sur une machine, cela pourrait être quelque chose que vous souhaitez mais sur les serveurs dédiés, vous pourriez vouloir augmenter cette limite.

D'un autre côté, certains systèmes autorisent l'ouverture d'un grand nombre de fichiers à des processus individuels ; si un plus grand nombre le font, alors les limites du système peuvent facilement être dépassées. Si vous rencontrez ce cas et que vous ne voulez pas modifier la limite du système, vous pouvez initialiser le paramètre de configuration max_files_per_process de PostgreSQL™ pour limiter la consommation de fichiers ouverts.

17.4.3. Linux memory overcommit

Dans Linux 2.4 et suivants, le comportement par défaut de la mémoire virtuelle n'est pas optimal pour PostgreSQL™. Du fait de l'implémentation du « memory overcommit » par le noyau, celui-ci peut arrêter le serveur PostgreSQL™ (le processus serveur maître, « postmaster ») si les demandes de mémoire de PostgreSQL™ ou d'un autre processus provoque un manque de mémoire virtuelle au niveau du système.

Si cela se produit, un message du noyau qui ressemble à ceci (consulter la documentation et la configuration du système pour savoir où chercher un tel message) :

```
Out of Memory: Killed process 12345 (postgres)
```

peut survenir. Ceci indique que le processus `postgres` a été terminé à cause d'un problème de mémoire. Bien que les connexions en cours continuent de fonctionner normalement, aucune nouvelle connexion n'est acceptée. Pour revenir à un état normal, PostgreSQL™ doit être relancé.

Une façon d'éviter ce problème revient à lancer PostgreSQL™ sur une machine où vous pouvez vous assurer que les autres processus ne mettront pas la machine en manque de mémoire. S'il y a peu de mémoire, augmenter la swap peut aider à éviter le problème car un système peut tuer des processus lorsque la mémoire physique et la mémoire swap sont utilisées entièrement.

Si PostgreSQL™ est lui-même la cause du manque mémoire du système, vous pouvez éviter le problème en modifiant votre configuration. Dans certains cas, il est intéressant de baisser les paramètres relatifs à la mémoire, en particulier `shared_buffers` et `work_mem`. Dans d'autres cas, le problème peut être causé en autorisant de trop nombreuses connexions au serveur. Dans un grand nombre de cas, il est préférable de réduire `max_connections` et d'utiliser à la place un pooler de connexions.

Sur Linux 2.6 et ultérieur, il est possible de modifier le comportement du noyau avec le « overcommit memory ». Bien que ce paramétrage n'empêchera pas ce *comportement*, il réduira sa fréquence de façon significative et contribuera du coup à un système plus robuste. Ceci se fait en sélectionnant le mode strict de l'overcommit via **sysctl** :

```
sysctl -w vm.overcommit_memory=2
```

ou en plaçant une entrée équivalente dans `/etc/sysctl.conf`. Vous pourriez souhaiter modifier le paramétrage relatif `vm.overcommit_ratio`. Pour les détails, voir la documentation du noyau (`documentation/vm/overcommit-accounting`).

Une autre approche, qui peut aussi utiliser la modification de `vm.overcommit_memory`, est de configurer la valeur de la variable `oom_adj`, valeur par processus, pour le processus postmaster à `-17`, garantissant ainsi qu'il ne sera pas la cible de OOM. La façon la plus simple de le faire est d'exécuter

```
echo -17 > /proc/self/oom_adj
```

dans le script de démarrage de postmaster juste avant d'appeler postmaster. Notez que cette action doit être faite en tant qu'utilisateur root. Dans le cas contraire, elle n'aura aucun effet. Du coup, un script de démarrage, exécuté par root, est le meilleur endroit où placer ce code. Si vous le faites, vous pourriez aussi souhaiter construire PostgreSQL™ avec l'option `-DLINUX_OOM_ADJ=0` ajoutée à `CPPFLAGS`. Cela fera en sorte que les processus enfants de postmaster seront exécutés avec la valeur `oom_adj` normale de zéro, pour que OOM puisse les cibler si nécessaire.

> **Note**
>
> Quelques noyaux 2.4 de vendeurs ont des pré-versions de l'overcommit du 2.6. Néanmoins, configurer `vm.overcommit_memory` à 2 sur un noyau 2.4 qui n'a pas le code correspondant rendra les choses pires qu'elles n'étaient. Il est recommandé d'inspecter le code source du noyau (voir la fonction `vm_enough_memory` dans le fichier `mm/mmap.c`) pour vérifier ce qui est supporté dans votre noyau avant d'essayer ceci avec une installation 2.4. La présence du fichier de documentation `overcommit-accounting` ne devrait *pas* être pris comme une preuve de la présence de cette fonctionnalité. En cas de doute, consultez un expert du noyau ou le vendeur de votre noyau.

17.5. Arrêter le serveur

Il existe plusieurs façons d'arrêter le serveur de bases de données. Vous contrôlez le type d'arrêt en envoyant différents signaux au processus serveur maître.

`sigterm`
: C'est le mode d'*arrêt intelligent*. Après réception de `sigterm`, le serveur désactive les nouvelles connexions mais permet aux sessions en cours de terminer leur travail normalement. Il s'arrête seulement après que toutes les sessions se sont terminées normalement. C'est l'arrêt intelligent (*smart shutdown*). Si le serveur est en mode de sauvegarde en ligne, il attends en plus la désactivation du mot de sauvegarde en ligne. Lorsque le mode de sauvegarde est actif, les nouvelles connexions sont toujours autorisées, mais seulement pour les superutilisateurs (cette exception permet à un superutilisateur de se connecter pour terminer le mode de sauvegarde en ligne). Si le serveur est en restauration quand une demande d'arrêt intelligent est envoyée, la restauration et la réplication en flux seront stoppées seulement une fois que toutes les autres sessions ont terminé.

`sigint`
: C'est le mode d'*arrêt rapide*. Le serveur désactive les nouvelles connexions et envoie à tous les processus serveur le signal `sigterm`, qui les fera annuler leurs transactions courantes pour quitter rapidement. Il attend ensuite la fin de tous les processus serveur et s'arrête finalement. Si le serveur est en mode de sauvegarde en ligne, le mode est annulé, rendant la sauvegarde inutilisable.

`sigquit`
: C'est le mode d'*arrêt immédiat*. Le processus **postgres** maître envoie un signal `sigquit` à tous les processus fils et à quitter immédiatement non proprement. Les processus fils quittent immédiatement à réception du signal `sigquit`. ceci amènera une tentative de récupération (en rejouant les traces WAL) au prochain lancement. Ceci n'est recommandé que dans les cas d'urgence.

Le programme pg_ctl(1) fournit une interface agréable pour envoyer ces signaux dans le but d'arrêter le serveur. Autrement, vous

pouvez envoyer le signal directement en utilisant **kill** sur les systèmes autres que Windows. Le PID du processus **postgres** peut être trouvé en utilisant le programme **ps** ou à partir du fichier `postmaster.pid` dans le répertoire des données. Par exemple, pour exécuter un arrêt rapide :

```
$ kill -int `head -1 /usr/local/pgsql/data/postmaster.pid`
```

> **Important**
>
> Il vaux mieux de ne pas utiliser `sigkill` pour arrêter le serveur. Le faire empêchera le serveur de libérer la mémoire partagée et les sémaphores, ce qui pourrait devoir être fait manuellement avant qu'un nouveau serveur ne soit lancé. De plus, `SIGKILL` tue le processus **postgres** sans que celui-ci ait le temps de relayer ce signal à ses sous-processus, donc il sera aussi nécessaire de tuer les sous-processus individuels à la main.

Pour terminer une session individuelle tout en permettant aux autres de continuer, utilisez `pg_terminate_backend()` (voir Tableau 9.55, « Fonctions d'envoi de signal au serveur ») ou envoyez un signal `SIGTERM` au processus fils associé à cette session.

17.6. Empêcher l'usurpation de serveur

Quand le serveur est en cours d'exécution, un utilisateur pernicieux ne peut pas interférer dans les communications client/serveur. Néanmoins, quand le serveur est arrêté, un utilisateur local peut usurper le serveur normal en lançant son propre serveur. Le serveur usurpateur pourrait lire les mots de passe et requêtes envoyées par les clients, mais ne pourrait pas renvoyer de données car le répertoire `PGDATA` serait toujours sécurisé grâce aux droits d'accès du répertoire. L'usurpation est possible parce que tout utilisateur peut lancer un serveur de bases de données ; un client ne peut pas identifier un serveur invalide sauf s'il est configuré spécialement.

Le moyen le plus simple d'empêcher les serveurs invalides pour des connexions `local`es est d'utiliser un répertoire de socket de domaine Unix (unix_socket_directory) qui a un droit en écriture accessible seulement par un utilisateur local de confiance. Ceci empêche un utilisateur mal intentionné de créer son propre fichier socket dans ce répertoire. Si vous êtes concerné que certaines applications pourraient toujours référencer `/tmp` pour le fichier socket et, du coup, être vulnérable au « spoofing », lors de la création du lien symbolique `/tmp/.s.PGSQL.5432` pointant vers le fichier socket déplacé. Vous pouvez aussi avoir besoin de modifier votre script de nettoyage de `/tmp` pour empêcher la suppression du lien symbolique.

Pour empêcher l'usurpation des connexions TCP, le mieux est d'utiliser des certificats SSL et de s'assurer que les clients vérifient le certificat du serveur. Pour cela, le serveur doit être configuré pour accepter les connexions `hostssl` (Section 19.1, « Le fichier `pg_hba.conf` ») et avoir les fichiers SSL pour la clé, `server.key`, et pour le certificat, `server.crt` (Section 17.8, « Connexions tcp/ip sécurisées avec ssl »). Le client TCP doit se connecter en utilisant `sslmode='verify-ca'` ou `'verify-full'` et avoir le certificat racine installé (Section 31.1, « Fonctions de contrôle de connexion à la base de données »).

17.7. Options de chiffrement

PostgreSQL™ offre du chiffrement sur plusieurs niveaux et fournit une flexibilité pour protéger les données d'être révélées suite à un vol du serveur de la base de données, des administrateurs non scrupuleux et des réseaux non sécurisés. Le chiffrement pourrait aussi être requis pour sécuriser des données sensibles, par exemple des informations médicales ou des transactions financières.

chiffrement du mot de passe stocké
: Par défaut, les mots de passe des utilisateurs de la base de données sont stockées suivant des hachages MD5, donc l'administrateur ne peut pas déterminer le mot de passe affecté à l'utilisateur. Si le cryptage MD5 est utilisé pour l'authentification du client, le mot de passe non crypté n'est jamais présent temporairement sur le serveur parce que le client le crypte en MD5 avant de l'envoyer sur le réseau.

chiffrement de colonnes spécifiques
: La bibliothèque de fonctions `contrib` `pgcrypto` autorise le stockage crypté de certains champs. Ceci est utile si seulement certaines données sont sensibles. Le client fournit la clé de décryptage et la donnée est décryptée sur le serveur puis elle est envoyée au client.

  La donnée décryptée et la clé de déchiffrement sont présente sur le serveur pendant un bref moment où la donnée est décryptée, puis envoyée entre le client et le serveur. Ceci présente un bref moment où la données et les clés peuvent être interceptées par quelqu'un ayant un accès complet au serveur de bases de données, tel que l'administrateur du système.

chiffrement de la partition de données
: Sur Linux, le chiffrement peut se faire au niveau du montage d'un système de fichiers en utilisant un « périphérique loopback ». Ceci permet à une partition entière du système de fichiers d'être cryptée et décryptée par le système d'exploitation. Sur FreeBSD, la fonctionnalité équivalent est appelé « geom based disk encryption » (gbde), et beaucoup d'autres systèmes d'exploitations, comme Windows, supportent cette fonctionnalité.

Ce mécanisme empêche les données non cryptées d'être lues à partir des lecteurs s'ils sont volés. Ceci ne protège pas contre les attaques quand le système de fichiers est monté parce que, une fois monté, le système d'exploitation fournit une vue non cryptée des données. Néanmoins, pour monter le système de fichiers, vous avez besoin d'un moyen pour fournir la clé de chiffrement au système d'exploitation et, quelque fois, la clé est stocké quelque part près de l'hôte qui monte le disque.

chiffrement des mots de passe sur le réseau
: La méthode d'authentification `md5` crypte deux fois le mot de passe sur le client avant de l'envoyer au serveur. Il le crypte tout d'abord à partir du nom de l'utilisateur puis il le crypte à partir d'un élément du hasard envoyé par le serveur au moment de la connexion. Cette valeur, deux fois cryptée, est envoyée sur le réseau au serveur. Le double chiffrement empêche non seulement la découverte du mot de passe, il empêche aussi une autre connexion en utilisant le même mot de passe crypté pour se connecter au serveur de bases de données lors d'une connexion future.

chiffrement des données sur le réseau
: Les connexions SSL cryptent toutes les données envoyées sur le réseau : le mot de passe, les requêtes et les données renvoyées. Le fichier `pg_hba.conf` permet aux administrateurs de spécifier quels hôtes peuvent utiliser des connexions non cryptées (`host`) et lesquels requièrent des connexions SSL (`hostssl`). De plus, les clients peuvent spécifier qu'ils se connectent aux serveurs seulement via SSL. stunnel ou ssh peuvent aussi être utilisés pour crypter les transmissions.

authentification de l'hôte ssl
: Il est possible que le client et le serveur fournissent des certificats SSL à l'autre. Cela demande une configuration supplémentaire de chaque côté mais cela fournit une vérification plus forte de l'identité que la simple utilisation de mots de passe. Cela empêche un ordinateur de se faire passer pour le serveur assez longtemps pour lire le mot de passe envoyé par le client. Cela empêche aussi les attaques du type « man in the middle » où un ordinateur, entre le client et le serveur, prétend être le serveur, lit et envoie les données entre le client et le serveur.

chiffrement côté client
: Si vous n'avez pas confiance en l'administrateur système du serveur, il est nécessaire que le client crypte les données ; de cette façon, les données non cryptées n'apparaissent jamais sur le serveur de la base de données. Les données sont cryptées sur le client avant d'être envoyé au serveur, et les résultats de la base de données doivent être décryptés sur le client avant d'être utilisés.

17.8. Connexions tcp/ip sécurisées avec ssl

PostgreSQL™ dispose d'un support natif pour l'utilisation de connexions ssl, cryptant ainsi les communications clients/serveurs pour une sécurité améliorée. Ceci requiert l'installation d'openssl™ à la fois sur le système client et sur le système serveur et que ce support soit activé au moment de la construction de PostgreSQL™ (voir le Chapitre 15, Procédure d'installation de PostgreSQL™ du code source).

Avec le support ssl compilé, le serveur PostgreSQL™ peut être lancé avec ssl activé en activant ssl dans `PostgreSQL.conf`. Le serveur écoutera les deux connexions, standard et SSL sur le même port TCP, et négociera avec tout client l'utilisation de SSL. Par défaut, le client peut choisir cette option ; voir Section 19.1, « Le fichier `pg_hba.conf` » sur la façon de configurer le serveur pour réclamer l'utilisation de SSL pour certaines, voire toutes les connexions.

PostgreSQL™ lit le fichier de configuration d'OpenSSL™ pour le serveur. Par défaut, ce fichier est nommé `openssl.cnf` et est situé dans le répertoire indiqué par `openssl version -d`. Cette valeur par défaut peut être surchargée en configurant la variable d'environnement `OPENSSL_CONF` avec le nom du fichier de configuration désiré.

OpenSSL™ accepte une gamme étendue d'algorithmes de chiffrement et d'authentification, de différentes forces. Bien qu'une liste d'algorithmes de chiffrement peut être indiquée dans le fichier de configuration d'OpenSSL™, vous pouvez spécifier des algorithmes spécifiques à utiliser par le serveur de la base de données en modifiant le paramètre ssl_ciphers dans `postgresql.conf`.

Note

Il est possible d'avoir une authentification sans le chiffrement en utilisant les algorithmes `NULL-SHA` ou `NULL-MD5`. Néanmoins, une attaque du type *man-in-the-middle* pourrait lire et passer les communications entre client et serveur. De plus, le temps pris par le chiffrement est minimal comparé à celui pris par l'authentification. Pour ces raisons, les algorithmes NULL ne sont pas recommandés.

Pour démarrer le mode SSL, les fichiers `server.crt` et `server.key` doivent exister dans le répertoire de données du serveur. Ces fichiers doivent contenir, respectivement, le certificat et la clé privée du serveur. Sur les systèmes Unix, les droits de `server.key` doivent interdire l'accès au groupe et au reste du monde ; cela se fait avec la commande **chmod 0600 server.key**. Si la clé privée est protégée par une phrase de passe, le serveur la demandera et ne se lancera pas tant qu'elle n'aura pas été saisie.

Dans certains cas, le certificat du serveur peut être signé par une autorité « intermédiaire » de certificats, plutôt que par un qui soit

directement de confiance par les clients. Pour utiliser un tel certificat, ajoutez le certificat de l'autorité signataire au fichier `server.crt`, puis le certificat de l'autorité parente, et ainsi de suite jusqu'à l'autorité racine qui est acceptée par les clients. Le certificat racine doit être inclus dans chaque cas où `server.crt` contient plus d'un certificat.

17.8.1. Utiliser des certificats clients

Pour réclamer l'envoi d'un certificat de confiance par le client, placez les certificats des autorités (CA) de confiance dans le fichier `root.crt` du répertoire des données, et configurez le paramètre `clientcert` à `1` sur la ligne `hostssl` appropriée dans le fichier `pg_hba.conf`. Un certificat pourra ensuite être réclamé lors du lancement de la connexion SSL. (Voir Section 31.17, « Support de SSL » pour une description de la configuration de certificats sur le client.) Le serveur vérifiera que le certificat du client est signé par une des autorités de confiance. Les entrées de la liste de révocation des certificats sont aussi vérifiées si le fichier `root.crl` existe. (Voir les *diagrammes montrant l'utilisation des certificats SSL*.)

L'option `clientcert` de `pg_hba.conf` est disponible pour toutes les méthodes d'authentification, mais seulement pour les lignes spécifiées `hostssl`. Quand `clientcert` n'est pas précisé ou qu'il est configuré à `0`, le serveur vérifiera toujours les certificats clients présentés avec `root.crt` si ce fichier existe -- mais il ne forcera pas la présentation d'un certificat client.

Notez que `root.crt` liste les autorités de certificats de haut-niveau, ceux suffisamment de confiance pour signer les certificats des clients. En principe, il n'a pas besoin de lister l'autorité de certificats qui a signé le certificat du serveur bien que dans la plupart des cas, cette autorité sera aussi de confiance pour les certificats de clients.

Si vous configurez les certificats de clients, vous pouvez utiliser la méthode d'authentification `cert`, de façon à ce que les certificats soient aussi utilisés pour contrôler l'authentification de l'utilisateur, tout en fournissant une sécurité de connexion. Voir Section 19.3.9, « Authentification de certificat » pour les détails.

17.8.2. Utilisation des fichiers serveur SSL

Les fichiers `server.key`, `server.crt`, `root.crt` et `root.crl` sont seulement examinés au démarrage du serveur ; donc vous devez démarrer le serveur pour que les changements prennent effet.

Tableau 17.3. Utilisation des fichiers serveur SSL

Fichier	Contenu	Effet
`server.crt`	certificat du serveur	envoyé au client pour indiquer l'identité du serveur
`server.key`	clé privée du serveur	prouve que le certificat serveur est envoyé par son propriétaire n'indique pas que le propriétaire du certificat est de confiance
`root.crt`	autorités de confiance pour les certificats	vérifie le certificat du client ; vérifie que le certificat du client est signé par une autorité de confiance
`root.crl`	certificats révoqués par les autorités de confiance	le certificat du client ne doit pas être sur cette liste

17.8.3. Créer un certificat auto-signé

Pour créer rapidement un certificat signé soi-même pour le serveur, utilisez la commande OpenSSL™ suivante :

```
openssl req -new -text -out server.req
```

Remplissez l'information que openssl demande. Assurez-vous de saisir le nom de l'hôte local dans « Common Name » ; le mot de passe peut ne pas être saisi. Le programme générera une clé qui est protégée par une phrase de passe ; il n'acceptera pas une phrase de passe qui fait moins de quatre caractères de long. Pour la supprimer (vous le devez si vous voulez un démarrage automatique du serveur), exécutez les commandes suivantes :

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

Saisissez l'ancienne phrase de passe pour déverrouiller la clé existante. Maintenant, lancez :

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

pour transformer le certificat en un certificat auto-signé et pour copier la clé et le certificat là où le serveur les cherchera. Enfin, faites :

```
chmod og-rwx server.key
```

car le serveur rejetera le fichier si ses droits sont plus importants. Pour plus de détails sur la façon de créer la clé privée et le certificat de votre serveur, référez-vous à la documentation d'OpenSSL™.

Un certificat auto-signé peut être utilisé pour tester, mais un certificat signé par une autorité (CA) (un des CAs global ou un local) devra être utilisé lorsque le serveur sera en production pour que le client puisse vérifier l'identité du serveur. Si tous les clients sont locaux à l'organisation, utiliser un CA local est recommandé.

17.9. Connexions tcp/ip sécurisées avec des tunnels ssh tunnels

Il est possible d'utiliser ssh pour chiffrer la connexion réseau entre les clients et un serveur PostgreSQL™. Réalisé correctement, ceci fournit une connexion réseau sécurisée, y compris pour les clients non SSL.

Tout d'abord, assurez-vous qu'un serveur ssh est en cours d'exécution sur la même machine que le serveur PostgreSQL™ et que vous pouvez vous connecter via **ssh** en tant qu'un utilisateur quelconque. Ensuite, vous pouvez établir un tunnel sécurisé avec une commande comme ceci sur la machine cliente :

```
ssh -L 63333:localhost:5432 joe@foo.com
```

Le premier numéro de l'argument `-l`, 63333, est le numéro de port de votre bout du tunnel ; il peut être choisi parmi tous les ports non utilisés. (IANA réserve les ports 49152 à 65535 pour une utilisation privée.) Le second numéro, 5432, est le bout distant du tunnel : le numéro de port que votre serveur utilise. Le nom ou l'adresse entre les numéros de port est l'hôte disposant du serveur de bases de données auquel vous souhaitez vous connecter, comme vu à partir de l'hôte où vous vous connectez, qui est `foo.com` dans cet exemple. Pour vous connecter au serveur en utilisant ce tunnel, vous vous connectez au port 63333 de la machine locale :

```
psql -h localhost -p 63333 postgres
```

Sur le serveur de bases de données, il semblera que vous êtes réellement l'utilisateur `joe` sur l'hôte `foo.com` en vous connectant à `localhost` dans ce contexte, et il utilisera la procédure d'authentification configurée pour les connexions de cet utilisateur et de cet hôte. Notez que le serveur ne pensera pas que la connexion est chiffrée avec SSL car, en effet, elle n'est pas chiffrée entre le serveur SSH et le serveur PostgreSQL™. Cela ne devrait pas poser un risque de sécurité supplémentaire si les deux serveurs sont sur la même machine.

Pour réussir la configuration du tunnel, vous devez être autorisé pour vous connecter via **ssh** sur `joe@foo.com`, comme si vous aviez tenté d'utiliser **ssh** pour créer une session de terminal.

Vous pouvez aussi configurer la translation de port de cette façon :

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

mais alors le serveur de la base de données verra la connexion venir de son interface `foo.com` qui n'est pas ouverte par son paramétrage par défaut `listen_addresses = 'localhost'`. Ceci n'est pas habituellement ce que vous êtes.

Si vous devez vous connecter au serveur de bases de données via un hôte de connexion, une configuration possible serait :

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Notez que de cette façon la connexion de `shell.foo.com` à `db.foo.com` ne sera pas chiffrée par le tunnel SSH. SSH offre un certain nombre de possibilités de configuration quand le réseau est restreint. Merci de vous référer à la documentation de SSH pour les détails.

Astuce

Plusieurs autres applications existantes peuvent fournir des tunnels sécurisés en utilisant une procédure similaire dans le concept à celle que nous venons de décrire.

Chapitre 18. Configuration du serveur

Un grand nombre de paramètres de configuration permettent de modifier le comportement du système de bases de données. Dans la première section de ce chapitre, les méthodes de configuration de ces paramètres sont décrites ; les sections suivantes discutent de chaque paramètre en détail.

18.1. Paramètres de configuration

Tous les noms de paramètres sont insensibles à la casse. Chaque paramètre prend une valeur d'un de ces cinq types : booléen, entier, nombre à virgule flottante, chaîne de caractères ou énumération. Les unités par défaut peuvent être récupérées en référençant `pg_settings.unit`. Les valeurs booléennes peuvent être `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` ou tout préfixe non ambigü de celles-ci (toutes ces écritures sont insensibles à la casse).

Certains paramètres indiquent une valeur de taille mémoire ou de durée. Ils ont chacun une unité implicite, soit Ko, soit blocs (typiquement 8 Ko), soit millisecondes, soit secondes, soit minutes. Les unités par défaut peuvent être obtenues en interrogeant pg_settings.*`unit`*. Pour simplifier la saisie, une unité différente peut être indiquée de façon explicite. Les unités mémoire valides sont `kB` (kilo-octets), `MB` (Méga-octets) et `GB` (Giga-octets) ; les unités de temps valides sont `ms` (millisecondes), `s` (secondes), `min` (minutes), `h` (heures), et `d` (jours). Les unités de mémoire sont des multiples de 1024, pas de 1000.

Les paramètres de type « enum » sont spécifiés de la même façon que les paramètres de type chaîne, mais sont restreints à un jeu limité de valeurs. Les valeurs autorisées peuvent être obtenues de pg_settings.*`enumvals`*. Les paramètres enum sont insensibles à la casse.

Une façon d'initialiser ces paramètres est d'éditer le fichier `postgresql.conf` qui est normalement placé dans le répertoire des données (une copie par défaut est installé ici quand le répertoire des données est initialisé). Un exemple de contenu peut être :

```
# Ceci est un commentaire
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Un paramètre est indiqué par ligne. Le signe égal entre le nom et la valeur est optionnel. Les espaces n'ont pas de signification et les lignes vides sont ignorées. Les symboles dièse (#) désignent le reste de la ligne comme un commentaire. Les valeurs des paramètres qui ne sont pas des identificateurs simples ou des nombres doivent être placées entre guillemets simples. Pour intégrer un guillemet simple dans la valeur d'un paramètre, on écrit soit deux guillemets (c'est la méthode préférée) soit un antislash suivi du guillemet.

En plus de la configuration des paramètres, le fichier `postgresql.conf` peut contenir des *directives d'inclusion* indiquant un autre fichier à lire et dont le contenu doit être traité à ce niveau comme partie intégrante du fichier de configuration. Les directives d'inclusion ressemblent simplement à :

```
include 'nom_fichier'
```

Si le nom du fichier n'est pas un chemin absolu, il est pris comme relatif au répertoire contenant le fichier de configuration le référençant. Les inclusions peuvent être imbriquées.

Le fichier de configuration est relu à chaque fois que le processus serveur principal reçoit un signal `SIGHUP` (`pg_ctl reload` est le moyen le plus simple de l'envoyer). Le processus serveur principal propage aussi ce signal aux processus serveur en cours d'exécution de façon à ce que les sessions existantes obtiennent aussi la nouvelle valeur. Il est également possible d'envoyer le signal directement à un seul processus serveur. Quelques paramètres ne peuvent être initialisés qu'au lancement du serveur ; tout changement de leur valeur dans le fichier de configuration est ignoré jusqu'au prochain démarrage du serveur.

Une autre façon de configurer ces paramètres est de les passer comme option de la commande **postgres** :

```
postgres -c log_connections=yes -c log_destination='syslog'
```

Les options de la ligne de commande surchargent le paramétrage effectué dans le fichier `postgresql.conf`. Ce qui signifie que la valeur d'un paramètre passé en ligne de commande ne peut plus être modifiée et rechargée à la volée à l'aide du fichier `postgresql.conf`. C'est pourquoi, bien que la méthode de la ligne de commande paraisse pratique, elle peut coûter en flexibilité par la suite.

Il est parfois utile de donner une option en ligne de commande pour une session particulière unique. La variable d'environnement `PGOPTIONS` est utilisée côté client à ce propos :

```
env PGOPTIONS='-c geqo=off' psql
```

(Cela fonctionne pour toute application client fondée sur libpq, et non pas seulement pour psql.) Cela ne fonctionne pas pour les paramètres fixés au démarrage du serveur ou qui doivent être précisés dans `postgresql.conf`.

De plus, il est possible d'affecter un ensemble de paramètres à un utilisateur ou à une base de données. Quand une session est lancée, les paramètres par défaut de l'utilisateur et de la base de données impliqués sont chargés. Les commandes ALTER USER(7) et ALTER DATABASE(7) sont respectivement utilisées pour configurer ces paramètres. Les paramètres par base de données surchargent ceux passés sur la ligne de commande de **postgres** ou du fichier de configuration et sont à leur tour surchargés par ceux de l'utilisateur ; les deux sont surchargés par les paramètres de session.

Quelques paramètres peuvent être modifiés dans les sessions SQL individuelles avec la commande SET(7), par exemple :

```
SET ENABLE_SEQSCAN TO OFF;
```

Si **SET** est autorisé, il surcharge toutes les autres sources de valeurs pour le paramètre. Quelques paramètres ne peuvent pas être changés via **SET** : s'ils contrôlent un comportement qui ne peut pas être modifié sans relancer le serveur PostgreSQL™, par exemple. De plus, quelques paramètres peuvent être modifiés via **SET** ou **ALTER** par les superutilisateurs.

La commande SHOW(7) permet d'inspecter les valeurs courantes de tous les paramètres.

La table virtuelle pg_settings (décrite dans la Section 45.55, « pg_settings ») autorise aussi l'affichage et la mise à jour de paramètres de session à l'exécution. Elle est équivalente à **SHOW** et **SET** mais peut être plus facile à utiliser parce qu'elle peut être jointe avec d'autres tables et que ses lignes peuvent être sélectionnées en utilisant des conditions personnalisées. Elle contient aussi davantage d'informations sur les valeurs autorisées pour les paramètres.

18.2. Emplacement des fichiers

En plus du fichier `postgresql.conf` déjà mentionné, PostgreSQL™ utilise deux autres fichiers de configuration éditables manuellement. Ces fichiers contrôlent l'authentification du client (leur utilisation est discutée dans le Chapitre 19, Authentification du client). Par défaut, les trois fichiers de configuration sont stockés dans le répertoire data du cluster de bases de données. Les paramètres décrits dans cette section permettent de déplacer les fichiers de configuration. Ce qui peut en faciliter l'administration. Il est, en particulier, souvent plus facile de s'assurer que les fichiers de configuration sont correctement sauvegardés quand ils sont conservés à part.

`data_directory` (string)

Indique le répertoire à utiliser pour le stockage des données. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

`config_file` (string)

Indique le fichier de configuration principal du serveur (appelé `postgresql.conf`). Ce paramètre ne peut être initialisé que sur la ligne de commande de **postgres**.

`hba_file` (string)

Indique le fichier de configuration de l'authentification fondée sur l'hôte (appelé `pg_hba.conf`). Ce paramètre ne peut être initialisé qu'au lancement du serveur.

`ident_file` (string)

Indique le fichier de configuration pour la correspondance des noms d'utilisateurs, fichier appelé `pg_ident.conf`). Voir Section 19.2, « Correspondances d'utilisateurs » pour plus de détails. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

`external_pid_file` (string)

Indique le nom d'un fichier supplémentaire d'identifiant de processus (PID) créé par le serveur à l'intention des programmes d'administration du serveur. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

Dans une installation par défaut, aucun des paramètres ci-dessus n'est configuré explicitement. À la place, le répertoire des données est indiqué par l'option `-D` en ligne de commande ou par la variable d'environnement `PGDATA`. Les fichiers de configuration sont alors tous disponibles dans le répertoire des données.

Pour conserver les fichiers de configuration dans un répertoire différent de data, l'option `-D` de la ligne de commande **postgres** ou la variable d'environnement `PGDATA` doit pointer sur le répertoire contenant les fichiers de configuration. Le paramètre `data_directory` doit alors être configuré dans le fichier `postgresql.conf` (ou sur la ligne de commande) pour préciser où est réellement situé le répertoire des données. `data_directory` surcharge `-D` et `PGDATA` pour l'emplacement du répertoire des données, mais pas pour l'emplacement des fichiers de configuration.

les noms des fichiers de configuration et leur emplacement peuvent être indiqués individuellement en utilisant les paramètres `config_file`, `hba_file` et/ou `ident_file`. `config_file` ne peut être indiqué que sur la ligne de commande de

postgres mais les autres peuvent être placés dans le fichier de configuration principal. Si les trois paramètres et `data_directory` sont configurés explicitement, alors il n'est pas nécessaire d'indiquer `-D` ou `PGDATA`.

Lors de la configuration de ces paramètres, un chemin relatif est interprété d'après le répertoire d'où est lancé **postgres**.

18.3. Connexions et authentification

18.3.1. Paramètres de connexion

`listen_addresses` (string)

Indique les adresses TCP/IP sur lesquelles le serveur écoute les connexions en provenance d'applications clientes. La valeur prend la forme d'une liste de noms d'hôte ou d'adresses IP numériques séparés par des virgules. L'entrée spéciale `*` correspond à toutes les interfaces IP disponibles. Si la liste est vide, le serveur n'écoute aucune interface IP, auquel cas seuls les sockets de domaine Unix peuvent être utilisées pour s'y connecter. La valeur par défaut est `localhost`, ce qui n'autorise que les connexions TCP/IP locales de type « loopback ». Bien que l'authentification client (Chapitre 19, Authentification du client) permet un contrôle très fin sur les accès au serveur, `listen_addresses` contrôle les interfaces pouvant accepter des tentatives de connexion, ce qui permet d'empêcher des demandes de connexion amlignes sur des interfaces réseau non sécurisées. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`port` (integer)

Le port TCP sur lequel le serveur écoute ; 5432 par défaut. Le même numéro de port est utilisé pour toutes les adresses IP que le serveur écoute. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`max_connections` (integer)

Indique le nombre maximum de connexions concurrentes au serveur de base de données. La valeur par défaut typique est de 100 connexions, mais elle peut être moindre si les paramètres du noyau ne le supportent pas (ce qui est déterminé lors de l'initdb). Ce paramètre ne peut être configuré qu'au lancement du serveur.

L'augmentation de ce paramètre peut obliger PostgreSQL™ à réclamer plus de mémoire partagée `System V` ou de sémaphores que ne le permet la configuration par défaut du système d'exploitation. Voir la Section 17.4.1, « Mémoire partagée et sémaphore » pour plus d'informations sur la façon d'ajuster ces paramètres, si nécessaire.

Lors de l'exécution d'un serveur en attente, vous devez configurer ce paramètre à la même valeur ou à une valeur plus importante que sur le serveur maître. Sinon, des requêtes pourraient ne pas être autorisées sur le serveur en attente.

`superuser_reserved_connections` (integer)

Indique le nombre de connecteurs (« slots ») réservés aux connexions des superutilisateurs PostgreSQL™. Au plus max_connections connexions peuvent être actives simultanément. Dès que le nombre de connexions simultanément actives atteint `max_connections` moins `superuser_reserved_connections`, les nouvelles connexions ne sont plus acceptées que pour les superutilisateurs, et aucune nouvelle connexion de réplication ne sera acceptée.

La valeur par défaut est de trois connexions. La valeur doit être plus petite que la valeur de `max_connections`. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`unix_socket_directory` (string)

Indique le répertoire du socket de domaine Unix sur lequel le serveur écoute les connexions des applications client. Par défaut, il s'agit de `/tmp` mais cela peut être modifié au moment de la construction. Ce paramètre ne peut être configuré qu'au lancement du serveur.

En plus du fichier socket, qui est nommé `.s.PGSQL.`*`nnnn`* où *`nnnn`* est le numéro de port du serveur, un fichier ordinaire nommé `.s.PGSQL.`*`nnnn`*`.lock` sera créé dans le répertoire `unix_socket_directory`. Les deux fichiers ne doivent pas être supprimés manuellement.

Ce paramètre n'a aucun intérêt sous Windows car ce système n'a pas de sockets domaine Unix.

`unix_socket_group` (string)

Configure le groupe propriétaire du socket de domaine Unix (l'utilisateur propriétaire de la socket est toujours l'utilisateur qui lance le serveur). En combinaison avec le paramètre `unix_socket_permissions`, ceci peut être utilisé comme un mécanisme de contrôle d'accès supplémentaire pour les connexions de domaine Unix. Par défaut, il s'agit d'une chaîne vide, ce qui sélectionne le groupe par défaut de l'utilisateur courant. Ce paramètre ne peut être configuré qu'au lancement du serveur.

Ce paramètre n'a aucun intérêt sous Windows car ce système n'a pas de sockets domaine Unix.

`unix_socket_permissions` (integer)

Configure les droits d'accès au socket de domaine Unix. Ce socket utilise l'ensemble habituel des droits du système de fichiers Unix. Ce paramètre doit être indiqué sous une forme numérique telle qu'acceptée par les appels système `chmod` et `umask` (pour utiliser le format octal, ce nombre doit commencer avec un `0` (zéro)).

Les droits par défaut sont `0777`, signifiant que tout le monde peut se connecter. Les alternatives raisonnables sont `0770` (utilisateur et groupe uniquement, voir aussi `unix_socket_group`) et `0700` (utilisateur uniquement) (pour un socket de domaine Unix, seul le droit d'accès en écriture importe ; il n'est donc pas nécessaire de donner ou de révoquer les droits de lecture ou d'exécution).

Ce mécanisme de contrôle d'accès est indépendant de celui décrit dans le Chapitre 19, Authentification du client.

Ce paramètre ne peut être configuré qu'au lancement du serveur.

Ce paramètre n'a aucun intérêt sous Windows car ce système n'a pas de sockets domaine Unix.

`bonjour` (boolean)

Active la promotion de l'existence du serveur via le protocole Bonjour™. Désactivé par défaut, ce paramètre ne peut être configuré qu'au lancement du serveur.

`bonjour_name` (string)

Indique le nom du service Bonjour™. Le nom de l'ordinateur est utilisé si ce paramètre est configuré avec une chaîne vide (ce qui est la valeur par défaut). Ce paramètre est ignoré si le serveur n'est pas compilé avec le support Bonjour™. Ce paramètre ne peut être configuré qu'au lancement du serveur.

Ce paramètre n'a pas de sens sur certains systèmes, notamment Solaris depuis la version 10, qui ignore complètement les droits sur les sockets. Il est possible d'arriver au même résultat en faisant pointer `unix_socket_directory` vers un répertoire ayant des droits limités pour une audience particulière. Ce paramètre est aussi inutile pour Windows qui ne dispose pas des sockets de domaine Unix.

`tcp_keepalives_idle` (integer)

Indique le nombre de secondes avant l'envoi d'un paquet keepalive sur une connexion qui semble inutilisée. Une valeur de 0 revient à utiliser la valeur système par défaut. Ce paramètre est seulement supporté par les systèmes qui supportent les symboles TCP_KEEPIDLE ou TCP_KEEPALIVE et sur Windows ; sur les autres systèmes, ce paramètre doit valoir zéro. Pour les sessions connectées via une socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

Note

Sur Windows, une valeur de 0 configurera ce paramètre à deux heures car Windows ne fournit pas un moyen de lire la valeur par défaut du système.

`tcp_keepalives_interval` (integer)

Indique le nombre de secondes entre chaque envoi d'un paquet keepalives sur une connexion qui semble inutilisée. Une valeur de 0 revient à utiliser la valeur système par défaut. Ce paramètre est seulement supporté par les systèmes qui supportent le symbole TCP_KEEPINTVL et sur Windows ; sur les autres systèmes, ce paramètre doit valoir zéro. Pour les sessions connectées via une socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

Note

Sur Windows, une valeur de 0 configurera ce paramètre à une seconde car Windows ne fournit pas un moyen de lire la valeur par défaut du système.

`tcp_keepalives_count` (integer)

Indique le nombre de paquets TCP keepalive packets à envoyer sur une connexion qui semble inutilisée. Une valeur de 0 revient à utiliser la valeur système par défaut. Ce paramètre est seulement supporté par les systèmes qui supportent le symbole TCP_KEEPCNT ; sur les autres systèmes, ce paramètre doit valoir zéro. Pour les sessions connectées via une socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

Note

Ce paramètre n'est pas supporté sur Windows et doit donc valoir zéro.

18.3.2. Sécurité et authentification

`authentication_timeout` (integer)

Temps maximum pour terminer l'authentification du client, en secondes. Si un client n'a pas terminé le protocole d'authentification dans ce délai, le serveur ferme la connexion. Cela protège le serveur des clients bloqués occupant une connexion indéfiniment. La valeur par défaut est d'une minute. Ce paramètre peut être configuré au lancement du serveur et dans le fichier `postgresql.conf`.

`ssl` (boolean)

Active les connexions SSL. Lire la Section 17.8, « Connexions tcp/ip sécurisées avec ssl » avant de l'utiliser. Désactivé par défaut. Ce paramètre ne peut être configuré qu'au lancement du serveur. La communication SSL n'est possible qu'avec des connexions TCP/IP.

`ssl_ciphers` (string)

Indique une liste de chiffrements SSL dont l'utilisation est autorisée sur des connexions sécurisées. Voir la page de manuel openssl pour la liste des chiffrements supportés. Ce paramètre n'est pas disponible si le serveur n'a pas été compilé avec le support de SSL.

`ssl_renegotiation_limit` (integer)

Specifies how much data can flow over an SSL-encrypted connection before renegotiation of the session keys will take place. Renegotiation decreases an attacker's chances of doing cryptanalysis when large amounts of traffic can be examined, but it also carries a large performance penalty. The sum of sent and received traffic is used to check the limit. If this parameter is set to 0, renegotiation is disabled. The default is `512MB`.

Note

SSL libraries from before November 2009 are insecure when using SSL renegotiation, due to a vulnerability in the SSL protocol. As a stop-gap fix for this vulnerability, some vendors shipped SSL libraries incapable of doing renegotiation. If any such libraries are in use on the client or server, SSL renegotiation should be disabled.

`password_encryption` (boolean)

Ce paramètre détermine si un mot de passe, indiqué dans CREATE USER(7) ou ALTER USER(7) sans qu'il soit précisé `ENCRYPTED` ou `UNENCRYPTED`, doit être chiffré. Actif par défaut (chiffre le mot de passe).

`krb_server_keyfile` (string)

Configure l'emplacement du fichier contenant la clé secrète du serveur Kerberos. Voir la Section 19.3.5, « Authentification Kerberos » et la Section 19.3.3, « Authentification GSSAPI » pour les détails. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`krb_srvname` (string)

Configure le nom du service Kerberos. Voir la Section 19.3.5, « Authentification Kerberos » pour les détails. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`krb_caseins_users` (boolean)

Indique si les noms des utilisateurs Kerberos et GSSAPI doivent être traités en respectant la casse. Désactivé par défaut (insensible à la casse, valeur `off`), Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`db_user_namespace` (boolean)

Active les noms d'utilisateur par base de données. Désactivé par défaut, ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Si ce paramètre est activé, les utilisateurs doivent être créés sous la forme `nomutilisateur@nom_base`. Quand `nomutilisateur` est passé par un client se connectant, `@` et le nom de la base de données sont ajoutés au nom de l'utilisateur et ce nom d'utilisateur spécifique à la base est recherché par le serveur. Lorsque des utilisateurs dont le nom contient un `@` sont créés dans l'environnement SQL, ce nom doit être placé entre guillemets.

`db_user_namespace` permet aux représentations des noms d'utilisateurs du client et du serveur de différer. Les vérifications sont toujours faites avec les noms d'utilisateurs du serveur, ce qui fait que les méthodes d'authentification doivent être configurées pour le nom d'utilisateur du serveur, pas pour celui du client. Comme `md5` utilise le nom d'utilisateur comme sel à la fois sur le client et le serveur, `md5` ne peut pas être utilisé conjointement avec `db_user_namespace`.

Ce paramètre activé, il reste possible de créer des utilisateurs globaux ordinaires. Il suffit pour cela d'ajouter `@` au nom du

client, e.g. `joe@`. Le `@` est supprimé avant que le serveur ne recherche ce nom.

Note

Cette fonctionnalité, temporaire, sera supprimée lorsqu'une solution complète sera trouvée.

18.4. Consommation des ressources

18.4.1. Mémoire

`shared_buffers` (integer)

Initialise la quantité de mémoire que le serveur de bases de données utilise comme mémoire partagée. La valeur par défaut, en général 32 Mo, peut être automatiquement abaissée si la configuration du noyau ne la supporte pas (déterminé lors de l'exécution de l'initdb). Ce paramètre doit être au minimum de 128 Ko + 16 Ko par max_connections. (Des valeurs personnalisées de BLCKSZ agissent sur ce minimum.) Des valeurs significativement plus importantes que ce minimum sont généralement nécessaires pour de bonnes performances. Ce paramètre ne peut être configuré qu'au lancement du serveur.

Si vous disposez d'un serveur dédié à la base de données, avec 1 Go de mémoire ou plus, une valeur de départ raisonnable pour ce paramètre est de 25% la mémoire de votre système. Certains cas peuvent nécessiter une valeur encore plus importante pour le `shared_buffers` mais comme PostgreSQL™ profite aussi du cache du système d'exploitation, il est peu probable qu'une allocation de plus de 40% de la mémoire fonctionnera mieux qu'une valeur plus restreinte. Des valeurs importantes pour le paramètre `shared_buffers` requièrent généralement une augmentation proportionnelle du `checkpoint_segments`, pour étendre dans le temps les écritures de grandes quantités de données, nouvelles ou modifiées.

Sur des systèmes comprenant moins d'1 Go de mémoire, un pourcentage plus restreint est approprié pour laisser une place suffisante au système d'exploitation. De plus, sur Windows, les grandes valeurs pour `shared_buffers` ne sont pas aussi efficaces. Vous pouvez avoir de meilleurs résultats en conservant un paramétrage assez bas et en utilisant le cache du système d'exploitation à la place. L'échelle habituelle pour `shared_buffers` sur des systèmes Windows va de 64 Mo à 512 Mo.

L'augmentation de ce paramètre peut obliger PostgreSQL™ à réclamer plus de mémoire partagée `System V` que ce que la configuration par défaut du système d'exploitation ne peut gérer. Voir la Section 17.4.1, « Mémoire partagée et sémaphore » pour de plus amples informations sur l'ajustement de ces paramètres, si nécessaire.

`temp_buffers` (integer)

Configure le nombre maximum de tampons temporaires utilisés par chaque session de la base de données. Ce sont des tampons locaux à la session utilisés uniquement pour accéder aux tables temporaires. La valeur par défaut est de 8 Mo. Ce paramètre peut être modifié à l'intérieur de sessions individuelles mais seulement jusqu'à la première utilisation des tables temporaires dans une session ; les tentatives suivantes de changement de cette valeur n'ont aucun effet sur cette session.

Une session alloue des tampons temporaires en fonction des besoins jusqu'à atteindre la limite donnée par `temp_buffers`. Positionner une valeur importante pour les sessions qui ne le nécessitent pas ne coûte qu'un descripteur de tampon, soit environ 64 octets, par incrément de `temp_buffers`. Néanmoins, si un tampon est réellement utilisé, 8192 autres octets sont consommés pour celui-ci (ou, plus généralement, BLCKSZ octets).

`max_prepared_transactions` (integer)

Configure le nombre maximum de transactions simultanément dans l'état « préparées » (voir PREPARE TRANSACTION(7)). Zéro, la configuration par défaut, désactive la fonctionnalité des transactions préparées Ce paramètre ne peut être configuré qu'au lancement du serveur.

Si vous ne prévoyez pas d'utiliser les transactions préparées, ce paramètre devrait être positionné à zéro pour éviter toute création accidentelle de transactions préparées. Au contraire, si vous les utilisez, il peut être intéressant de positionner `max_prepared_transactions` au minimum à au moins max_connections pour que chaque session puisse avoir sa transaction préparée.

Augmenter ce paramètre peut conduire PostgreSQL™ à réclamer plus de mémoire partagée `System V` que ne le permet la configuration par défaut du système d'exploitation. Voir la Section 17.4.1, « Mémoire partagée et sémaphore » pour les informations concernant la façon d'ajuster ces paramètres, si nécessaire.

Lors de l'exécution d'un serveur en attente, vous devez configurer ce paramètre à la même valeur ou à une valeur plus importante que sur le serveur maître. Sinon, des requêtes pourraient ne pas être autorisées sur le serveur en attente.

`work_mem` (integer)

Indique la quantité de mémoire que les opérations de tri interne et les tables de hachage peuvent utiliser avant de basculer sur

des fichiers disque temporaires. La valeur par défaut est de 1 Mo. Pour une requête complexe, il peut y avoir plusieurs opérations de tri ou de hachage exécutées en parallèle ; chacune peut utiliser de la mémoire à hauteur de cette valeur avant de commencer à placer les données dans des fichiers temporaires. De plus, de nombreuses sessions peuvent exécuter de telles opérations simultanément. La mémoire totale utilisée peut, de ce fait, atteindre plusieurs fois la valeur de `work_mem` ; il est nécessaire de garder cela à l'esprit lors du choix de cette valeur. Les opérations de tri sont utilisées pour `ORDER BY`, `DISTINCT` et les jointures de fusion. Les tables de hachage sont utilisées dans les jointures de hachage, les agrégations et le traitement des sous-requêtes `IN` fondés sur le hachage.

`maintenance_work_mem` (integer)

Indique la quantité maximale de mémoire que peuvent utiliser les opérations de maintenance telles que **VACUUM**, **CREATE INDEX** et **ALTER TABLE ADD FOREIGN KEY**. La valeur par défaut est de 16 Mo. Puisque seule une de ces opérations peut être exécutée à la fois dans une session et que, dans le cadre d'un fonctionnement normal, peu d'opérations de ce genre sont exécutées concurrentiellement sur une même installation, il est possible d'initialiser cette variable à une valeur bien plus importante que `work_mem`. Une grande valeur peut améliorer les performances des opérations VACUUM et de la restauration des sauvegardes.

Quand autovacuum fonctionne, un maximum de autovacuum_max_workers fois cette quantité de mémoire peut être utilisé. Il convient donc de s'assurer de ne pas configurer la valeur par défaut de façon trop importante.

`max_stack_depth` (integer)

Indique la profondeur maximale de la pile d'exécution du serveur. La configuration idéale pour ce paramètre est la limite réelle de la pile assurée par le noyau (configurée par `ulimit -s` ou équivalent local) à laquelle est soustraite une marge de sécurité d'un Mo environ. La marge de sécurité est nécessaire parce que la profondeur de la pile n'est pas vérifiée dans chaque routine du serveur mais uniquement dans les routines clés potentiellement récursives telles que l'évaluation d'une expression. Le paramétrage par défaut est de 2 Mo, valeur faible qui implique peu de risques. Néanmoins, elle peut s'avérer trop petite pour autoriser l'exécution de fonctions complexes. Seuls les superutilisateurs peuvent modifier ce paramètre.

Configurer ce paramètre à une valeur plus importante que la limite réelle du noyau signifie qu'une fonction récursive peut occasionner un arrêt brutal d'un processus serveur particulier. Sur les plateformes où PostgreSQL™ peut déterminer la limite du noyau, il interdit de positionner cette variable à une valeur inadéquate. Néanmoins, toutes les plateformes ne fournissent pas cette information, et une grande attention doit être portée au choix de cette valeur.

18.4.2. Usage des ressources du noyau

`max_files_per_process` (integer)

Positionne le nombre maximum de fichiers simultanément ouverts par sous-processus serveur. La valeur par défaut est de 1000 fichiers. Si le noyau assure une limite par processus, il n'est pas nécessaire de s'intéresser à ce paramètre. Toutefois, sur certaines plateformes (notamment les systèmes BSD) le noyau autorise les processus individuels à ouvrir plus de fichiers que le système ne peut effectivement en supporter lorsqu'un grand nombre de processus essayent tous d'ouvrir ce nombre de fichiers. Si le message « Too many open files » (« Trop de fichiers ouverts ») apparaît, il faut essayer de réduire ce paramètre. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`shared_preload_libraries` (string)

Indique les bibliothèques partagées à précharger au démarrage du serveur. Par exemple, `'$libdir/malib'` implique le préchargement de `malib.so` (ou, sur certaines plateformes, `malib.sl`) depuis le répertoire d'installation des bibliothèques standard. Tous les noms de bibliothèques sont convertis en minuscule sauf s'ils sont compris entre des guillemets doubles. S'il faut précharger plusieurs bibliothèques, leurs noms doivent être séparés par des virgules. Ce paramètre ne peut être configuré qu'au lancement du serveur.

Les bibliothèques des langages procéduraux de PostgreSQL™ peuvent être préchargées ainsi, typiquement en utilisant la syntaxe `'$libdir/plXXX'` où `XXX` est `pgsql`, `perl`, `tcl` ou `python`.

Le préchargement d'une bibliothèque partagée permet d'éviter le temps de chargement de la bibliothèque à sa première utilisation. Toutefois, la durée de démarrage de chaque nouveau processus serveur peut augmenter légèrement, même si aucun de ces processus n'utilise la bibliothèque. Ce paramètre n'est réellement recommandé que pour les bibliothèques utilisées dans la plupart des sessions.

Note

Sur un hôte Windows, le préchargement d'une bibliothèque au lancement du serveur ne réduit pas le temps nécessaire au lancement de chaque nouveau processus serveur ; chaque processus serveur recharge toutes les bibliothèques déjà chargées. Néanmoins, `shared_preload_libraries` est toujours utile sur les hôtes

> Windows car certaines bibliothèques partagées peuvent nécessiter des opérations qui ne peuvent avoir lieu qu'au lancement du serveur (par exemple, une bibliothèque partagée peut réserver des verrous légers ou de la mémoire partagée, ce qui ne peut être fait une fois le serveur démarré).

Si une bibliothèque indiquée est introuvable, le démarrage du serveur échoue.

Chaque bibliothèque supportée par PostgreSQL possède un « bloc magique » qui est vérifié pour garantir la compatibilité. Pour cette raison, seules les bibliothèques PostgreSQL peuvent être chargées de cette façon.

18.4.3. Report du VACUUM en fonction de son coût

Lors de l'exécution des commandes VACUUM(7) et ANALYZE(7), le système maintient un compteur interne qui conserve la trace du coût estimé des différentes opérations d'entrée/sortie réalisées. Quand le coût accumulé atteint une limite (indiquée par `vacuum_cost_limit`), le processus traitant l'opération s'arrête un court moment (précisé par `vacuum_cost_delay`). Puis, il réinitialise le compteur et continue l'exécution.

Le but de cette fonctionnalité est d'autoriser les administrateurs à réduire l'impact des entrées/sorties de ces commandes en fonction de l'activité des bases de données. Nombreuses sont les situations pour lesquelles il n'est pas très important que les commandes de maintenance telles que **VACUUM** et **ANALYZE** se finissent rapidement, mais il est généralement très important que ces commandes n'interfèrent pas de façon significative avec la capacité du système à réaliser d'autres opérations sur les bases de données. Le report du VACUUM en fonction de son coût fournit aux administrateurs un moyen d'y parvenir.

Cette fonctionnalité est désactivée par défaut pour les commandes **VACUUM** lancées manuellement. Pour l'activer, la variable `vacuum_cost_delay` doit être initialisée à une valeur différente de zéro.

`vacuum_cost_delay` (integer)

Indique le temps, en millisecondes, de repos du processus quand la limite de coût a été atteinte. La valeur par défaut est zéro, ce qui désactive la fonctionnalité de report du VACUUM en fonction de son coût. Une valeur positive active cette fonctionnalité. Sur de nombreux systèmes, la résolution réelle du **sleep** est de 10 millisecondes ; configurer `vacuum_cost_delay` à une valeur qui n'est pas un multiple de 10 conduit alors au même résultat que de le configurer au multiple de 10 supérieur.

Lors d'utilisation de vacuum basée sur le coût, les valeurs appropriées pour `vacuum_cost_delay` sont habituellement assez petites, de l'ordre de 10 à 20 millisecondes. Il est préférable d'ajuster la consommation de ressource de vacuum en changeant les autres paramètres de coût de vacuum.

`vacuum_cost_page_hit` (integer)

Indique Le coût estimé du nettoyage par VACUUM d'un tampon trouvé dans le cache des tampons partagés. Cela représente le coût de verrouillage de la réserve de tampons, la recherche au sein de la table de hachage partagée et le parcours du contenu de la page. La valeur par défaut est 1.

`vacuum_cost_page_miss` (integer)

Indique le coût estimé du nettoyage par VACUUM d'un tampon qui doit être lu sur le disque. Cela représente l'effort à fournir pour verrouiller la réserve de tampons, rechercher dans la table de hachage partagée, lire le bloc désiré sur le disque et parcourir son contenu. La valeur par défaut est 10.

`vacuum_cost_page_dirty` (integer)

Indique le coût estimé de modification par VACUUM d'un bloc précédemment vide (*clean block*). Cela représente les entrées/sorties supplémentaires nécessaires pour vider à nouveau le bloc modifié (*dirty block*) sur le disque. La valeur par défaut est 20.

`vacuum_cost_limit` (integer)

Indique Le coût cumulé qui provoque l'endormissement du processus de VACUUM. La valeur par défaut est 200.

> **Note**
>
> Certaines opérations détiennent des verrous critiques et doivent donc se terminer le plus vite possible. Les reports de VACUUM en fonction du coût ne surviennent pas pendant ces opérations. De ce fait, il est possible que le coût cumulé soit bien plus important que la limite indiquée. Pour éviter des délais inutilement longs dans de tels cas, le délai réel est calculé de la façon suivante : `vacuum_cost_delay * accumulated_balance / vacuum_cost_limit` avec un maximum de `vacuum_cost_delay * 4`.

18.4.4. Processus d'écriture en arrière-plan

Il existe un processus serveur séparé appelé *background writer* dont le but est d'écrire les tampons « sales » (parce que nouveaux ou modifiés). Ce processus écrit les tampons partagés pour que les processus serveur gérant les requêtes des utilisateurs n'aient jamais ou peu fréquemment à attendre qu'une écriture se termine. Néanmoins, ce processus d'écriture en tâche de fond implique une augmentation globale de la charge des entrées/sorties disque car, quand une page fréquemment modifiée pourrait n'être écrite qu'une seule fois par CHECKPOINT, le processus d'écriture en tâche de fond pourrait l'avoir écrit plusieurs fois si cette page a été modifiée plusieurs fois dans le même intervalle. Les paramètres discutés dans cette sous-section peuvent être utilisés pour configurer finement son comportement pour les besoins locaux.

`bgwriter_delay` (integer)

Indique le délai entre les tours d'activité du processus d'écriture en arrière-plan. À chaque tour, le processus écrit un certain nombre de tampons modifiés (contrôlable par les paramètres qui suivent). Puis, il s'endort pour `bgwriter_delay` millisecondes et recommence. La valeur par défaut est de 200 millisecondes. Sur de nombreux systèmes, la résolution réelle du **sleep** est de 10 millisecondes ; positionner `bgwriter_delay` à une valeur qui n'est pas un multiple de 10 peut avoir le même résultat que de le positionner au multiple de 10 supérieur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`bgwriter_lru_maxpages` (integer)

Nombre maximum de tampons qui peuvent être écrits à chaque tour par le processus d'écriture en tâche de fond. Le configurer à zéro désactive l'écriture en tâche de fond (sauf en ce qui concerne l'activité des points de vérification). La valeur par défaut est de 100 tampons. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`bgwriter_lru_multiplier` (floating point)

Le nombre de tampons sales écrits à chaque tour est basé sur le nombre de nouveaux tampons qui ont été requis par les processus serveur lors des derniers tours. Le besoin récent moyen est multiplié par `bgwriter_lru_multiplier` pour arriver à une estimation du nombre de tampons nécessaire au prochain tour. Les tampons sales sont écrits pour qu'il y ait ce nombre de tampons propres, réutilisables. (Néanmoins, au maximum `bgwriter_lru_maxpages` tampons sont écrits par tour.) De ce fait, une configuration de 1.0 représente une politique d'écriture « juste à temps » d'exactement le nombre de tampons prédits. Des valeurs plus importantes fournissent une protection contre les pics de demande, alors qu'une valeur plus petite laisse intentionnellement des écritures aux processus serveur. La valeur par défaut est de 2. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Des valeurs plus faibles de `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier` réduisent la charge supplémentaire des entrées/sorties induite par le processus d'écriture en arrière-plan. En contrepartie, la probabilité que les processus serveurs effectuent plus d'écritures par eux-mêmes augmente, ce qui retarde les requêtes interactives.

18.4.5. Comportement asynchrone

`effective_io_concurrency` (integer)

Positionne le nombre d'opérations d'entrées/sorties disque concurrentes que PostgreSQL™ pense pouvoir exécuter simultanément. Augmenter cette valeur va augmenter le nombre d'opérations d'entrée/sortie que chaque session PostgreSQL™ individuelle essayera d'exécuter en parallèle. Les valeurs autorisées vont de 1 à 1000, ou zéro pour désactiver l'exécution de requêtes d'entrée/sortie asynchrones.

Un bon point départ pour ce paramètre est le nombre de disques que comprend un agrégat par bande RAID 0 ou miroir RAID 1 utilisé pour la base de données. (Pour du RAID 5, le disque de parité ne devrait pas être pris en compte.) Toutefois, si la base est souvent occupée par de nombreuses requêtes exécutées dans des sessions concurrentes, des valeurs plus basses peuvent être suffisantes pour maintenir le groupe de disques occupé. Une valeur plus élevée que nécessaire pour maintenir les disques occupés n'aura comme seul résultat que de surcharger le processeur.

Pour des systèmes plus exotiques, comme du stockage mémoire ou un groupement RAID qui serait limité par la bande passante du bus, la valeur correcte pourrait être le nombre de chemins d'entrées/sorties disponibles. Il pourrait être nécessaire d'expérimenter afin d'obtenir la valeur idéale.

Les entrées/sorties asynchrones dépendent de la présence d'une fonction `posix_fadvise` efficace, ce que n'ont pas certains systèmes d'exploitation. Si la fonction n'est pas présente, alors positionner ce paramètre à une valeur autre que zéro entraînera une erreur. Sur certains systèmes (par exemple Solaris), cette fonction est présente mais n'a pas d'effet.

18.5. Write Ahead Log

Voir aussi la Section 29.4, « Configuration des journaux de transaction » pour les détails concernant l'optimisation des WAL.

18.5.1. Paramètres

`wal_level` (enum)

`wal_level` détermine la quantité d'informations écrite dans les journaux de transactions. La valeur par défaut est `minimal`, ce qui permet d'écrire seulement les informations nécessaires pour survivre à un arrêt brutal ou à un arrêt immédiat. `archive` ajoute quelques enregistrements supplémentaires pour permettre l'archivage des journaux de transactions. `hot_standby` en ajoute encore plus pour permettre l'exécution de requêtes en lecture seule sur le serveur en attente. Ce paramètre peut seulement être configuré au lancement du serveur.

Au niveau `minimal`, certains enregistrements dans les journaux de transactions peuvent être ignorés sans risque, ce qui rend ces opérations plus rapides (voir Section 14.4.7, « Désactiver l'archivage des journaux de transactions et la réplication en flux »). Les opérations pour lesquelles cette optimisation s'applique incluent :

CREATE TABLE AS
CREATE INDEX
CLUSTER
COPY dans des tables qui ont été créées ou tronquées dans la même transaction

Mais, du coup, les journaux au niveau minimal ne contiennent pas suffisamment d'informations pour reconstruire les données à partir d'une sauvegarde de base et des journaux de transactions. Donc, les niveaux `archive` ou `hot_standby` doivent être utilisés pour activer l'archivage des journaux de transactions (archive_mode) et la réplication en flux.

Au niveau `hot_standby`, en plus des informations que trace déjà le niveau `archive`, plus d'informations sont nécessaires pour reconstruire le statut des transactions en cours à partir du journal de transactions. Pour activer les requêtes en lecture seule sur un serveur en attente, `wal_level` doit être configuré à `hot_standby` sur le serveur principal et hot_standby doit être activé sur le serveur en attente. Il existe une différence mesurable de performances entre l'utilisation des niveaux `hot_standby` et `archive`, donc un retour d'expérience serait apprécié si l'impact est ressenti en production.

`fsync` (boolean)

Si ce paramètre est activé, le serveur PostgreSQL™ tente de s'assurer que les mises à jour sont écrites physiquement sur le disque à l'aide d'appels système `fsync()` ou de méthodes équivalentes (voir wal_sync_method). Cela permet de s'assurer que le cluster de bases de données peut revenir à un état cohérent après une panne matérielle ou du système d'exploitation.

Bien que désactiver `fsync` améliore fréquemment les performances, cela peut avoir pour conséquence une corruption des données non récupérables dans le cas d'une perte de courant ou d'un crash du système. Donc, il est seulement conseillé de désactiver `fsync` si vous pouvez facilement recréer la base de données complète à partir de données externes.

Quelques exemples de circonstances permettant de désactiver `fsync` : le chargement initial d'une nouvelle instance à partir d'une sauvegarde, l'utilisation de l'instance pour traiter un flot de données après quoi la base sera supprimée puis recréée, la création d'un clone d'une base en lecture seule, clone qui serait recréé fréquemment et n'est pas utilisé pour du failover. La haute qualité du matériel n'est pas une justification suffisante pour désactiver `fsync`.

Dans de nombreuses situations, désactiver synchronous_commit pour les transactions non critiques peut fournir une grande partie des performances de la désactivation de `fsync`, sans les risques associés de corruption de données.

`fsync` ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Si ce paramètre est désactivé (`off`), il est intéressant de désactiver aussi full_page_writes.

`synchronous_commit` (boolean)

Indique si la validation des transactions doit attendre l'écriture des enregistrements WAL avant que la commande ne renvoie une indication de « réussite » au client. La configuration par défaut, et la plus sûre, est `on`. Quand ce paramètre est désactivé (`off`), il peut exister un délai entre le moment où le succès est rapporté et le moment où la transaction est vraiment protégée d'un arrêt brutal du serveur. (Le délai maximum est de trois fois wal_writer_delay.) Contrairement à fsync, la configuration de ce paramètre à `off` n'implique aucun risque d'incohérence dans la base de données : un arrêt brutal du système d'exploitation ou d'une base de données peut résulter en quelques transactions récentes prétendument validées perdues malgré tout. Cependant, l'état de la base de données est identique à celui obtenu si les transactions avaient été correctement annulées. C'est pourquoi la désactivation de `synchronous_commit` est une alternative utile quand la performance est plus importante que la sûreté de la transaction. Pour plus de discussion, voir Section 29.3, « Validation asynchrone (Asynchronous Commit) ».

Ce paramètre peut être changé à tout moment ; le comportement pour toute transaction est déterminé par la configuration en cours lors de la validation. Il est donc possible et utile d'avoir certaines validations validées en synchrone et d'autres en asynchrone. Par exemple, pour réaliser une validation asynchrone de transaction à plusieurs instructions avec une valeur par défaut inverse, on exécute l'instruction **SET LOCAL synchronous_commit TO OFF** dans la transaction.

`wal_sync_method` (enum)

Méthode utilisée pour forcer les mises à jour des WAL sur le disque. Si `fsync` est désactivé, alors ce paramètre est inapplicable, car les mises à jour des journaux de transactions ne sont pas du tout forcées. Les valeurs possibles sont :

- `open_datasync` (écrit les fichiers WAL avec l'option O_DSYNC de `open()`)
- `fdatasync` (appelle `fdatasync()` à chaque validation)
- `fsync_writethrough` (appelle `fsync()` à chaque validation, forçant le mode *write-through* de tous les caches disque en écriture)
- `fsync` (appelle `fsync()` à chaque validation)
- `open_sync` (écrit les fichiers WAL avec l'option O_SYNC de `open()`)

Ces options ne sont pas toutes disponibles sur toutes les plateformes. La valeur par défaut est la première méthode de la liste ci-dessus supportée par la plateforme. Les options `open_*` utilisent aussi `O_DIRECT` s'il est disponible. L'outil `src/tools/fsync` disponible dans le code source de PostgreSQL permet de tester les performances des différentes méthodes de synchronisation. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`full_page_writes` (boolean)

Quand ce paramètre est activé, le serveur écrit l'intégralité du contenu de chaque page disque dans les WAL lors de la première modification de cette page qui intervient après un point de vérification. C'est nécessaire car l'écriture d'une page lors d'un plantage du système d'exploitation peut n'être que partielle, ce qui conduit à une page sur disque qui contient un mélange d'anciennes et de nouvelles données. Les données de modification de niveau ligne stockées habituellement dans les WAL ne sont pas suffisantes pour restaurer complètement une telle page lors de la récupération qui suit la panne. Le stockage de l'image de la page complète garantit une restauration correcte de la page, mais au prix d'un accroissement de la quantité de données à écrire dans les WAL. (Parce que la relecture des WAL démarre toujours à un point de vérification, il suffit de faire cela lors de la première modification de chaque page survenant après un point de vérification. De ce fait, une façon de réduire le coût d'écriture de pages complètes consiste à augmenter le paramètre réglant les intervalles entre points de vérification.)

La désactivation de ce paramètre accélère les opérations normales, mais peut aboutir soit à une corruption impossible à corriger soit à une corruption silencieuse, après un échec système. Les risques sont similaires à la désactivation de `fsync`, bien que moindres. Sa désactivation devrait se faire en se basant sur les mêmes recommandations que cet autre paramètre.

La désactivation de ce paramètre n'affecte pas l'utilisation de l'archivage des WAL pour la récupération d'un instantané, aussi appelé PITR (voir Section 24.3, « Archivage continu et récupération d'un instantané (PITR) »).

Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Activé par défaut (`on`).

`wal_buffers` (integer)

Quantité de mémoire utilisée en mémoire partagée pour les données WAL. La valeur par défaut est de 64 Ko. Ce paramètre nécessite uniquement d'être assez important pour contenir toutes les données WAL engendrées par une transaction typique, car les données sont écrites sur le disque à chaque validation de transaction. Ce paramètre ne peut être configuré qu'au lancement du serveur.

L'augmentation de ce paramètre peut conduire PostgreSQL™ à réclamer plus de tampons partagés `System V` que ne le permet la configuration par défaut du système d'exploitation. Voir la Section 17.4.1, « Mémoire partagée et sémaphore » pour les informations sur la façon d'ajuster ces paramètres, si nécessaire.

`wal_writer_delay` (integer)

Indique le délai entre les tours d'activité pour l'enregistreur des WAL. À chaque tour, l'enregistreur place les WAL sur disque. Il s'endort ensuite pour `wal_writer_delay` millisecondes et recommence. La valeur par défaut est de 200 millisecondes (`200ms`). Pour de nombreux systèmes, la résolution réelle du **sleep** est de 10 millisecondes ; configurer `wal_writer_delay` à une valeur qui n'est pas un multiple de 10 a le même résultat que de le configurer au multiple de 10 immédiatement supérieur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`commit_delay` (integer)

Délai entre l'enregistrement d'une validation dans le tampon WAL et le vidage du tampon sur le disque, en microsecondes. Un délai différent de zéro peut autoriser la validation de plusieurs transactions en un seul appel système `fsync()`, si la charge système est assez importante pour que des transactions supplémentaires soient prêtes dans l'intervalle donné. Mais le délai est perdu si aucune autre transaction n'est prête à être validée. De ce fait, le délai n'est traité que si, au minimun, `commit_siblings` autres transactions sont actives au moment où le processus serveur a écrit son enregistrement de validation.

La valeur par défaut est zéro (pas de délai).

`commit_siblings` (integer)

Nombre minimum de transactions concurrentes ouvertes en même temps nécessaires avant d'attendre le délai `commit_delay`. Une valeur plus importante rend plus probable le fait qu'au moins une autre transaction soit prête à valider pendant le délai. La valeur par défaut est de cinq transactions.

18.5.2. Points de vérification

`checkpoint_segments` (integer)

Nombre maximum de journaux de transaction entre deux points de vérification automatique des WAL (chaque segment fait normalement 16 Mo). La valeur par défaut est de trois segments. Augmenter ce paramètre peut accroitre le temps nécessaire à une récupération après un arrêt brutal. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`checkpoint_timeout` (integer)

Temps maximum entre deux points de vérification automatique des WAL, en secondes. La valeur par défaut est de cinq minutes. Augmenter ce paramètre peut accroitre le temps nécessaire à une récupération après un arrêt brutal. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`checkpoint_completion_target` (floating point)

Précise la cible pour la fin du CHECKPOINT, sous la format d'une fraction de temps entre deux CHECKPOINT. La valeur par défaut est 0.5. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`checkpoint_warning` (integer)

Si deux points de vérification imposés par le remplissage des fichiers segment interviennent dans un délai plus court que celui indiqué par ce paramètre (ce qui laisse supposer qu'il faut augmenter la valeur du paramètre `checkpoint_segments`), un message est écrit dans le fichier de traces du serveur. Par défaut, 30 secondes. Une valeur nulle (0) désactive cet avertissement. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

18.5.3. Archivage

`archive_mode` (boolean)

Quand `archive_mode` est activé, les segments WAL remplis peuvent être archivés en configurant archive_command. `archive_mode` et `archive_command` sont des variables séparées de façon à ce que `archive_command` puisse être modifiée sans quitter le mode d'archivage. Ce paramètre ne peut être configuré qu'au lancement du serveur. `wal_level` doit être configuré à `archive` ou `hot_standby` pour activer `archive_mode`.

`archive_command` (string)

Commande shell à exécuter pour archiver un segment terminé de la série des fichiers WAL. Tout `%p` dans la chaîne est remplacé par le chemin du fichier à archiver et tout `%f` par le seul nom du fichier. (Le chemin est relatif au répertoire de travail du serveur, c'est-à-dire le répertoire de données du cluster.) `%%` est utilisé pour intégrer un caractère `%` dans la commande. Il est important que la commande renvoit un code zéro seulement si elle a réussit l'archivage. Pour plus d'informations, voir Section 24.3.1, « Configurer l'archivage WAL ».

Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est ignoré sauf si `archive_mode` a été activé au lancement du serveur. Si `archive_command` est une chaîne vide (la valeur par défaut) alors que `archive_mode` est activé, alors l'archivage des journaux de transactions est désactivé temporairement mais le serveur continue d'accumuler les fichiers des journaux de transactions dans l'espoir qu'une commande lui soit rapidement proposée. Configurer `archive_command` à une commande qui ne fait rien tout en renvoyant true, par exemple `/bin/true` (`REM` sur Windows), désactive l'archivage mais casse aussi la chaîne des fichiers des journaux de transactions nécessaires pour la restauration d'une archive. Cela ne doit donc être utilisé quand lors de circonstances inhabituelles.

`archive_timeout` (integer)

Le archive_command n'est appelé que pour les segments WAL remplis. De ce fait, si le serveur n'engendre que peu de trafic WAL (ou qu'il y a des périodes de plus faible activité), il se peut qu'un long moment s'écoule entre la fin d'une transaction et son archivage certain. Pour limiter l'âge des données non encore archivées, `archive_timeout` peut être configuré pour forcer le serveur à basculer périodiquement sur un nouveau segment WAL. Lorsque ce paramètre est positif, le serveur bascule sur un nouveau segment à chaque fois que `archive_timeout` secondes se sont écoulées depuis le dernier change-

ment de segment et qu'il n'y a pas eu d'activité de la base de données, y compris un seul CHECKPOINT. (augmenter `checkpoint_timeout` réduira les CHECKPOINT inutiles sur un système non utilisé.) Les fichiers archivés clos par anticipation suite à une bascule imposée sont toujours de la même taille que les fichiers complets. Il est donc déconseillé de configurer un temps très court pour `archive_timeout` -- cela va faire exploser la taille du stockage des archives. Un paramétrage d'`archive_timeout` de l'ordre de la minute est habituellement raisonnable. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

18.5.4. Réplication en flux

Ces paramètres contrôlent le comportement de la fonctionnalité interne de *réplication en flux*. Ces paramètres sont à configurer sur le serveur maître pour envoyer les données de réplication sur un ou plusieurs serveurs en attente.

`max_wal_senders` (integer)
: Indique le nombre maximum de connexions concurrentes à partir des serveurs en attente (c'est-à-dire le nombre maximum de processus walsender en cours d'exécution). La valeur par défaut est zéro, signifiant que la réplication est désactivée. Les processus walsender sont lançables jusquà atteindre le nombre total de connexions, donc ce paramètre ne peut pas être supérieur à max_connections. Ce paramètre peut seulement être configuré au lancement du serveur. `wal_level` doit être configuré à `archive` ou `hot_standby` pour permettre les connexions des serveurs en attente.

`wal_sender_delay` (integer)
: Précise le délai entre deux tours d'activité des processus walsender. À chaque tour, le processus walsender envoie tous les enregistrements WAL accumulés depuis sont dernier tour. Il s'endort ensuite pour `wal_sender_delay` millisecondes et recommence. La valeur par défaut est de 200 millisecondes (`200ms`). Notez que sur de nombreux systèmes la résolution réelle du délai d'endormissement en de 10 millisecondes ; configurer `wal_sender_delay` à une valeur qui n'est pas un multiple de 10 pourrait avoir le même résultat que de le configurer à la valeur suivante multiple de 10. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`wal_keep_segments` (integer)
: Indique le nombre minimum de journaux de transactions passés à conserver dans le répertoire `pg_xlog`, au cas où un serveur en attente a besoin de les récupérer pour la réplication en flux. Chaque fichier fait normalement 16 Mo. Si un serveur en attente connecté au primaire se laisse distancer par le primaire pour plus de `wal_keep_segments` fichiers, le primaire pourrait supprimer un journal de transactions toujours utile au serveur en attente, auquel cas la connexion de réplication serait fermée. (Néanmoins, le serveur en attente peut continuer la restauration en récupérant le segment des archives si l'archivage des journaux de transactions est utilisé.)

  Cela configure seulement le nombre minimum de fichiers à conserver dans `pg_xlog` ; le système pourrait avoir besoin de conserver plus de fichiers pour l'archivage ou pour restaurer à partir d'un CHECKPOINT. Si `wal_keep_segments` vaut zéro (ce qui est la valeur par défaut), le système ne conserve aucun fichier supplémentaire pour les serveurs en attente et le nombre des anciens journaux disponibles pour les serveurs en attente est seulement basé sur l'emplacement du dernier CHECKPOINT ainsi que sur l'état de l'archivage des journaux de transactions. Ce paramètre n'a aucun effet sur les restartpoints. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`vacuum_defer_cleanup_age` (integer)
: Indique le nombre de transactions pendant lesquelles les **VACUUM** et les mises à jour HOT reporteront à plus tard le nettoyage des versions de lignes mortes. La valeur par défaut est de zéro transaction. Cela veut dire que les versions de lignes mortes peuvent être supprimées dès que possible, autrement dit à partir du moment où elles ne sont plus visibles par les transactions en cours d'exécution. Vous pourriez augmenter la valeur de ce paramètre sur un serveur maître qui accepte des serveurs en attente de type hotstandby, comme décrit dans Section 25.5, « Hot Standby ». Ceci donne plus de temps aux requêtes sur les serveurs hotstandby pour qu'elles se terminent avec succès, sans conflit relatif à un nettoyage des lignes. Néanmoins, comme la valeur est mesurée en terme de nombres de transactions en écriture survenant sur le serveur maître, il est difficile de prédire le temps supplémentaire que cela met à disposition des requêtes sur les serveurs hotstandby. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

18.5.5. Serveurs en attente

Ces paramètres contrôlent le comportement d'un serveur en attente pour qu'il puisse recevoir les données de réplication.

`hot_standby` (boolean)
: Indique si vous pouvez vous connecter et exécuter des requêtes lors de la restauration, comme indiqué dans Section 25.5, « Hot Standby ». Désactivé par défaut. Ce paramètre peut seulement être configuré au lancement du serveur. Il a un effet seulement lors de la restauration des archives ou en mode serveur en attente.

`max_standby_archive_delay` (integer)

Quand le Hot Standby est activé, ce paramètre détermine le temps maximum d'attente que le serveur esclave doit observer avant d'annuler les requêtes en lecture qui entreraient en conflit avec des enregistrements des journaux de transactions à appliquer, comme c'est décrit dans Section 25.5.2, « Gestion des conflits avec les requêtes ». `max_standby_archive_delay` est utilisé quand les données de journaux de transactions sont lues à partir des archives de journaux de transactions (et du coup accuse un certain retard par rapport au serveur maître). La valeur par défaut est de 30 secondes. L'unité est la milliseconde si cette dernière n'est pas spécifiée. Une valeur de -1 autorise le serveur en attente à attendre indéfiniment la fin d'exécution des requêtes en conflit. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Notez que `max_standby_archive_delay` ne correspond pas au temps d'exécution maximum d'une requête avant son annulation ; il s'agit plutôt du temps maximum autorisé pour enregistrer les données d'un journal de transactions. Donc, si une requête a occasionné un délai significatif au début du traitement d'un journal de transactions, les requêtes suivantes auront un délai beaucoup moins important.

`max_standby_streaming_delay` (integer)

Quand Hot Standby est activé, ce paramètre détermine le délai maximum d'attente que le serveur esclave doit observer avant d'annuler les requêtes en lecture qui entreraient en conflit avec les enregistrements de transactions à appliquer, comme c'est décrit dans Section 25.5.2, « Gestion des conflits avec les requêtes ». `max_standby_streaming_delay` est utilisé quand les données des journaux de données sont reçues via la connexion de la réplication en flux. La valeur par défaut est de 30 secondes. L'unité est la milliseconde si cette dernière n'est pas spécifiée. Une valeur de -1 autorise le serveur en attente à attendre indéfiniment la fin d'exécution des requêtes en conflit. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Notez que `max_standby_streaming_delay` ne correspond pas au temps d'exécution maximum d'une requête avant son annulation ; il s'agit plutôt du temps maximum autorisé pour enregistrer les données d'un journal de transactions une fois qu'elles ont été récupérées du serveur maître. Donc, si une requête a occasionné un délai significatif au début du traitement d'un journal de transactions, les requêtes suivantes auront un délai beaucoup moins important.

18.6. Planification des requêtes

18.6.1. Configuration de la méthode du planificateur

Ces paramètres de configuration fournissent une méthode brutale pour influencer les plans de requête choisis par l'optimiseur de requêtes. Si le plan choisi par défaut par l'optimiseur pour une requête particulière n'est pas optimal, une solution *temporaire* peut provenir de l'utilisation de l'un de ces paramètres de configuration pour forcer l'optimiseur à choisir un plan différent. De meilleures façons d'améliorer la qualité des plans choisis par l'optimiseur passent par l'ajustement des constantes de coût du planificateur (voir Section 18.6.2, « Constantes de coût du planificateur »), le lancement plus fréquent de ANALYZE(7), l'augmentation de la valeur du paramètre de configuration default_statistics_target et l'augmentation du nombre de statistiques récupérées pour des colonnes spécifiques en utilisant **ALTER TABLE SET STATISTICS**.

`enable_bitmapscan` (boolean)

Active ou désactive l'utilisation des plans de parcours de bitmap (*bitmap-scan*) par le planificateur de requêtes. Activé par défaut (`on`).

`enable_hashagg` (boolean)

Active ou désactive l'utilisation des plans d'agrégation hachée (*hashed aggregation*) par le planificateur. Activé par défaut (`on`).

`enable_hashjoin` (boolean)

Active ou désactive l'utilisation des jointures de hachage (*hash-join*) par le planificateur. Activé par défaut (`on`).

`enable_indexscan` (boolean)

Active ou désactive l'utilisation des parcours d'index (*index-scan*) par le planificateur. Activé par défaut (`on`).

`enable_material` (boolean)

Active ou désactive l'utilisation de la matérialisation par le planificateur. Il est impossible de supprimer complètement son utilisation mais la désactivation de cette variable permet d'empêcher le planificateur d'insérer des nœuds de matérialisation sauf dans le cas où son utilisation est obligatoire pour des raisons de justesse de résultat. Activé par défaut (`on`).

`enable_mergejoin` (boolean)

Active ou désactive l'utilisation des jointures de fusion (*merge-join*)par le planificateur. Activé par défaut (`on`).

`enable_nestloop` (boolean)

Active ou désactive l'utilisation des jointures de boucles imbriquées (*nested-loop*) par le planificateur. Il n'est pas possible de supprimer complètement les jointures de boucles imbriquées mais la désactivation de cette variable décourage le planificateur d'en utiliser une si d'autres méthodes sont disponibles. Activé par défaut (`on`).

`enable_seqscan` (boolean)

Active ou désactive l'utilisation des parcours séquentiels (*sequential scan*) par le planificateur. Il n'est pas possible de supprimer complètement les parcours séquentiels mais la désactivation de cette variable décourage le planificateur d'n utiliser un si d'autres méthodes sont disponibles. Activé par défaut (`on`).

`enable_sort` (boolean)

Active ou désactive l'utilisation des étapes de tri explicite par le planificateur. Il n'est pas possible de supprimer complètement ces tris mais la désactivation de cette variable décourage le planificateur d'en utiliser un si d'autres méthodes sont disponibles. Activé par défaut (`on`).

`enable_tidscan` (boolean)

Active ou désactive l'utilisation des parcours de TID par le planificateur. Activé par défaut (`on`).

18.6.2. Constantes de coût du planificateur

Les variables de *coût* décrites dans cette section sont mesurées sur une échelle arbitraire. Seules leurs valeurs relatives ont un intérêt. De ce fait, augmenter ou diminuer leurs valeurs d'un même facteur n'occasione aucun changement dans les choix du planificateur. Par défaut, ces variables de coût sont basées sur le coût de récupération séquentielle d'une page ; c'est-à-dire que `seq_page_cost` est, par convention, positionné à `1.0` et les autres variables de coût sont configurées relativement à cette référence. Il est toutefois possible d'utiliser une autre échelle, comme les temps d'exécution réels en millisecondes sur une machine particulière.

Note

Il n'existe malheureuresement pas de méthode bien définie pour déterminer les valeurs idéales des variables de coût. Il est préférable de les considérer comme moyennes sur un jeu complet de requêtes d'une installation particulière. Cela signifie que modifier ces paramètres sur la seule base de quelques expériences est très risqué.

`seq_page_cost` (floating point)

Initialise l'estimation faite par le planificateur du coût de récupération d'une page disque incluse dans une série de récupérations séquentielles. La valeur par défaut est 1.0. Cette valeur peut être surchargé par un tablespace spécifique en configurant le paramètre du même nom pour un tablespace (voir ALTER TABLESPACE(7)).

`random_page_cost` (floating point)

Initialise l'estimation faite par le planificateur du coût de récupération non-séquentielle d'une page disque. Mesurée comme un multiple du coût de récupération d'une page séquentielle, sa valeur par défaut est 4.0. Cette valeur peut être surchargé par un tablespace spécifique en configurant le paramètre du même nom pour un tablespace (voir ALTER TABLESPACE(7)).

Réduire cette valeur par rapport à `seq_page_cost` incite le système à privilégier les parcours d'index ; l'augmenter donne l'impression de parcours d'index plus coûteux. Les deux valeurs peuvent être augmentées ou diminuées concomitament pour modifier l'importance des coûts d'entrées/sorties disque par rapport aux coûts CPU, décrits par les paramètres qui suivent.

Astuce

Bien que le système permette de configurer `random_page_cost` à une valeur inférieure à celle de `seq_page_cost`, cela n'a aucun intérêt. En revanche, les configurer à des valeurs identiques prend tout son sens si la base tient entièrement dans le cache en RAM. En effet, dans ce cas, il n'est pas pénalisant d'atteindre des pages qui ne se suivent pas. De plus, dans une base presque entièrement en cache, ces valeurs peuvent être abaissées relativement aux paramètres CPU car le coût de récupération d'une page déjà en RAM est bien moindre à celui de sa récupération sur disque.

`cpu_tuple_cost` (floating point)

Initialise l'estimation faite par le planificateur du coût de traitement de chaque ligne lors d'une requête. La valeur par défaut est 0.01.

`cpu_index_tuple_cost` (floating point)

Initialise l'estimation faite par le planificateur du coût de traitement de chaque entrée de l'index lors d'un parcours d'index. La valeur par défaut est 0.005.

`cpu_operator_cost` (floating point)

Initialise l'estimation faite par le planificateur du coût de traitement de chaque opérateur ou fonction exécutée dans une requête. La valeur par défaut est 0.0025.

`effective_cache_size` (integer)

Initialise l'estimation faite par le planificateur de la taille réelle du cache disque disponible pour une requête. Ce paramètre est lié à l'estimation du coût d'utilisation d'un index ; une valeur importante favorise les parcours d'index, une valeur faible les parcours séquentiels. Pour configurer ce paramètre, il est important de considérer à la fois les tampons partagés de PostgreSQL™ et la portion de cache disque du noyau utilisée pour les fichiers de données de PostgreSQL™. Il faut également tenir compte du nombre attendu de requêtes concurrentes sur des tables différentes car elles partagent l'espace disponible. Ce paramètre n'a pas d'inluence sur la taille de la mémoire partagée allouée par PostgreSQL™, et ne réserve pas non plus le cache disque du noyau ; il n'a qu'un rôle estimatif. La valeur par défaut est de 128 Mo.

18.6.3. Optimiseur génétique de requêtes

L'optimiseur génétique de requête (GEQO) est un algorithme qui fait la planification d'une requête en utilisant une recherche heuristique. Cela réduit le temps de planification pour les requêtes complexes (celles qui joignent de nombreuses relations), au prix de plans qui sont quelques fois inférieurs à ceux trouver par un algorithme exhaustif. Pour plus d'informations, voir Chapitre 50, Optimiseur génétique de requêtes (*Genetic Query Optimizer*).

`geqo` (boolean)

Active ou désactive l'optimisation génétique des requêtes. Activé par défaut. Il est généralement préférable de ne pas le désactiver sur un serveur en production. La variable `geqo_threshold` fournit un moyen plus granulaire de désactiver le GEQO.

`geqo_threshold` (integer)

L'optimisation génétique des requêtes est utilisée pour planifier les requêtes si, au minimum, ce nombre d'éléments est impliqué dans la clause `FROM` (une construction `FULL OUTER JOIN` ne compte que pour un élément du `FROM`). La valeur par défaut est 12. Il est généralement préférable d'utiliser le planificateur standard, exhaustif, pour les requêtes plus simples, mais pour les requêtes impliquant autant de tables, celui-ci prend trop de temps, fréquemment plus longtemps que la pénalité dû à l'exécution d'un plan non optimal. Du coup, une limite sur la taille de la requête est un moyen simple de contrôler l'utilisation de GEQO.

`geqo_effort` (integer)

Contrôle le compromis entre le temps de planification et l'efficacité du plan de requête dans GEQO. Cette variable est un entier entre 1 et 10. La valeur par défaut est de cinq. Des valeurs plus importantes augmentent le temps passé à la planification de la requête mais aussi la probabilité qu'un plan de requête efficace soit choisi.

`geqo_effort` n'a pas d'action directe ; il est simplement utilisé pour calculer les valeurs par défaut des autres variables influençant le comportement de GEQO (décrites ci-dessous). Il est également possible de les configurer manuellement.

`geqo_pool_size` (integer)

Contrôle la taille de l'ensemble utilisé par GEQO. C'est-à-dire le nombre d'individus au sein d'une population génétique. Elle doit être au minimum égale à deux, les valeurs utiles étant généralement comprises entre 100 et 1000. Si elle est configurée à zéro (valeur par défaut), alors une valeur convenable est choisie en fonction de `geqo_effort` et du nombre de tables dans la requête.

`geqo_generations` (integer)

Contrôle le nombre de générations utilisées par GEQO. C'est-à-dire le nombre d'itérations de l'algorithme. Il doit être au minimum de un, les valeurs utiles se situent dans la même plage que la taille de l'ensemble. S'il est configuré à zéro (valeur par défaut), alors une valeur convenable est choisie en fonction de `geqo_pool_size`.

`geqo_selection_bias` (floating point)

Contrôle le biais de sélection utilisé par GEQO. C'est-à-dire la pression de sélectivité au sein de la population. Les valeurs s'étendent de 1.50 à 2.00 (valeur par défaut).

`geqo_seed` (floating point)

Contrôle la valeur initiale du générateur de nombres aléatoires utilisé par GEQO pour sélectionner des chemins au hasard dans l'espace de recherche des ordres de jointures. La valeur peut aller de zéro (valeur par défaut) à un. Varier la valeur modifie l'ensemble des chemins de jointure explorés et peut résulter en des chemins meilleurs ou pires.

18.6.4. Autres options du planificateur

`default_statistics_target` (integer)

Initialise la cible de statistiques par défaut pour les colonnes de table pour lesquelles aucune cible de colonne spécifique n'a été configurée via **ALTER TABLE SET STATISTICS**. Des valeurs élevées accroissent le temps nécessaire à l'exécution d'**ANALYZE** mais peuvent permettre d'améliorer la qualité des estimations du planificateur. La valeur par défaut est 100. Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes, se référer à la Section 14.2, « Statistiques utilisées par le planificateur ».

`constraint_exclusion` (enum)

Contrôle l'utilisation par le planificateur de requête des contraintes pour optimiser les requêtes. Les valeurs autorisées de `constraint_exclusion` sont `on` (examiner les contraintes pour toutes les tables), `off` (ne jamais examiner les contraintes) et `partition` (n'examiner les contraintes que pour les tables enfants d'un héritage et pour les sous-requêtes `UNION ALL`). `partition` est la valeur par défaut. C'est souvent utilisé avec l'héritage et les tables partitionnées pour améliorer les performances.

Quand ce paramètre l'autorise pour une table particulière, le planificateur compare les conditions de la requête avec les contraintes `CHECK` sur la table, et omet le parcourt des tables pour lesquelles les conditions contredisent les contraintes. Par exemple :

```
CREATE TABLE parent(clef integer, ...);
CREATE TABLE fils1000(check (clef between 1000 and 1999)) INHERITS(parent);
CREATE TABLE fils2000(check (clef between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE clef = 2400;
```

Avec l'activation de l'exclusion par contraintes, ce **SELECT** ne parcourt pas fils1000, ce qui améliore les performances.

À l'heure actuelle, l'exclusion de contraintes est activée par défaut seulement pour les cas qui sont souvent utilisés pour implémenter le partitionnement de tables. L'activer pour toutes les tables impose un surcoût pour la planification qui est assez mesurable pour des requêtes simples, et le plus souvent n'apportera aucun bénéfice aux requêtes simples. Si vous n'avez pas de tables partitionnées, vous voudrez peut-être le désactiver entièrement.

Reportez vous à Section 5.9.4, « Partitionnement et exclusion de contrainte » pour plus d'informations sur l'utilisation d'exclusion de contraintes et du partitionnement.

`cursor_tuple_fraction` (floating point)

Positionne la fraction, estimée par le planificateur, de la fraction d'enregistrements d'un curseur qui sera récupérée. La valeur par défaut est 0.1. Des valeurs plus petites de ce paramètre rendent le planificateur plus enclin à choisir des plans à démarrage rapide (« fast start »), qui récupèreront les premiers enregistrement rapidement, tout en mettant peut être un temps plus long à récupérer tous les enregistrements. Des valeurs plus grandes mettent l'accent sur le temps total estimé. À la valeur maximum 1.0 du paramètre, les curseurs sont planifiés exactement comme des requêtes classiques, en ne prenant en compte que le temps total estimé et non la vitesse à laquelle les premiers enregistrements seront fournis.

`from_collapse_limit` (integer)

Le planificateur assemble les sous-requêtes dans des requêtes supérieures si la liste `FROM` résultante contient au plus ce nombre d'éléments. Des valeurs faibles réduisent le temps de planification mais conduisent à des plans de requêtes inférieurs. La valeur par défaut est de 8. Pour plus d'informations, voir Section 14.3, « Contrôler le planificateur avec des clauses `JOIN` explicites ».

Configurer cette valeur à geqo_threshold ou plus pourrait déclencher l'utilisation du planificateur GEQO, ce qui pourrait aboutir à la génération de plans non optimaux. Voir Section 18.6.3, « Optimiseur génétique de requêtes ».

`join_collapse_limit` (integer)

Le planificateur réécrit les constructions `JOIN` explicites (à l'exception de `FULL JOIN`) en une liste d'éléments `FROM` à chaque fois qu'il n'en résulte qu'une liste ne contenant pas plus de ce nombre d'éléments. Des valeurs faibles réduisent le temps de planification mais conduisent à des plans de requêtes inférieurs.

Par défaut, cette variable a la même valeur que `from_collapse_limit`, valeur adaptée à la plupart des utilisations. Configurer cette variable à 1 empêche le réordonnancement des `JOIN`tures explicites. De ce fait, l'ordre des jointures explicites indiqué dans la requête est l'ordre réel dans lequel les relations sont jointes. Le planificateur de la requête ne choisit pas toujours l'ordre de jointure optimal ; les utilisateurs aguerris peuvent choisir d'initialiser temporairement cette variable à 1 et d'indiquer explicitement l'ordre de jointure souhaité. Pour plus d'informations, voir Section 14.3, « Contrôler le planificateur avec des clauses `JOIN` explicites ».

Configurer cette valeur à geqo_threshold ou plus pourrait déclencher l'utilisation du planificateur GEQO, ce qui pourrait

aboutir à la génération de plans non optimaux. Voir Section 18.6.3, « Optimiseur génétique de requêtes ».

18.7. Remonter et tracer les erreurs

18.7.1. Où tracer

`log_destination` (string)

PostgreSQL™ supporte plusieurs méthodes pour la journalisation des messages du serveur, dont `stderr`, `csvlog` et `syslog`. Sur Windows, `eventlog` est aussi supporté. Ce paramètre se configure avec la liste des destinations souhaitées séparées par des virgules. Par défaut, les traces ne sont dirigées que vers `stderr`. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Si `csvlog` est la valeur de `log_destination`, les entrées du journal applicatif sont enregistrées dans le format CSV (« comma separated value »), ce qui est bien pratique pour les charger dans des programmes. Voir Section 18.7.4, « Utiliser les journaux au format CSV » pour les détails. `logging_collector` doit être activé pour produire des journaux applicatifs au format CSV.

Note

Sur la plupart des systèmes Unix, il est nécessaire de modifier la configuration du démon syslog pour utiliser l'option `syslog` de `log_destination`. PostgreSQL™ peut tracer dans les niveaux syslog `LOCAL0` à `LOCAL7` (voir syslog_facility) mais la configuration par défaut de syslog sur la plupart des plateformes ignore de tels messages. Il faut ajouter une ligne similaire à :

```
local0.*    /var/log/postgresql
```

dans le fichier de configuration de syslog pour obtenir ce type de journalisation.

`logging_collector` (boolean)

Ce paramètre capture les messages au format texte et CSV envoyé à stderr et les redirige dans des journaux applicatifs. Cette approche est souvent plus utile que la journalisation avec syslog, car certains messages peuvent ne pas apparaître dans syslog (les messages d'échec de l'éditeur de liens en sont un bon exemple). Ce paramètre ne peut être configuré qu'au lancement du serveur.

Note

Le collecteur des traces est conçu pour ne jamais perdre de messages. Cela signifie que, dans le cas d'une charge extrêmement forte, les processus serveur pourraient se trouver bloqués à cause de l'envoi de messages de trace supplémentaires. Le collecteur pourrait accumuler dans ce cas du retard. syslog préfère supprimer des messages s'il ne peut pas les écrire. Il est donc moins fiable dans ces cas mais il ne bloquera pas le reste du système.

`log_directory` (string)

Lorsque `logging_collector` est activé, ce paramètre détermine le répertoire dans lequel les fichiers de trace sont créés. Il peut s'agir d'un chemin absolu ou d'un chemin relatif au répertoire des données du cluster. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `pg_log`.

`log_filename` (string)

Lorsque `logging_collector` est activé, ce paramètre indique les noms des journaux applicatifs créés. La valeur est traitée comme un motif `strftime`. Ainsi les échappements `%` peuvent être utilisés pour indiquer des noms de fichiers horodatés. (S'il y a des échappements `%` dépendant des fuseaux horaires, le calcul se fait dans le fuseau précisé par log_timezone.) Notez que la fonction `strftime` du système n'est pas utilisée directement, ce qui entraîne que les extensions spécifiques à la plateforme (non-standard) ne fonctionneront pas. La valeur par défaut est `postgresql-%Y-%m-%d_%H%M%S.log`.

Si vous spécifiez un nom de fichier sans échappements, vous devriez prévoir d'utiliser un utilitaire de rotation des journaux pour éviter le risque de remplir le disque entier. Dans les versions précédentes à 8.4, si aucun échappement `%` n'était présent, PostgreSQL™ aurait ajouté l'epoch de la date de création du nouveau journal applicatif mais ce n'est plus le cas.

Si la sortie au format CSV est activée dans `log_destination`, `.csv` est automatiquement ajouté au nom du journal horodaté. (Si `log_filename` se termine en `.log`, le suffixe est simplement remplacé.)

Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou en ligne de commande.

`log_rotation_age` (integer)

Lorsque `logging_collector` est activé, ce paramètre détermine la durée de vie maximale (en minutes) d'un journal individuel. Passé ce délai, un nouveau journal est créé. Initialiser ce paramètre à zéro désactive la création en temps compté de nouveaux journaux. Ce paramètre ne peut qu'être configuré dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_rotation_size` (integer)

Lorsque `logging_collector` est activé, ce paramètre détermine la taille maximale (en kilooctets) d'un journal individuel. Passé cette taille, un nouveau journal est créé. Initialiser cette taille à zéro désactive la création en taille comptée de nouveaux journaux. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_truncate_on_rotation` (boolean)

Lorsque `logging_collector` est activé, ce paramètre impose à PostgreSQL™ de vider (écraser), plutôt qu'ajouter à, tout fichier journal dont le nom existe déjà. Toutefois, cet écrasement ne survient qu'à partir du moment où un nouveau fichier doit être ouvert du fait d'une rotation par temps compté, et non pas à la suite du démarrage du serveur ou d'une rotation par taille comptée. Si ce paramètre est désactivé (off), les traces sont, dans tous les cas, ajoutées aux fichiers qui existent déjà.

Par exemple, si ce paramètres est utilisé en combinaison avec un `log_filename` tel que `postgresql-%H.log`, il en résulte la génération de 24 journaux (un par heure) écrasés de façon cyclique.

Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Exemple : pour conserver sept jours de traces, un fichier par jour nommé `server_log.Mon`, `server_log.Tue`, etc. et écraser automatiquement les traces de la semaine précédente avec celles de la semaine courante, on positionne `log_filename` à `server_log.%a`, `log_truncate_on_rotation` à `on` et `log_rotation_age` à `1440`.

Exemple : pour conserver 24 heures de traces, un journal par heure, toute en effectuant la rotation plus tôt si le journal dépasse 1 Go, on positionne `log_filename` à `server_log.%H%M`, `log_truncate_on_rotation` à `on`, `log_rotation_age` à `60` et `log_rotation_size` à `1000000`. Inclure `%M` dans `log_filename` permet à toute rotation par taille comptée qui survient d'utiliser un nom de fichier distinct du nom initial horodaté.

`syslog_facility` (enum)

Lorsque les traces syslog sont activées, ce paramètre fixe le niveau (« facility ») utilisé par syslog. Les différentes possibilités sont `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7` ; `LOCAL0` étant la valeur par défaut. Voir aussi la documentation du démon syslog du serveur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Ce paramètre n'est pas disponible si le serveur n'a pas été compilé avec le support de syslog.

`syslog_ident` (string)

Si syslog est activé, ce paramètre fixe le nom du programme utilisé pour identifier les messages PostgreSQL™ dans les traces de syslog. La valeur par défaut est `postgres`. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Ce paramètre n'est pas disponible si le serveur n'a pas été compilé avec le support de syslog.

`silent_mode` (boolean)

Exécute silencieusement le serveur. Si ce paramètre est configuré, le serveur démarre automatiquement en tâche de fond et tout terminal de contrôle est dissocié. Ce paramètre ne peut être configuré qu'au démarrage du serveur.

Attention

Quand ce paramètre est activé, la sortie standard et l'erreur standard sont redirigées vers le fichier `postmaster.log` dans le répertoire des données. Ce fichier ne bénéficie pas du système de rotation, donc il grossira indéfiniment sauf si la sortie du serveur est renvoyée ailleurs par d'autres paramétrages. Il est recommandé de configurer à `log_destination` à `syslog` ou que `logging_collector` soit activé lors de l'utilisation de cette option. Même avec ces mesures, les erreurs rapportées très tôt lors du démarrage pourraient apparaître dans le fichier `postmaster.log` plutôt que dans la destination normal des traces.

18.7.2. Quand tracer

`client_min_messages` (enum)

Contrôle les niveaux de message envoyés au client. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING`, `ERROR`, `FATAL`, et `PANIC`. Chaque niveau inclut tous les niveaux qui le suivent. Plus on progresse dans la liste, plus le nombre de messages envoyés est faible. `NOTICE` est la valeur par défaut. `LOG` a ici une portée différente de celle de `log_min_messages`.

`log_min_messages` (enum)

Contrôle les niveaux de message écrits dans les traces du serveur. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` et `PANIC`. Chaque niveau inclut tous les niveaux qui le suivent. Plus on progresse dans la liste, plus le nombre de messages envoyés est faible. `WARNING` est la valeur par défaut. `LOG` a ici une portée différente de celle de `client_min_messages`. Seuls les superutilisateurs peuvent modifier la valeur de ce paramètre.

`log_min_error_statement` (enum)

Contrôle si l'instruction SQL à l'origine d'une erreur doit être enregistrée dans les traces du serveur. L'instruction SQL en cours est incluse dans les traces pour tout message de sévérité indiquée ou supérieure. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` et `PANIC`. `ERROR` est la valeur par défaut, ce qui signifie que les instructions à l'origine d'erreurs, de messages applicatifs, d'erreurs fatales ou de paniques sont tracées. Pour réellement désactiver le traçage des instructions échouées, ce paramètre doit être positionné à `PANIC`. Seuls les superutilisateurs peuvent modifier la valeur de ce paramètre.

`log_min_duration_statement` (integer)

Trace la durée de toute instruction terminée dont le temps d'exécution égale ou dépasse ce nombre de millisecondes. Positionné à zéro, les durées de toutes les instructions sont tracées. -1 (valeur par défaut) désactive ces traces.

Par exemple, si le paramètre est positionné à `250ms`, alors toutes les instructions SQL dont la durée est supérieure ou égale à 250 ms sont tracées.

Il est utile d'activer ce paramètre pour tracer les requêtes non optimisées des applications. Seuls les superutilisateurs peuvent modifier cette configuration.

Pour les clients utilisant le protocole de requêtage étendu, les durées des étapes Parse (analyse), Bind (lien) et Execute (exécution) sont tracées indépendamment.

Note

Lorsque cette option est utilisée avec log_statement, le texte des instructions tracées du fait de `log_statement` n'est pas répété dans le message de trace de la durée. Si syslog n'est pas utilisé, il est recommandé de tracer le PID ou l'ID de session à l'aide de log_line_prefix de façon à pouvoir lier le message de l'instruction au message de durée par cet identifiant.

Tableau 18.1, « Niveaux de sévérité des messages » explique les niveaux de sévérité des messages utilisés par PostgreSQL™. Si la journalisation est envoyée à`syslog` ou à l'`eventlog` de Windows, les niveaux de sévérité sont traduits comme indiqué ci-dessous.

Tableau 18.1. Niveaux de sévérité des messages

Sévérité	Usage	`syslog`	`eventlog`
`DEBUG1..DEBUG5`	Fournit des informations successivement plus détaillées à destination des développeurs.	`DEBUG`	`INFORMATION`
`INFO`	Fournit des informations implicitement demandées par l'utilisateur, par exemple la sortie de **VACUUM VERBOSE**.	`INFO`	`INFORMATION`
`NOTICE`	Fournit des informations éventuellement utiles aux utilisateurs, par exemple la troncature des identifiants longs.	`NOTICE`	`INFORMATION`
`WARNING`	Fournit des messages	`NOTICE`	`WARNING`

Sévérité	Usage	syslog	eventlog
	d'avertissement sur d'éventuels problèmes. Par exemple, un **COMMIT** en dehors d'un bloc de transaction.		
ERROR	Rapporte l'erreur qui a causé l'annulation de la commande en cours.	WARNING	ERROR
LOG	Rapporte des informations à destination des administrateurs. Par exemple, l'activité des points de vérification.	INFO	INFORMATION
FATAL	Rapporte l'erreur qui a causé la fin de la session en cours.	ERR	ERROR
PANIC	Rapporte l'erreur qui a causé la fin de toutes les sessions.	CRIT	ERROR

18.7.3. Que tracer

`application_name` (string)

Le paramètre `application_name` peut être tout chaîne de moins de NAMEDATALEN caractères (64 caractères après une compilation standard). Il est typiquement configuré lors de la connexion d'une application au serveur. Le nom sera affiché dans la vue pg_stat_activity et inclus dans les traces du journal au format CSV. Il peut aussi être inclus dans les autres formats de traces en configurant le paramètre log_line_prefix. Tout caractère ASCII affichable peut être utilisé. Les autres caractères seront remplacés par des points d'interrogation (?).

`debug_print_parse` (boolean), `debug_print_rewritten` (boolean), `debug_print_plan` (boolean)

Ces paramètres activent plusieurs sorties de débogage. Quand positionnés, il affichent l'arbre d'interprétation résultant, la sortie de la réécriture de requête, ou le plan d'exécution pour chaque requête exécutée. Ces messages sont émis au niveau de trace `LOG` , par conséquent ils apparaîtront dans le journal applicatif du serveur, mais ne seront pas envoyés au client. Vous pouvez changer cela en ajustant client_min_messages et/ou log_min_messages. Ces paramètres sont désactivés par défaut.

`debug_pretty_print` (boolean)

Quand positionné, `debug_pretty_print` indente les messages produits par `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan`. Le résultat est une sortie plus lisible mais plus verbeuse que le format « compact » utilisé quand ce paramètre est à off. La valeur par défaut est 'on'.

`log_checkpoints` (boolean)

Trace les points de vérification dans les journaux applicatifs. Diverses statistiques concernant chaque point de vérification sont incluses dans les journaux applicatifs, dont le nombre de tampons écrits et le temps passé à les écrire. Désactivé par défaut, ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_connections` (boolean)

Trace chaque tentative de connexion sur le serveur, ainsi que la réussite de l'authentification du client. Désactivé par défaut, ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Note

Quelques programmes clients, comme psql, tentent de se connecter deux fois pour déterminer si un mot de passe est nécessaire, des messages « connection received » dupliqués n'indiquent donc pas forcément un problème.

`log_disconnections` (boolean)

Affiche dans les traces du serveur une ligne similaire à `log_connections` mais à la fin d'une session, en incluant la durée de la session. Désactivé par défaut, ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_duration` (boolean)

Trace la durée de toute instruction exécutée. Désactivé par défaut (`off`), seuls les superutilisateurs peuvent modifier ce paramètre.

Pour les clients utilisant le protocole de requêtage étendu, les durées des étapes Parse (analyse), Bind (lien) et Execute (exécution) sont tracées indépendamment.

Note

À la différence de log_min_duration_statement, ce paramètre ne force pas le traçage du texte des requêtes. De ce fait, si `log_duration` est activé (`on`) et que `log_min_duration_statement` a une valeur positive, toutes les durées sont tracées mais le texte de la requête n'est inclus que pour les instructions qui dépassent la limite. Ce comportement peut être utile pour récupérer des statistiques sur les installations à forte charge.

`log_error_verbosity` (enum)

Contrôle la quantité de détails écrit dans les traces pour chaque message tracé. Les valeurs valides sont `TERSE`, `DEFAULT` et `VERBOSE`, chacun ajoutant plus de champs aux messages affichés. `TERSE` exclut des traces les informations de niveau `DETAIL`, `HINT`, `QUERY` et `CONTEXT`. La sortie `VERBOSE` inclut le code d'erreur SQLSTATE (voir aussi Annexe A, Codes d'erreurs de PostgreSQL™), le nom du code source, le nom de la fonction et le numéro de la ligne qui a généré l'erreur. Seuls les superutilisateurs peuvent modifier ce paramètre.

`log_hostname` (boolean)

Par défaut, les traces de connexion n'affichent que l'adresse IP de l'hôte se connectant. Activer ce paramètre permet de tracer aussi le nom de l'hôte. En fonction de la configuration de la résolution de nom d'hôte, les performances peuvent être pénalisées. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_line_prefix` (string)

Il s'agit d'une chaîne de style `printf` affichée au début de chaque ligne de trace. Les caractères `%` débutent des « séquences d'échappement » qui sont remplacées avec l'information de statut décrite ci-dessous. Les échappement non reconnus sont ignorés. Les autres caractères sont copiés directement dans la trace. Certains échappements ne sont reconnus que par les processus de session et ne s'appliquent pas aux processus en tâche de fond comme le processus serveur principal. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est une chaîne vide.

Échappement	Produit	Session seule
`%a`	Nom de l'application	yes
`%u`	Nom de l'utilisateur	oui
`%d`	Nom de la base de données	oui
`%r`	Nom ou adresse IP de l'hôte distant et port distant	oui
`%h`	Nom d'hôte distant ou adresse IP	yes
`%p`	ID du processus	non
`%t`	Estampille temporelle sans millisecondes	non
`%m`	Estampille temporelle avec millisecondes	non
`%i`	Balise de commande : type de commande	oui
`%e`	code d'erreur correspondant à l'état SQL	no
`%c`	ID de session : voir ci-dessous	non
`%l`	Numéro de la ligne de trace de chaque session ou processus, commençant à 1	non
`%s`	Estampille temporelle du lancement du processus	oui
`%v`	Identifiant virtuel de transaction (backendID/localXID)	no
`%x`	ID de la transaction (0 si aucune affectée)	non
`%q`	Ne produit aucune sortie, mais indique aux autres processus de stopper à cet endroit de la chaîne. Ignoré par les processus de session.	non
`%%`	`%`	non

L'échappement `%c` affiche un identifiant de session quasi-unique constitué de deux nombres hexadécimaux sur quatre octets (sans les zéros initiaux) et séparés par un point. Les nombres représentent l'heure de lancement du processus et l'identifiant du

processus, `%c` peut donc aussi être utilisé comme une manière de raccourcir l'affichage de ces éléments. Par exemple, pour générer l'identifiant de session à partir de `pg_stat_activity`, utilisez cette requête :

```
SELECT to_hex(EXTRACT(EPOCH FROM backend_start)::integer) || '.' ||
       to_hex(procpid)
FROM pg_stat_activity;
```

Astuce

Si `log_line_prefix` est différent d'une chaîne vide, il est intéressant d'ajouter une espace en fin de chaîne pour créer une séparation visuelle avec le reste de la ligne. Un caractère de ponctuation peut aussi être utilisé.

Astuce

syslog produit ses propres informations d'horodatage et d'identifiant du processus. Ces échappements n'ont donc que peu d'intérêt avec syslog.

`log_lock_waits` (boolean)

Contrôle si une trace applicative est écrite quand une session attend plus longtemps que deadlock_timeout pour acquérir un verrou. Ceci est utile pour déterminer si les attentes de verrous sont la cause des pertes de performance. Désactivé (`off`) par défaut.

`log_statement` (enum)

Contrôle les instructions SQL à tracer. Les valeurs valides sont `none` (off), `ddl`, `mod` et `all` (toutes les instructions). `ddl` trace toutes les commandes de définition comme **CREATE**, **ALTER** et **DROP**. `mod` trace toutes les instructions `ddl` ainsi que les instructions de modification de données **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE** et **COPY FROM**. Les instructions **PREPARE**, **EXECUTE** et **EXPLAIN ANALYZE** sont aussi tracées si la commande qui les contient est d'un type approprié. Pour les clients utilisant le protocole de requêtage étendu, la trace survient quand un message Execute est reçu et les valeurs des paramètres de Bind sont incluses (avec doublement de tout guillemet simple embarqué).

La valeur par défaut est `none`. Seuls les superutilisateurs peuvent changer ce paramétrage.

Note

Les instructions qui contiennent de simples erreurs de syntaxe ne sont pas tracées même si `log_statement` est positionné à `all` car la trace n'est émise qu'après qu'une analyse basique soit réalisée pour déterminer le type d'instruction. Dans le cas du protocole de requêtage étendu, ce paramètre ne trace pas les instructions qui échouent avant la phase Execute (c'est-à-dire pendant l'analyse et la planification). `log_min_error_statement` doit être positionné à `ERROR` pour tracer ce type d'instructions.

`log_temp_files` (integer)

Contrôle l'écriture de traces sur l'utilisation des fichiers temporaires (noms et tailles). Les fichiers temporaires peuvent être créés pour des tris, des hachages et des résultats temporaires de requête. Une entrée de journal est générée pour chaque fichier temporaire au moment ou il est effacé. Zéro implique une trace des informations sur tous les fichiers temporaires alors qu'une valeur positive ne trace que les fichiers dont la taille est supérieure ou égale au nombre indiqué (en kilo-octets). La valeur par défaut est `-1`, ce qui a pour effet de désactiver les traces. Seuls les superutilisateurs peuvent modifier ce paramètre.

`log_timezone` (string)

Configure le fuseau horaire utilisé par l'horodatage des traces. Contrairement à timezone, cette valeur est valable pour le cluster complet, de façon à ce que toutes les sessions utilisent le même. La valeur par défaut est `unknown`, ce qui signifie que l'environnement système sera utilisé pour trouver cette information. Voir Section 8.5.3, « Fuseaux horaires » pour plus d'informations. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

18.7.4. Utiliser les journaux au format CSV

L'ajout de `csvlog` dans la liste `log_destination` est une manière simple d'importer des journaux dans une table de base de données. Cette option permet de créer des journaux au format CSV avec les colonnes : l'horodatage en millisecondes, le nom de l'utilisateur, le nom de la base de données, le PID du processus serveur, l'hôte et le numéro de port du client, l'identifiant de la session, le numéro de ligne dans la session, le tag de la commande, l'horodatage de début de la session, l'identifiant de transaction vir-

tuelle, l'identifiant de transaction standard, la sévérité de l'erreur, le code SQLSTATE, le message d'erreur, les détails du message d'erreur, une astuce, la requête interne qui a amené l'erreur (si elle existe), le nombre de caractères pour arriver à la position de l'erreur, le contexte de l'erreur, la requête utilisateur qui a amené l'erreur (si elle existe et si `log_min_error_statement` est activé), le nombre de caractères pour arriver à la position de l'erreur, l'emplacement de l'erreur dans le code source de PostgreSQL (si `log_error_verbosity` est configuré à `verbose`) et le nom de l'application.

Exemple de définition d'une table de stockage de journaux au format CSV :

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

Pour importer un journal dans cette table, on utilise la commande **COPY FROM** :

```
COPY postgres_log FROM '/chemin/complet/vers/le/logfile.csv' WITH csv;
```

Quelques conseils pour simplifier et automatiser l'import des journaux CVS :

1. configurer `log_filename` et `log_rotation_age` pour fournir un schéma de nommage cohérent et prévisible des journaux. Cela permet de prédire le nom du fichier et le moment où il sera complet (et donc prêt à être importé) ;
2. initialiser `log_rotation_size` à 0 pour désactiver la rotation par taille comptée, car elle rend plus difficile la prévision du nom du journal ;
3. positionner `log_truncate_on_rotation` à `on` pour que les données anciennes ne soient pas mélangées aux nouvelles dans le même fichier ;
4. la définition de la table ci-dessus inclut une clé primaire. C'est utile pour se protéger de l'import accidentel de la même information à plusieurs reprises. La commande **COPY** valide toutes les données qu'elle importe en une fois. Toute erreur annule donc l'import complet. Si un journal incomplet est importé et qu'il est de nouveau importé lorsque le fichier est complet, la violation de la clé primaire cause un échec de l'import. Il faut attendre que le journal soit complet et fermé avant de l'importer. Cette procédure protége aussi de l'import accidentel d'une ligne partiellement écrite, qui causerait aussi un échec de **COPY**.

18.8. Statistiques d'exécution

18.8.1. Collecteur de statistiques sur les requêtes et les index

Ces paramètres contrôlent la collecte de statistiques de niveau serveur. Lorsque celle-ci est activée, les données produites peuvent être visualisées à travers la famille de vues systèmes pg_stat et pg_statio. On peut se reporter à Chapitre 27, Surveiller l'activité de la base de données pour plus d'informations.

`track_activities` (boolean)

Active la collecte d'informations sur la commande en cours d'exécution dans chaque session, avec l'heure de démarrage de la commande. Ce paramètre est activé par défaut. Même si le paramètre est activé, cette information n'est pas visible par tous les utilisateurs, mais uniquement par les superutilisateurs et l'utilisateur possédant la session traitée ; de ce fait, cela ne représente pas une faille de sécurité. Seuls les superutilisateurs peuvent modifier ce paramètre.

`track_activity_query_size` (integer)

Spécifie le nombre d'octets réservés pour suivre la commande en cours d'exécution pour chaque session active, pour le champ pg_stat_activity.*`current_query`*. La valeur par défaut est 1024. Ce paramètre ne peut être positionné qu'au démarrage du serveur.

`track_counts` (boolean)

Active la récupération de statistiques sur l'activité de la base de données. Ce paramètre est activé par défaut car le processus autovacuum utilise les informations ainsi récupérées. Seuls les super-utilisateurs peuvent modifier ce paramètre.

`track_functions` (enum)

Active le suivi du nombre et de la durée des appels aux fonctions. Précisez `pl` pour ne tracer que les fonctions de langages procéduraux, ou `all` pour suivre aussi les fonctions SQL et C. La valeur par défaut est `none`, qui désactive le suivi des statistiques de fonctions. Seuls les superutilisateurs peuvent modifier ce paramètre.

Note

Les fonctions en langage SQL qui sont assez simples pour être « inlined », c'est à dire substituées dans le code de la requête appelante, ne seront pas suivies, quelle que soit la valeur de ce paramètre.

`update_process_title` (boolean)

Active la mise à jour du titre du processus à chaque fois qu'une nouvelle commande SQL est reçue par le serveur. Le titre du processus est visible typiquement avec la commande **ps** sur Unix ou en utilisant l'explorateur de processus sur Windows. Seuls les superutilisateurs peuvent modifier ce paramètre.

`stats_temp_directory` (string)

Précise le répertoire dans lequel stocker les données temporaires de statistiques. Cela peut être un chemin relatif au répertoire de données ou un chemin absolu. La valeur par défaut est `pg_stat_tmp`. Faire pointer ceci vers un système de fichiers mémoire diminuera les entrées/sorties physiques et peut améliorer les performances. Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou sur la ligne de commande.

18.8.2. Surveillance et statistiques

`log_statement_stats` (boolean), `log_parser_stats` (boolean), `log_planner_stats` (boolean), `log_executor_stats` (boolean)

Écrivent, pour chaque requête, les statistiques de performance du module respectif dans les traces du serveur. C'est un outil de profilage très simpliste, similaire aux possibilités de l'appel `getrusage()` du système d'exploitation Unix. `log_statement_stats` rapporte les statistiques d'instructions globales, tandis que les autres fournissent un rapport par module. `log_statement_stats` ne peut pas être activé conjointement à une option de module. Par défaut, toutes ces options sont désactivées. Seuls les superutilisateurs peuvent modifier ces paramètres.

18.9. Nettoyage (vacuum) automatique

Ces paramètres contrôlent le comportement de la fonctionnalité appelée *autovacuum*. Se référer à la Section 23.1.5, « Le démon auto-vacuum » pour plus de détails.

`autovacuum` (boolean)

Contrôle si le serveur doit démarrer le démon d'autovacuum. Celui-ci est activé par défaut. track_counts doit aussi être activé pour que ce démon soit démarré. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Même si ce paramètre est désactivé, le système lance les processus autovacuum nécessaires pour empêcher le bouclage des identifiants de transaction. Voir Section 23.1.4, « Éviter les cycles des identifiants de transactions » pour plus d'informations.

`log_autovacuum_min_duration` (integer)

Trace chaque action réalisée par l'autovacuum si elle dure chacune plus de ce nombre de millisecondes. Le configurer à zéro trace toutes les actions de l'autovacuum. La valeur par défaut, -1, désactive les traces des actions de l'autovacuum.

Par exemple, s'il est configuré à `250ms`, toutes les opérations VACUUM et ANALYZE qui durent plus de 250 ms sont tracées. Activer ce paramètre peut être utile pour tracer l'activité de l'autovacuum. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`autovacuum_max_workers` (integer)

Indique le nombre maximum de processus autovacuum (autre que le lanceur d'autovacuum) qui peuvent être exécutés simultanément. La valeur par défaut est 3. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`autovacuum_naptime` (integer)

Indique le délai minimum entre les tours d'activité du démon autovacuum sur une base. À chaque tour, le démon examine une base de données et lance les commandes **VACUUM** et **ANALYZE** nécessaires aux tables de cette base. Le délai, mesuré en secondes, vaut, par défaut, une minute (`1min`). Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`autovacuum_vacuum_threshold` (integer)

Indique le nombre minimum de lignes mises à jour ou supprimées nécessaire pour déclencher un **VACUUM** sur une table. La valeur par défaut est de 50 lignes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage.

`autovacuum_analyze_threshold` (integer)

Indique le nombre minimum de lignes insérées, mises à jour ou supprimées nécessaire pour déclencher un **ANALYZE** sur une table. La valeur par défaut est de 50 lignes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage.

`autovacuum_vacuum_scale_factor` (floating point)

Indique la fraction de taille de la table à ajouter à `autovacuum_vacuum_threshold` pour décider du moment auquel déclencher un **VACUUM**. La valeur par défaut est 0.2 (20 % de la taille de la table). Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage.

`autovacuum_analyze_scale_factor` (floating point)

Indique la fraction de taille de la table à ajouter à `autovacuum_analyze_threshold` pour décider du moment auquel déclencher une commande **ANALYZE**. La valeur par défaut est 0.1 (10 % de la taille de la table). Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage.

`autovacuum_freeze_max_age` (integer)

Indique l'âge maximum (en transactions) que le champ pg_class.*`relfrozenxid`* d'une table peut atteindre avant qu'une opération **VACUUM** ne soit forcée pour empêcher la réinitialisation de l'ID de transaction sur cette table. Le système lance les processus autovacuum pour éviter ce bouclage même si l'autovacuum est désactivé.

L'opération VACUUM supprime aussi les anciens fichiers du sous-répertoire `pg_clog`, ce qui explique pourquoi la valeur par défaut est relativement basse (200 millions de transactions). Ce paramètre n'est lu qu'au démarrage du serveur, mais il peut être diminué pour toute table en modifiant les paramètres de stockage. Pour plus d'informations, voir Section 23.1.4, « Éviter les cycles des identifiants de transactions ».

`autovacuum_vacuum_cost_delay` (integer)

Indique la valeur du coût de délai utilisée dans les opérations de **VACUUM**. Si `-1` est indiqué, la valeur habituelle de vacuum_cost_delay est utilisée. La valeur par défaut est 20 millisecondes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est possible de le surcharger pour toute table en modifiant les paramètres de stockage.

`autovacuum_vacuum_cost_limit` (integer)

Indique la valeur de coût limite utilisée dans les opérations de **VACUUM** automatiques. Si `-1` est indiqué (valeur par défaut), la valeur courante de vacuum_cost_limit est utilisée. La valeur est distribuée proportionnellement entre les processus autovacuum en cours d'exécution, s'il y en a plus d'un, de sorte que la somme des limites de chaque processus ne dépasse jamais la limite de cette variable. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Il est possible de le surcharger pour toute tableen modifiant les paramètres de stockage.

18.10. Valeurs par défaut des connexions client

18.10.1. Comportement des instructions

`search_path` (string)

Cette variable précise l'ordre dans lequel les schémas sont parcourus lorsqu'un objet (table, type de données, fonction, etc.) est référencé par un simple nom sans précision du schéma. Lorsque des objets de noms identiques existent dans plusieurs schémas, c'est le premier trouvé dans le chemin de recherche qui est utilisé. Il ne peut être fait référence à un objet qui ne fait partie d'aucun des schémas indiqués dans le chemin de recherche qu'en précisant son schéma conteneur avec un nom qualifié (avec un point).

`search_path` doit contenir une liste de noms de schémas séparés par des virgules. Si un des éléments de la liste est la valeur spéciale `$user`, alors le schéma dont le nom correspond à la valeur retournée par `SESSION_USER` est substitué, s'il existe (sinon `$user` est ignoré).

Le schéma du catalogue système, `pg_catalog`, est toujours parcouru, qu'il soit ou non mentionné dans le chemin. Mentionné, il est alors parcouru dans l'ordre indiqué. Dans le cas contraire, il est parcouru *avant* tout autre élément du chemin.

De même, le schéma des tables temporaires, `pg_temp_`*nnn*, s'il existe, est toujours parcouru. Il peut être explicitement ajouté au chemin à l'aide de l'alias `pg_temp`. S'il n'en fait pas partie, la recherche commence par lui (avant même `pg_catalog`). Néanmoins, seuls les noms de relation (table, vue, séquence, etc.) et de type de données sont recherchés dans le schéma temporaire. Aucune fonction et aucun opérateur n'y est jamais recherché.

Lorsque des objets sont créés sans précision de schéma cible particulier, ils sont placés dans le premier schéma listé dans le chemin de recherche. Une erreur est rapportée si le chemin de recherche est vide.

La valeur par défaut de ce paramètre est `'"$user", public'` (la deuxième partie est ignorée s'il n'existe pas de schéma nommé `public`). Elle permet l'utilisation partagée d'une base de données (dans laquelle aucun utilisateur n'a de schéma privé et tous partagent l'utilisation de `public`), les schémas privés d'utilisateur ainsi qu'une combinaison de ces deux modes. D'autres effets peuvent être obtenus en modifiant le chemin de recherche par défaut, globalement ou par utilisateur.

La valeur courante réelle du chemin de recherche peut être examinée via la fonction SQL `current_schemas` (voir Section 9.23, « Fonctions d'informations système »). Elle n'est pas identique à la valeur de `search_path`, car `current_schemas` affiche la façon dont les requêtes apparaissant dans `search_path` sont résolues.

Pour plus d'informations sur la gestion des schémas, voir la Section 5.7, « Schémas ».

`default_tablespace` (string)

Cette variable indique le *tablespace* par défaut dans lequel sont créés les objets (tables et index) quand une commande **CREATE** ne l'explicite pas.

La valeur est soit le nom d'un *tablespace* soit une chaîne vide pour indiquer l'utilisation du *tablespace* par défaut de la base de données courante. Si la valeur ne correspond pas au nom d'un tablespace existant, PostgreSQL™ utilise automatiquement le tablespace par défaut de la base de données courante. Si un *tablespace* différent de celui par défaut est indiqué, l'utilisateur doit avoir le droit `CREATE`. Dans le cas contraire, la tentative de création échouera.

Cette variable n'est pas utilisée pour les tables temporaires ; pour elles, temp_tablespaces est consulté à la place.

`temp_tablespaces` (string)

Cette variable indique le (ou les) *tablespace*(s) dans le(s)quel(s) créer les objets temporaires (tables temporaires et index sur des tables temporaires) quand une commande **CREATE** n'en explicite pas. Les fichiers temporaires créés par les tris de gros ensembles de données sont aussi créés dans ce *tablespace*.

Cette valeur est une liste de noms de *tablespace*s. Quand cette liste contient plus d'un nom, PostgreSQL™ choisit un membre de la liste au hasard à chaque fois qu'un objet temporaire doit être créé. En revanche, dans une transaction, les objets temporaires créés successivement sont placés dans les *tablespace*s successifs de la liste. Si l'élément sélectionné de la liste est une chaîne vide, PostgreSQL™ utilise automatiquement le *tablespace* par défaut de la base en cours.

Si `temp_tablespaces` est configuré interactivement, l'indication d'un *tablespace* inexistant est une erreur. Il en est de même si l'utilisateur n'a pas le droit `CREATE` sur le *tablespace* indiqué. Néanmoins, lors de l'utilisation d'une valeur précédemment configurée, les *tablespace*s qui n'existent pas sont ignorés comme le sont les *tablespace*s pour lesquels l'utilisateur n'a pas le droit `CREATE`. Cette règle s'applique, en particulier, lors de l'utilisation d'une valeur configurée dans le fichier `postgresql.conf`.

La valeur par défaut est une chaîne vide. De ce fait, tous les objets temporaires sont créés dans le *tablespace* par défaut de la base de données courante.

Voir aussi default_tablespace.

`check_function_bodies` (boolean)

Ce paramètre est habituellement positionné à `on`. Positionné à `off`, il désactive la validation du corps de la fonction lors de CREATE FUNCTION(7). Désactiver la validation évite les effets de bord du processus de validation et évite les faux positifs dûs aux problèmes, par exemple les références. Configurer ce paramètre à `off` avant de charger les fonctions à la place des autres utilisateurs ; pg_dump le fait automatiquement.

`default_transaction_isolation` (enum)

Chaque transaction SQL a un niveau d'isolation. Celui-ci peut être « read uncommitted », « read committed », « repeatable read » ou « serializable ». Ce paramètre contrôle le niveau d'isolation par défaut de chaque nouvelle transaction. La valeur par défaut est « read committed ».

Consulter le Chapitre 13, Contrôle d'accès simultané et SET TRANSACTION(7) pour plus d'informations.

`default_transaction_read_only` (boolean)

Une transaction SQL en lecture seule ne peut pas modifier les tables permanentes. Ce paramètre contrôle le statut de lecture seule par défaut de chaque nouvelle transaction. La valeur par défaut est `off` (lecture/écriture).

Consulter SET TRANSACTION(7) pour plus d'informations.

`session_replication_role` (enum)

Contrôle l'exécution des triggers et règles relatifs à la réplication pour la session en cours. Seul un superutilisateur peut configurer cette variable. Sa modification résulte en l'annulation de tout plan de requête précédemment mis en cache. Les valeurs possibles sont `origin` (la valeur par défaut), `replica` et `local`. Voir ALTER TABLE(7) pour plus d'informations.

`statement_timeout` (integer)

Interrompt toute instruction qui dure plus longtemps que ce nombre (indiqué en millisecondes). Le temps est décompté à partir du moment où la commande en provenance du client arrive sur le serveur. Si `log_min_error_statement` est configuré à `ERROR`, ou plus bas, l'instruction en cause est tracée. La valeur zéro (par défaut) désactive le décompte.

Il n'est pas recommandé de configurer `statement_timeout` dans `postgresql.conf` car cela affecte toutes les sessions.

`vacuum_freeze_table_age` (integer)

VACUUM effectuera un parcours complet de la table si le champ pg_class.*`relfrozenxid`* de la table a atteint l'âge spécifié par ce paramètre. La valeur par défaut est 150 millions de transactions. Même si les utilisateurs peuvent positionner cette valeur à n'importe quelle valeur comprise entre zéro et 1 milliard, **VACUUM** limitera silencieusement la valeur effective à 95% de autovacuum_freeze_max_age, afin qu'un vacuum périodique manuel ait une chance de s'exécuter avant un autovacuum anti-bouclage ne soit lancé pour la table. Pour plus d'informations voyez Section 23.1.4, « Éviter les cycles des identifiants de transactions ».

`vacuum_freeze_min_age` (integer)

Indique l'âge limite (en transactions) que **VACUUM** doit utiliser pour décider de remplacer les ID de transaction par `FrozenXID` lors du parcours d'une table. La valeur par défaut est 50 millions. Bien que les utilisateurs puissent configurer une valeur quelconque comprise entre zéro et 1 milliard, **VACUUM** limite silencieusement la valeur réelle à la moitié de la valeur de autovacuum_freeze_max_age afin que la valeur entre deux autovacuums forcés ne soit pas déraisonnablement courte. Pour plus d'informations, voir Section 23.1.4, « Éviter les cycles des identifiants de transactions ».

`bytea_output` (enum)

Configure le format de sortie pour les valeurs de type bytea. Les valeurs valides sont `hex` (la valeur par défaut) et `escape` (le format traditionnel de PostgreSQL). Voir Section 8.4, « Types de données binaires » pour plus d'informations. Le type bytea accepte toujours les deux formats en entrée, quelque soit la valeur de cette configuration.

`xmlbinary` (enum)

Définit la manière de coder les valeurs binaires en XML. Ceci s'applique, par exemple, quand les valeurs bytea sont converties en XML par les fonctions `xmlelement` et `xmlforest`. Les valeurs possibles sont `base64` et `hex`, qui sont toutes les deux définies dans le standard XML Schema. La valeur par défaut est `base64`. Pour plus d'informations sur les fonctions relatives à XML, voir Section 9.14, « Fonctions XML ».

Le choix effectif de cette valeur est une affaire de sensibilité, la seule restriction provenant des applications clientes. Les deux méthodes supportent toutes les valeurs possibles, et ce bien que le codage hexadécimal soit un peu plus grand que le codage en base64.

`xmloption` (enum)

Définit si DOCUMENT ou CONTENT est implicite lors de la conversion entre XML et valeurs chaînes de caractères. Voir Section 8.13, « Type XML » pour la description. Les valeurs valides sont DOCUMENT et CONTENT. La valeur par défaut est CONTENT.

D'après le standard SQL, la commande pour configurer cette option est :

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Cette syntaxe est aussi disponible dans PostgreSQL.

18.10.2. Locale et formatage

datestyle (string)

Configure le format d'affichage des valeurs de type date et heure, ainsi que les règles d'interprétation des valeurs ambiguës de dates saisies. Pour des raisons historiques, cette variable contient deux composantes indépendantes : la spécification du format en sortie (ISO, Postgres, SQL ou German) et la spécification en entrée/sortie de l'ordre année/mois/jour (DMY, MDY ou YMD). Elles peuvent être configurées séparément ou ensemble. Les mots clés Euro et European sont des synonymes de DMY ; les mots clés US, NonEuro et NonEuropean sont des synonymes de MDY. Voir la Section 8.5, « Types date/heure » pour plus d'informations. La valeur par défaut est ISO, MDY, mais initdb initialise le fichier de configuration avec une valeur qui correspond au comportement de la locale lc_time choisie.

IntervalStyle (enum)

Positionne le format d'affichage pour les valeurs de type intervalle. La valeur sql_standard produira une sortie correspondant aux litéraux d'intervalles du standard SQL. La valeur postgres (qui est la valeur par défaut) produira une sortie correspondant à celle des versions de PostgreSQL™ antérieures à la 8.4 quand le paramètre datestyle était positionné à ISO. La valeur postgres_verbose produira une sortie correspondant à celle des versions de PostgreSQL™ antérieures à la 8.4 quand le paramètre DateStyle était positionné à une valeur autre que ISO La valeur iso_8601 produira une sortie correspondant au « format avec designateurs » d'intervalle de temps défini dans le paragraphe 4.4.3.2 de l'ISO 8601.

Le paramètre IntervalStyle affecte aussi l'interprétation des entrées ambigües d'intervalles. Voir Section 8.5.4, « Saisie d'intervalle » pour plus d'informations.

timezone (string)

Configure le fuseau horaire pour l'affichage et l'interprétation de la date et de l'heure. Par défaut, unknown, ce qui signifie que le fuseau horaire utilisé est celui défini par le système. Voir la Section 8.5.3, « Fuseaux horaires » pour plus d'informations.

timezone_abbreviations (string)

Configure la liste des abréviations de fuseaux horaires acceptés par le serveur pour la saisie de données de type datetime. La valeur par défaut est 'Default', qui est une liste qui fonctionne presque dans le monde entier ; il y a aussi 'Australia' et 'India'. D'autres listes peuvent être définies pour une installation particulière. Voir Annexe B, Support de date/heure pour plus d'informations.

extra_float_digits (integer)

Ce paramètre ajuste le nombre de chiffres affichés par les valeurs à virgule flottante, ce qui inclut float4, float8 et les types de données géométriques. La valeur du paramètre est ajoutée au nombre standard de chiffres (FLT_DIG ou DBL_DIG). La valeur peut être initialisée à une valeur maximale de 3 pour inclure les chiffres partiellement significatifs ; c'est tout spécialement utile pour sauvegarder les données à virgule flottante qui ont besoin d'être restaurées exactement. Cette variable peut aussi être négative pour supprimer les chiffres non souhaités. Voir aussi Section 8.1.3, « Types à virgule flottante ».

client_encoding (string)

Initialise l'encodage client (jeu de caractères). Par défaut, il s'agit de celui de la base de données.

lc_messages (string)

Initialise la langue d'affichage des messages. Les valeurs acceptables dépendent du système ; voir Section 22.1, « Support des locales » pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

Avec certains systèmes, cette catégorie de locale n'existe pas. Initialiser cette variable fonctionne toujours mais n'a aucun effet. De même, il est possible qu'il n'existe pas de traduction des messages dans la langue sélectionnée. Dans ce cas, les messages sont affichés en anglais.

Seuls les superutilisateurs peuvent modifier ce paramètre car il affecte aussi bien les messages envoyés dans les traces du ser-

veur que ceux envoyés au client.

`lc_monetary` (string)

Initialise la locale à utiliser pour le formatage des montants monétaires (pour la famille de fonctions `to_char`, par exemple). Les valeurs acceptables dépendent du système ; voir la Section 22.1, « Support des locales » pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur, et une valeur incorrecte pourrait dégrader la lisibilité des traces du serveur.

`lc_numeric` (string)

Initialise la locale à utiliser pour le formatage des nombres (pour la famille de fonctions `to_char`, par exemple). Les valeurs acceptables dépendent du système ; voir la Section 22.1, « Support des locales » pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

`lc_time` (string)

Initialise la locale à utiliser pour le formatage des valeurs de date et d'heure, par exemple avec la famille de fonctions `to_char`. Les valeurs acceptables dépendent du système ; voir la Section 22.1, « Support des locales » pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

`default_text_search_config` (string)

Sélectionne la configuration de recherche plein texte utilisée par les variantes des fonctions de recherche plein texte qui n'ont pas d'argument explicite pour préciser la configuration. Voir Chapitre 12, Recherche plein texte pour plus d'informations. La valeur par défaut est `pg_catalog.simple` mais initdb initialise le fichier de configuration avec une valeur qui correspond à la locale choisie pour `lc_ctype` s'il est possible d'identifier une configuration correspondant à la locale.

18.10.3. Autres valeurs par défaut

`dynamic_library_path` (string)

Chemin de recherche utilisé lorsqu'un module chargeable dynamiquement doit être ouvert et que le nom de fichier indiqué dans la commande **CREATE FUNCTION** ou **LOAD** ne contient pas d'indication de répertoire (c'est-à-dire que le nom ne contient pas de slash).

La valeur de `dynamic_library_path` doit être une liste de chemins absolus séparés par des virgules (ou des points virgules sous Windows). Si un élément de la liste débute par la chaîne spéciale `$libdir`, le répertoire des bibliothèques internes du paquetage PostgreSQL™ est substitué à `$libdir`. C'est l'emplacement où sont installés les modules fournis par la distribution PostgreSQL™ standard. (La commande `pg_config --pkglibdir` permet de connaître le nom de ce répertoire.) Par exemple :

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

ou dans un environnement Windows :

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

La valeur par défaut de ce paramètre est `'$libdir'`. Si la valeur est une chaîne vide, la recherche automatique du chemin est désactivée.

Ce paramètre peut être modifié à l'exécution par les superutilisateurs, mais un tel paramétrage ne persiste que pour la durée de la connexion du client. Il est donc préférable de ne réserver cette méthode qu'à des fins de développement. Il est recommandé d'initialiser ce paramètre dans le fichier de configuration `postgresql.conf`.

`gin_fuzzy_search_limit` (integer)

Limite souple haute de la taille de l'ensemble renvoyé par un index GIN. Pour plus d'informations, voir Section 53.4, « Conseils et astuces GIN ».

`local_preload_libraries` (string)

Indique les bibliothèques partagées à charger à l'initialisation d'une connexion. Si plusieurs bibliothèques doivent être chargées, leurs noms sont séparés par des virgules. Tous les noms de bibliothèques sont convertis en minuscule sauf s'ils sont compris entre des guillemets doubles. Ce paramètre ne peut pas être modifié après le démarrage de la session.

Comme il ne s'agit pas d'une option réservée aux superutilisateurs, les bibliothèques pouvant être chargées sont restreintes à celles du sous-répertoire `plugins` du répertoire standard des bibliothèques. (Il est de la responsabilité de l'administrateur de bases de données de s'assurer que seules des bibliothèques « saines » s'y trouvent.) Les entrées dans `lo-`

cal_preload_libraries peuvent indiquer ce répertoire explicitement, par exemple $libdir/plugins/ma_lib, ou n'indiquer que le nom de la bibliothèque -- ma_lib a le même effet que $libdir/plugins/ma_lib.

Contrairement à shared_preload_libraries, il n'y a pas de gain de performance à charger une bibliothèque au démarrage d'une session ou à sa première utilisation. Le but de cette fonctionnalité est d'autoriser le chargement de bibliothèques de débogage ou de mesure de performance dans certaines sessions spécifiques sans commande **LOAD** explicite. Par exemple, le débogage peut être activé pour toutes les sessions d'un nom d'utilisateur donné en configurant ce paramètre avec **ALTER USER SET**.

Si une bibliothèque indiquée n'est pas trouvée, la tentative de connexion échoue.

Toute bibliothèque supportée par PostgreSQL contient un « bloc magique » qui permet d'en garantir la compatibilité. Pour cette raison, les bibliothèques non PostgreSQL ne peuvent pas être chargées de cette façon.

18.11. Gestion des verrous

deadlock_timeout (integer)

Temps total, en millisecondes, d'attente d'un verrou avant de tester une condition de verrou mort (*deadlock*). Le test de verrou mort est très coûteux, le serveur ne l'effectue donc pas à chaque fois qu'il attend un verrou. Les développeurs supposent (de façon optimiste ?) que les verrous morts sont rares dans les applications en production et attendent simplement un verrou pendant un certain temps avant de lancer une recherche de blocage. Augmenter cette valeur réduit le temps perdu en recherches inutiles de verrous morts mais retarde la détection de vraies erreurs de verrous morts. La valeur par défaut est une seconde (1s), ce qui est probablement la plus petite valeur pratique. Sur un serveur en pleine charge, elle peut être augmentée. Idéalement, ce paramétrage peut dépasser le temps typique d'une transaction de façon à augmenter la probabilité qu'un verrou soit relâché avant que le processus en attente ne décide de lancer une recherche de verrous morts.

Quand log_lock_waits est configuré, ce paramètre détermine aussi le temps d'attente avant qu'un message ne soit enregistré dans les journaux concernant cette attente. Pour comprendre ces délais de verrouillage, il peut être utile de configurer deadlock_timeout à une valeur extraordinairement basse.

max_locks_per_transaction (integer)

La table des verrous partagés trace les verrous sur max_locks_per_transaction * (max_connections + max_prepared_transactions) objets (c'est-à-dire des tables) ; de ce fait, au maximum ce nombre d'objets distincts peuvent être verrouillés simultanément. Ce paramètre contrôle le nombre moyen de verrous d'objets alloués pour chaque transaction ; des transactions individuelles peuvent verrouiller plus d'objets tant que l'ensemble des verrous de toutes les transactions tient dans la table des verrous. Il *ne* s'agit *pas* du nombre de lignes qui peuvent être verrouillées ; cette valeur n'a pas de limite. La valeur par défaut, 64, s'est toujours avérée suffisante par le passé, mais il est possible de l'augmenter si des clients accèdent à de nombreuses tables différentes au sein d'une unique transaction. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

Augmenter ce paramètre peut obliger PostgreSQL™ à réclamer plus de mémoire partagée System V ou de sémaphores que ne le permet la configuration par défaut du système d'exploitation. Voir la Section 17.4.1, « Mémoire partagée et sémaphore » pour plus d'informations sur la façon d'ajuster ces paramètres, si nécessaire.

Lors de l'exécution d'un serveur en attente, vous devez configurer ce paramètre à la même valeur ou à une valeur plus importante que sur le serveur maître. Sinon, des requêtes pourraient ne pas être autorisées sur le serveur en attente.

18.12. Compatibilité de version et de plateforme

18.12.1. Versions précédentes de PostgreSQL

array_nulls (boolean)

Contrôle si l'analyseur de saisie de tableau reconnaît NULL non-encadré par des guillemets comme élément de tableaux NULL. Activé par défaut (on), il autorise la saisie de valeurs NULL dans un tableau. Néanmoins, les versions de PostgreSQL™ antérieures à la 8.2 ne supportent pas les valeurs NULL dans les tableaux. De ce fait, ces versions traitent NULL comme une chaîne dont le contenu est « NULL ». Pour une compatibilité ascendante avec les applications nécessitant l'ancien comportement, ce paramètre peut être désactivé (off).

Il est possible de créer des valeurs de tableau contenant des valeurs NULL même quand cette variable est à off.

backslash_quote (enum)

Contrôle si un guillemet simple peut être représenté par un \' dans une chaîne. Il est préférable, et conforme au standard SQL, de représenter un guillemet simple en le doublant ('') mais, historiquement, PostgreSQL™ a aussi accepté \'. Néan-

moins, l'utilisation de `\'` présente des problèmes de sécurité car certains encodages client contiennent des caractères multi-octets dans lesquels le dernier octet est l'équivalent ASCII numérique d'un `\`. Si le code côté client ne fait pas un échappement correct, alors une attaque par injection SQL est possible. Ce risque peut être évité en s'assurant que le serveur rejette les requêtes dans lesquelles apparaît un guillemet échappé avec un antislash. Les valeurs autorisées de `backslash_quote` sont `on` (autorise `\'` en permanence), `off` (le rejette en permanence) et `safe_encoding` (ne l'autorise que si l'encodage client n'autorise pas l'ASCII `\` dans un caractère multioctet). `safe_encoding` est le paramétrage par défaut.

Dans une chaîne littérale conforme au standard, `\` ne signifie que `\`. Ce paramètre affecte seulement la gestion des chaînes non conformes, incluant la syntaxe de chaînes d'échappement (`E'...'`).

`default_with_oids` (boolean)

Contrôle si les commandes **CREATE TABLE** et **CREATE TABLE AS** incluent une colonne OID dans les tables nouvellement créées, lorsque ni `WITH OIDS` ni `WITHOUT OIDS` ne sont précisées. Ce paramètre détermine également si les OID sont inclus dans les tables créées par **SELECT INTO**. Ce paramètre est désactivé (`off`) par défaut ; avec PostgreSQL™ 8.0 et les versions précédentes, il était activé par défaut.

L'utilisation d'OID dans les tables utilisateur est considérée comme obsolète. Il est donc préférable pour la plupart des installations de laisser ce paramètre désactivé. Les applications qui requièrent des OID pour une table particulière doivent préciser `WITH OIDS` lors de la création de la table. Cette variable peut être activée pour des raisons de compatibilité avec les anciennes applications qui ne suivent pas ce comportement.

`escape_string_warning` (boolean)

S'il est activé (`on`), un message d'avertissement est affiché lorsqu'un antislash (`\`) apparaît dans une chaîne littérale ordinaire (syntaxe `'...'`) et que `standard_conforming_strings` est désactivé. Il est activé par défaut (`on`).

Les applications qui souhaitent utiliser l'antislash comme échappement doivent être modifiées pour utiliser la syntaxe de chaîne d'échappement (`E'...'`) car le comportement par défaut des chaînes ordinaires changera dans une prochaine version pour des raisons de compatibilité avec le standard SQL. Cette variable peut être activée pour aider à la détection des applications non conformes.

`regex_flavor` (enum)

La « flaveur » des expressions rationnelles peut être configurée à `advanced` (avancée), `extended` (étendue) ou `basic` (basique). La valeur par défaut est `advanced`. La configuration `extended` peut être utile pour une compatibilité ascendante avec les versions antérieures à PostgreSQL™ 7.4. Voir Section 9.7.3.1, « Détails des expressions rationnelles » pour plus de détails.

`lo_compat_privileges` (boolean)

Dans les versions antérieures à la 9.0, les « Large Objects » n'avaient pas de droits d'accès et étaient, en réalité, lisibles et modifiables par tous les utilisateurs. L'activation de cette variable désactive les nouvelles vérifications sur les droits, pour améliorer la compatibilité avec les versions précédentes. Désactivé par défaut.

Configurer cette variable ne désactive pas toutes les vérifications de sécurité pour les « Large Objects » -- seulement ceux dont le comportement par défaut a changé avec PostgreSQL™ 9.0. Par exemple, `lo_import()` et `lo_export()` ont besoin de droits superutilisateur indépendants de cette configuration.

`sql_inheritance` (boolean)

Ce paramètre contrôle si les références de table doivent inclure les tables filles. La valeur par défaut est `on`, signifiant que les tables filles sont incluses (et de ce fait, un suffixe `*` est supposé par défaut. Si ce paramètre est désactivé (à `off`), les tables filles ne sont pas inclus (et de ce fait, le préfixe `ONLY` est ajouté). Le standard SQL requiert que les tables filles soient inclues, donc le paramètrages `off` n'est pas conforme au standard. Cependant, il est fourni par compatibilité avec les versions PostgreSQL™ antérieures à la 7.1. Voir Section 5.8, « L'héritage » pour plus d'informations.

Désactiver `sql_allance` n'est pas conseillé car le comportement induis par cette configuration porte à faire beaucoup d'erreurs. Ceci n'est pas constaté lorsque ce paramètre est activé comme le demande le standard SQL. Les discussions sur l'héritage dans ce manuel supposent généralement que ce paramètre est configuré à `on`.

`standard_conforming_strings` (boolean)

Contrôle si les chaînes ordinaires (`'...'`) traitent les antislashs littéralement, comme cela est indiqué dans le standard SQL. Désactivé par défaut (`off`), ce paramètre ramène PostgreSQL™ à son comportement historique pour le traitement des antislashs comme caractères d'échappement. La valeur par défaut sera passée à `on` dans une prochaine version pour améliorer la compatibilité avec le standard SQL. Les applications peuvent vérifier ce paramètre pour déterminer la façon dont elles doivent traiter les chaînes littérales. La présence de ce paramètre indique aussi que la syntaxe de chaîne d'échappement (`E'...'`) est supportée. La syntaxe de chaîne d'échappement (Section 4.1.2.2, « Constantes chaîne avec des échappements de style C ») doit être utilisée pour les applications traitant les antislashs comme des caractères d'échappement.

`synchronize_seqscans` (boolean)

Cette variable permet la synchronisation des parcours séquentiels de grosses tables pour que les parcours concurrents lisent le même bloc à peu près au même moment, et donc partagent la charge d'entrées/sorties. Quand ce paramètre est activé, un parcours peut commencer au milieu de la table, aller jusqu'à la fin, puis « revenir au début » pour récupérer toutes les lignes, ce qui permet de le synchroniser avec l'activité de parcours déjà entamés. Il peut en résulter des modifications non prévisibles dans l'ordre des lignes renvoyées par les requêtes qui n'ont pas de clause `ORDER BY`. Désactiver ce paramètre assure un comportement identique aux versions précédant la 8.3 pour lesquelles un parcours séquentiel commence toujours au début de la table. Activé par défaut (`on`).

18.12.2. Compatibilité entre la plateforme et le client

`transform_null_equals` (boolean)

Lorsque ce paramètre est activé (`on`), les expressions de la forme *`expr`* `= NULL` (ou `NULL =` *`expr`*) sont traitées comme *`expr`* `IS NULL`, c'est-à-dire qu'elles renvoient vrai si *`expr`* s'évalue à la valeur NULL, et faux sinon. Le bon comportement, compatible avec le standard SQL, de *`expr`* `= NULL` est de toujours renvoyer NULL (inconnu). De ce fait, ce paramètre est désactivé par défaut.

Toutefois, les formulaires filtrés dans Microsoft Access™ engendrent des requêtes qui utilisent *`expr`* `= NULL` pour tester les valeurs NULL. Il peut donc être souhaitable, lorsque cette intarface est utilisée pour accéder à une base de données, d'activer ce paramètre. Comme les expressions de la forme *`expr`* `= NULL` renvoient toujours la valeur NULL (en utilisant l'interprétation du standard SQL), elles ne sont pas très utiles et n'apparaissent pas souvent dans les applications normales. De ce fait, ce paramètre a peu d'utilité en pratique. Mais la sémantique des expressions impliquant des valeurs NULL est souvent source de confusion pour les nouveaux utilisateurs. C'est pourquoi ce paramètre n'est pas activé par défaut.

Ce paramètre n'affecte que la forme exacte `= NULL`, pas les autres opérateurs de comparaison ou expressions équivalentes en terme de calcul à des expressions qui impliquent l'opérateur égal (tels que `IN`). De ce fait, ce paramètre ne doit pas être considéré comme un correctif général à une mauvaise programmation.

De plus amples informations sont disponibles dans la Section 9.2, « Opérateurs de comparaison ».

18.13. Options préconfigurées

Les « paramètres » qui suivent sont en lecture seule. Ils sont déterminés à la compilation ou à l'installation de PostgreSQL™. De ce fait, ils sont exclus du fichier `postgresql.conf` d'exemple. Ces paramètres décrivent différents aspects du comportement de PostgreSQL™ qui peuvent s'avérer intéressants pour pour certaines applications, en particulier pour les interfaces d'administration.

`block_size` (integer)

Informe sur la taille d'un bloc disque. Celle-ci est déterminée par la valeur de `BLCKSZ` à la construction du serveur. La valeur par défaut est de 8192 octets. La signification de diverses variables de configuration (shared_buffers, par exemple) est influencée par `block_size`. Voir la Section 18.4, « Consommation des ressources » pour plus d'informations.

`integer_datetimes` (boolean)

Informe sur la construction de PostgreSQL™ avec le support des dates et heures sur des entiers de 64 bits. Ceci peut être désactivé avec l'option `--disable-integer-datetimes` au moment de la construction de PostgreSQL™. La valeur par défaut est `on`.

`lc_collate` (string)

Affiche la locale utilisée pour le tri des données de type texte. Voir la Section 22.1, « Support des locales » pour plus d'informations. La valeur est déterminée lors de la création d'une base de données.

`lc_ctype` (string)

Affiche la locale qui détermine les classifications de caractères. Voir la Section 22.1, « Support des locales » pour plus d'informations. La valeur est déterminée lors de la création d'une base de données. Elle est habituellement identique à `lc_collate`. Elle peut, toutefois, pour des applications particulières, être configurée différemment.

`max_function_args` (integer)

Affiche le nombre maximum d'arguments des fonctions. Ce nombre est déterminé par la valeur de `FUNC_MAX_ARGS` lors de la construction du serveur. La valeur par défaut est de 100 arguments.

`max_identifier_length` (integer)

Affiche la longueur maximale d'un identifiant. Elle est déterminée à NAMEDATALEN - 1 lors de la construction du serveur. La valeur par défaut de NAMEDATALEN est 64 ; la valeur par défaut de max_identifier_length est, de ce fait, de 63 octets mais peut être moins de 63 caractères lorsque des encodages multi-octets sont utilisés.

max_index_keys (integer)

Affiche le nombre maximum de clés d'index. Ce nombre est déterminé par la valeur de INDEX_MAX_KEYS lors de la construction du serveur. La valeur par défaut est de 32 clés.

segment_size (integer)

Retourne le nombre de blocs (pages) qui peuvent être stockés dans un segment de fichier. C'est déterminé par la valeur de RELSEG_SIZE à la compilation du serveur. La valeur maximum d'un fichier de segment en octet est égal à segment_size multiplié par block_size ; par défaut, c'est 1 Go.

server_encoding (string)

Affiche l'encodage de la base de données (jeu de caractères). Celui-ci est déterminé lors de la création de la base de données. Les clients ne sont généralement concernés que par la valeur de client_encoding.

server_version (string)

Affiche le numéro de version du serveur. Celui-ci est déterminé par la valeur de PG_VERSION lors de la construction du serveur.

server_version_num (integer)

Affiche le numéro de version du serveur sous la forme d'un entier. Celui-ci est déterminé par la valeur de PG_VERSION_NUM lors de la construction du serveur.

wal_block_size (integer)

Retourne la taille d'un bloc disque de WAL. C'est déterminé par la valeur XLOG_BLCKSZ à la compilation du serveur. La valeur par défaut est 8192 octets.

wal_segment_size (integer)

Retourne le nombre de blocs (pages) dans un fichier de segment WAL. La taille totale d'un fichier de segment WAL en octets est égale à wal_segment_size multiplié par wal_block_size ; Par défaut, c'est 16 Mo. Voir Section 29.4, « Configuration des journaux de transaction » pour plus d'informations.

18.14. Options personnalisées

Cette fonctionnalité a été conçue pour permettre l'ajout de paramètres habituellement inconnus de PostgreSQL™ par des modules complémentaires (comme les langages procéduraux). Cela permet de configurer ces modules de façon standard.

custom_variable_classes (string)

Cette variable indique les noms de classe à utiliser pour les variables personnalisées, sous la forme d'une liste séparée par des virgules. Une variable personnalisée est une variable habituellement inconnue de PostgreSQL™ mais utilisée par certains modules complémentaires. Les noms de ces variables doivent être constitués d'un nom de classe, d'un point et d'un nom de variable. custom_variable_classes indique tous les noms de classe utilisés dans une installation particulière. Ce paramètre ne peut être configuré que dans le fichier postgresql.conf ou indiqué sur la ligne de commande.

La difficulté de configurer des variables personnalisées dans postgresql.conf se situe dans le fait que le fichier doit être lu avant que les modules complémentaires ne soient chargés. De ce fait, les variables sont habituellement rejetées parce qu'inconnues. Lorsque custom_variable_classes est initialisé, le serveur accepte les définitions de variables arbitraires à l'intérieur de chaque classe indiquée. Ces variables sont traitées comme des emplacements et n'ont aucune fonction tant que le module qui les définit n'est pas chargé. Quand un module d'une classe spécifique est chargé, il ajoute les bonnes définitions de variables pour son nom de classe, convertit les valeurs des emplacements en fonction de leurs définitions et émet des messages d'avertissement pour tout emplacement non reconnu de la classe restant.

Exemple de ce que peut contenir postgresql.conf lorsque les variables personnalisées sont utilisées :

```
custom_variable_classes = 'plpgsql,plperl'
plpgsql.variable_conflict = use_variable
plperl.use_strict = true
plruby.use_strict = true        # generates error, unknown class name
```

18.15. Options pour les développeurs

Les paramètres qui suivent permettent de travailler sur les sources de PostgreSQL™ et, dans certains cas, fournissent une aide à la récupération de bases de données sévèrement endommagées. Il n'y a aucune raison de les utiliser en configuration de production. En tant que tel, ils sont exclus du fichier d'exemple de `postgresql.conf`. Un certain nombre d'entre eux requièrent des options de compilation spéciales pour fonctionner.

`allow_system_table_mods` (boolean)

Autorise la modification de la structure des tables systèmes. Ce paramètre, utilisé par **initdb**, n'est modifiable qu'au démarrage du serveur.

`debug_assertions` (boolean)

Active différentes vérifications d'affectations. C'est une aide au débogage. En cas de problèmes étranges ou d'arrêts brutaux, ce paramètre peut être activé car il permet de remonter des erreurs de programmation. Pour utiliser ce paramètre, la macro USE_ASSERT_CHECKING doit être définie lors de la construction de PostgreSQL™ (à l'aide de l'option `--enable-cassert` de **configure**). Ce paramètre est activé par défaut si PostgreSQL™ a été construit avec l'activation des assertions.

`ignore_system_indexes` (boolean)

Ignore les index système lors de la lecture des tables système (mais continue de les mettre à jour lors de modifications des tables). Cela s'avère utile lors de la récupération d'index système endommagés. Ce paramètre ne peut pas être modifié après le démarrage de la session.

`post_auth_delay` (integer)

Si ce paramètre est différent de zéro, un délai de ce nombre de secondes intervient, après l'étape d'authentification, lorsqu'un nouveau processus serveur est lancé. Ceci a pour but de donner l'opportunité aux développeurs d'attacher un débogueur au processus serveur. Ce paramètre ne peut pas être modifié après le démarrage de la session.

`pre_auth_delay` (integer)

Si ce paramètre est différent de zéro, un délai de ce nombre de secondes intervient juste après la création d'un nouveau processus, avant le processus d'authentification. Ceci a pour but de donner une opportunité aux développeurs d'attacher un débogueur au processus serveur pour tracer les mauvais comportements pendant l'authentification. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`trace_notify` (boolean)

Produit un grand nombre de sorties de débogage pour les commandes **LISTEN** et **NOTIFY**. client_min_messages ou log_min_messages doivent être positionnées à `DEBUG1` ou plus bas pour envoyer cette sortie sur les traces client ou serveur, respectivement.

`trace_recovery_messages` (enum)

Contrôle les niveaux des traces écrites dans le journal applicatif pour les modules nécessaires lors du traitement de la restauration. Cela permet à l'utilisateur de surcharger la configuration normale de log_min_messages, mais seulement pour des messages spécifiques. Ça a été ajouté principalement pour débugger Hot Standby. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1` et `LOG`. La valeur par défaut, `LOG`, n'affecte pas les décisions de trace. Les autres valeurs causent l'apparition de messages de débogage relatifs à la restauration pour tous les messages de ce niveau et des niveaux supérieurs. Elles utilisent malgré tout le niveau `LOG` ; pour les configurations habituelles de `log_min_messages`, cela résulte en un envoi sans condition dans les traces du serveur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`trace_sort` (boolean)

Si ce paramètre est actif, des informations concernant l'utilisation des ressources lors des opérations de tri sont émises. Ce paramètre n'est disponible que si la macro TRACE_SORT a été définie lors de la compilation de PostgreSQL™ (néanmoins, TRACE_SORT est actuellement définie par défaut).

`trace_locks` (boolean)

Si activé, émet des informations à propos de l'utilisation des verrous. L'information fournie inclut le type d'opération de verrouillage, le type de verrou et l'identifiant unique de l'objet verrouillé ou déverrouillé. Sont aussi inclus les masques de bits pour les types de verrous déjà accordés pour cet objet, ainsi que pour les types de verrous attendus sur cet objet. Pour chaque type de verrou un décompte du nombre de verrous accordés et en attente est aussi retourné, ainsi que les totaux. Un exemple de sortie dans le journal applicatif est montré ici :

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
```

```
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

Les détails de la structure retournée peuvent être trouvés dans `src/include/storage/lock.h`.

Ce paramètre n'est disponible que si la macro LOCK_DEBUG a été définie quand PostgreSQL™ a été compilé.

`trace_lwlocks` (boolean)
Si à on, génère des informations à propos de l'utilisation de verrous légers (lightweight lock). Les verrous légers servent principalement à fournir une exclusion mutuelle d'accès aux structures de données en mémoire partagée.

Ce paramètre n'est disponible que si la macro LOCK_DEBUG a été définie quand PostgreSQL™ a été compilé.

`trace_userlocks` (boolean)
Si activé, génère des informations à propos de l'utilisation de verrous utilisateurs. La sortie est la même que pour trace_locks, mais restreinte aux verrous informatifs.

`trace_lock_oidmin` (integer)
Si positionné, ne trace pas les verrouillages pour des tables en dessous de cet OID. (à utiliser pour ne pas avoir de sortie pour les tables systèmes)

Ce paramètre n'est disponible que si la macro LOCK_DEBUG a été définie quand PostgreSQL™ a été compilé.

`trace_lock_table` (integer)
Tracer les verrouillages sur cette table de façon inconditionnelle.

Ce paramètre n'est disponible que si la macro LOCK_DEBUG a été définie quand PostgreSQL™ a été compilé.

`debug_deadlocks` (boolean)
Si positionné, génère des informations à propos de tous les verrous en cours quand l'expiration de temps d'attente d'un verrou mortel se produit.

Ce paramètre n'est disponible que si la macro LOCK_DEBUG a été définie quand PostgreSQL™ a été compilé.

`log_btree_build_stats` (boolean)
Si positionné, trace des statistiques d'utilisation de ressource système (mémoire et processeur) sur différentes opérations B-tree.

Ce paramètre n'est disponible que si la macro BTREE_BUILD_STATS a été définie quand PostgreSQL™ a été compilé.

`wal_debug` (boolean)

Si ce paramètre est positionné à `on`, une sortie de débogage relative aux WAL est émise. Ce paramètre n'est disponible que si la macro WAL_DEBUG a été définie au moment de la compilation de PostgreSQL™.

`zero_damaged_pages` (boolean)

La détection d'une en-tête de page endommagée cause généralement le rapport d'une erreur par PostgreSQL™ et l'annulation de la commande courante. Initialiser `zero_damaged_pages` à `on` oblige le système à remonter un avertissement, à vider la page endommagée et à continuer le traitement. Ce comportement *détruit les données* (à savoir, toutes les lignes de la page endommagée). Mais cela permet de passer outre l'erreur et de récupérer les lignes des pages non endommagées qui peuvent être présentes dans la table. Ce paramètre s'avère donc utile pour récupérer les données si la corruption est due à une erreur matérielle ou logicielle. Il est généralement préférable de ne pas configurer ce paramètre à `on`, sauf à abandonner l'espoir de récupérer les données des pages endommagées d'une table. Par défaut ce paramètre est désactivé. Seul un superutilisateur peut modifier cette configuration.

18.16. Options courtes

Pour des raisons pratiques, il existe également des commutateurs en ligne de commandes sur une seule lettre pour certains paramètres. Ceux-ci sont décrits dans le Tableau 18.2, « Clé d'option courte ». Certaines des options existent pour des raisons historiques et leur présence en tant qu'option courte ne doit pas être vue comme une incitation à son utilisation massive.

Tableau 18.2. Clé d'option courte

Option courte	Équivalent
`-A` *x*	`debug_assertions =` *x*
`-B` *x*	`shared_buffers =` *x*
`-d` *x*	`log_min_messages = DEBUG`*x*
`-e`	`datestyle = euro`
`-fb`, `-fh`, `-fi`, `-fm`, `-fn`, `-fs`, `-ft`	`enable_bitmapscan = off`, `enable_hashjoin = off`, `enable_indexscan = off`, `enable_mergejoin = off`, `enable_nestloop = off`, `enable_seqscan = off`, `enable_tidscan = off`
`-F`	`fsync = off`
`-h` *x*	`listen_addresses =` *x*
`-i`	`listen_addresses = '*'`
`-k` *x*	`unix_socket_directory =` *x*
`-l`	`ssl = on`
`-N` *x*	`max_connections =` *x*
`-O`	`allow_system_table_mods = on`
`-p` *x*	`port =` *x*
`-P`	`ignore_system_indexes = on`
`-s`	`log_statement_stats = on`
`-S` *x*	`work_mem =` *x*
`-tpa`, `-tpl`, `-te`	`log_parser_stats = on`, `log_planner_stats = on`, `log_executor_stats = on`
`-W` *x*	`post_auth_delay =` *x*

Chapitre 19. Authentification du client

Quand une application client se connecte au serveur de bases de données, elle indique le nom de l'utilisateur de base de données à utiliser pour la connexion, de la même façon qu'on se connecte à un ordinateur Unix sous un nom d'utilisateur particulier. Au sein de l'environnement SQL, le nom d'utilisateur de la base de données active détermine les droits régissant l'accès aux objets de la base de données -- voir le Chapitre 20, Rôles et droits de la base de données pour plus d'informations. Ainsi, il est essentiel de limiter le nombre de bases de données auxquelles les utilisateurs peuvent se connecter.

> **Note**
>
> Comme expliqué dans le Chapitre 20, Rôles et droits de la base de données, PostgreSQL™ gère les droits par l'intermédiaire des « rôles ». Dans ce chapitre, le terme *utilisateur de bases de données* est utilisé pour signifier « rôle disposant du droit `LOGIN` ».

L'*authentification* est le processus par lequel le serveur de bases de données établit l'identité du client et, par extension, détermine si l'application client (ou l'utilisateur qui l'utilise) est autorisée à se connecter avec le nom d'utilisateur de bases de données indiqué.

PostgreSQL™ offre quantité de méthodes d'authentification différentes. La méthode utilisée pour authentifier une connexion client particulière peut être sélectionnée d'après l'adresse (du client), la base de données et l'utilisateur.

Les noms d'utilisateur de bases de données sont séparés de façon logique des noms d'utilisateur du système d'exploitation sur lequel tourne le serveur. Si tous les utilisateurs d'un serveur donné ont aussi des comptes sur la machine serveur, il peut être pertinent d'attribuer aux utilisateurs de bases de données des noms qui correspondent à ceux des utilisateurs du système d'exploitation. Cependant, un serveur qui accepte des connexions distantes peut avoir des utilisateurs de bases de données dépourvus de compte correspondant sur le système d'exploitation. Dans ce cas, aucune correspondance entre les noms n'est nécessaire.

19.1. Le fichier `pg_hba.conf`

L'authentification du client est contrôlée par un fichier, traditionnellement nommé `pg_hba.conf` et situé dans le répertoire data du groupe de bases de données, par exemple `/usr/local/pgsql/data/pg_hba.conf` (HBA signifie « host-based authentication » : authentification fondée sur l'hôte.) Un fichier `pg_hba.conf` par défaut est installé lorsque le répertoire data est initialisé par **initdb**. Néanmoins, il est possible de placer le fichier de configuration de l'authentification ailleurs ; voir le paramètre de configuration hba_file.

Le format général du fichier `pg_hba.conf` est un ensemble d'enregistrements, un par ligne. Les lignes vides sont ignorées tout comme n'importe quel texte placé après le caractère de commentaire #. Un enregistrement est constitué d'un certain nombre de champs séparés par des espace et/ou des tabulations. Les enregistrements ne peuvent pas être continués sur plusieurs lignes. Les champs peuvent contenir des espaces si la valeur du champ est mise entre guillemets. Mettre entre guillemets un des mots-clés dans un champ base de données ou utilisateur (par exemple, `all` ou `replication`) fait que le mot perd son interprétation spéciale, ou correspond à la base de données ou à l'utilisateur ayant ce nom.

Chaque enregistrement précise un type de connexion, une plage d'adresses IP (si approprié au type de connexion), un nom de base de données, un nom d'utilisateur et la méthode d'authentification à utiliser pour les connexions correspondant à ces paramètres. Le premier enregistrement qui correspond au type de connexion, à l'adresse client, à la base de données demandée et au nom d'utilisateur est utilisé pour effectuer l'authentification. Il n'y a pas de suite après une erreur (« fall-through » ou « backup ») : si un enregistrement est choisi et que l'authentification échoue, les enregistrements suivants ne sont pas considérés. Si aucun enregistrement ne correspond, l'accès est refusé.

Un enregistrement peut avoir l'un des sept formats suivants.

```
local      database  user  auth-method  [auth-options]
host       database  user  CIDR-address  auth-method  [auth-options]
hostssl    database  user  CIDR-address  auth-method  [auth-options]
hostnossl  database  user  CIDR-address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

La signification des champs est la suivante :

`local`
: Cet enregistrement intercepte les tentatives de connexion qui utilise les sockets du domaine Unix. Sans enregistrement de ce type, les connexions de sockets du domaine Unix ne sont pas autorisées.

`host`
Cet enregistrement intercepte les tentatives de connexion par TCP/IP. Les lignes `host` s'appliquent à toute tentative de connexion, SSL ou non.

> **Note**
>
> Les connexions TCP/IP ne sont pas autorisées si le serveur n'est pas démarré avec la valeur appropriée du paramètre de configuration listen_addresses. En effet, par défaut, le serveur n'écoute que les connexions TCP/IP en provenance de l'adresse `loopback` locale, `localhost`.

`hostssl`
Cet enregistrement intercepte les seules tentatives de connexions par TCP/IP qui utilisent le chiffrement SSL.

Pour utiliser cette fonction, le serveur doit être compilé avec le support de SSL. De plus, SSL doit être activé au démarrage du serveur en positionnant le paramètre de configuration ssl (voir la Section 17.8, « Connexions tcp/ip sécurisées avec ssl » pour plus d'informations).

`hostnossl`
Cet enregistrement a un comportement opposé à `hostssl` : il n'intercepte que les tentatives de connexion qui n'utilisent pas SSL.

`database`
Indique les noms des bases de données concernées par l'enregistrement. La valeur `all` indique qu'il concerne toutes les bases de données. Le terme `sameuser` indique que l'enregistrement coïncide si la base de données demandée a le même nom que l'utilisateur demandé. Le terme `samerole` indique que l'utilisateur demandé doit être membre du rôle portant le même nom que la base de données demandée (`samegroup` est obsolète bien qu'il soit toujours accepté comme écriture alternative de `samerole`.). La valeur `replication` indique que l'enregistrement établit une correspondance si une connexion de réplication est demandée (notez que les connexions de réplication ne ciblent pas une base de données particulière). Dans tous les autres cas, c'est le nom d'une base de données particulière. Plusieurs noms de base de données peuvent être fournis en les séparant par des virgules. Un fichier contenant des noms de base de données peut être indiqué en faisant précéder le nom du fichier de `@`.

`user`
Indique les utilisateurs de bases de données auxquels cet enregistrement correspond. La valeur `all` indique qu'il concerne tous les utilisateurs. Dans le cas contraire, il s'agit soit du nom d'un utilisateur spécifique de bases de données ou d'un nom de groupe précédé par un `+` (il n'existe pas de véritable distinction entre les utilisateurs et les groupes dans PostgreSQL™ ; un `+` signifie exactement « établit une correspondance pour tous les rôles faisant parti directement ou indirectement de ce rôle » alors qu'un nom sans `+` établit une correspondance avec ce rôle spécifique). Plusieurs noms d'utilisateurs peuvent être fournis en les séparant par des virgules. Un fichier contenant des noms d'utilisateurs peut être indiqué en faisant précéder le nom du fichier de `@`.

`CIDR-address`
Indique la plage d'adresses IP client à laquelle correspond cet enregistrement. Ce champ contient une adresse IP dans la notation décimale standard et une longueur de masque CIDR (les adresses IP ne peuvent qu'être indiquées numériquement, pas en tant que nom d'hôte ou de domaine). La longueur du masque indique le nombre de bits forts pour lesquels une correspondance doit être trouvée avec l'adresse IP du client. Les bits de droite doivent valoir zéro dans l'adresse IP indiquée. Il ne doit y avoir aucune espace entre l'adresse IP, le `/` et la longueur du masque CIDR.

À la place du *`CIDR-address`*, vous pouvez écrire `samehost` pour correspondre aux adresses IP du serveur ou `samenet` pour correspondre à toute adresse du sous-réseau auquel le serveur est directement connecté.

Une adresse CIDR (*`CIDR-address`*) est typiquement `172.20.143.89/32` pour un hôte seul, `172.20.143.0/24` pour un petit réseau ou `10.6.0.0/16` pour un réseau plus grand. `0.0.0.0/0` (« que des zéros ») représente toutes les adresses. Pour n'indiquer qu'un seul hôte, on utilise un masque de 32 pour IPv4 ou 128 pour IPv6. Dans une adresse réseau, ne pas oublier les zéros finaux.

Une adresse IP indiquée au format IPv4 coïncide avec les connexions IPv6 d'adresse correspondante. Par exemple, `127.0.0.1` correspond à l'adresse IPv6 `::ffff:127.0.0.1`. Une entrée donnée au format IPv6 correspond uniquement aux connexions IPv6 même si l'adresse représentée est dans le domaine IPv4-vers-IPv6. Les adresses au format IPv6 sont rejetées si la bibliothèque système C ne supporte pas les adresses IPv6.

Ce champ ne concerne que les enregistrements `host`, `hostssl` et `hostnossl`.

`IP-address`, *`IP-mask`*
Ces champs peuvent être utilisés comme alternative à la notation *`CIDR-address`*. Au lieu de spécifier la longueur du masque, le masque réel est indiquée dans une colonne distincte. Par exemple, `255.0.0.0` représente une longueur de masque CIDR IPv4 de 8, et `255.255.255.255` représente une longueur de masque de 32.

Ces champs ne concernent que les enregistrements `host`, `hostssl` et `hostnossl`.

`auth-method`
Indique la méthode d'authentification à utiliser lors d'une connexion via cet enregistrement. Les choix possibles sont résumés ici ; les détails se trouvent dans la Section 19.3, « Méthodes d'authentification ».

`trust`
Autorise la connexion sans condition. Cette méthode permet à quiconque peut se connecter au serveur de bases de données de s'enregistrer sous n'importe quel utilisateur PostgreSQL™ de son choix sans mot de passe ou autre authentification. Voir la Section 19.3.1, « Authentification trust » pour les détails.

`reject`
Rejette la connexion sans condition. Ce cas est utile pour « filtrer » certains hôtes d'un groupe, par exemple une ligne `reject` peut bloquer la connexion d'un hôte spécifique alors qu'une ligne plus bas permettra aux autres hôtes de se connecter à partir d'un réseau spécifique.

`md5`
Demande au client de fournir un mot de passe chiffré MD5 pour l'authentification. Voir la Section 19.3.2, « Authentification par mot de passe » pour les détails.

`password`
Requiert que le client fournisse un mot de passe non chiffré pour l'authentification. Comme le mot de passe est envoyé en clair sur le réseau, ceci ne doit pas être utilisé sur des réseaux non dignes de confiance. Voir la Section 19.3.2, « Authentification par mot de passe » pour les détails.

`gss`
Utilise GSSAPI pour authentifier l'utilisateur. Disponible uniquement pour les connexions TCP/IP. Voir Section 19.3.3, « Authentification GSSAPI » pour les détails.

`sspi`
Utilise SSPI pour authentifier l'utilisateur. Disponible uniquement sur Windows. Voir Section 19.3.4, « Authentification SSPI » pour plus de détails.

`krb5`
Utilise Kerberos V5 pour authentifier l'utilisateur. Ceci n'est disponible que pour les connexions TCP/IP. Voir la Section 19.3.5, « Authentification Kerberos » pour les détails.

`ident`
Récupère le nom de l'utilisateur du système d'exploitation du client (pour les connexions TCP/IP en contactant le serveur d'identification sur le client, pour les connexions locales en l'obtenant du système d'exploitation) et vérifie que cela correspond au nom d'utilisateur de base de données demandé. Voir la Section 19.3.6, « Authentification fondée sur ident » ci-dessous pour les détails.

`ldap`
Authentification par un serveur LDAP. Voir la Section 19.3.7, « Authentification LDAP » pour les détails.

`radius`
Authentification par un serveur RADIUS. Voir Section 19.3.8, « Authentification RADIUS » pour les détails.

`cert`
Authentification par certificat client SSL. Voir Section 19.3.9, « Authentification de certificat » pour les détails.

`pam`
Authentification par les Pluggable Authentification Modules (PAM) fournis par le système d'exploitation. Voir la Section 19.3.10, « Authentification PAM » pour les détails.

`auth-options`
Après le champ *`auth-method`*, on peut trouver des champs de la forme *`nom=valeur`* qui spécifient des options pour la méthode d'authentification. Les détails sur les options disponibles apparaissent ci-dessous pour chaque méthode d'authentification.

Les fichiers inclus par les constructions `@` sont lus comme des listes de noms, séparés soit par des espaces soit par des virgules. Les commentaires sont introduits par le caractère # comme dans `pg_hba.conf`, et les constructions `@` imbriquées sont autorisées. À moins que le nom du fichier qui suit `@` ne soit un chemin absolu, il est supposé relatif au répertoire contenant le fichier le référençant.

Les enregistrements du fichier `pg_hba.conf` sont examinés séquentiellement à chaque tentative de connexion, l'ordre des enregistrements est donc significatif. Généralement, les premiers enregistrements ont des paramètres d'interception de connexions stricts et des méthodes d'authentification peu restrictives tandis que les enregistrements suivants ont des paramètres plus larges et

des méthodes d'authentification plus fortes. Par exemple, on peut souhaiter utiliser l'authentification `trust` pour les connexions TCP/IP locales mais demander un mot de passe pour les connexion TCP/IP distantes. Dans ce cas, l'enregistrement précisant une authentification `trust` pour les connexions issues de 127.0.0.1 apparaît avant un enregistrement indiquant une authentification par mot de passe pour une plage plus étendue d'adresses IP client autorisées.

Le fichier `pg_hba.conf` est lu au démarrage et lorsque le processus serveur principal reçoit un signal `SIGHUP`. Si le fichier est édité sur un système actif, on peut signaler au postmaster (en utilisant `pg_ctl reload` ou `kill -HUP`) de relire le fichier.

Astuce

Pour se connecter à une base particulière, un utilisateur doit non seulement passer les vérifications de `pg_hba.conf` mais doit également avoir le droit `CONNECT` sur cette base. Pour contrôler qui peut se connecter à quelles bases, il est en général plus facile de le faire en donnant ou retirant le privilège `CONNECT` plutôt qu'en plaçant des règles dans le fichier `pg_hba.conf`.

Quelques exemples d'entrées de `pg_hba.conf` sont donnés ci-dessous dans l'Exemple 19.1, « Exemple d'entrées de `pg_hba.conf` ». Voir la section suivante pour les détails des méthodes d'authentification.

Exemple 19.1. Exemple d'entrées de `pg_hba.conf`

```
# Permettre à n'importe quel utilisateur du système local de se connecter
# à la base de données sous n'importe quel nom d'utilisateur au travers
# des sockets de domaine Unix (par défaut pour les connexions locales).
#
# TYPE  DATABASE    USER        CIDR-ADDRESS        METHOD
local   all         all                             trust

# La même chose en utilisant les connexions TCP/IP locales loopback.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS        METHOD
host    all         all         127.0.0.1/32          trust

# Pareil mais en utilisant une colonne netmask distincte.
#
# TYPE  DATABASE    USER        IP-ADDRESS          IP-mask              METHOD
host    all         all         127.0.0.1           255.255.255.255      trust

# Permettre à n'importe quel utilisateur de n'importe quel hôte d'adresse IP
# 192.168.93.x de se connecter à la base de données "postgres" sous le nom
# d'utilisateur qu'ident signale à la connexion (généralement le
# nom utilisateur du système d'exploitation).
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    postgres    all         192.168.93.0/24       ident

# Permet à un utilisateur de l'hôte 192.168.12.10 de se connecter à la base de
# données "postgres" si le mot de passe de l'utilisateur est correctement fourni.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    postgres    all         192.168.12.10/32      md5

# Si aucune ligne "host" ne précède, ces deux lignes rejettent toutes
# les connexions en provenance de 192.168.54.1 (puisque cette entrée déclenche
# en premier), mais autorisent les connexions Kerberos 5 de n'importe où
# ailleurs sur l'Internet. Le masque zéro signifie qu'aucun bit de l'ip de
# l'hôte n'est considéré, de sorte à correspondre à tous les hôtes.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         192.168.54.1/32       reject
host    all         all         0.0.0.0/0             krb5

# Permettre à tous les utilisateurs de se connecter depuis 192.168.x.x à n'importe
# quelle base de données s'ils passent la verification d'identification. Si,
# par exemple, ident indique que l'utilisateur est "bryanh" et qu'il
# demande à se connecter en tant qu'utilisateur PostgreSQL "guest1", la
# connexion n'est permise que s'il existe une entrée dans pg_ident.conf pour la
# correspondance "omicron" disant que "bryanh" est autorisé à se connecter en
```

```
# tant que "guest1".
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         192.168.0.0/16        ident map=omicron

# Si ces trois lignes traitent seules les connexions locales, elles
# n'autorisent les utilisateurs locaux qu'à se connecter à leur propre
# base de données (base ayant le même nom que leur nom
# d'utilisateur) exception faite des administrateurs
# et des membres du rôle "support" qui peuvent se connecter à toutes les bases
# de données. Le fichier $PGDATA/admins contient une liste de noms
# d'administrateurs. Un mot de passe est requis dans tous les cas.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
local   sameuser    all                               md5
local   all         @admins                           md5
local   all         +support                          md5

# Les deux dernières lignes ci-dessus peuvent être combinées en une seule ligne :
local   all         @admins,+support                  md5

# La colonne database peut aussi utiliser des listes et des noms de fichiers :
local   db1,db2,@demodbs  all                         md5
```

19.2. Correspondances d'utilisateurs

Lorsqu'on utilise une authentification externe telle que Ident ou GSSAPI, le nom de l'utilisateur du système d'exploitation qui a initié la connexion peut ne pas être le même que celui de l'utilisateur de la base à laquelle il tente de se connecter. Dans ce cas, une table de correspondance d'identités peut être mise en place pour faire correspondre le nom d'utilisateur système au nom d'utilisateur base de donnée. Pour utiliser une table de correspondance d'identités, spécifiez `map=`*`nom-table`* dans le champ options de `pg_hba.conf`. Cette option est supportée pour toutes les méthodes d'authentification qui reçoivent des noms d'utilisateurs externes. Comme différentes correspondances peuvent être nécessaires pour différentes connexions, le nom de la table à utiliser doit être spécifié dans le paramètre *`nom-table`* de `pg_hba.conf` afin d'indiquer quelle table utiliser pour chaque connexion.

Les tables de correspondance de noms d'utilisateurs sont définies dans le fichier de correspondance, qui par défaut s'appelle `pg_ident.conf` et est stocké dans le répertoire de données du cluster. (Toutefois, il est possible de placer la table de correspondance ailleurs ; voir le paramètre de configuration ident_file.) Le fichier de table de correspondance contient des lignes de la forme suivante :

```
nom-table nom-d-utilisateur-systeme nom-d-utilisateur-base
```

Les commentaires et les blancs sont traités de la même façon que dans `pg_hba.conf`. Le *`nom-table`* est un nom arbitraire qui sera utilisé pour faire référence à cette table de correspondance dans `pg_hba.conf`. Les deux autres champs spécifient un nom d'utilisateur du système d'exploitation et un nom d'utilisateur de la base de données correspondant. Le même *`nom-correspondance`* peut être utilisé de façon répétée pour indiquer plusieurs correspondances d'utilisateur dans la même carte.

Il n'y a aucune restriction sur le nombre d'utilisateurs de base de données auxquels un utilisateur du système d'exploitation peut correspondre et vice-versa. Du coup, les entrées dans une carte signifient que « cet utilisateur du système d'exploitation est autorisé à se connecter en tant que cet utilisateur de la base de données », plutôt que supposer qu'ils sont équivalents. La connexion sera autorisée s'il existe une entrée dans la carte qui correspond au nom d'utilisateur obtenu à partir du système d'authentification externe pour le nom de l'utilisateur de la base de données que l'utilisateur a indiqué.

Si le champ *`system-username`* commence avec un slash (`/`), le reste du champ est traité comme une expression rationnelle. (Voir Section 9.7.3.1, « Détails des expressions rationnelles » pour les détails de la syntaxe des expressions rationnelles avec PostgreSQL™.). L'expression rationnelle peut inclure une copie (sous-expression entre parenthèses), qui peut ensuite être référencée dans le champ *`database-username`* avec le joker `\1` (antislash-un). Ceci permet la correspondance de plusieurs noms d'utilisateurs sur une seule ligne, ce qui est particulièrement utile pour les substitutions simples. Par exemple, ces entrées

```
mymap   /^(.*)@mydomain\.com$      \1
mymap   /^(.*)@otherdomain\.com$   guest
```

supprimeront la partie domaine pour les utilisateurs de système d'exploitation dont le nom finissent avec `@mydomain.com`, et permettront aux utilisateurs dont le nom se termine avec `@otherdomain.com` de se connecter en tant que `guest`.

> **Astuce**
>
> Gardez en tête que, par défaut, une expression rationnelle peut correspondre à une petite partie d'une chaîne. Il est généralement conseillé d'utiliser les jokers `^` et `$`, comme indiqué dans l'exemple ci-dessus, pour forcer une correspondance sur le nom entier de l'utilisateur du système d'exploitation.

Le fichier `pg_ident.conf` est lu au démarrage et quand le processus principal du serveur reçoit un signal `SIGHUP`. Si vous éditez le fichier sur un système en cours d'utilisation, vous devez notifier le postmaster (en utilisant`pg_ctl reload` ou `kill -HUP`) pour lui faire relire le fichier.

Un fichier `pg_ident.conf` qui pourrait être utilisé avec le fichier `pg_hba.conf` de Exemple 19.1, « Exemple d'entrées de `pg_hba.conf` » est montré en Exemple 19.2, « Un exemple de fichier `pg_ident.conf` ». Dans cet exemple, toute personne connectée sur une machine du réseau 192.168 qui n'a pas le nom d'utilisateur du système d'exploitation `bryanh`, `ann`, ou `robert` verrait son accès refusé. L'utilisateur Unix `robert` ne verrait son accès autorisé que lorsqu'il essaye de se connecter en tant qu'utilisateur PostgreSQL™ `bob`, pas en tant que `robert` ou qui que ce soit d'autre. `ann` ne serait autorisée à se connecter qu'en tant que `ann`. L'utilisateur `bryanh` aurait le droit de se connecter soit en tant que `bryanh`, soit en tant que `guest1`.

Exemple 19.2. Un exemple de fichier `pg_ident.conf`

```
# MAPNAME     SYSTEM-USERNAME   PG-USERNAME

omicron       bryanh            bryanh
omicron       ann               ann
# bob has user name robert on these machines
omicron       robert            bob
# bryanh can also connect as guest1
omicron       bryanh            guest1
```

19.3. Méthodes d'authentification

Les sous-sections suivantes décrivent les méthodes d'authentification en détail.

19.3.1. Authentification trust

Quand l'authentification `trust` est utilisée, PostgreSQL™ considère que quiconque peut se connecter au serveur est autorisé à accéder à la base de données quel que soit le nom d'utilisateur de bases de données qu'il fournit (même les noms des super-utilisateurs). Les restrictions apportées dans les colonnes `database` et `user` continuent évidemment de s'appliquer. Cette méthode ne doit être utilisée que si le système assure un contrôle adéquat des connexions au serveur.

L'authentification `trust` est appropriée et très pratique pour les connexions locales sur une station de travail mono-utilisateur. Elle n'est généralement *pas* appropriée en soi sur une machine multi-utilisateur. Cependant, `trust` peut tout de même être utilisé sur une machine multi-utilisateur, si l'accès au fichier socket de domaine Unix est restreint par les permissions du système de fichiers. Pour ce faire, on peut positionner les paramètres de configuration `unix_socket_permissions` (et au besoin `unix_socket_group`) comme cela est décrit dans la Section 18.3, « Connexions et authentification ». On peut également positionner le paramètre de configuration `unix_socket_directory` pour placer le fichier de socket dans un répertoire à l'accès convenablement restreint.

Le réglage des droits du système de fichiers n'a d'intérêt que le cas de connexions par les sockets Unix. Les droits du système de fichiers ne restreignent pas les connexions TCP/IP locales. Ainsi, pour utiliser les droits du système de fichiers pour assurer la sécurité locale, il faut supprimer la ligne `host ...127.0.0.1 ...` de `pg_hba.conf` ou la modifier pour utiliser une méthode d'authentification différente de `trust`.

L'authentification `trust` n'est envisageable, pour les connexions TCP/IP, que si chaque utilisateur de chaque machine autorisée à se connecter au serveur par les lignes `trust` du fichier `pg_hba.conf` est digne de confiance. Il est rarement raisonnable d'utiliser `trust` pour les connexions autres que celles issues de `localhost` (127.0.0.1).

19.3.2. Authentification par mot de passe

Les méthodes fondées sur une authentification par mot de passe sont `md5` et `password`. Ces méthodes fonctionnent de façon analogue à l'exception du mode d'envoi du mot de passe à travers la connexion : respectivement, hachage MD5 et texte en clair.

S'il existe un risque d'attaque par « interception (sniffing) » des mots de passe, il est préférable d'utiliser `md5`. L'utilisation de `password`, en clair, est toujours à éviter quand c'est possible. Néanmoins, `md5` ne peut pas être utilisé avec la fonctionnalité

db_user_namespace. Si la connexion est protégée par un chiffrement SSL, alors `password` peut être utilisé avec sûreté (bien que l'authentification par certificat SSL pourrait être un meilleur choix s'il y a dépendance au sujet de l'utilisation de SSL).

Les mots de passe PostgreSQL™ sont distincts des mots de passe du système d'exploitation. Le mot de passe de chaque utilisateur est enregistré dans le catalogue système `pg_authid`. Ils peuvent être gérés avec les commandes SQL CREATE USER(7) et ALTER USER(7). Ainsi, par exemple, **`CREATE USER foo WITH PASSWORD 'secret';`**. Si aucun mot de passe n'est enregistré pour un utilisateur, le mot de passe enregistré est nul et l'authentification par mot de passe échoue systématiquement pour cet utilisateur.

19.3.3. Authentification GSSAPI

GSSAPI™ est un protocole du standard de l'industrie pour l'authentification sécurisée définie dans RFC 2743. PostgreSQL™ supporte GSSAPI™ avec l'authentification Kerberos™ suivant la RFC 1964. GSSAPI™ fournit une authentification automatique (*single sign-on*) pour les systèmes qui le supportent. L'authentification elle-même est sécurisée mais les données envoyées sur la connexion seront en clair sauf si SSL est utilisé.

Quand GSSAPI™ passe par Kerberos™, il utilise un principal standard dans le format *`nomservice/nomhôte@domaine`*. Pour des informations sur les parties du principal et sur la façon de configurer les clés requises, voir Section 19.3.5, « Authentification Kerberos ».

Le support de GSSAPI doit être activé lors de la construction de PostgreSQL™ ; voir Chapitre 15, Procédure d'installation de PostgreSQL™ du code source pour plus d'informations.

Les options de configuration suivantes sont supportées pour GSSAPI™ :

`include_realm`
: Si configuré à `1`, le nom du royaume provenant du principal de l'utilisateur authentifié est inclus dans le nom de l'utilisateur système qui est passé au système de correspondance d'utilisateur (Section 19.2, « Correspondances d'utilisateurs »). Ceci est utile pour gérer des utilisateurs provenant de plusieurs royaumes.

`map`
: Permet la mise en correspondance entre les noms système et base de données. Voir Section 19.2, « Correspondances d'utilisateurs » pour plus de détails. Pour un principal Kerberos `username/hostbased@EXAMPLE.COM`, le nom d'utilisateur utilisé pour la correspondance est `nomutilisateur/hote` si `include_realm` est désactivé, et `username/hostbased@EXAMPLE.COM` si `include_realm` est activé.

`krb_realm`
: Configure le royaume pour la correspondance du principal de l'utilisateur. Si ce paramètre est configuré, seuls les utilisateurs de ce royaume seront acceptés. S'il n'est pas configuré, les utilisateurs de tout royaume peuvent se connecter, à condition que la correspondance du nom de l'utilisateur est faite.

19.3.4. Authentification SSPI

SSPI™ est une technologie Windows™ pour l'authentification sécurisée avec *single sign-on*. PostgreSQL™ utilise SSPI dans un mode de négociation (`negotiate`) qui utilise Kerberos™ si possible et NTLM™ sinon. L'authentification SSPI™ ne fonctionne que lorsque serveur et client utilisent Windows™.

Lorsque Kerberos™ est utilisé, SSPI™ fonctionne de la même façon que GSSAPI™. Voir Section 19.3.3, « Authentification GSSAPI » pour les détails.

Les options de configuration suivantes sont supportées pour SSPI™ :

`include_realm`
: Si configuré à `1`, le nom du royaume provenant du principal de l'utilisateur authentifié est inclus dans le nom de l'utilisateur système qui est passé au système de correspondance d'utilisateur (Section 19.2, « Correspondances d'utilisateurs »). Ceci est utile pour gérer des utilisateurs provenant de plusieurs royaumes.

`map`
: Permet la mise en correspondance entre les noms système et base de données. Voir Section 19.2, « Correspondances d'utilisateurs » pour plus de détails.

`krb_realm`
: Configure le royaume pour la correspondance du principal de l'utilisateur. Si ce paramètre est configuré, seuls les utilisateurs de ce royaume seront acceptés. S'il n'est pas configuré, les utilisateurs de tout royaume peuvent se connecter, à condition que la correspondance du nom de l'utilisateur est faite.

19.3.5. Authentification Kerberos

Note

L'authentification Kerberos native est obsolète et ne doit être utilisée que pour assurer la compatibilité ascendante. Les nouvelles installations et les mises à jour utiliseront préférentiellement le standard d'authentification de l'industrie, GSSAPI™ (voir Section 19.3.3, « Authentification GSSAPI »).

Kerberos™ est un système d'authentification sécurisée de standard industriel destiné à l'informatique distribuée sur un réseau public. La description de Kerberos™ dépasse les objectifs de ce document même dans les généralités, c'est assez complexe (bien que puissant). La *FAQ Kerberos* ou la *page Kerberos du MIT* sont un bon point de départ à l'exploration. Il existe plusieurs sources de distribution Kerberos™. Kerberos™ fournit une authentification sécurisée mais ne chiffre pas les requêtes ou les données passées sur le réseau ; pour cela, SSL doit être utilisé.

PostgreSQL™ supporte Kerberos version 5. Le support de Kerberos doit être activé lors de la construction de PostgreSQL™ ; voir le Chapitre 15, Procédure d'installation de PostgreSQL™ du code source pour plus d'informations.

PostgreSQL™ opère comme un service Kerberos normal. Le nom du service principal est `nomservice/nomhôte@domaine`.

`nomservice` peut être configuré du côté serveur en utilisant le paramètre de configuration krb_srvname (voir aussi Section 31.1, « Fonctions de contrôle de connexion à la base de données »). La valeur par défaut à l'installation, `postgres`, peut être modifiée lors de la construction avec `./configure --with-krb-srvnam=quelquechose`. Dans la plupart des environnements, il est inutile de modifier cette valeur. Néanmoins, cela devient nécessaire pour supporter plusieurs installations de PostgreSQL™ sur le même hôte. Quelques implantations de Kerberos peuvent imposer un nom de service différent, comme Microsoft Active Directory qui réclame un nom du service en majuscules (`POSTGRES`).

`nom_hote` est le nom de l'hôte pleinement qualifié (*fully qualified host name*) de la machine serveur. Le domaine du service principal (client) est le domaine préféré du serveur.

Les principaux (clients) doivent contenir le nom de leur utilisateur PostgreSQL™ comme premier composant, `nomutilisateurpg@domaine`, par exemple. Sinon, vous pouvez utiliser une correspondance du nom d'utilisateur qui établit la correspondance à partir du premier composant du nom du principal vers le nom d'utilisateur de la base de données Par défaut, le domaine du client n'est pas vérifié par PostgreSQL™. Si l'authentification inter-domaine (*cross-realm*) est activée, on utilise le paramètre `krb_realm` ou activer `include_realm` et utiliser la correspondance du nom d'utilisateur pour vérifier le royaume.

Le fichier de clés du serveur doit être lisible (et de préférence uniquement lisible) par le compte serveur PostgreSQL™ (voir aussi la Section 17.1, « Compte utilisateur PostgreSQL™ »). L'emplacement du fichier de clés est indiqué grâce au paramètre de configuration krb_server_keyfile fourni à l'exécution. La valeur par défaut est `/etc/srvtab`, si Kerberos 4 est utilisé, et `/usr/local/pgsql/etc/krb5.keytab` sinon (ou tout autre répertoire indiqué comme `sysconfdir` à la compilation).

Le fichier de clés est engendré par le logiciel Kerberos ; voir la documentation de Kerberos pour les détails. L'exemple suivant correspond à des implantations de Kerberos 5 compatibles avec MIT :

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

Lors de la connexion à la base de données, il faut s'assurer de posséder un ticket pour le principal correspondant au nom d'utilisateur de base de données souhaité. Par exemple, pour le nom d'utilisateur PostgreSQL `fred`, le principal `fred@EXAMPLE.COM` pourrait se connecter. Pour autoriser aussi le principal `fred/users.example.com@EXAMPLE.COM`, utiliser une correspondance de nom d'utilisateur, comme décrit dans Section 19.2, « Correspondances d'utilisateurs ».

Si *mod_auth_kerb* et mod_perl sont utilisés sur le serveur web Apache™, `AuthType KerberosV5SaveCredentials` peut être utilisé avec un script mod_perl. Cela fournit un accès sûr aux bases de données, sans demander de mot de passe supplémentaire.

Les options de configuration suivantes sont supportées pour Kerberos™ :

`map`
: Permet la mise en correspondance entre les noms système et base de données. Voir Section 19.2, « Correspondances d'utilisateurs » pour plus de détails.

`include_realm`
: Si configuré à `1`, le nom du royaume provenant du principal de l'utilisateur authentifié est inclus dans le nom de l'utilisateur système qui est passé au système de correspondance d'utilisateur (Section 19.2, « Correspondances d'utilisateurs »). Ceci est utile pour gérer des utilisateurs provenant de plusieurs royaumes.

`krb_realm`
: Configure le royaume pour la correspondance du principal de l'utilisateur. Si ce paramètre est configuré, seuls les utilisateurs

de ce royaume seront acceptés. S'il n'est pas configuré, les utilisateurs de tout royaume peuvent se connecter, à condition que la correspondance du nom de l'utilisateur est faite.

`krb_server_hostname`
: Précise le nom d'hôte du service principal. Cela, combiné avec `krb_srvname`, est utilisé pour généré le nom complet du service principal, qui est `krb_srvname/krb_server_hostname@`REALM. S'il n'est pas renseigné, la valeur par défaut est le nom d'hôte du serveur.

19.3.6. Authentification fondée sur ident

La méthode d'authentification par ident fonctionne en obtenant les noms des utilisateurs du système d'exploitation, et en les utilisant comme le nom de l'utilisateur de bases de données autorisé (avec une correspondance de nom d'utilisateur optionnel). La résolution du nom d'utilisateur du client est le point de sécurité critique. Elle fonctionne différemment selon le type de connexion, comme décrit ci-dessous.

Les options de configuration suivantes sont supportées pour ident™ :

`map`
: Permet la mise en correspondance entre les noms système et base de données. Voir Section 19.2, « Correspondances d'utilisateurs » pour plus de détails.

19.3.6.1. Authentification par ident en TCP/IP

Le « protocole d'identification » est décrit dans la RFC 1413. Théoriquement, chaque système d'exploitation de type Unix contient un serveur ident qui écoute par défaut sur le port TCP 113. La fonctionnalité basique d'un serveur ident est de répondre aux questions telles que : « Quel utilisateur a initié la connexion qui sort du port *X* et se connecte à mon port *Y*? ». Puisque PostgreSQL™ connaît *X* et *Y* dès lors qu'une connexion physique est établie, il peut interroger le serveur ident de l'hôte du client qui se connecte et peut ainsi théoriquement déterminer l'utilisateur du système d'exploitation pour n'importe quelle connexion.

Le revers de cette procédure est qu'elle dépend de l'intégrité du client : si la machine cliente est douteuse ou compromise, un attaquant peut lancer n'importe quel programme sur le port 113 et retourner un nom d'utilisateur de son choix. Cette méthode d'authentification n'est, par conséquent, appropriée que dans le cas de réseaux fermés dans lesquels chaque machine cliente est soumise à un contrôle strict et dans lesquels les administrateurs système et de bases de données opèrent en étroite collaboration. En d'autres mots, il faut pouvoir faire confiance à la machine hébergeant le serveur d'identification. Cet avertissement doit être gardé à l'esprit :

> Le protocole d'identification n'a pas vocation à être un protocole d'autorisation ou de contrôle d'accès.
>
> —RFC 1413

Certains serveurs ident ont une option non standard qui chiffre le nom de l'utilisateur retourné à l'aide d'une clé connue du seul administrateur de la machine dont émane la connexion. Cette option *ne doit pas* être employée lorsque le serveur ident est utilisé avec PostgreSQL™ car PostgreSQL™ n'a aucun moyen de déchiffré la chaîne renvoyée pour déterminer le nom réel de l'utilisateur.

19.3.6.2. Authentification par ident sur sockets locaux

Sur les systèmes qui supportent les requêtes SO_PEERCRED pour les sockets de domaine Unix (actuellement `Linux`, `FreeBSD`, `NetBSD`, `OpenBSD`, `BSD/OS` et `Solaris`), l'authentification par ident peut aussi être appliquée aux connexions locales. PostgreSQL™ utilise SO_PEERCRED pour trouver le nom du système d'exploiration du processus client connecté. Dans ce cas, l'utilisation de l'authentification par ident n'ajoute aucun risque de sécurité en fait, c'est même un choix préférable sur ce genre de système.

Sur les systèmes sans requête SO_PEERCRED, l'authentification par ident n'est disponible que pour les connexions TCP/IP. Pour pallier ceci, il est possible de préciser l'adresse `localhost` `127.0.0.1` et d'établir une connexion à cette adresse. Si le serveur ident local est digne de confiance, alors cette méthode l'est aussi.

19.3.7. Authentification LDAP

Ce mécanisme d'authentification opère de façon similaire à `password` à ceci près qu'il utilise LDAP comme méthode de vérification des mots de passe. LDAP n'est utilisé que pour valider les paires nom d'utilisateur/mot de passe. De ce fait, pour pouvoir utiliser LDAP comme méthode d'authentification, l'utilisateur doit préalablement exister dans la base.

L'authentification LDAP peut opérer en deux modes. Dans le premier mode, le serveur fera un « bind » sur le nom distingué comme *`préfixe nom_utilisateur suffixe`*. Typiquement, le paramètre *`prefix`* est utilisé pour spécifier `cn=` ou *`DOMAIN`*\ dans un environnement Active Directory. *`suffix`* est utilisé pour spécifier le reste du DN dans un environnement autre

qu'Active Directory.

Dans le second mode, le serveur commence un « bind » sur le répertoire LDAP avec un nom d'utilisateur et un mot de passe fixés, qu'il indique à *`ldapbinddn`* et *`ldapbindpasswd`*. Il réalise une recherche de l'utilisateur en essayant de se connecter à la base de données. Si aucun utilisateur et aucun mot de passe n'est configuré, un « bind » anonyme sera tenté sur le répertoire. La recherche sera réalisée sur le sous-arbre sur *`ldapbasedn`*, et essaiera une correspondance exacte de l'attribut indiqué par *`ldapsearchattribute`*. Si aucun attribut n'est indiqué, l'attribut `uid` sera utilisé. Une fois que l'utilisateur a été trouvé lors de cette recherche, le serveur se déconnecte et effectue un nouveau « bind » au répertoire en tant que cet utilisateur, en utilisant le mot de passe indiqué par le client pour vérifier que la chaîne de connexion est correcte. Cette méthode permet une plus grande flexibilité sur l'emplacement des objets utilisateurs dans le répertoire mais demandera deux connexions au serveur LDAP.

Les options de configuration suivantes sont supportées pour LDAP :

`ldapserver`
: Noms ou adresses IP des serveurs LDAP auxquels se connecter. Plusieurs serveurs peuvent être indiqués, en les séparant par des espaces.

`ldapport`
: Numéro de port du serveur LDAP auquel se connecter. Si aucun port n'est spécifié, le port par défaut de la bibliothèque LDAP sera utilisé.

`ldaptls`
: Positionnez à `1` pour que la connexion entre PostgreSQL et le serveur LDAP utilise du chiffrage TLS. Notez que ceci ne chiffre que le trafic jusqu'au serveur LDAP -- la connexion vers le client peut toujours ne pas être chiffrée sauf si SSL est utilisé.

`ldapprefix`
: Chaîne à préfixer au nom de l'utilisateur pour former le DN utilisé comme lien lors d'une simple authentification *bind*.

`ldapsuffix`
: Chaîne à suffixer au nom de l'utilisateur pour former le DN utilisé comme lien lors d'une simple authentification *bind*.

`ldapbasedn`
: Racine DN pour commencer la recherche de l'utilisateur lors d'une authentification *search+bind*.

`ldapbinddn`
: DN de l'utilisateur pour se lier au répertoire avec lequel effectuer la recherche lors d'une authentification *search+bind*.

`ldapbindpasswd`
: Mot de passe de l'utilisateur pour se lier au répertoire avec lequel effectuer la recherche lors d'une authentification *search+bind*.

`ldapsearchattribute`
: Attribut à faire correspondre au nom d'utilisateur dans la recherche lors d'une authentification *search+bind*.

Note

Comme LDAP utilise souvent des virgules et des espaces pour séparer les différentes parties d'un DN, il est souvent nécessaire d'utiliser des paramètres entourés de guillements durant le paramétrage des options LDAP, comme par exemple :

```
ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example,
dc=net"
```

19.3.8. Authentification RADIUS

Cette méthode d'authentification opère de façon similaire à `password` sauf qu'il existe la méthode RADIUS pour la vérification du mot de passe. RADIUS est seulement utilisé pour valider des pairs nom utilisateur / mot de passe. Du coup, l'utilisateur doit déjà exister dans la base de données avant que RADIUS puisse être utilisé pour l'authentification.

Lors de l'utilisation de l'authentification RADIUS, un message de demande d'accès (*Access Request*) sera envoyé au serveur RADIUS configuré. Cette demande sera du type « authentification seule » (`Authenticate Only`) et incluera les paramètres pour le nom de l'utilisateur, son mot de passe (chiffré) et un identifiant NAS (`NAS Identifier`). La demande sera chiffrée en utilisant un secret partagé avec le serveur. Le serveur RADIUS répondre au serveur soit la réussite (`Access Accept`) soit l'échec (`Access Reject`) de l'accès. Il n'y a pas de support des comptes RADIUS.

Les options de configuration suivantes sont supportées par RADIUS :

`radiusserver`
Le nom ou l'adresse IP sur serveur RADIUS pour l'authentification. Ce paramètre est requis.

`radiussecret`
Le secret partagé utilisé lors de discussions sécurisées avec le serveur RADIUS. Il doit y avoir exactement la même valeur sur le serveur PostgreSQL et sur le serveur RADIUS. Il est recommandé d'utiliser une chaîne d'au moins 16 caractères. Ce paramètre est requis.

> **Note**
>
> Le vecteur de chiffrement utilisé sera un chiffrement fort seulement si PostgreSQL™ a été compilé avec le support d'OpenSSL™. Dans les autres cas, la transmission au serveur RADIUS peut seulement être considérée comme caché, et non pas sécurisé, et des mesures de sécurité externes doivent être appliquées si nécessaire.

`radiusport`
Le numéro de port sur le serveur RADIUS pour la connexion. Si aucun port n'est indiqué, le port par défaut, `1812`, sera utilisé.

`radiusidentifier`
La chaîne utilisée comme identifiant NAS (`NAS Identifier`) dans les demandes RADIUS. Ce paramètre peut être utilisé comme second paramètre identifiant par exemple l'utilisateur de bases de données pour la connexion. C'est utilisable pour des vérifications sur le serveur RADIUS. Si aucune identifiant n'est spécifié, la valeur par défaut, `postgresql`, sera utilisée.

19.3.9. Authentification de certificat

Cette méthode d'authentification utilise des clients SSL pour procéder à l'authentification. Elle n'est par conséquent disponible que pour les connexions SSL. Quand cette méthode est utilisée, le serveur exigera que le client fournisse un certificat valide. Aucune invite de saisie de mot de passe ne sera envoyée au client. L'attribut `cn` (Common Name) du certificat sera comparé au nom d'utilisateur de base de données demandé. S'ils correspondent, la connexion sera autorisée. La correspondance des noms d'utilisateurs peut être utilisé pour permettre au `cn` d'être différent du nom d'utilisateur de la base de données.

Les options de configuration suivantes sont supportées pour l'authentification par certificat SSL :

`map`
Permet la correspondance entre les noms d'utilisateur système et les noms d'utilisateurs de bases de données. Voir Section 19.2, « Correspondances d'utilisateurs » pour les détails.

19.3.10. Authentification PAM

Ce mécanisme d'authentification fonctionne de façon similaire à `password` à ceci près qu'il utilise PAM (Pluggable Authentication Modules) comme méthode d'authentification. Le nom du service PAM par défaut est `postgresql`. PAM n'est utilisé que pour valider des paires nom utilisateur/mot de passe. De ce fait, avant de pouvoir utiliser PAM pour l'authentification, l'utilisateur doit préalablement exister dans la base de données. Pour plus d'informations sur PAM, merci de lire la *page Linux-PAM*™ et la *page PAM* `Solaris`.

Les options suivantes sont supportées pour PAM :

`pamservice`
Nom de service PAM.

> **Note**
>
> Si PAM est configuré pour lire `/etc/shadow`, l'authentification échoue car le serveur PostgreSQL est exécuté en tant qu'utilisateur standard. Ce n'est toutefois pas un problème quand PAM est configuré pour utiliser LDAP ou les autres méthodes d'authentification.

19.4. Problèmes d'authentification

Les erreurs et problèmes d'authentification se manifestent généralement par des messages d'erreurs tels que ceux qui suivent.

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database
"testdb"
```

ou, en français,

```
FATAL:  pas d'entrée pg_hba.conf pour l'hôte "123.123.123.123", utilisateur "andym",
base "testdb"
```

C'est le message le plus probable lorsque le contact peut être établi avec le serveur mais qu'il refuse de communiquer. Comme le suggère le message, le serveur a refusé la demande de connexion parce qu'il n'a trouvé aucune entrée correspondante dans son fichier de configuration `pg_hba.conf`.

```
FATAL:  password authentication failed for user "andym"
```

ou, en français,

```
FATAL:  l'authentification par mot de passe a échoué pour l'utilisateur "andym"
```

Les messages de ce type indiquent que le serveur a été contacté et qu'il accepte la communication, mais pas avant que la méthode d'authentification indiquée dans le fichier `pg_hba.conf` n'ait été franchie avec succès. Le mot de passe fourni, le logiciel d'identification ou le logiciel Kerberos doivent être vérifiés en fonction du type d'authentification mentionné dans le message d'erreur.

```
FATAL:  user "andym" does not exist
```

ou, en français,

```
FATAL:  l'utilisateur "andym" n'existe pas
```

Le nom d'utilisateur indiqué n'a pas été trouvé.

```
FATAL:  database "testdb" does not exist
```

ou, en français,

```
FATAL:  la base "testdb" n'existe pas
```

La base de données utilisée pour la tentative de connexion n'existe pas. Si aucune base n'est précisée, le nom de la base par défaut est le nom de l'utilisateur, ce qui peut être approprié ou non.

Astuce

Les traces du serveur contiennent plus d'informations sur une erreur d'authentification que ce qui est rapporté au client. En cas de doute sur les raisons d'un échec, il peut s'avérer utile de les consulter.

Chapitre 20. Rôles et droits de la base de données

PostgreSQL™ gère les droits d'accès aux bases de données en utilisant le concept de *rôles*. Un rôle peut être vu soit comme un utilisateur de la base de données, soit comme un groupe d'utilisateurs de la base de données, suivant la façon dont le rôle est configuré. Les rôles peuvent posséder des objets de la base de données (par exemple des tables) et peuvent affecter des droits sur ces objets à d'autres rôles pour contrôler qui a accès à ces objets. De plus, il est possible de donner l'*appartenance* d'un rôle à un autre rôle, l'autorisant du coup à utiliser les droits affectés à un autre rôle.

Le concept des rôles comprends les concepts des « utilisateurs » et des « groupes ». Dans les versions de PostgreSQL™ antérieures à la 8.1, les utilisateurs et les groupes étaient des types d'entité distincts mais, maintenant, ce ne sont que des rôles. Tout rôle peut agir comme un utilisateur, un groupe ou les deux.

Ce chapitre décrit comment créer et gérer des rôles, et introduit le système des droits. Plus d'informations sur les différents types d'objets de la base de données et les effets des droits sont disponibles dans le Chapitre 5, Définition des données.

20.1. Rôles de la base de données

Conceptuellement, les rôles de la base sont totalement séparés des utilisateurs du système d'exploitation. En pratique, il peut être commode de maintenir une correspondance mais cela n'est pas requis. Les rôles sont globaux à toute une installation de groupe de bases de données (et non individuelle pour chaque base). Pour créer un rôle, utilisez la commande SQL CREATE ROLE(7) :

```
CREATE ROLE nom_utilisateur;
```

nom_utilisateur suit les règles des identifiants SQL : soit sans guillemets et sans caractères spéciaux, soit entre double-guillemets (en pratique, vous voudrez surtout ajouter des options supplémentaires, comme `LOGIN`, à cette commande. Vous trouverez plus de détails ci-dessous). Pour supprimer un rôle existant, utilisez la commande analogue DROP ROLE(7) :

```
DROP ROLE nom_utilisateur;
```

Pour une certaine facilité d'utilisation, les programmes createuser(1) et dropuser(1) sont fournis comme emballage de ces commandes SQL et peuvent être appelés depuis la ligne de commande du shell :

```
createuser nom_utilisateur
dropuser nom_utilisateur
```

Pour déterminer l'ensemble des rôles existants, examinez le catalogue système pg_roles existant, par exemple

```
SELECT rolname FROM pg_roles;
```

La méta-commande `\du` du programme psql(1) est aussi utile pour lister les rôles existants.

Afin d'amorcer le système de base de données, un système récemment installé contient toujours un rôle prédéfini. Ce rôle est un superutilisateur et aura par défaut le même nom que l'utilisateur du système d'exploitation qui a initialisé le groupe de bases de données (à moins que cela ne soit modifié en lançant la commande **initdb**). Par habitude, ce rôle sera nommé `postgres`. Pour créer plus de rôles, vous devez d'abord vous connecter en tant que ce rôle initial.

Chaque connexion au serveur de la base de données est fait au nom d'un certain rôle et ce rôle détermine les droits d'accès initiaux pour les commandes lancées sur cette connexion. Le nom du rôle à employer pour une connexion à une base particulière est indiqué par le client initialisant la demande de connexion et ce, de la manière qui lui est propre. Par exemple, le programme **psql** utilise l'option de ligne de commandes `-U` pour préciser sous quel rôle il se connecte. Beaucoup d'applications (incluant **createuser** et **psql**) utilisent par défaut le nom courant de l'utilisateur du système d'exploitation. Par conséquence, il peut souvent être pratique de maintenir une correspondance de nommage entre les rôles et les utilisateurs du système d'exploitation.

La configuration de l'authentification du client détermine avec quel rôle de la base, la connexion cliente donnée se connectera, comme cela est expliqué dans le Chapitre 19, Authentification du client (donc, un client n'est pas obligé de se connecter avec le rôle du même nom que son nom d'utilisateur dans le système d'exploitation ; de la même façon que le nom de connexion d'un utilisateur peut ne pas correspondre à son vrai nom). Comme le rôle détermine l'ensemble des droits disponibles pour le client connecté, il est important de configurer soigneusement les droits quand un environnement multi-utilisateurs est mis en place.

20.2. Attributs des rôles

Un rôle de bases de données peut avoir un certain nombre d'attributs qui définissent ses droits et interagissent avec le système d'authentification du client.

droit de connexion
: Seuls les rôles disposant de l'attribut `LOGIN` peuvent être utilisés comme nom de rôle initial pour une connexion à une base

de données. Un rôle avec l'attribut `LOGIN` peut être considéré de la même façon qu'un « utilisateur de la base de données ». Pour créer un rôle disposant du droit de connexion, utilisez :

```
CREATE ROLE nom LOGIN;
CREATE USER nom;
```

(**CREATE USER** est équivalent à **CREATE ROLE** sauf que **CREATE USER** utilise `LOGIN` par défaut alors que **CREATE ROLE** ne le fait pas)

statut de superutilisateur
: Les superutilisateurs ne sont pas pris en compte dans les vérifications des droits. Ceci est un droit dangereux et ne devrait pas être utilisé sans faire particulièrement attention ; il est préférable de faire la grande majorité de votre travail avec un rôle qui n'est pas superutilisateur. Pour créer un nouveau superutilisateur, utilisez `CREATE ROLE nom SUPERUSER`. Vous devez le faire en tant que superutilisateur.

création de bases de données
: Les droits de création de bases doivent être explicitement données à un rôle (à l'exception des super-utilisateurs qui passent au travers de toute vérification de droits). Pour créer un tel rôle, utilisez `CREATE ROLE nom_utilisateur CREATEDB`.

création de rôle
: Un rôle doit se voir explicitement donné le droit de créer plus de rôles (sauf pour les superutilisateurs vu qu'ils ne sont pas pris en compte lors des vérifications de droits). Pour créer un tel rôle, utilisez `CREATE ROLE nom CREATEROLE`. Un rôle disposant du droit `CREATEROLE` peut aussi modifier et supprimer d'autres rôles, ainsi que donner ou supprimer l'appartenance à ces rôles. Néanmoins, pour créer, modifier, supprimer ou changer l'appartenance à un rôle superutilisateur, le statut de superutilisateur est requis ; `CREATEROLE` n'est pas suffisant pour cela.

mot de passe
: Un mot de passe est seulement significatif si la méthode d'authentification du client exige que le client fournisse un mot de passe quand il se connecte à la base. Les méthodes d'authentification par `mot de passe` et `md5` utilisent des mots de passe. Les mots de passe de la base de données ne sont pas les mêmes que ceux du système d'exploitation. Indiquez un mots de passe lors de la création d'un rôle avec `CREATE ROLE nom_utilisateur PASSWORD 'le_mot_de_passe'`.

Les attributs d'un rôle peuvent être modifiés après sa création avec **ALTER ROLE**. Regardez les pages de références de CREATE ROLE(7) et de ALTER ROLE(7) pour plus de détails.

Astuce

Une bonne pratique est de créer un rôle qui dispose des droits `CREATEDB` et `CREATEROLE` mais qui n'est pas un superutilisateur, et d'utiliser ce rôle pour toute la gestion des bases de données et des rôles. Cette approche évite les dangers encourus en travaillant en tant que superutilisateur pour des tâches qui n'ont pas besoin de cet état.

Un rôle peut aussi configurer ses options par défaut pour de nombreux paramètres de configuration décris dans le Chapitre 18, Configuration du serveur. Par exemple, si, pour une raison ou une autre, vous voulez désactiver les parcours d'index (conseil : ce n'est pas une bonne idée) à chaque fois que vous vous connectez, vous pouvez utiliser :

```
ALTER ROLE myname SET enable_indexscan TO off;
```

Cela sauve les paramètres (mais ne les applique pas immédiatement). Dans les connexions ultérieures de ce rôle, c'est comme si `SET enable_indexscan TO off` avait été appelé juste avant le démarrage de la session. Vous pouvez toujours modifier les paramètres durant la session. Pour supprimer une configuration par défaut spécifique à un rôle, utilisez `ALTER ROLE nom_utilisateur RESET nom_variable`. Notez que les valeurs par défaut spécifiques aux rôles sans droit de connexion (`LOGIN`) sont vraiment inutiles car ils ne seront jamais appelés.

20.3. Droits

Quand un objet est créé, il est affecté à un propriétaire. Ce dernier est habituellement le rôle qui a exécuté l'instruction de création. Pour la plupart des objets, l'état initial est tel que seul le propriétaire (ou un superutilisateur) peut faire quelque chose avec cet objet. Afin de laisser les autres rôles utiliser l'objet, des *droits* doivent être accordés. Il existe différents droits : `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, and `USAGE`. Pour plus d'informations sur le support des différents types de droits par PostgreSQL™, regardez la page de référence GRANT(7).

Pour affecter des droits, la commande **GRANT** est utilisée. Ainsi, si `joe` est un rôle existant et `comptes` est une table existante, le droit pour mettre à jour la table peut être accordé avec :

```
GRANT UPDATE ON comptes TO joe;
```

Le nom spécial `PUBLIC` peut être utilisé pour accorder un droit à chaque rôle du système. Écrire `ALL` à la place d'un droit spécifique signifie que tous les droits s'appliquant à l'objet seront accordés.

Pour révoquer un droit, utilisez la commande nommée REVOKE(7) :

```
REVOKE ALL ON comptes FROM PUBLIC;
```

Les droits spéciaux d'un propriétaire d'un objet (par exemple, le droit de modifier ou détruire un objet) sont toujours implicites et ne peuvent pas être accordés ou révoqués. Mais le propriétaire peut choisir de révoquer ses propres droits ordinaires, par exemple mettre une table en lecture seule pour lui-même aussi bien que pour les autres.

Un objet peut être affecté à un nouveau propriétaire avec une commande **ALTER** du genre approprié pour l'objet. Les superutilisateurs peuvent toujours le faire ; les rôles ordinaires peuvent seulement le faire s'ils sont le propriétaire actuel de l'objet (ou un membre du rôle propriétaire) et un membre du nouveau rôle propriétaire.

20.4. Appartenance d'un rôle

Il est souvent intéressant de grouper les utilisateurs pour faciliter la gestion des droits : de cette façon, les droits peuvent être donnés ou supprimés pour tout un groupe. Dans PostgreSQL™, ceci se fait en créant un rôle représentant le groupe, puis en ajoutant les rôles utilisateurs individuels *membres* de ce groupe.

Pour configurer un rôle en tant que groupe, créez tout d'abord le rôle :

```
CREATE ROLE nom;
```

Typiquement, un rôle utilisé en tant que groupe n'aura pas l'attribut `LOGIN` bien que vous puissiez le faire si vous le souhaitez.

Une fois que ce rôle existe, vous pouvez lui ajouter et lui supprimer des membres en utilisant les commandes GRANT(7) et REVOKE(7) :

```
GRANT role_groupe TO role1, ... ;
REVOKE role_groupe FROM role1, ... ;
```

Vous pouvez aussi faire en sorte que d'autres rôles groupes appartiennent à ce groupe (car il n'y a pas réellement de distinction entre les rôles groupe et les rôles non groupe). La base de données ne vous laissera pas configurée des boucles circulaires d'appartenance. De plus, il est interdit de faire en sorte qu'un membre appartienne à `PUBLIC`.

Les membres d'un rôle groupe peuvent utiliser les droits du rôle de deux façons. Tout d'abord, chaque membre d'un groupe peut exécuter explicitement SET ROLE(7) pour « devenir » temporairement le rôle groupe. Dans cet état, la session de la base de données a accès aux droits du rôle groupe plutôt qu'à ceux du rôle de connexion original et tous les objets créés sont considérés comme appartenant au rôle groupe, et non pas au rôle utilisé lors de la connexion. Deuxièmement, les rôles membres qui ont l'attribut `INHERIT` peuvent utiliser automatiquement les droits des rôles dont ils sont membres, ceci incluant les droits hérités par ces rôles. Comme exemple, supposons que nous avons lancé les commandes suivantes :

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Immédiatement après connexion en tant que `joe`, la session de la base de données peut utiliser les droits donnés directement à `joe` ainsi que ceux donnés à `admin` parce que `joe` « hérite » des droits de `admin`. Néanmoins, les droits donnés à `wheel` ne sont pas disponibles parce que, même si `joe` est un membre indirect de `wheel`, l'appartenance se fait via `admin` qui dispose de l'attribut `NOINHERIT`. Après :

```
SET ROLE admin;
```

la session aura la possibilité d'utiliser les droits donnés à `admin` mais n'aura plus accès à ceux de `joe`. Après :

```
SET ROLE wheel;
```

la session pourra utiliser uniquement ceux de `wheel`, mais ni ceux de `joe` ni ceux de `admin`. L'état du droit initial peut être restauré avec une des instructions suivantes :

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

Note

La commande **SET ROLE** autorisera toujours la sélection de tout rôle dont le rôle de connexion est membre directement ou indirectement. Du coup, dans l'exemple précédent, il n'est pas nécessaire de devenir `admin` pour devenir `wheel`.

> **Note**
>
> Dans le standard SQL, il existe une distinction claire entre les utilisateurs et les rôles. Les utilisateurs ne peuvent pas hériter automatiquement alors que les rôles le peuvent. Ce comportement est obtenu dans PostgreSQL™ en donnant aux rôles utilisés comme des rôles SQL l'attribut `INHERIT`, mais en donnant aux rôles utilisés en tant qu'utilisateurs SQL l'attribut `NOINHERIT`. Néanmoins, par défaut, PostgreSQL™ donne à tous les rôles l'attribut `INHERIT` pour des raisons de compatibilité avec les versions précédant la 8.1 dans lesquelles les utilisateurs avaient toujours les droits des groupes dont ils étaient membres.

Les attributs `LOGIN`, `SUPERUSER`, `CREATEDB` et `CREATEROLE` peuvent être vus comme des droits spéciaux qui ne sont jamais hérités contrairement aux droits ordinaires sur les objets de la base. Vous devez réellement utiliser **SET ROLE** vers un rôle spécifique pour avoir un de ces attributs et l'utiliser. Pour continuer avec l'exemple précédent, nous pourrions très bien choisir de donner les droits `CREATEDB` et `CREATEROLE` au rôle `admin`. Puis, une session connectée en tant que le rôle `joe` n'aurait pas ces droits immédiatement, seulement après avoir exécuté **SET ROLE admin**.

Pour détruire un rôle groupe, utilisez DROP ROLE(7):

```
DROP ROLE nom;
```

Toute appartenance à ce rôle est automatiquement supprimée (mais les rôles membres ne sont pas autrement affectés). Notez néanmoins que tous les objets dont le groupe était propriétaire doivent d'abord être supprimés ou réaffectés ; et tous les droits accordés au rôle groupe doivent être supprimés.

20.5. Sécurité des fonctions et déclencheurs (triggers)

Les fonctions et les déclencheurs autorisent à l'intérieur du serveur les utilisateurs à insérer du code que d'autres utilisateurs peuvent exécuter sans en avoir l'intention. Par conséquent, les deux mécanismes permettent aux utilisateurs d'utiliser un « cheval de Troie » contre d'autres avec une relative facilité. La seule protection réelle est d'effectuer un fort contrôle sur ceux qui peuvent définir des fonctions.

Les fonctions sont exécutées à l'intérieur du processus serveur avec les droits au niveau système d'exploitation du démon serveur de la base de données. Si le langage de programmation utilisé par la fonction autorise les accès mémoire non contrôlés, il est possible de modifier les structures de données internes du serveur. Du coup, parmi d'autres choses, de telles fonctions peuvent dépasser les contrôles d'accès au système. Les langages de fonctions qui permettent un tel accès sont considérées « sans confiance » et PostgreSQL™ autorise uniquement les superutilisateurs à écrire des fonctions dans ces langages.

Chapitre 21. Administration des bases de données

Chaque instance d'un serveur PostgreSQL™ gère une ou plusieurs bases de données. Les bases de données sont donc le niveau hiérarchique le plus élevé pour organiser des objets SQL (« objets de base de données »). Ce chapitre décrit les propriétés des bases de données et comment les créer, les administrer et les détruire.

21.1. Aperçu

Une base de données est un ensemble nommé d'objets SQL (« objets de base de données »). En général, chaque objet de base de données (table, fonction etc.) appartient à une et une seule base de données (néanmoins certains catalogues système, par exemple `pg_database`, appartiennent à tout le groupe et sont accessibles depuis toutes les bases de données du groupe). Plus précisément, une base de données est une collection de schémas et les schémas contiennent les tables, fonctions, etc. Ainsi, la hiérarchie complète est : serveur, base de données, schéma, table (ou un autre type d'objet, comme une fonction).

Lors de la connexion au serveur de bases de données, une application cliente doit spécifier dans sa requête de connexion la base de données à laquelle elle veut se connecter. Il n'est pas possible d'accéder à plus d'une base de données via la même connexion. Néanmoins une application n'est pas limitée dans le nombre de connexions qu'elle établit avec une ou plusieurs bases de données. Les bases de données sont séparées physiquement et le contrôle d'accès est géré au niveau de la connexion. Si une instance de serveur PostgreSQL™ doit héberger des projets ou des utilisateurs censés rester séparés et sans interaction, il est recommandé de les répartir sur plusieurs bases de données. Si les projets ou les utilisateurs sont reliés et doivent pouvoir partager leurs ressources, alors ils devraient être placés dans la même base de données mais éventuellement dans des schémas différents. Les schémas sont une structure purement logique et qui peut accéder à ce qui est géré par le système des droits. Pour plus d'informations sur la manipulation des schémas, voir la Section 5.7, « Schémas ».

Les bases de données sont créées avec la commande **CREATE DATABASE** (voir la Section 21.2, « Création d'une base de données ») et détruites avec la commande **DROP DATABASE** (voir la Section 21.5, « Détruire une base de données »). Pour déterminer l'ensemble des bases de données existantes, examinez le catalogue système pg_database, par exemple

```
SELECT datname FROM pg_database;
```

La méta-commande `\l` du programme psql(1) et l'option en ligne de commande `-l` sont aussi utiles pour afficher les bases de données existantes.

> **Note**
>
> Le standard SQL appelle les bases de données des « catalogues » mais il n'y a aucune différence en pratique.

21.2. Création d'une base de données

Pour pouvoir créer une base de données, il faut que le serveur PostgreSQL™ soit lancé (voir la Section 17.3, « Lancer le serveur de bases de données »).

Les bases de données sont créées à l'aide de la commande SQL CREATE DATABASE(7):

```
CREATE DATABASE nom;
```

ou *nom* suit les règles habituelles pour les identifiants SQL. Le rôle actuel devient automatiquement le propriétaire de la nouvelle base de données. C'est au propriétaire de la base de données qu'il revient de la supprimer par la suite (ce qui supprime aussi tous les objets qu'elle contient, même s'ils ont un propriétaire différent).

La création de bases de données est une opération protégée. Voir la Section 20.2, « Attributs des rôles » sur la manière d'attribuer des droits.

Comme vous devez être connecté au serveur de base de données pour exécuter la commande **CREATE DATABASE**, reste à savoir comment créer la première base de données d'un site. La première base de données est toujours créée par la commande **initdb** quand l'aire de stockage des données est initialisée (voir la Section 17.2, « Créer un groupe de base de données »). Cette base de données est appelée `postgres`. Donc, pour créer la première base de données « ordinaire », vous pouvez vous connecter à `postgres`.

Une deuxième base de données, `template1`, est aussi créée durant l'initialisation du cluster de bases de données. Quand une nouvelle base de données est créée à l'intérieur du groupe, `template1` est généralement cloné. Cela signifie que tous les changements effectués sur `template1` sont propagés à toutes les bases de données créées ultérieurement. À cause de cela, évitez de créer des objets dans `template1` sauf si vous voulez les propager à chaque nouvelle base de données créée. Pour plus de détails, voir la Section 21.3, « Bases de données modèles ».

Pour plus de confort, il existe aussi un programme que vous pouvez exécuter à partir du shell pour créer de nouvelles bases de

données, **createdb**.

```
createdb nom_base
```

createdb ne fait rien de magique. Il se connecte à la base de données `postgres` et exécute la commande **CREATE DATABASE**, exactement comme ci-dessus. La page de référence sur createdb(1) contient les détails de son invocation. Notez que **createdb** sans aucun argument crée une base de donnée portant le nom de l'utilisateur courant.

> **Note**
>
> Le Chapitre 19, Authentification du client contient des informations sur la manière de restreindre l'accès à une base de données.

Parfois, vous voulez créer une base de données pour quelqu'un d'autre. Ce rôle doit devenir le propriétaire de la nouvelle base de données afin de pouvoir la configurer et l'administrer lui-même. Pour faire ceci, utilisez l'une des commandes suivantes :

```
CREATE DATABASE nom_base OWNER nom_role;
```

dans l'environment SQL ou

```
createdb -O nom_role nom_base
```

dans le shell. Seul le super-utilisateur est autorisé à créer une base de données pour quelqu'un d'autre c'est-à-dire pour un rôle dont vous n'êtes pas membre.

21.3. Bases de données modèles

En fait, **CREATE DATABASE** fonctionne en copiant une base de données préexistante. Par défaut, cette commande copie la base de données système standard `template1`. Ainsi, cette base de données est le « modèle » à partir duquel de nouvelles bases de données sont créées. Si vous ajoutez des objets à `template1`, ces objets seront copiés dans les bases de données utilisateur créées ultérieurement. Ce comportement permet d'apporter des modifications locales au jeu standard d'objets des bases de données. Par exemple, si vous installez le langage de procédures PL/Perl dans `template1`, celui-ci sera automatiquement disponible dans les bases de données utilisateur sans qu'il soit nécessaire de faire quelque chose de spécial au moment où ces bases de données sont créées.

Il y a une seconde base de données système standard appelée `template0`. Cette base de données contient les mêmes données que le contenu initial de `template1`, c'est-à-dire seulement les objets standards prédéfinis dans votre version de PostgreSQL™. `template0` ne devrait jamais être modifiée après que le cluster des bases de données ait été créé. En indiquant à **CREATE DATABASE** de copier `template0` au lieu de `template1`, vous pouvez créer une base de données utilisateur « vierge » qui ne contient aucun des ajouts locaux à `template1`. Ceci est particulièrement pratique quand on restaure une sauvegarde réalisé avec `pg_dump` : le script de dump devrait être restauré dans une base de données vierge pour être sûr de recréer le contenu correct de la base de données sauvegardée, sans survenue de conflits avec des objets qui auraient été ajoutés à `template1`.

Une autre raison habituelle de copier `template0` au lieu de `template1` est que les nouvelles options d'encodage et de locale peuvent être indiquées lors de la copie de `template0`, alors qu'une copie de `template1` doit utiliser les même options. Ceci est dû au fait que `template1` pourrait conteinur des données spécifiques à l'encodage ou à la locale alors que `template0` n'est pas modifiable.

Pour créer une base de données à partir de `template0`, on écrit :

```
CREATE DATABASE nom_base TEMPLATE template0;
```

dans l'environnement SQL ou

```
createdb -T template0 nom_base
```

dans le shell.

Il est possible de créer des bases de données modèles supplémentaires et, à vrai dire, on peut copier n'importe quelle base de données d'un cluster en la désignant comme modèle pour la commande **CREATE DATABASE**. Cependant, il importe de comprendre, que ceci n'est pas (encore) à prendre comme une commande « **COPY DATABASE** » de portée générale. La principale limitation est qu'aucune autre session ne peut être connectée à la base source tant qu'elle est copiée. **CREATE DATABASE** échouera si une autre connexion existe à son lancement. Lors de l'opération de copie, les nouvelles connexions à la base source sont empêchées.

Deux drapeaux utiles existent dans `pg_database` pour chaque base de données : les colonnes `datistemplate` et `datallowconn`. `datistemplate` peut être positionné à vrai pour indiquer qu'une base de données a vocation à servir de modèle à **CREATE DATABASE**. Si ce drapeau est positionné à vrai, la base de données peut être clonée par tout utilisateur ayant le droit `CREATEDB` ; s'il est positionné à faux, seuls les super-utilisateurs et le propriétaire de la base de données peuvent la cloner. Si

`datallowconn` est positionné à faux, alors aucune nouvelle connexion à cette base de données n'est autorisée (mais les sessions existantes ne sont pas terminées simplement en positionnant ce drapeau à faux). La base de données `template0` est normalement marquée `datallowconn = false` pour empêcher qu'elle ne soit modifiée. Aussi bien `template0` que `template1` devraient toujours être marquées `datistemplate = true`.

> **Note**
>
> `template1` et `template0` n'ont pas de statut particulier en dehors du fait que `template1` est la base de données source par défaut pour la commande **CREATE DATABASE**. Par exemple, on pourrait supprimer `template1` et la recréer à partir de `template0` sans effet secondaire gênant. Ce procédé peut être utile lorsqu'on a encombré `template1` d'objets inutiles. (Pour supprimer `template1`, cette dernière doit avoir le statut `pg_database.datistemplate` à `false`.
>
> La base de données `postgres` est aussi créé quand le groupe est initialisé. Cette base de données a pour but de devenir une base de données par défaut pour la connexion des utilisateurs et applications. C'est une simple copie de `template1` et peut être supprimée et re-créée si nécessaire.

21.4. Configuration d'une base de données

Comme il est dit dans le Chapitre 18, Configuration du serveur, le serveur PostgreSQL™ offre un grand nombre de variables de configuration à chaud. Vous pouvez spécifier des valeurs par défaut, valables pour une base de données particulière, pour nombre de ces variables.

Par exemple, si pour une raison quelconque vous voulez désactiver l'optimiseur GEQO pour une base de donnée particulière, vous n'avez pas besoin de le désactiver pour toutes les bases de données ou de faire en sorte que tout client se connectant exécute la commande `SET geqo TO off;`. Pour appliquer ce réglage par défaut à la base de données en question, vous pouvez exécuter la commande :

```
ALTER DATABASE ma_base SET geqo TO off;
```

Cela sauvegarde le réglage (mais ne l'applique pas immédiatement). Lors des connexions ultérieures à cette base de données, tout se passe comme si la commande `SET geqo TO off` est exécutée juste avant de commencer la session. Notez que les utilisateurs peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut. Pour annuler un tel réglage par défaut, utilisez `ALTER DATABASE` *`nom_base`* `RESET` *`nomvariable`*.

21.5. Détruire une base de données

Les bases de données sont détruites avec la commande DROP DATABASE(7) :

```
DROP DATABASE nom;
```

Seul le propriétaire de la base de données ou un superutilisateur peut supprimer une base de données. Supprimer une base de données supprime tous les objets qui étaient contenus dans la base. La destruction d'une base de données ne peut pas être annulée.

Vous ne pouvez pas exécuter la commande **DROP DATABASE** en étant connecté à la base de données cible. Néanmoins, vous pouvez être connecté à une autre base de données, ceci incluant la base `template1`. `template1` pourrait être la seule option pour supprimer la dernière base utilisateur d'un groupe donné.

Pour une certaine facilité, il existe un script shell qui supprime les bases de données, dropdb(1) :

```
dropdb nom_base
```

(Contrairement à **createdb**, l'action par défaut n'est pas de supprimer la base possédant le nom de l'utilisateur en cours.)

21.6. Tablespaces

Les tablespaces dans PostgreSQL™ permettent aux administrateurs de bases de données de définir l'emplacement dans le système de fichiers où seront stockés les fichiers représentant les objets de la base de données. Une fois créé, un tablespace peut être référencé par son nom lors de la création d'objets.

En utilisant les tablespaces, un administrateur peut contrôler les emplacements sur le disque d'une installation PostgreSQL™. Ceci est utile dans au moins deux cas. Tout d'abord, si la partition ou le volume sur lequel le groupe a été initialisé arrive à court d'espace disque mais ne peut pas être étendu, un tablespace peut être créé sur une partition différente et utilisé jusqu'à ce que le système soit reconfiguré.

Deuxièmement, les tablespaces permettent à un administrateur d'utiliser sa connaissance des objets de la base pour optimiser les performances. Par exemple, un index qui est très utilisé peut être placé sur un disque très rapide et disponible, comme un périphé-

rique mémoire. En même temps, une table stockant des données archivées et peu utilisée ou dont les performances ne portent pas à conséquence pourra être stockée sur un disque système plus lent, moins cher.

Pour définir un tablespace, utilisez la commande CREATE TABLESPACE(7), par exemple :

```
CREATE TABLESPACE espace_rapide LOCATION '/mnt/sda1/postgresql/data';
```

L'emplacement doit être un répertoire existant, possédé par l'utilisateur système d'exploitation de PostgreSQL™. Tous les objets créés par la suite dans le tablespace seront stockés dans des fichiers contenus dans ce répertoire.

Note

Il n'y a généralement aucune raison de créer plus d'un tablespace sur un système de fichiers logique car vous ne pouvez pas contrôler l'emplacement des fichiers individuels à l'intérieur de ce système de fichiers logique. Néanmoins, PostgreSQL™ ne vous impose aucune limitation et, en fait, il n'est pas directement conscient des limites du système de fichiers sur votre système. Il stocke juste les fichiers dans les répertoires que vous lui indiquez.

La création d'un tablespace lui-même doit être fait en tant que superutilisateur de la base de données mais, après cela, vous pouvez autoriser des utilisateurs standards de la base de données à l'utiliser. Pour cela, donnez-leur le droit `CREATE` sur le tablespace.

Les tables, index et des bases de données entières peuvent être affectés à des tablespaces particuliers. Pour cela, un utilisateur disposant du droit `CREATE` sur un tablespace donné doit passer le nom du tablespace comme paramètre de la commande. Par exemple, ce qui suit crée une table dans le tablespace `espace1` :

```
CREATE TABLE foo(i int) TABLESPACE espace1;
```

Autrement, utilisez le paramètre default_tablespace :

```
SET default_tablespace = espace1;
CREATE TABLE foo(i int);
```

Quand `default_tablespace` est configuré avec autre chose qu'une chaîne vide, il fournit une clause `TABLESPACE` implicite pour les commandes **CREATE TABLE** et **CREATE INDEX** qui n'en ont pas d'explicites.

Il existe aussi un paramètre temp_tablespaces, qui détermine l'emplacement des tables et index temporaires, ainsi les fichiers temporaires qui sont utilisés pour le tri de gros ensembles de données. Ce paramètre peut aussi contenir une liste de tablespaces, plutôt qu'une seule, pour que la charge associée aux objets temporaires soit répartie sur plusieurs tablespaces. Un membre de la liste est pris au hasard à chaque fois qu'un objet temporaire doit être créé.

Le tablespace associé avec une base de données est utilisé pour stocker les catalogues système de la base. De plus, il est l'espace par défaut pour les tables, index et fichiers temporaires créés à l'intérieur de cette base de données si aucune clause `TABLESPACE` n'est fournie et qu'aucune sélection n'est spécifiée par `default_tablespace` ou `temp_tablespaces` (comme approprié). Si une base de données est créée sans spécifier de tablespace pour elle, le serveur utilise le même tablespace que celui de la base modèle utilisée comme copie.

Deux tablespaces sont automatiquement créés lors de l'initialisation du cluster de bases de données. Le tablespace `pg_global` est utilisé pour les catalogues système partagés. Le tablespace `pg_default` est l'espace logique par défaut des bases de données `template1` et `template0` (et, du coup, sera le tablespace par défaut pour les autres bases de données sauf en cas de surcharge par une clause `TABLESPACE` dans **CREATE DATABASE**).

Une fois créé, un tablespace peut être utilisé à partir de toute base de données si l'utilisateur le souhaitant dispose du droit nécessaire. Ceci signifie qu'un tablespace ne peut pas supprimé tant que tous les objets de toutes les bases de données utilisant le tablespace n'ont pas été supprimés.

Pour supprimer un tablespace vide, utilisez la commande DROP TABLESPACE(7).

Pour déterminer l'ensemble des tablespaces existants, examinez le catalogue système pg_tablespace, par exemple

```
SELECT spcname FROM pg_tablespace;
```

La métacommande `\db` du programme psql(1) est aussi utile pour afficher les tablespaces existants.

PostgreSQL™ utilise des liens symboliques pour simplifier l'implémentation des tablespaces. Ceci signifie que les tablespaces peuvent être utilisés *seulement* sur les systèmes supportant les liens symboliques.

Le répertoire `$PGDATA/pg_tblspc` contient des liens symboliques qui pointent vers chacun des tablespaces utilisateur dans le groupe. Bien que non recommandé, il est possible d'ajuster la configuration des tablespaces à la main en redéfinissant ces liens. Deux avertissements : ne pas le faire alors que le serveur est en cours d'exécution, mettez à jour le catalogue pg_tablespace pour indiquer les nouveaux emplacements (si vous ne le faites pas, `pg_dump` continuera à afficher les anciens emplacements des tablespaces).

Chapitre 22. Localisation

Ce chapitre décrit, du point de vue de l'administrateur, les fonctionnalités de régionalisation (ou localisation) disponibles. PostgreSQL™ fournit deux approches différentes pour la gestion de la localisation :

- l'utilisation des fonctionnalités de locales du système d'exploitation pour l'ordonnancement du tri, le formatage des chiffres, les messages traduits et autres aspects spécifiques à la locale ;
- la fourniture d'un certain nombre d'encodages différents pour permettre le stockage de texte dans toutes les langues et fournir la traduction de l'encodage entre serveur et client.

22.1. Support des locales

Le support des *locales* fait référence à une application respectant les préférences culturelles au regard des alphabets, du tri, du format des nombres, etc. PostgreSQL™ utilise les possibilités offertes par C et POSIX du standard ISO fournies par le système d'exploitation du serveur. Pour plus d'informations, consulter la documentation du système.

22.1.1. Aperçu

Le support des locales est automatiquement déclenché lorsqu'un cluster de base de données est créé avec **initdb**. **initdb** initialise le cluster avec la valeur des locales de son environnement d'exécution par défaut. Si le système est déjà paramétré pour utiliser la locale souhaitée pour le cluster, il n'y a donc rien d'autre à faire. Si une locale différente est souhaitée (ou que celle utilisée par le serveur n'est pas connue avec certitude), il est possible d'indiquer à **initdb** la locale à utiliser à l'aide de l'option `--locale`. Par exemple :

```
initdb --locale=sv_SE
```

Cet exemple pour les systèmes Unix positionne la locale au suédois (`sv`) tel que parlé en Suède (`SE`). Parmi les autres possibilités, on peut envisager `en_US` (l'anglais américain) ou `fr_CA` (français canadien). Si plus d'un ensemble de caractères peuvent être utilisés pour une locale, alors les spécifications peuvent prendre la forme *`langage_territoire.codeset`*. Par exemple, `fr_BE.UTF-8` représente la langue française telle qu'elle est parlée en Belgique (BE), avec un encodage UTF-8.

Les locales disponibles et leurs noms dépendent de l'éditeur du système d'exploitation et de ce qui est installé. Sur la plupart des systèmes Unix, la commande `locale -a` fournit la liste des locales disponibles. Windows utilise des noms de locale plus verbeux, comme `German_Germany` ou `Swedish_Sweden.1252` mais le principe est le même.

Il est parfois utile de mélanger les règles de plusieurs locales, par exemple d'utiliser les règles de tri anglais avec des messages en espagnol. Pour cela, des sous-catégories de locales existent qui ne contrôlent que certains aspects des règles de localisation :

`LC_COLLATE`	Ordre de tri des chaînes de caractères
`LC_CTYPE`	Classification de caractères (Qu'est-ce qu'une lettre ? La majuscule équivalente ?)
`LC_MESSAGES`	Langue des messages
`LC_MONETARY`	Formatage des valeurs monétaires
`LC_NUMERIC`	Formatage des nombres
`LC_TIME`	Formatage des dates et heures

Les noms des catégories se traduisent en noms d'options **initdb** pour surcharger le choix de locale pour une catégorie donnée. Par exemple, pour utiliser la locale français canadien avec des règles américaines pour le formatage monétaire, on utilise `initdb --locale=fr_CA --lc-monetary=en_US`.

Pour bénéficier d'un système qui se comporte comme s'il ne disposait pas du support des locales, on utilise les locales spéciales `C` ou `POSIX`.

Certaines catégories de locales doivent fixer leur valeurs lors de la création de la base de données. Vous pouvez utiliser des paramétrages différents pour chaque bases de données. Par contre, une fois que la base est créée, les paramétrages de locales ne peuvent plus être modifiés. `LC_COLLATE` et `LC_CTYPE` sont de ces types de catégorie. Elles affectent l'ordre de tri des index et doivent donc rester inchangées, les index sur les colonnes de texte risquant d'être corrompus dans le cas contraire. La valeur par défaut pour ces catégories est déterminée lors de l'exécution d'**initdb**. Ces valeurs sont utilisées quand de nouvelles bases de données sont créées, sauf si d'autres valeurs sont indiquées avec la commande **CREATE DATABASE**.

Les autres catégories de locale peuvent être modifiées à n'importe quel moment en configurant les variables d'environnement de même nom (voir la Section 18.10.2, « Locale et formatage » pour de plus amples détails). Les valeurs par défaut choisies par

initdb sont en fait écrites dans le fichier de configuration `postgresql.conf` pour servir de valeurs par défaut au démarrage du serveur. Si ces déclarations sont désactivées du fichier `postgresql.conf`, le serveur hérite des paramètres de son environnement d'exécution.

Le comportement des locales du serveur est déterminé par les variables d'environnement vues par le serveur, pas par celles de l'environnement d'un quelconque client. Il est donc important de configurer les bons paramètres de locales avant le démarrage du serveur. Cela a pour conséquence que, si les locales du client et du serveur diffèrent, les messages peuvent apparaître dans des langues différentes en fonction de leur provenance.

> **Note**
>
> Hériter la locale de l'environnement d'exécution signifie, sur la plupart des systèmes d'exploitation, la chose suivante : pour une catégorie de locales donnée (l'ordonnancement par exemple) les variables d'environnement `LC_ALL`, `LC_COLLATE` (ou la variable qui correspond à la catégorie) et `LANG` sont consultées dans cet ordre jusqu'à en trouver une qui est fixée. Si aucune de ces variables n'est fixée, c'est la locale par défaut, `C`, qui est utilisée.
>
> Certaines bibliothèques de localisation regardent aussi la variable d'environnement `LANGUAGE` qui surcharge tout autre paramètre pour fixer la langue des messages. En cas de doute, lire la documentation du système d'exploitation, en particulier la partie concernant gettext.

Pour permettre la traduction des messages dans la langue préférée de l'utilisateur, NLS doit avoir été activé pendant la compilation (`configure --enable-nls`). Tout autre support de la locale est construit automatiquement.

22.1.2. Comportement

Le paramétrage de la locale influence les fonctionnalités SQL suivantes :

- l'ordre de tri dans les requêtes utilisant `ORDER BY` ou les opérateurs de comparaison standards sur des données de type texte ;
- la possibilité d'utiliser des index avec les clauses `LIKE` ;
- les fonctions `upper`, `lower` et `initcap` ;
- la famille de fonctions `to_char`.

Le support des locales autres que `C` ou `POSIX` dans PostgreSQL™ a pour inconvénient son impact sur les performances. Il ralentit la gestion des caractères et empêche l'utilisation des index ordinaires par `LIKE`. Pour cette raison, il est préférable de n'utiliser les locales qu'en cas de réel besoin.

Toutefois, pour permettre à PostgreSQL™ d'utiliser des index avec les clauses `LIKE` et une locale différente de `C`, il existe plusieurs classes d'opérateurs personnalisées. Elles permettent la création d'un index qui réalise une stricte comparaison caractère par caractère, ignorant les règles de comparaison des locales. Se référer à la Section 11.9, « Classes et familles d'opérateurs » pour plus d'informations.

22.1.3. Problèmes

Si le support des locales ne fonctionne pas au regard des explications ci-dessus, il faut vérifier que le support des locales du système d'exploitation est correctement configuré. Pour vérifier les locales installées sur le système, on peut utiliser la commande `locale -a`, si elle est fournie avec le système d'exploitation.

Il faut vérifier que PostgreSQL™ utilise effectivement la locale supposée. Les paramètres `LC_COLLATE` et `LC_CTYPE` sont déterminés lors de la création de la base de données et ne peuvent pas être modifiés sauf en créant une nouvelle base de données. D'autres paramètres de locale, y compris `LC_MESSAGES` et `LC_MONETARY`, sont déterminés initialement par l'environnement dans lequel le serveur est lancé mais peuvent être modifiés pendant l'exécution. Pour vérifier le paramétrage de la locale active on utilise la commande **SHOW**.

Le répertoire `src/test/locale` de la distribution source contient une série de tests pour le support des locales dans PostgreSQL™.

Les applications clientes qui gèrent les erreurs en provenance du serveur par l'analyse du texte du message d'erreur vont certainement éprouver des difficultés lorsque les messages du serveur sont dans une langue différente. Les auteurs de telles applications sont invités à utiliser le schéma de code d'erreur à la place.

Le maintien de catalogues de traductions de messages nécessitent les efforts permanents de beaucoup de volontaires qui souhaitent voir PostgreSQL™ parler correctement leur langue préférée. Si certains messages dans une langue ne sont pas disponibles ou pas complètement traduits, toute aide est la bienvenue. Pour apporter son aide à ce projet, consulter le Chapitre 48, Support natif des langues ou écrire à la liste de diffusion des développeurs.

22.2. Support des jeux de caractères

Le support des jeux de caractères dans PostgreSQL™ permet d'insérer du texte dans différents jeux de caractères (aussi appelés encodages), dont ceux mono-octet tels que la série ISO 8859 et ceux multi-octets tels que EUC (Extended Unix Code), UTF-8 ou le codage interne Mule. Tous les jeux de caractères supportés peuvent être utilisés de façon transparente par les clients mais certains ne sont pas supportés par le serveur (c'est-à-dire comme encodage du serveur). Le jeu de caractères par défaut est sélectionné pendant l'initialisation du cluster de base de données avec **initdb**. Ce choix peut être surchargé à la création de la base. Il est donc possible de disposer de bases utilisant chacune un jeu de caractères différent.

Il existe, cependant une importante restriction : le jeu de caractère de la base de données doit être compatible avec les variables d'environnement `LC_CTYPE` (classification des caractères) et `LC_COLLATE` (ordre de tri des chaînes) de la base de données. Pour les locales `C` ou `POSIX`, tous les jeux de caractères sont autorisés, mais pour les locales, il n'y a qu'un seul jeux de caractères qui fonctionne correctement. (Néanmoins, sur Windows, l'encodage UTF-8 peut être utilisé avec toute locale.)

22.2.1. Jeux de caractères supportés

Le Tableau 22.1, « Jeux de caractères de PostgreSQL™ » présente les jeux de caractères utilisables avec PostgreSQL™.

Tableau 22.1. Jeux de caractères de PostgreSQL™

Nom	Description	Langue	Serveur ?	Octets/Caractère
`BIG5`	Big Five	Chinois traditionnel	Non	1-2
`EUC_CN`	Code-CN Unix étendu	Chinois simplifié	Oui	1-3
`EUC_JP`	Code-JP Unix étendu	Japonais	Oui	1-3
`EUC_JIS_2004`	Code-JP Unix étendu, JIS X 0213	Japonais	Oui	1-3
`EUC_KR`	Code-KR Unix étendu	Coréen	Oui	1-3
`EUC_TW`	Code-TW Unix étendu	Chinois traditionnel, taïwanais	Oui	1-3
`GB18030`	Standard national	Chinois	Non	1-2
`GBK`	Standard national étendu	Chinois simplifié	Non	1-2
`ISO_8859_5`	ISO 8859-5, ECMA 113	Latin/Cyrillique	Oui	1
`ISO_8859_6`	ISO 8859-6, ECMA 114	Latin/Arabe	Oui	1
`ISO_8859_7`	ISO 8859-7, ECMA 118	Latin/Grec	Oui	1
`ISO_8859_8`	ISO 8859-8, ECMA 121	Latin/Hébreu	Oui	1
`JOHAB`	JOHAB	Koréen (Hangul)	Non	1-3
`KOI8`	KOI8-R(U)	Cyrillique	Oui	1
`KOI8R`	KOI8-R	Cyrillique (Russie)	Oui	1
`KOI8U`	KOI8-U	Cyrillique (Ukraine)	Oui	1
`LATIN1`	ISO 8859-1, ECMA 94	Europe de l'ouest	Oui	1
`LATIN2`	ISO 8859-2, ECMA 94	Europe centrale	Oui	1
`LATIN3`	ISO 8859-3, ECMA 94	Europe du sud	Oui	1
`LATIN4`	ISO 8859-4, ECMA 94	Europe du nord	Oui	1
`LATIN5`	ISO 8859-9, ECMA 128	Turque	Oui	1
`LATIN6`	ISO 8859-10, ECMA 144	Nordique	Oui	1
`LATIN7`	ISO 8859-13	Baltique	Oui	1
`LATIN8`	ISO 8859-14	Celtique	Oui	1
`LATIN9`	ISO 8859-15	`ISO885915` avec l'Euro et les accents	Oui	1
`LATIN10`	ISO 8859-16, ASRO SR 14111	Roumain	Oui	1
`MULE_INTERNAL`	Code interne Mule	Emacs multi-langues	Oui	1-4
`SJIS`	Shift JIS	Japonais	Non	1-2

Nom	Description	Langue	Serveur ?	Octets/Caractère
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Japonais	Non	1-2
SQL_ASCII	non spécifié (voir le texte)	*tout*	Oui	1
UHC	Code unifié Hangul	Koréen	Non	1-2
UTF8	Unicode, 8-bit	*tous*	Oui	1-4
WIN866	Windows CP866	Cyrillique	Oui	1
WIN874	Windows CP874	Thai	Oui	1
WIN1250	Windows CP1250	Europe centrale	Oui	1
WIN1251	Windows CP1251	Cyrillique	Oui	1
WIN1252	Windows CP1252	Europe de l'ouest	Oui	1
WIN1253	Windows CP1253	Grec	Oui	1
WIN1254	Windows CP1254	Turque	Oui	1
WIN1255	Windows CP1255	Hébreux	Oui	1
WIN1256	Windows CP1256	Arabe	Oui	1
WIN1257	Windows CP1257	Baltique	Oui	1
WIN1258	Windows CP1258	Vietnamien	Oui	1

Toutes les API clients ne supportent pas tous les jeux de caractères de la liste. Le pilote JDBC de PostgreSQL™, par exemple, ne supporte pas MULE_INTERNAL, LATIN6, LATIN8 et LATIN10.

SQL_ASCII se comporte de façon considérablement différente des autres valeurs. Quand le jeu de caractères du serveur est SQL_ASCII, le serveur interprète les valeurs des octets 0-127 suivant le standard ASCII alors que les valeurs d'octets 128-255 sont considérées comme des caractères non interprétés. Aucune conversion de codage n'est effectuée avec SQL_ASCII. De ce fait, cette valeur ne déclare pas tant un encodage spécifique que l'ignorance de l'encodage. Dans la plupart des cas, si des données non ASCII doivent être traitées, il est déconseillé d'utiliser la valeur SQL_ASCII car PostgreSQL™ est alors incapable de convertir ou de valider les caractères non ASCII.

22.2.2. Choisir le jeu de caractères

initdb définit le jeu de caractères par défaut (encodage) pour un cluster. Par exemple,

```
initdb -E EUC_JP
```

paramètre le jeu de caractères à EUC_JP (Extended Unix Code for Japanese). L'option --encoding peut aussi être utilisée à la place de -E (options longues). Si aucune option -E ou --encoding n'est donnée, **initdb** tente de déterminer l'encodage approprié en fonction de la locale indiquée ou de celle par défaut.

Vous pouvez indiquer un encodage autre que celui par défaut lors de la création de la base de données, à condition que l'encodage soit compatible avec la locale sélectionnée :

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

Cela crée une base de données appelée korean qui utilise le jeu de caractères EUC_KR, et la locale ko_KR. Un autre moyen de réaliser cela est d'utiliser la commande SQL suivante :

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr'
LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

Notez que les commandes ci-dessus précisent de copier la base de données template0. Lors de la copie d'une autre base, les paramètres d'encodage et de locale ne peuvent pas être modifiés de ceux de la base de données source car cela pourrait corrompre les données. Pour plus d'informations, voir Section 21.3, « Bases de données modèles ».

L'encodage de la base de données est conservé dans le catalogue système pg_database. Cela est visible à l'aide de l'option -l ou de la commande **\l** de **psql**.

```
$ psql -l
                                        List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |          Access
Privileges
-----------+----------+-----------+-------------+-------------+-----------------------------
```

```
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
{=c/hlinnaka,hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
{=c/hlinnaka,hlinnaka=CTc/hlinnaka}
(7 rows)
```

Important

Sur la plupart des systèmes d'exploitation modernes, PostgreSQL™ peut déterminer le jeu de caractères impliqué par la variable `LC_CTYPE`, et s'assure que l'encodage correspondant de la base de données est utilisé. Sur les systèmes plus anciens, il est de la responsabilité de l'utilisateur de s'assurer que l'encodage attendu par la locale est bien utilisé. Une erreur à ce niveau risque fort de conduire à un comportement étrange des opérations liées à la locale, tel le tri.

PostgreSQL™ autorise les superutilisateurs à créer des bases de données avec le jeu de caractère `SQL_ASCII` même lorsque la variable `LC_CTYPE` n'est pas à `C` ou `POSIX`. Comme indiqué plus haut, `SQL_ASCII` n'impose aucun encodage particulier aux données stockées en base, ce qui rend ce paramétrage vulnérable aux comportements erratiques lors d'opérations liées à la locale. Cette combinaison de paramètres est dépréciée et pourrait un jour être interdite.

22.2.3. Conversion automatique d'encodage entre serveur et client

PostgreSQL™ automatise la conversion de jeux de caractères entre client et serveur pour certaines combinaisons de jeux de caractères. Les informations de conversion sont conservées dans le catalogue système `pg_conversion`. PostgreSQL™ est livré avec certaines conversions prédéfinies, conversions listées dans le Tableau 22.2, « Conversion de jeux de caractères client/serveur ». Une nouvelle conversion peut être créée en utilisant la commande SQL **CREATE CONVERSION**.

Tableau 22.2. Conversion de jeux de caractères client/serveur

Jeu de caractères serveur	Jeux de caractères client disponibles
`BIG5`	*non supporté comme encodage serveur*
`EUC_CN`	*EUC_CN*, `MULE_INTERNAL`, `UTF8`
`EUC_JP`	*EUC_JP*, `MULE_INTERNAL`, `SJIS`, `UTF8`
`EUC_KR`	*EUC_KR*, `MULE_INTERNAL`, `UTF8`
`EUC_TW`	*EUC_TW*, `BIG5`, `MULE_INTERNAL`, `UTF8`
`GB18030`	*non supporté comme encodage serveur*
`GBK`	*non supporté comme encodage serveur*
`ISO_8859_5`	*ISO_8859_5*, `KOI8R`, `MULE_INTERNAL`, `UTF8`, `WIN866`, `WIN1251`
`ISO_8859_6`	*ISO_8859_6*, `UTF8`
`ISO_8859_7`	*ISO_8859_7*, `UTF8`
`ISO_8859_8`	*ISO_8859_8*, `UTF8`
`JOHAB`	*JOHAB*, `UTF8`
`KOI8R`	*KOI8R*, `ISO_8859_5`, `MULE_INTERNAL`, `UTF8`, `WIN866`, `WIN1251`
`KOI8U`	*KOI8U*, `UTF8`
`LATIN1`	*LATIN1*, `MULE_INTERNAL`, `UTF8`
`LATIN2`	*LATIN2*, `MULE_INTERNAL`, `UTF8`, `WIN1250`
`LATIN3`	*LATIN3*, `MULE_INTERNAL`, `UTF8`
`LATIN4`	*LATIN4*, `MULE_INTERNAL`, `UTF8`
`LATIN5`	*LATIN5*, `UTF8`
`LATIN6`	*LATIN6*, `UTF8`

Jeu de caractères serveur	Jeux de caractères client disponibles
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 vers LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*non supporté comme encodage serveur*
SQL_ASCII	*tous (aucune conversion n'est réalisée)*
UHC	*non supporté comme encodage serveur*
UTF8	*tout encodage supporté*
WIN866	*WIN866*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

Pour activer la conversion automatique des jeux de caractères, il est nécessaire d'indiquer à PostgreSQL™ le jeu de caractères (encodage) souhaité côté client. Il y a plusieurs façons de le faire :

- en utilisant la commande **\encoding** dans psql. **\encoding** permet de changer l'encodage client à la volée. Par exemple, pour changer le codage en SJIS, taper :

  ```
  \encoding SJIS
  ```

- la libpq (Section 31.9, « Fonctions de contrôle ») a des fonctions de contrôle de l'encodage client ;
- en utilisant **SET client_encoding TO**. L'encodage client peut être fixé avec la commande SQL suivante :

  ```
  SET CLIENT_ENCODING TO 'valeur';
  ```

  La syntaxe SQL plus standard SET NAMES peut également être utilisée pour cela :

  ```
  SET NAMES 'valeur';
  ```

  Pour connaître l'encodage client courant :

  ```
  SHOW client_encoding;
  ```

  Pour revenir à l'encodage par défaut :

  ```
  RESET client_encoding;
  ```

- en utilisant PGCLIENTENCODING. Si la variable d'environnement PGCLIENTENCODING est définie dans l'environnement client, l'encodage client est automatiquement sélectionné lors de l'établissement d'une connexion au serveur (cette variable peut être surchargée à l'aide de toute autre méthode décrite ci-dessus) ;
- en utilisant la variable de configuration client_encoding. Si la variable client_encoding est définie, l'encodage client est automatiquement sélectionné lors de l'établissement d'une connexion au serveur (cette variable peut être surchargée à l'aide de toute autre méthode décrite ci-dessus).

Si la conversion d'un caractère particulier n'est pas possible -- dans le cas d'encodages EUC_JP pour le serveur et LATIN1 pour le client, et que certains caractères japonais renvoyés n'ont pas de représentation en LATIN1 -- une erreur est remontée.

Si l'encodage client est défini en tant que `SQL_ASCII`, la conversion de l'encodage est désactivée quelque soit celui du serveur. Comme pour le serveur, `SQL_ASCII` est déconseillé sauf à ne travailler qu'avec des données ASCII.

22.2.4. Pour aller plus loin

Il existe quelques sources intéressantes pour commencer à maîtriser les différents jeux de caractères.

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing
: Contient des explications détaillées de `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW`.

http://www.unicode.org/
: Le site web du Unicode Consortium.

RFC 3629
: UTF-8 (8-bit UCS/Unicode Transformation Format) est défini ici.

Chapitre 23. Planifier les tâches de maintenance

PostgreSQL™, comme tout SGBD, requiert que certains tâches soient réalisées de façon régulière pour atteindre les performances optimales. Ces tâches sont *requises*, mais elles sont répétitives par nature et peuvent être facilement automatisées en utilisant des outils standard comme les scripts cron ou le Task Scheduler de Windows. La responsabilité de la mise en place de ces scripts et du contrôle de leur bon fonctionnement relève de l'administrateur de la base.

Une opération de maintenance évidente est la sauvegarde régulière des données. Sans une sauvegarde récente, il est impossible de restaurer après un dommage grave (perte d'un disque, incendie, table supprimée par erreur, etc.). Les mécanismes de sauvegarde et restauration disponibles dans PostgreSQL™ sont détaillés dans le Chapitre 24, Sauvegardes et restaurations.

L'autre tâche primordiale est la réalisation périodique d'un « vacuum », c'est à dire « faire le vide » dans la base de données. Cette opération est détaillée dans la Section 23.1, « Nettoyages réguliers ». La mise à jour des statistiques utilisées par le planificateur de requêtes sera discutée dans Section 23.1.3, « Maintenir les statistiques du planificateur ».

La gestion du fichier de traces mérite aussi une attention régulière. Cela est détaillé dans la Section 23.3, « Maintenance du fichier de traces ».

check_postgres est disponible pour surveiller la santé des bases de données et pour rapporter des conditions inhabituelles. check_postgres s'intègre bien avec Nagios et MRTG, mais il peut aussi fonctionner de façon autonome.

PostgreSQL™ demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à rendre le système productif et agréable à utiliser.

23.1. Nettoyages réguliers

Le SGBD PostgreSQL™ nécessite des opérations de maintenance périodiques, connues sous le nom de *VACUUM*. Pour de nombreuses installations, il est suffisant de laisser travailler le *démon autovacuum*, qui est décrit dans Section 23.1.5, « Le démon auto-vacuum ». En fonction des cas, les paramètres de cet outil peuvent être optimisés pour obtenir de meilleurs résultats. Certains administrateurs de bases de données voudront suppléer ou remplacer les activités du démon avec une gestion manuelle des commandes **VACUUM**, qui seront typiquement exécutées suivant un planning par des scripts cron ou par le Task Scheduler. Pour configurer une gestion manuelle et correcte du VACUUM, il est essentiel de bien comprendre les quelques sous-sections suivantes. Les administrateurs qui se basent sur l'autovacuum peuvent toujours lire ces sections pour les aider à comprendre et à ajuster l'autovacuum.

23.1.1. Bases du VACUUM

La commande VACUUM(7) de PostgreSQL™ doit traiter chaque table régulièrement pour plusieurs raisons :

1. pour récupérer ou ré-utiliser l'espace disque occupé par les lignes supprimées ou mises à jour ;
2. pour mettre à jour les statistiques utilisées par l'optimiseur de PostgreSQL™ ;
3. pour prévenir la perte des données les plus anciennes à cause d'un *cycle de l'identifiant de transaction (XID)*.

Chacune de ces raisons impose de réaliser des opérations **VACUUM** de différentes fréquences et portées, comme expliqué dans les sous-sections suivantes.

Il y a deux variantes de la commande **VACUUM** : **VACUUM** standard et **VACUUM FULL**. **VACUUM FULL** peut récupérer davantage d'espace disque mais s'exécute beaucoup plus lentement. Par ailleurs, la forme standard de **VACUUM** peut s'exécuter en parallèle avec les opérations de production des bases. Des commandes comme **SELECT**, **INSERT**, **UPDATE** et **DELETE** continuent de fonctionner de façon normale, mais la définition d'une table ne peut être modifiée avec des commandes telles que **ALTER TABLE** pendant le **VACUUM**. **VACUUM FULL** nécessite un verrou exclusif sur la table sur laquelle il travaille, et ne peut donc pas être exécuté en parallèle avec une autre activité sur la table. En règle générale, par conséquent, les administrateurs doivent s'efforcer d'utiliser la commande standard **VACUUM** et éviter **VACUUM FULL**.

VACUUM produit un nombre important d'entrées/sorties, ce qui peut entraîner de mauvaises performances pour les autres sessions actives. Des paramètres de configuration peuvent être ajustés pour réduire l'impact d'une opération VACUUM en arrière plan sur les performances -- voir Section 18.4.3, « Report du VACUUM en fonction de son coût ».

23.1.2. Récupérer l'espace disque

Avec PostgreSQL™, les versions périmées des lignes ne sont pas immédiatement supprimées après une commande **UPDATE** ou **DELETE**. Cette approche est nécessaire pour la consistance des accès concurrents (MVCC, voir le Chapitre 13, Contrôle d'accès simultané) : la version de la ligne ne doit pas être supprimée tant qu'elle est susceptible d'être lue par une autre transaction. Mais finalement, une ligne qui est plus vieille que toutes les transactions en cours n'est plus utile du tout. La place qu'elle

utilise doit être rendue pour être réutilisée par d'autres lignes afin d'éviter un accroissement constant, sans limite, du volume occupé sur le disque. Cela est réalisé en exécutant **VACUUM**.

La forme standard de **VACUUM** élimine les versions d'enregistrements morts dans les tables et les index, et marque l'espace comme réutilisable. Néanmoins, il ne rend pas cet espace au système d'exploitation, sauf dans le cas spécial où des pages à la fin d'une table deviennent totalement vides et qu'un verrou exclusif sur la table peut être obtenu aisément. Par opposition, **VACUUM FULL** compacte activement les tables en écrivant une nouvelle version complète du fichier de la table, sans espace vide. Ceci réduit la taille de la table mais peut prendre beaucoup de temps. Cela requiert aussi un espace disque supplémentaire pour la nouvelle copie de la table jusqu'à la fin de l'opération.

Le but habituel d'un vacuum régulier est de lancer des **VACUUM** standard suffisamment souvent pour éviter d'avoir recours à **VACUUM FULL**. Le démon autovacuum essaie de fonctionner de cette façon, et n'exécute jamais de **VACUUM FULL**. Avec cette approche, l'idée directrice n'est pas de maintenir les tables à leur taille minimale, mais de maintenir l'utilisation de l'espace disque à un niveau constant : chaque table occupe l'espace équivalent à sa taille minimum plus la quantité d'espace consommée entre deux vacuums. Bien que **VACUUM FULL** puisse être utilisé pour retourner une table à sa taille minimale et rendre l'espace disque au système d'exploitation, cela ne sert pas à grand chose, si cette table recommence à grossir dans un futur proche. Par conséquent, cette approche s'appuyant sur des commandes **VACUUM** exécutées à intervalles modérément rapprochés est une meilleure approche que d'exécuter des **VACUUM FULL** espacés pour des tables mises à jour de façon intensive.

Certains administrateurs préfèrent planifier le passage de **VACUUM** eux-mêmes, par exemple faire le travail de nuit, quand la charge est faible. La difficulté avec cette stratégie est que si une table a un pic d'activité de mise à jour inattendu, elle peut grossir au point qu'un **VACUUM FULL** soit vraiment nécessaire pour récupérer l'espace. L'utilisation du démon d'autovacuum minore ce problème, puisque le démon planifie les vacuum de façon dynamique, en réponse à l'activité de mise à jour. Il est peu raisonnable de désactiver totalement le démon, sauf si l'activité de la base est extrêmement prévisible. Un compromis possible est de régler les paramètres du démon afin qu'il ne réagisse qu'à une activité exceptionnellement lourde de mise à jour, de sorte à éviter seulement de perdre totalement le contrôle de la volumétrie, tout en laissant les **VACUUM** planifiés faire le gros du travail quand la charge est normale.

Pour ceux qui n'utilisent pas autovacuum, une approche typique alternative est de planifier un **VACUUM** sur la base complète une fois par jour lorsque l'utilisation n'est pas grande, avec en plus des opérations de **VACUUM** plus fréquentes pour les tables très impactées par des mises à jour, de la façon adéquate. (Certaines installations avec énormément de mises à jour peuvent exécuter des VACUUM toutes les quelques minutes.) Lorsqu'il y a plusieurs bases dans un cluster, il faut penser à exécuter un **VACUUM** sur chacune d'elles ; le programme vacuumdb(1) peut être utile.

Astuce

Le **VACUUM** simple peut ne pas suffire quand une table contient un grand nombre d'enregistrements morts comme conséquence d'une mise à jour ou suppression massive. Dans ce cas, s'il est nécessaire de récupérer l'espace disque gaspillé, **VACUUM FULL** peut être utilisé, CLUSTER(7) ou une des variantes de ALTER TABLE(7). Ces commandes écrivent une nouvelle copie de la table et lui adjoignent de nouveaux index. Toutes ces options nécessitent un verrou exclusif. Elles utilisent aussi temporairement un espace disque supplémentaire, approximativement égal à la taille de la table, car les anciennes copies de la table et des index ne peuvent pas être supprimées avant la fin de l'opération.

Astuce

Si le contenu d'une table est supprimé périodiquement, il est préférable d'envisager l'utilisation de TRUNCATE(7), plutôt que **DELETE** suivi de **VACUUM**. **TRUNCATE** supprime le contenu entier de la table immédiatement sans nécessiter de **VACUUM** ou **VACUUM FULL** pour réclamer l'espace disque maintenant inutilisé. L'inconvénient est la violation des sémantiques MCC strictes.

23.1.3. Maintenir les statistiques du planificateur

L'optimiseur de requêtes de PostgreSQL™ s'appuie sur des informations statistiques sur le contenu des tables dans l'optique de produire des plans d'exécutions efficaces pour les requêtes. Ces statistiques sont collectées par la commande ANALYZE(7), qui peut être invoquée seule ou comme option de **VACUUM**. Il est important d'avoir des statistiques relativement à jour, ce qui permet d'éviter les choix de mauvais plans d'exécution, pénalisant les performances de la base.

Le démon d'autovacuum, si activé, va automatiquement exécuter des commandes **ANALYZE** à chaque fois que le contenu d'une table aura changé suffisamment. Toutefois, un administrateur peut préférer se fier à des opérations **ANALYZE** planifiées manuellement, en particulier s'il est connu que l'activité de mise à jour de la table n'a pas d'impact sur les statistiques des colonnes « intéressantes ». Le démon planifie des **ANALYZE** uniquement en fonction du nombre d'enregistrements insérés, mis à jour ou supprimés

À l'instar du nettoyage pour récupérer l'espace, les statistiques doivent être plus souvent collectées pour les tables intensément modifiées que pour celles qui le sont moins. Mais même si la table est très modifiée, il se peut que ces collectes soient inutiles si la distribution probabiliste des données évolue peu. Une règle simple pour décider est de voir comment évoluent les valeurs minimum et maximum des données. Par exemple, une colonne de type timestamp qui contient la date de mise à jour de la ligne aura une valeur maximum en continuelle croissance au fur et à mesure des modifications ; une telle colonne nécessitera plus de collectes statistiques qu'une colonne qui contient par exemple les URL des pages accédées sur un site web. La colonne qui contient les URL peut très bien être aussi souvent modifiée mais la distribution probabiliste des données changera certainement moins rapidement.

Il est possible d'exécuter **ANALYZE** sur des tables spécifiques, voire des colonnes spécifiques ; il a donc toute flexibilité pour mettre à jour certaines statistiques plus souvent que les autres en fonction des besoins de l'application. Quoi qu'il en soit, dans la pratique, il est généralement mieux de simplement analyser la base entière car il s'agit d'une opération rapide. **ANALYZE** utilise un système d'échantillonage des lignes d'une table, ce qui lui évite de lire chaque ligne.

Astuce

Même si il n'est pas très productif de règler précisément la fréquence de **ANALYZE** pour chaque colonne, il peut être intéressant d'ajuster le niveau de détail des statistiques collectées pour chaque colonne. Les colonnes très utilisées dans les clauses `WHERE` et dont la distribution n'est pas uniforme requièrent des histogrammes plus précis que les autres colonnes. Voir **ALTER TABLE SET STATISTICS**, ou modifier les paramètres par défaut de la base de données en utilisant le paramètre de configuration default_statistics_target.

De plus, par défaut, il existe peu d'informations sur la sélectivité des fonctions. Néanmoins, si vous créez un index qui utilise une fonction, des statistiques utiles seront récupérées de la fonction, ce qui peut grandement améliorer les plans de requêtes qui utilisent l'index.

23.1.4. Éviter les cycles des identifiants de transactions

Le mécanisme de contrôle de concurrence multiversion (MVCC) de PostgreSQL™ s'appuie sur la possibilité de comparer des identifiants de transactions (XID) ; c'est un nombre croissant : la version d'une ligne dont le XID d'insertion est supérieur au XID de la transaction en cours est « dans le futur » et ne doit pas être visible de la transaction courante. Comme les identifiants ont une taille limitée (32 bits), un groupe qui est en activité depuis longtemps (plus de 4 milliards de transactions) pourrait connaître un cycle des identifiants de transaction : le XID reviendra à 0 et soudainement les transactions du passé sembleront appartenir au futur - ce qui signifie qu'elles deviennent invisibles. En bref, perte de données totale. (En réalité, les données sont toujours là mais c'est un piètre réconfort puisqu'elles resteront inaccessibles.) Pour éviter ceci, il est nécessaire d'exécuter un VACUUM sur chaque table de chaque base au moins au moins une fois à chaque milliard de transactions.

La raison pour laquelle un VACUUM périodique résout le problème est que PostgreSQL™ réserve un ID de transaction spécial, `FrozenXID`. Ce XID ne suit pas les règles standards de comparaison des identifiants de transactions et est toujours considéré comme plus âgé que les XID normaux. Les XID normaux sont comparés sur une base modulo-2^{32}. Cela signifie que pour chaque XID normal, il y en a deux milliards qui sont plus vieux et deux milliards qui sont plus récents. Une autre manière de le dire est que l'ensemble de définition des XID est circulaire et sans limite. De plus, une ligne créée avec un XID normal donné, la version de la ligne apparaîtra comme appartenant au passé pour les deux milliards de transactions qui suivront quelque soit le XID. Si la ligne existe encore après deux milliards de transactions, elle apparaîtra soudainement comme appartenant au futur. Pour éviter ceci, les versions trop anciennes doivent se voir affecter le XID `FrozenXID` avant d'atteindre le seuil fatidique des deux milliards de transactions. Une fois qu'elles ont ce XID spécifique, elles appartiendront au passé pour toutes les transactions même en cas de cycle. Cette affectation des anciens XID est réalisée par **VACUUM**.

vacuum_freeze_min_age contrôle l'âge que doit avoir une valeur XID avant qu'elle soit remplacée par `FrozenXID`. Les valeurs plus importantes de ces deux paramètres préservent l'information transactionnelle plus longtemps alors que les valeurs plus petites augmentent le nombre de transactions qui peuvent survenir avant un nouveau VACUUM de la table.

VACUUM ignore habituellement les pages qui n'ont pas de lignes mortes, mais ces pages pourraient toujours avoir des versions de lignes avec d'anciennes valeurs XID. Pour s'assurer que tous les anciens XID ont été remplacés par `FrozenXID`, un parcours complet de la table est nécessaire. vacuum_freeze_table_age contrôle quand **VACUUM** fait cela : un parcours complet est forcé si la table n'a pas été parcourue complètement pendant `vacuum_freeze_table_age` - `vacuum_freeze_min_age` transactions. En le configurant à zéro, cela force **VACUUM** à toujours parcourir toutes les pages, ignorant de ce fait la carte de visibilité.

Le temps maximum où une table peut rester sans VACUUM est de deux millions de transactions moins `vacuum_freeze_min_age` quand **VACUUM** a parcouru la table complètement pour la dernière fois. Si elle devait rester sans VACUUM après cela, des pertes de données pourraient survenir. Pour s'assurer que cela n'arrive pas, autovacuum est appelé sur chaque table qui pourrait contenir des XID plus âgés que ne l'indique le paramètre de configuration autovacuum_freeze_max_age. (Ceci arrivera même si autovacuum est désactivé.)

Ceci implique que, si aucune opération de VACUUM n'est demandée sur une table, l'autovacuum sera automatiquement déclenché

une fois toutes les autovacuum_freeze_max_age moins vacuum_freeze_min_age transactions. Pour les tables qui ont régulièrement l'opération de VACUUM pour réclamer l'espace perdu, ceci a peu d'importance. Néanmoins, pour les tables statiques (ceci incluant les tables qui ont des INSERT mais pas d'UPDATE ou de DELETE), il n'est pas nécessaire d'exécuter un VACUUM pour récupérer de la place et donc il peut être utile d'essayer de maximiser l'interval entre les autovacuums forcés sur de très grosses tables statiques. Évidemment, vous pouvez le faire soit en augmentant autovacuum_freeze_max_age soit en diminuant vacuum_freeze_min_age.

Le maximum efficace pour vacuum_freeze_table_age est 0.95 * autovacuum_freeze_max_age ; un paramétrage plus haut que ça sera limité à ce maximum. Une valeur plus importante qie autovacuum_freeze_max_age n'aurait pas de sens car un autovacuum de préservation contre la ré-utilisation des identifiants de transactions serait déclenché, et le multiplicateur 0,95 laisse un peu de place pour exécuter un **VACUUM** manuel avant que cela ne survienne. Comme règle d'or, **vacuum_freeze_table_age** devrait être configuré à une valeur légèrement inférieure à autovacuum_freeze_max_age, laissant suffisamment d'espace pour qu'un **VACUUM** planifié régulièrement ou pour qu'un autovacuum déclenché par des activités normales de suppression et de mise à jour puissent être activés pendant ce laps de temps. Le configurer de façon trop proche pourrait déclencher des autovacuum de protection contre la ré-utilisation des identifiants de transactions, même si la table a été récemment l'objet d'un VACUUM pour récupérer l'espace, alors que des valeurs basses amènent à des parcours complets de table plus fréquents.

Le seul inconvénient à augmenter autovacuum_freeze_max_age (et vacuum_freeze_table_age avec elle) est que le sous-répertoire pg_clog du cluster prendre plus de place car il doit stocker le statut du COMMIT pour toutes les transactions depuis autovacuum_freeze_max_age. L'état de COMMIT utilise deux bits par transaction, donc si autovacuum_freeze_max_age et vacuum_freeze_table_age ont une valeur maximum permise de deux milliards, pg_clog peut grossir jusqu'à la moitié d'un Go. Si c'est rien comparé à votre taille de base totale, configurer autovacuum_freeze_max_age à son maximum permis est recommandé. Sinon, le configurer suivant ce que vous voulez comme stockage maximum dans pg_clog. (La valeur par défaut, 200 millions de transactions, se traduit en à peu près 50 Mo de stockage dans pg_clog.)

Un inconvénient causé par la diminution de vacuum_freeze_min_age est que cela pourrait faire que **VACUUM** travaille sans raison : modifier le XID de la ligne d'une table à FrozenXID est une perte de temps si la ligne est modifiée rapidement après (ce qui fait qu'elle obtiendra un nouveau XID). Donc ce paramètre doit être suffisamment important pour que les lignes ne soient pas gelées jusqu'à ce qu'il soit pratiquement certain qu'elles ne seront plus modifiées. Un autre inconvénient en diminuant ce paramètre est que les détails sur la transaction exacte qui a inséré ou modifié une ligne seront perdus plus tôt. Cette information est quelque fois utile, particulièrement lors d'une analyse de ce qui s'est mal passé sur une base après un problème. Pour ces deux raisons, baisser ce paramètre n'est pas recommandé sauf pour les tables entièrement statiques.

Pour tracer l'âge des plus anciens XID de la base, **VACUUM** stocke les statistiques sur XID dans les tables systèmes pg_class et pg_database. En particulier, la colonne *relfrozenxid* de la ligne pg_class d'une table contient le XID final du gel qui a été utilisé par le dernier **VACUUM** pour cette table. Il est garanti que tous les XID plus anciens que ce XID ont été remplacés par FrozenXID pour cette table. De façon similaire, la colonne *datfrozenxid* de la ligne pg_database de la base est une limite inférieure des XID normaux apparaissant dans cette base -- c'est tout simplement le minimum des valeurs *relfrozenxid* par table dans cette base. Pour examiner cette information, le plus simple est d'exécuter des requêtes comme :

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind = 'r';

SELECT datname, age(datfrozenxid) FROM pg_database;
```

La colonne age mesure le nombre de transactions à partir du XID final vers le XID de transaction en cours.

VACUUM parcourt habituellement seulement les pages qui ont été modifiées depuis le dernier VACUUM mais *relfrozenxid* peut seulement être avancé quand la table est parcourue complètement. La table est parcourue entièrement quand *relfrozenxid* est agée de plus de vacuum_freeze_table_age transactions, quand l'option FREEZE de la commande **VACUUM** est utilisée ou quand toutes les pages se trouvent nécessiter un VACUUM pour supprimer les versions mortes des lignes. Après que **VACUUM** ait parcouru la table complète age(relfrozenxid) devrait être un peu plus grande que le paramètre vacuum_freeze_min_age qui a été utilisé (la différence étant due au nombre de transactions démarrées depuis que **VACUUM** a commencé son travail). Si aucun parcours de table complet ne se trouve exécuté via un **VACUUM** sur cette table, lorsque autovacuum_freeze_max_age est atteint, un autovacuum sera rapidement forcé sur la table.

Si, pour une certaine raison, l'autovacuum échoue à effacer les anciens XID d'une table, le système commencera à émettre des messages d'avertissement comme ceci quand les plus anciens XID de la base atteignent les 10 millions de transactions à partir du point de réinitialisation :

```
WARNING:  database "mydb" must be vacuumed within 177009986 transactions
```

```
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

(Une commande **VACUUM** manuelle devrait résoudre le problème, comme suggéré par l'indice ; mais notez que la commande **VACUUM** doit être exécutée par un superutilisateur, sinon elle échouera à mettre à jour les catalogues systèmes et ne pourra donc pas faire avancer le *datfrozenxid* de la base.) Si ces avertissements sont ignorés, le système s'arrêtera et refusera de commencer toute nouvelle transaction dès qu'il n'en restera qu'un million avant la réinitialisation :

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database
"mydb"
HINT:  Stop the postmaster and use a standalone backend to VACUUM in "mydb".
```

La marge de sécurité de un million de transactions existe pour permettre à l'administrateur de récupérer ces données sans perte en exécutant manuellement les commandes **VACUUM** requises. Néanmoins, comme le système n'exécute pas de commandes tant qu'il n'est pas sorti du mode d'arrêt par sécurité, la seule façon de le faire est de stopper le serveur et d'utiliser un moteur simple utilisateur pour exécuter le **VACUUM**. Le mode d'arrêt n'est pas pris en compte par le moteur simple utilisateur. Voir la page de référence de postgres(1) pour des détails sur l'utilisation du moteur simple utilisateur.

23.1.5. Le démon auto-vacuum

PostgreSQL ™ dispose d'une fonctionnalité optionnelle mais hautement recommandée appelée *autovacuum*, dont le but est d'automatiser l'exécution des commandes **VACUUM** et **ANALYZE** . Une fois activé, autovacuum vérifie les tables ayant un grand nombre de lignes insérées, mises à jour ou supprimées. Ces vérifications utilisent la fonctionnalité de récupération de statistiques ; du coup, autovacuum ne peut pas être utilisé sauf si track_counts est configuré à `true`. Dans la configuration par défaut, l'autovacuum est activé et les paramètres liés sont correctement configurés.

Le « démon autovacuum » est constitué de plusieurs processus. Un processus démon permanent appelé *autovacuum launcher* (autrement dit le lanceur d'autovacuum) est en charge de lancer des processus travailleur (*autovacuum worker*) pour toutes les bases de données. Le lanceur distribuera le travail dans le temps mais essaiera de lancer un nouveau travailleur sur chaque base de données chaque autovacuum_naptime secondes. (Du coup, si l'installation a *N* bases de données, un nouveau *autovacuum worker* sera lancé tous les `autovacuum_naptime`/*N* secondes.) Un maximum de autovacuum_max_workers processus *autovacuum worker* est autorisé à s'exécuter en même temps. S'il y a plus de `autovacuum_max_workers` bases à traiter, la prochaine base de données sera traitée dès qu'un autre travailleur a terminé. Chaque processus travailleur vérifiera chaque table de leur base de données et exécutera un **VACUUM** et/ou un **ANALYZE** suivant les besoins.

Si plusieurs grosses tables deviennent toutes éligibles pour un VACUUM dans un court espace de temps, tous les processus travailleurs pourraient avoir à exécuter des VACUUM sur ces tables pendant un long moment. Ceci aura pour résultat que d'autres tables et d'autres bases de données ne pourront pas être traitées tant qu'un processus travailleur ne sera pas disponible. Il n'y a pas de limite sur le nombre de processus travailleurs sur une seule base, mais ils essaient d'éviter de répéter le travail qui a déjà été fait par d'autres. Notez que le nombre de processus travailleurs en cours d'exécution n'est pas décompté des limites max_connections et superuser_reserved_connections.

Les tables dont la valeur de *relfrozenxid* est plus importante que autovacuum_freeze_max_age sont toujours l'objet d'un VACUUM (cela s'applique aux tables dont le 'freeze max age' a été modifié par les paramètres de stockage ; voyez plus bas). Sinon, si le nombre de lignes obsolètes depuis le dernier **VACUUM** dépasse une « limite de vacuum », la table bénéficie d'un VACUUM. La limite est définie ainsi :

```
limite du vacuum = limite de base du vacuum + facteur d'échelle du vacuum * nombre de
lignes
```

où la limite de base du vacuum est autovacuum_vacuum_threshold, le facteur d'échelle du vacuum est autovacuum_vacuum_scale_factor et le nombre de lignes est pg_class.*reltuples*. Le nombre de lignes obsolètes est obtenu à partir du récupérateur de statistiques ; c'est un nombre à peu près précis, mis à jour après chaque instruction **UPDATE** et **DELETE** (il est seulement à peu près précis car certaines informations pourraient être perdues en cas de grosse charge). Si la valeur de *relfrozenxid* pour la table est supérieure à `vacuum_freeze_table_age`, la table complète est parcourue pour geler les anciennes lignes et pour avancer *relfrozenxid*, sinon seules les pages qui ont été modifiées depuis le dernier VACUUM sont parcourues par l'opération de VACUUM.

Pour ANALYZE, une condition similaire est utilisée : la limite, définie comme

```
limite du analyze = limite de base du analyze + facteur d'échelle du analyze * nombre
de lignes
```

est comparée au nombre de lignes insérées, mises à jour et supprimées depuis le dernier **ANALYZE**.

Les tables temporaires ne peuvent pas être accédées par l'autovacuum. Du coup, les opérations appropriées de VACUUM et d'ANALYZE devraient être traitées par des commandes SQL de session.

Les limites et facteurs d'échelle par défaut sont pris dans `postgresql.conf`, mais il est possible de les surcharger table par

table ; voir la section intitulée « Paramètres de stockage » pour plus d'informations. Si un paramètre a été modifié via les paramètres de stockage, cette valeur est utilisée ; sinon les paramètres globaux sont utilisés. Voir Section 18.9, « Nettoyage (**vacuum**) automatique » pour plus d'informations sur les paramètres globaux.

En plus des valeurs de la limite de base et des facteurs d'échelle, il existe six autres paramètres autovacuum pouvant être configurés pour chaque table via les paramètres de stockage. Le premier paramètre, `autovacuum_enabled`, peut être configuré à `false` pour instruire le démon autovacuum de laisser cette table particulière. Dans ce cas, autovacuum touchera seulement la table quand il devra le faire pour prévenir la réinitialisation de l'ID de transaction. Deux autres paramètres, le délai du coût du VACUUM (`autovacuum_vacuum_cost_delay`) et la limite du coût du VACUUM (`autovacuum_vacuum_cost_limit`), sont utilisés pour configurer des valeurs spécifiques aux tables pour la fonctionnalité de délai de VACUUM basé sur le coût (voir Section 18.4.3, « Report du VACUUM en fonction de son coût »). `autovacuum_freeze_min_age`, `autovacuum_freeze_max_age` et `autovacuum_freeze_table_age` sont utilisés pour configurer des valeurs par table, respectivement vacuum_freeze_min_age, autovacuum_freeze_max_age et vacuum_freeze_table_age.

Lorsque plusieurs processus autovacuum sont en cours d'exécution, la limite de coût est « répartie » parmi tous les processus pour que l'impact total sur le système soit identique quelque soit le nombre de processus en cours d'exécution.

23.2. Ré-indexation régulière

Dans certains cas, reconstruire périodiquement les index par la commande REINDEX(7) vaut la peine.

Les pages de l'index B-tree, qui sont devenues complètement vides, sont réclamées pour leur ré-utilisation. Mais, il existe toujours une possibilité d'utilisation peu efficace de l'espace : si, sur une page, seulement plusieurs clés d'index ont été supprimés, la page reste allouée. En conséquence, si seulement quelques clés sont supprimées, vous devrez vous attendre à ce que l'espace disque soit très mal utilisé. Dans de tels cas, la réindexation périodique est recommandée.

Le potentiel d'inflation des index qui ne sont pas des index B-tree n'a pas été particulièrement analysé. Surveiller périodiquement la taille physique de ces index est une bonne idée.

De plus, pour les index B-tree, un index tout juste construit est légèrement plus rapide qu'un index qui a été mis à jour plusieurs fois parce que les pages adjacentes logiquement sont habituellement aussi physiquement adjacentes dans un index nouvellement créé (cette considération ne s'applique pas aux index non B-tree). Il pourrait être intéressant de ré-indexer périodiquement simplement pour améliorer la vitesse d'accès.

23.3. Maintenance du fichier de traces

Sauvegarder les journaux de trace du serveur de bases de données dans un fichier plutôt que dans `/dev/NULL` est une bonne idée. Les journaux sont d'une utilité incomparable lorsqu'arrive le moment où des problèmes surviennent. Néanmoins, les journaux ont tendance à être volumineux (tout spécialement à des niveaux de débogage importants) et vous ne voulez pas les sauvegarder indéfiniment. Vous avez besoin de faire une « rotation » des journaux pour que les nouveaux journaux sont commencés et que les anciens soient supprimés après une période de temps raisonnable.

Si vous redirigez simplement `stderr` du **postgres** dans un fichier, vous aurez un journal des traces mais la seule façon de le tronquer sera d'arrêter et de relancer le serveur. Ceci peut convenir si vous utilisez PostgreSQL™ dans un environnement de développement mais peu de serveurs de production trouveraient ce comportement acceptable.

Une meilleure approche est d'envoyer la sortie `stderr` du serveur dans un programme de rotation de journaux. Il existe un programme interne de rotation que vous pouvez utiliser en configurant le paramètre `logging_collector` à `true` dans `postgresql.conf`. Les paramètres de contrôle pour ce programme sont décrits dans Section 18.7.1, « Où tracer ». Vous pouvez aussi utiliser cette approche pour capturer les données des journaux applicatifs dans un format CSV (valeurs séparées par des virgules) lisible par une machine

Sinon, vous pourriez préférer utiliser un programme externe de rotation de journaux si vous en utilisez déjà un avec d'autres serveurs. Par exemple, l'outil rotatelogs inclus dans la distribution Apache™ peut être utilisé avec PostgreSQL™. Pour cela, envoyez via un tube la sortie `stderr` du serveur dans le programme désiré. Si vous lancez le serveur avec **pg_ctl**, alors `stderr` est déjà directement renvoyé dans `stdout`, donc vous avez juste besoin d'ajouter la commande via un tube, par exemple :

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Une autre approche de production pour la gestion des journaux de trace est de les envoyer à syslog et de laisser syslog gérer la rotation des fichiers. Pour cela, initialisez le paramètre de configuration `log_destination` à `syslog` (pour tracer uniquement via syslog) dans `postgresql.conf`. Ensuite, vous pouvez envoyer un signal `SIGHUP` au démon syslog quand vous voulez le forcer à écrire dans un nouveau fichier. Si vous voulez automatiser la rotation des journaux, le programme logrotate peut être configuré pour fonctionner avec les journaux de traces provenant de syslog.

Néanmoins, sur beaucoup de systèmes, syslog n'est pas très fiable, particulièrement avec les messages très gros ; il pourrait tronquer ou supprimer des messages au moment où vous en aurez le plus besoin. De plus, sur Linux™, syslog synchronisera tout mes-

sage sur disque, amenant des performances assez pauvres. (Vous pouvez utiliser un « - » au début du nom de fichier dans le fichier de configuration syslog pour désactiver la synchronisation.)

Notez que toutes les solutions décrites ci-dessus font attention à lancer de nouveaux journaux de traces à des intervalles configurables mais ils ne gèrent pas la suppression des vieux fichiers de traces, qui ne sont probablement plus très utiles. Vous voudrez probablement configurer un script pour supprimer périodiquement les anciens journaux. Une autre possibilité est de configurer le programme de rotation pour que les anciens journaux de traces soient écrasés de façon cyclique.

pgFouine est un projet externe qui analyse les journaux applicatifs d'une façon très poussée. *check_postgres* fournit des alertes Nagios quand des messages importants apparaît dans les journaux applicatifs, mais détecte aussi de nombreux autres cas.

Chapitre 24. Sauvegardes et restaurations

Comme tout ce qui contient des données importantes, les bases de données PostgreSQL™ doivent être sauvegardées régulièrement. Bien que la procédure soit assez simple, il est important de comprendre les techniques et hypothèses sous-jacentes.

Il y a trois approches fondamentalement différentes pour sauvegarder les données de PostgreSQL™ :

- la sauvegarde SQL ;
- la sauvegarde au niveau du système de fichiers ;
- l'archivage continu.

Chacune a ses avantages et ses inconvénients. Elles sont toutes analysées, chacune leur tour, dans les sections suivantes.

24.1. Sauvegarde SQL

Le principe est de produire un fichier texte de commandes SQL (appelé « fichier dump »), qui, si on le renvoie au serveur, recrée une base de données identique à celle sauvegardée. PostgreSQL™ propose pour cela le programme utilitaire pg_dump(1). L'usage basique est :

```
pg_dump base_de_donnees > fichier_de_sortie
```

pg_dump écrit son résultat sur la sortie standard. Son utilité est expliquée plus loin.

pg_dump est un programme client PostgreSQL™ classique (mais plutôt intelligent). Cela signifie que la sauvegarde peut être effectuée depuis n'importe quel ordinateur ayant accès à la base. Mais pg_dump n'a pas de droits spéciaux. Il doit, en particulier, avoir accès en lecture à toutes les tables à sauvegarder, si bien qu'il doit être lancé pratiquement toujours en tant que superutilisateur de la base.

Pour préciser le serveur de bases de données que pg_dump doit contacter, on utilise les options de ligne de commande `-h` *`serveur`* et `-p` *`port`*. Le serveur par défaut est le serveur local ou celui indiqué par la variable d'environnement `PGHOST`. De la même façon, le port par défaut est indiqué par la variable d'environnement `PGPORT` ou, en son absence, par la valeur par défaut précisée à la compilation. Le serveur a normalement reçu les mêmes valeurs par défaut à la compilation.

Comme tout programme client PostgreSQL™, pg_dump se connecte par défaut avec l'utilisateur de base de données de même nom que l'utilisateur système courant. L'utilisation de l'option `-U` ou de la variable d'environnement `PGUSER` permettent de modifier le comportement par défaut. Les connexions de pg_dump sont soumises aux mécanismes normaux d'authentification des programmes clients (décrits dans le Chapitre 19, Authentification du client).

Un des gros avantages de pg_dump sur les autres méthodes de sauvegarde décrites après est que la sortie de pg_dump peut être généralement re-chargée dans des versions plus récentes de PostgreSQL™, alors que les sauvegardes au niveau fichier et l'archivage continu sont tous les deux très spécifique à la version du serveur pg_dump est aussi la seule méthode qui fonctionnera lors du transfert d'une base de données vers une machine d'une architecture différente (comme par exemple d'un serveur 32 bits à un serveur 64 bits).

Les sauvegardes créées par pg_dump sont cohérentes, ce qui signifie que la sauvegarde représente une image de la base de données au moment où commence l'exécution de pg_dump. pg_dump ne bloque pas les autres opérations sur la base lorsqu'il fonctionne (sauf celles qui nécessitent un verrou exclusif, comme la plupart des formes d'**ALTER TABLE**.)

> **Important**
>
> Si la base de données utilise les OID (par exemple en tant que clés étrangères), il est impératif d'indiquer à pg_dump de sauvegarder aussi les OID. Pour cela, on utilise l'option `-o` sur la ligne de commande.

24.1.1. Restaurer la sauvegarde

Les fichiers texte créés par pg_dump peuvent être lus par le programme psql. La syntaxe générale d'une commande de restauration est

```
psql base_de_donnees < fichier_d_entree
```

où *`fichier_d_entree`* est le fichier en sortie de la commande pg_dump. La base de données *`base_de_donnees`* n'est pas créée par cette commande. Elle doit être créée à partir de `template0` avant d'exécuter psql (par exemple avec `createdb -T template0` *`base_de_donnees`*). psql propose des options similaires à celles de pg_dump pour indiquer le serveur de bases de données sur lequel se connecter et le nom d'utilisateur à utiliser. La page de référence de psql(1) donne plus d'informations.

Tous les utilisateurs possédant des objets ou ayant certains droits sur les objets de la base sauvegardée doivent exister préalablement à la restauration de la sauvegarde. S'ils n'existent pas, la restauration échoue pour la création des objets dont ils sont propriétaires ou sur lesquels ils ont des droits (quelque fois, cela est souhaitable mais ce n'est habituellement pas le cas).

Par défaut, le script psql continue de s'exécuter après la détection d'une erreur SQL. Vous pouvez exécuter psql avec la variable `ON_ERROR_STOP` configurée pour modifier ce comportement. psql quitte alors avec un code d'erreur 3 si une erreur SQL survient :

```
psql --set ON_ERROR_STOP=on base_de_données < infile
```

Dans tous les cas, une sauvegarde partiellement restaurée est obtenue. Si cela n'est pas souhaitable, il est possible d'indiquer que la sauvegarde complète doit être restaurée au cours d'une transaction unique. De ce fait, soit la restauration est validée dans son ensemble, soit elle est entièrement annulée. Ce mode est choisi en passant l'option `-1` ou `--single-transaction` en ligne de commande à psql. Dans ce mode, la plus petite erreur peut annuler une restauration en cours depuis plusieurs heures. Néanmoins, c'est probablement préférable au nettoyage manuel d'une base rendue complexe par une sauvegarde partiellement restaurée.

La capacité de pg_dump et psql à écrire et à lire dans des tubes permet de sauvegarder une base de données directement d'un serveur sur un autre. Par exemple :

```
pg_dump -h serveur1 base_de_donnees | psql -h serveur2 base_de_donnees
```

Important

Les fichiers de sauvegarde produits par pg_dump sont relatifs à `template0`. Cela signifie que chaque langage, procédure, etc. ajouté à `template1` est aussi sauvegardé par pg_dump. En conséquence, si une base `template1` modifiée est utilisée lors de la restauration, il faut créer la base vide à partir de `template0`, comme dans l'exemple plus haut.

Après la restauration d'une sauvegarde, il est conseillé d'exécuter ANALYZE(7) sur chaque base de données pour que l'optimiseur de requêtes dispose de statistiques utiles ; voir Section 23.1.3, « Maintenir les statistiques du planificateur » et Section 23.1.5, « Le démon auto-vacuum » pour plus d'informations. Pour plus de conseils sur le chargement efficace de grosses quantités de données dans PostgreSQL™, on peut se référer à la Section 14.4, « Remplir une base de données ».

24.1.2. Utilisation de pg_dumpall

pg_dump ne sauvegarde qu'une seule base à la fois, et ne sauvegarde pas les informations relatives aux rôles et *tablespaces* (parce que ceux-ci portent sur l'ensemble des bases du cluster, et non sur une base particulière). Pour permettre une sauvegarde aisée de tout le contenu d'un cluster, le programme pg_dumpall(1) est fourni. pg_dumpall sauvegarde toutes les bases de données d'un cluster (ensemble des bases d'une instance) PostgreSQL™ et préserve les données communes au cluster, telles que les rôles et tablespaces. L'utilisation basique de cette commande est :

```
pg_dumpall > fichier_de_sortie
```

Le fichier de sauvegarde résultant peut être restauré avec psql :

```
psql -f fichier_d_entree postgres
```

(N'importe quelle base de données peut être utilisée pour la connexion mais si le rechargement est exécuté sur un cluster vide, il est préférable d'utiliser `postgres`.) Il faut obligatoirement avoir le profil superutilisateur pour restaurer une sauvegarde faite avec pg_dumpall, afin de pouvoir restaurer les informations sur les rôles et les tablespaces. Si les *tablespaces* sont utilisés, il faut s'assurer que leurs chemins sauvegardés sont appropriés à la nouvelle installation.

pg_dumpall fonctionne en émettant des commandes pour recréer les rôles, les *tablespaces* et les bases vides, puis en invoquant pg_dump pour chaque base de données. Cela signifie que, bien que chaque base de données est cohérente en interne, les images des différentes bases de données peuvent ne pas être tout à fait synchronisées.

24.1.3. Gérer les grosses bases de données

Certains systèmes d'exploitation ont des limites sur la taille maximum des fichiers qui posent des problème lors de la création de gros fichiers de sauvegarde avec pg_dump. Heureusement, pg_dump peut écrire sur la sortie standard, donc vous pouvez utiliser les outils Unix standards pour contourner ce problème potentiel. Il existe plusieurs autres méthodes :

Compresser le fichier de sauvegarde. Tout programme de compression habituel est utilisable. Par exemple gzip :

```
pg_dump base_de_donnees | gzip > nom_fichier.gz
```

Pour restaurer :

```
gunzip -c nom_fichier.gz | psql base_de_donnees
```

ou

```
cat nom_fichier.gz | gunzip | psql base_de_donnees
```

Couper le fichier avec split. La commande **split** permet de découper le fichier en fichiers plus petits, de taille acceptable par le système de fichiers sous-jacent. Par exemple, pour faire des morceaux de 1 Mo :

```
pg_dump base_de_donnees | split -b 1m - nom_fichier
```

Pour restaurer :

```
cat nom_fichier* | psql base_de_donnees
```

Utilisation du format de sauvegarde personnalisé de pg_dump. Si PostgreSQL™ est installé sur un système où la bibliothèque de compression zlib est disponible, le format de sauvegarde personnalisé peut être utilisé pour compresser les données à la volée. Pour les bases de données volumineuses, cela produit un fichier de sauvegarde d'une taille comparable à celle du fichier produit par **gzip**, avec l'avantage supplémentaire de permettre de restaurer des tables sélectivement. La commande qui suit sauvegarde une base de données en utilisant ce format de sauvegarde :

```
pg_dump -Fc base_de_donnees > nom_fichier
```

Le format de sauvegarde personnalisé ne produit pas un script utilisable par psql. Ce script doit être restauré avec pg_restore, par exemple :

```
pg_restore -d nom_base nom_fichier
```

Voir les pages de référence de pg_dump(1) et pg_restore(1) pour plus de détails.

Pour les très grosses bases de données, il peut être nécessaire de combiner **split** avec une des deux autres approches.

24.2. Sauvegarde de niveau système de fichiers

Une autre stratégie de sauvegarde consiste à copier les fichiers utilisés par PostgreSQL™ pour le stockage des données. Dans la Section 17.2, « Créer un groupe de base de données », l'emplacement de ces fichiers est précisé. N'importe quelle méthode de sauvegarde peut être utilisée, par exemple :

```
tar -cf sauvegarde.tar /usr/local/pgsql/data
```

Cependant, deux restrictions rendent cette méthode peu pratique ou en tout cas inférieure à la méthode pg_dump.

1. Le serveur de base de données *doit* être arrêté pour obtenir une sauvegarde utilisable. Toutes les demi-mesures, comme la suppression des connexions, ne fonctionnent *pas* (principalement parce que **tar** et les outils similaires ne font pas une image atomique de l'état du système de fichiers, mais aussi à cause du tampon interne du serveur). Les informations concernant la façon d'arrêter le serveur PostgreSQL™ se trouvent dans la Section 17.5, « Arrêter le serveur ».

   Le serveur doit également être arrêté avant de restaurer les données.

2. Quiconque s'est aventuré dans les détails de l'organisation de la base de données peut être tenté de ne sauvegarder et restaurer que certaines tables ou bases de données particulières. Ce n'est *pas* utilisable sans les fichiers journaux de validation `pg_clog/*` qui contiennent l'état de la validation de chaque transaction. Un fichier de table n'est utilisable qu'avec cette information. Bien entendu, il est impossible de ne restaurer qu'une table et les données `pg_clog` associées car cela rendrait toutes les autres tables du serveur inutilisables. Les sauvegardes du système de fichiers fonctionnent, de ce fait, uniquement pour les sauvegardes et restaurations complètes d'un cluster de bases de données.

Une autre approche à la sauvegarde du système de fichiers consiste à réaliser une « image cohérente » (*consistent snapshot*) du répertoire des données. Il faut pour cela que le système de fichiers supporte cette fonctionnalité (et qu'il puisse lui être fait confiance). La procédure typique consiste à réaliser une « image gelée » (*frozen snapshot*) du volume contenant la base de données et ensuite de copier entièrement le répertoire de données (pas seulement quelques parties, voir plus haut) de l'image sur un périphérique de sauvegarde, puis de libérer l'image gelée. Ceci fonctionne même si le serveur de la base de données est en cours d'exécution. Néanmoins, une telle sauvegarde copie les fichiers de la base de données dans un état où le serveur n'est pas correctement arrêté ; du coup, au lancement du serveur à partir des données sauvegardées, PostgreSQL peut penser que le serveur s'est stoppé brutalement et rejouer les journaux WAL. Ce n'est pas un problème, mais il faut en être conscient (et s'assurer d'inclure les fichiers WAL dans la sauvegarde).

Si la base de données est répartie sur plusieurs systèmes de fichiers, il n'est peut-être pas possible d'obtenir des images gelées exactement simultanées de tous les disques. Si les fichiers de données et les journaux WAL sont sur des disques différents, par exemple, ou si les tablespaces sont sur des systèmes de fichiers différents, une sauvegarde par images n'est probablement pas utilisable parce que ces dernières *doivent* être simultanées. La documentation du système de fichiers doit être étudiée avec attention

avant de faire confiance à la technique d'images cohérentes dans de telles situations.

S'il n'est pas possible d'obtenir des images simultanées, il est toujours possible d'éteindre le serveur de bases de données suffisamment longtemps pour établir toutes les images gelées. Une autre possibilité est de faire une sauvegarde de la base en archivage continu (Section 24.3.2, « Réaliser une sauvegarde de base ») parce que ces sauvegardes ne sont pas sensibles aux modifications des fichiers pendant la sauvegarde. Cela n'impose d'activer l'archivage en continu que pendant la période de sauvegarde ; la restauration est faite en utilisant la restauration d'archive en ligne (Section 24.3.3, « Récupération à partir d'un archivage continu »).

Une autre option consiste à utiliser rsync pour réaliser une sauvegarde du système de fichiers. Ceci se fait tout d'abord en lançant rsync alors que le serveur de bases de données est en cours d'exécution, puis en arrêtant le serveur juste assez longtemps pour lancer rsync une deuxième fois. Le deuxième rsync est beaucoup plus rapide que le premier car il a relativement peu de données à transférer et le résultat final est cohérent, le serveur étant arrêté. Cette méthode permet de réaliser une sauvegarde du système de fichiers avec un arrêt minimal.

Une sauvegarde des fichiers de données va être généralement plus volumineuse qu'une sauvegarde SQL. (pg_dump ne sauvegarde pas le contenu des index, mais la commande pour les recréer). Cependant, une sauvegarde des fichiers de données peut être plus rapide.

24.3. Archivage continu et récupération d'un instantané (PITR)

PostgreSQL™ maintient en permanence des journaux WAL (*write ahead log*) dans le sous-répertoire `pg_xlog/` du répertoire de données du cluster. Ces journaux enregistrent chaque modification effectuée sur les fichiers de données des bases. Ils existent principalement pour se prémunir des suites d'un arrêt brutal : si le système s'arrête brutalement, la base de données peut être restaurée dans un état cohérent en « rejouant » les entrées des journaux enregistrées depuis le dernier point de vérification. Néanmoins, l'existence de ces journaux rend possible l'utilisation d'une troisième stratégie pour la sauvegarde des bases de données : la combinaison d'une sauvegarde de niveau système de fichiers avec la sauvegarde des fichiers WAL. Si la récupération est nécessaire, la sauvegarde des fichiers est restaurée, puis les fichiers WAL sauvegardés sont rejoués pour amener la sauvegarde jusqu'à la date actuelle. Cette approche est plus complexe à administrer que toutes les autres approches mais elle apporte des bénéfices significatifs :

- Il n'est pas nécessaire de disposer d'une sauvegarde des fichiers parfaitement cohérente comme point de départ. Toute incohérence dans la sauvegarde est corrigée par la ré-exécution des journaux (ceci n'est pas significativement différent de ce qu'il se passe lors d'une récupération après un arrêt brutal). La fonctionnalité d'image du système de fichiers n'est alors pas nécessaire, tar ou tout autre outil d'archivage est suffisant.
- Puisqu'une longue séquence de fichiers WAL peut être assemblée pour être rejouée, une sauvegarde continue est obtenue en continuant simplement à archiver les fichiers WAL. C'est particulièrement intéressant pour les grosses bases de données dont une sauvegarde complète fréquente est difficilement réalisable.
- Les entrées WAL ne doivent pas obligatoirement être rejouées intégralement. La ré-exécution peut être stoppée en tout point, tout en garantissant une image cohérente de la base de données telle qu'elle était à ce moment-là. Ainsi, cette technique autorise la *récupération d'un instantané* (PITR) : il est possible de restaurer l'état de la base de données telle qu'elle était en tout point dans le temps depuis la dernière sauvegarde de base.
- Si la série de fichiers WAL est fournie en continu à une autre machine chargée avec le même fichier de sauvegarde de base, on obtient un système « de reprise intermédiaire » (*warm standby*) : à tout moment, la deuxième machine peut être montée et disposer d'une copie quasi-complète de la base de données.

> **Note**
>
> pg_dump et pg_dumpall ne font pas de sauvegardes au niveau système de fichiers. Ce type de sauvegarde est qualifié de *logique* et ne contiennent pas suffisamment d'informations pour permettre le rejeu des journaux de transactions.

Tout comme la technique de sauvegarde standard du système de fichiers, cette méthode ne supporte que la restauration d'un cluster de bases de données complet, pas d'un sous-ensemble. De plus, un espace d'archivage important est requis : la sauvegarde de la base peut être volumineuse et un système très utilisé engendre un trafic WAL à archiver de plusieurs Mo. Malgré tout, c'est la technique de sauvegarde préférée dans de nombreuses situations où une haute fiabilité est requise.

Une récupération fructueuse à partir de l'archivage continu (aussi appelé « sauvegarde à chaud » par certains vendeurs de SGBD) nécessite une séquence ininterrompue de fichiers WAL archivés qui s'étend au moins jusqu'au point de départ de la sauvegarde. Pour commencer, il faut configurer et tester la procédure d'archivage des journaux WAL *avant* d'effectuer la première sauvegarde de base. C'est pourquoi la suite du document commence par présenter les mécanismes d'archivage des fichiers WAL.

24.3.1. Configurer l'archivage WAL

Au sens abstrait, un système PostgreSQL™ fonctionnel produit une séquence infinie d'enregistrements WAL. Le système divise physiquement cette séquence en *fichiers de segment* WAL de 16 Mo chacun (en général, mais cette taille peut être modifiée lors de la construction de PostgreSQL™). Les fichiers segment reçoivent des noms numériques pour refléter leur position dans la séquence abstraite des WAL. Lorsque le système n'utilise pas l'archivage des WAL, il ne crée que quelques fichiers segment, qu'il « recycle » en renommant les fichiers devenus inutiles. Un fichier segment dont le contenu précède le dernier point de vérification est supposé inutile et peut être recyclé.

Lors de l'archivage des données WAL, le contenu de chaque fichier segment doit être capturé dès qu'il est rempli pour sauvegarder les données ailleurs avant son recyclage. En fonction de l'application et du matériel disponible, « sauvegarder les données ailleurs » peut se faire de plusieurs façons : les fichiers segment peuvent être copiés dans un répertoire NFS monté sur une autre machine, être écrits sur une cartouche (après s'être assuré qu'il existe un moyen d'identifier le nom d'origine de chaque fichier) ou être groupés pour gravage sur un CD, ou toute autre chose. Pour fournir autant de flexibilité que possible à l'administrateur de la base de données, PostgreSQL™ essaie de ne faire aucune supposition sur la façon dont l'archivage est réalisé. À la place, PostgreSQL™ permet de préciser la commande shell à exécuter pour copier le fichier segment complet à l'endroit désiré. La commande peut être aussi simple qu'un `cp` ou impliquer un shell complexe -- c'est l'utilisateur qui décide.

Pour activer l'archivage des journaux de transaction, on positionne le paramètre de configuration wal_level à `archive` (ou à `hot_standby`), archive_mode à `on`, et on précise la commande shell à utiliser dans le paramètre archive_command de la configuration. En fait, ces paramètres seront toujours placés dans le fichier `postgresql.conf`. Dans cette chaîne, un `%p` est remplacé par le chemin absolu de l'archive alors qu'un `%f` n'est remplacé que par le nom du fichier. (Le nom du chemin est relatif au répertoire de travail du serveur, c'est-à-dire le répertoire des données du cluster.) `%%` est utilisé pour écrire le caractère `%` dans la commande. La commande la plus simple ressemble à :

```
archive_command = 'test ! -f /mnt/serveur/repertoire_archive/%f && cp -i %p
/mnt/serveur/repertoire_archive/%f </dev/null'  # Unix
archive_command = 'copy "%p" "C:\\serveur\\repertoire_archive\\%f"'  # Windows
```

qui copie les segments WAL archivables dans le répertoire `/mnt/serveur/repertoire_archive`. (Ceci est un exemple, pas une recommandation, et peut ne pas fonctionner sur toutes les plateformes.) Après le remplacement des paramètres `%p` et `%f`, la commande réellement exécutée peut ressembler à :

```
test ! -f /mnt/serveur/repertoire_archive/00000001000000A900000065 && cp
pg_xlog/00000001000000A900000065
/mnt/serveur/repertoire_archive/00000001000000A900000065
```

Une commande similaire est produite pour chaque nouveau fichier à archiver.

La commande d'archivage est exécutée sous l'identité de l'utilisateur propriétaire du serveur PostgreSQL™. La série de fichiers WAL en cours d'archivage contient absolument tout ce qui se trouve dans la base de données, il convient donc de s'assurer que les données archivées sont protégées des autres utilisateurs ; on peut, par exemple, archiver dans un répertoire sur lequel les droits de lecture ne sont positionnés ni pour le groupe ni pour le reste du monde.

Il est important que la commande d'archivage ne renvoie le code de sortie zéro que si, et seulement si, l'exécution a réussi. En obtenant un résultat zéro, PostgreSQL™ suppose que le fichier segment WAL a été archivé avec succès et qu'il peut le supprimer ou le recycler. Un statut différent de zéro indique à PostgreSQL™ que le fichier n'a pas été archivé ; il essaie alors périodiquement jusqu'à la réussite de l'archivage.

La commande d'archivage doit, en général, être conçue pour refuser d'écraser tout fichier archive qui existe déjà. C'est une fonctionnalité de sécurité importante pour préserver l'intégrité de l'archive dans le cas d'une erreur de l'administrateur (comme l'envoi de la sortie de deux serveurs différents dans le même répertoire d'archivage).

Il est conseillé de tester la commande d'archivage proposée pour s'assurer, qu'en effet, elle n'écrase pas un fichier existant, *et qu'elle retourne un statut différent de zéro dans ce cas*. La commande de l'exemple pour Unix ci-dessus s'en assure en ajoutant une étape de test séparé. Sur certaines plateformes Unix, **cp** dispose d'options comme `-i` pouvant être utilisées pour obtenir le même résultat, avec une commande moins longue, mais vous ne devez pas vous baser sur ces options sans tester que le bon code de statut est renvoyé. (En particulier, GNU **cp** renverra toujours le code de statut 0 quand l'option `-i` est utilisée, même si le fichier cible existe, ce qui n'est *pas* le comportement désiré.)

Lors de la conception de la configuration d'archivage, il faut considérer ce qui peut se produire si la commande d'archivage échoue de façon répétée, que ce soit parce qu'une intervention de l'opérateur s'avère nécessaire ou par manque d'espace dans le répertoire d'archivage. Ceci peut arriver, par exemple, lors de l'écriture sur une cartouche sans changeur automatique ; quand la cartouche est pleine, rien ne peut être archivé tant que la cassette n'est pas changée. Toute erreur ou requête à un opérateur humain doit être rapportée de façon appropriée pour que la situation puisse être résolue rapidement. Le répertoire `pg_xlog/` continue à se remplir de fichiers de segment WAL jusqu'à la résolution de la situation. (Si le système de fichiers contenant `pg_xlog/` se remplit, PostgreSQL™ s'arrête en mode PANIC. Aucune transaction validée n'est perdue mais la base de données est inaccessible tant que de l'espace n'a pas été libéré.)

La vitesse de la commande d'archivage n'est pas importante, tant qu'elle suit le rythme de génération des données WAL du ser-

veur. Les opérations normales continuent même si le processus d'archivage est un peu plus lent. Si l'archivage est significativement plus lent, alors la quantité de données qui peut être perdue croît. Cela signifie aussi que le répertoire `pg_xlog/` contient un grand nombre de fichiers segment non archivés, qui peuvent finir par dépasser l'espace disque disponible. Il est conseillé de surveiller le processus d'archivage pour s'assurer que tout fonctionne normalement.

Lors de l'écriture de la commande d'archivage, il faut garder à l'esprit que les noms de fichier à archiver peuvent contenir jusqu'à 64 caractères et être composés de toute combinaison de lettres ASCII, de chiffres et de points. Il n'est pas nécessaire de conserver le chemin relatif original (`%p`) mais il est nécessaire de se rappeler du nom du fichier (`%f`).

Bien que l'archivage WAL autorise à restaurer toute modification réalisée sur les données de la base, il ne restaure pas les modifications effectuées sur les fichiers de configuration (c'est-à-dire `postgresql.conf`, `pg_hba.conf` et `pg_ident.conf`) car ceux-ci sont édités manuellement et non au travers d'opérations SQL. Il est souhaitable de conserver les fichiers de configuration à un endroit où ils sont sauvegardés par les procédures standard de sauvegarde du système de fichiers. Voir la Section 18.2, « Emplacement des fichiers » pour savoir comment modifier l'emplacement des fichiers de configuration.

La commande d'archivage n'est appelée que sur les segments WAL complets. Du coup, si le serveur engendre peu de trafic WAL (ou qu'il y a des périodes de calme où le trafic WAL est léger), il peut y avoir un long délai entre la fin d'une transaction et son enregistrement sûr dans le stockage d'archive. Pour placer une limite sur l'ancienneté des données archivées, on configure archive_timeout qui force le serveur à changer de fichier segment WAL passé ce délai. Les fichiers archivés lors d'un tel forçage ont toujours la même taille que les fichiers complets. Il est donc déconseillé de configurer un délai `archive_timeout` trop court -- cela fait grossir anormalement le stockage. Une minute pour `archive_timeout` est généralement raisonnable.

De plus, le changement d'un segment peut être forcé manuellement avec `pg_switch_xlog`. Cela permet de s'assurer qu'une transaction tout juste terminée est archivée aussi vite que possible. D'autres fonctions utilitaires relatives à la gestion des WAL sont disponibles dans Tableau 9.56, « Fonctions de contrôle de la sauvegarde ».

Quand `wal_level` est configuré à `minimal`, certaines commandes SQL sont optimisées pour éviter la journalisation des transactions, de la façon décrite dans Section 14.4.7, « Désactiver l'archivage des journaux de transactions et la réplication en flux ». Si l'archivage ou la réplication en flux est activé lors de l'exécution d'une de ces instructions, les journaux de transaction ne contiennent pas suffisamment d'informations pour une récupération via les archives. (La récupération après un arrêt brutal n'est pas affectée.) Pour cette raison, `wal_level` ne peut être modifié qu'au lancement du serveur. Néanmoins, `archive_command` peut être modifié par rechargement du fichier de configuration. Pour arrêter temporairement l'archivage, on peut placer une chaîne vide (`''`) pour `archive_command`. Les journaux de transaction sont alors accumulés dans `pg_xlog/` jusqu'au rétablissement d'un paramètre `archive_command` fonctionnel.

24.3.2. Réaliser une sauvegarde de base

La procédure pour réaliser une sauvegarde de base est relativement simple :

1. S'assurer que l'archivage WAL est activé et fonctionnel.
2. Se connecter à la base de données en tant que superutilisateur et lancer la commande :

   ```
   SELECT pg_start_backup('label');
   ```

   où `label` est une chaîne utilisée pour identifier de façon unique l'opération de sauvegarde (une bonne pratique est d'utiliser le chemin complet du fichier de sauvegarde). `pg_start_backup` crée un fichier *de label de sauvegarde* nommé `backup_label` dans le répertoire du cluster. Ce fichier contient les informations de la sauvegarde, ceci incluant le moment du démarrage et le label.

   La base de données de connexion utilisée pour lancer cette commande n'a aucune importance. Le résultat de la fonction peut être ignoré, mais il faut gérer l'erreur éventuelle avant de continuer.

   Par défaut, `pg_start_backup` peut prendre beaucoup de temps pour arriver à son terme. Ceci est dû au fait qu'il réalise un point de vérification (*checkpoint*), et que les entrées/sorties pour l'établissement de ce point de vérification seront réparties sur une grande durée, par défaut la moitié de l'intervalle entre deux points de vérification (voir le paramètre de configuration checkpoint_completion_target). Habituellement, ce comportement est appréciable, car il minimise l'impact du traitement des requêtes. Pour commencer la sauvegarde aussi rapidement que possible, utiliser :

   ```
   SELECT pg_start_backup('label', true);
   ```

   Cela force l'exécution du point de vérification aussi rapidement que possible.
3. Effectuer la sauvegarde à l'aide de tout outil de sauvegarde du système de fichiers, tel tar ou cpio (mais ni pg_dump ni pg_dumpall). Il n'est ni nécessaire ni désirable de stopper les opérations normales de la base de données pour cela.
4. Se connecter à nouveau à la base de données en tant que superutilisateur et lancer la commande :

```
SELECT pg_stop_backup();
```

Cela met fin au processus de sauvegarde et réalise une bascule automatique vers le prochain segment WAL. Cette bascule est nécessaire pour permettre au dernier fichier de segment WAL écrit pendant la sauvegarde d'être immédiatement archivable.

5. Une fois que les fichiers des segments WAL utilisés lors de la sauvegarde sont archivés, c'est terminé. Le fichier identifié par le résultat de `pg_stop_backup` est le dernier segment nécessaire pour produire un jeu complet de fichiers de backup. Si `archive_mode` est activé, `pg_stop_backup` ne rend pas la main avant que le dernier segment n'ait été archivé. L'archivage de ces fichiers est automatique puisque `archive_command` est configuré. Dans la plupart des cas, c'est rapide, mais il est conseillé de surveiller le système d'archivage pour s'assurer qu'il n'y a pas de retard. Si le processus d'archivage a pris du retard en raison d'échecs de la commande d'archivage, il continuera d'essayer jusqu'à ce que l'archive réussisse et que le backup soit complet. Pour positionner une limite au temps d'exécution de `pg_stop_backup`, il faut positionner `statement_timeout` à une valeur appropriée.

Certains outils de sauvegarde de fichiers émettent des messages d'avertissement ou d'erreur si les fichiers qu'ils essaient de copier sont modifiés au cours de la copie. Cette situation, normale lors de la sauvegarde d'une base active, ne doit pas être considérée comme une erreur ; il suffit de s'assurer que ces messages puissent être distingués des autres messages. Certaines versions de rsync, par exemple, renvoient un code de sortie distinct en cas de « disparition de fichiers source ». Il est possible d'écrire un script qui considère ce code de sortie comme normal.

De plus, certaines versions de GNU tar retournent un code d'erreur qu'on peut confondre avec une erreur fatale si le fichier a été tronqué pendant sa copie par tar. Heureusement, les versions 1.16 et suivantes de GNU tar retournent `1` si le fichier a été modifié pendant la sauvegarde et `2` pour les autres erreurs.

Il n'est pas utile d'accorder de l'importance au temps passé entre `pg_start_backup` et le début réel de la sauvegarde, pas plus qu'entre la fin de la sauvegarde et `pg_stop_backup` ; un délai de quelques minutes ne pose pas de problème. (Néanmoins, si le serveur est normalement utilisé alors que `full_page_writes` est désactivé, une perte de performances entre `pg_start_backup` et `pg_stop_backup` peut être constatée car l'activation du paramètre `full_page_writes` est forcée lors du mode de sauvegarde.) Il convient toutefois de s'assurer que ces étapes sont effectuées séquentiellement, sans chevauchement. Dans le cas contraire, la sauvegarde est invalidée.

La sauvegarde doit inclure tous les fichiers du répertoire du groupe de bases de données (`/usr/local/pgsql/data`, par exemple). Si des *tablespaces* qui ne se trouvent pas dans ce répertoire sont utilisés, il ne faut pas oublier de les inclure (et s'assurer également que la sauvegarde archive les liens symboliques comme des liens, sans quoi la restauration va corrompre les *tablespaces*).

Néanmoins, les fichiers du sous-répertoire `pg_xlog/`, contenu dans le répertoire du cluster, peuvent être omis. Ce léger ajustement permet de réduire le risque d'erreurs lors de la restauration. C'est facile à réaliser si `pg_xlog/` est un lien symbolique vers quelque endroit extérieur au répertoire du cluster, ce qui est toutefois une configuration courante, pour des raisons de performance.

La sauvegarde n'est utilisable que si les fichiers de segment WAL engendrés pendant ou après cette sauvegarde sont préservés. Pour faciliter cela, la fonction `pg_stop_backup` crée un *fichier d'historique de la sauvegarde* immédiatement stocké dans la zone d'archivage des WAL. Ce fichier est nommé d'après le nom du premier fichier segment WAL nécessaire à l'utilisation de la sauvegarde. Ainsi, si le fichier WAL de démarrage est `000000010000012340000 55CD`, le nom du fichier d'historique ressemble à `000000010000012340000 55CD.007C9330.backup` (le deuxième nombre dans le nom de ce fichier contient la position exacte à l'intérieur du fichier WAL et peut en général être ignoré). Une fois que la sauvegarde du système de fichiers et des segments WAL utilisés pendant celle-ci (comme précisé dans le fichier d'historique des sauvegardes) est archivée de façon sûre, tous les segments WAL archivés de noms numériquement plus petits ne sont plus nécessaires à la récupération de la sauvegarde du système de fichiers et peuvent être supprimés. Toutefois, il est préférable de conserver plusieurs jeux de sauvegarde pour être absolument certain de pouvoir récupérer les données.

Le fichier d'historique de la sauvegarde est un simple fichier texte. Il contient le label passé à `pg_start_backup`, l'heure et les segments WAL de début et de fin de la sauvegarde. Si le label est utilisé pour identifier l'endroit où le fichier de sauvegarde associé est conservé, alors le fichier d'historique archivé est suffisant pour savoir quel fichier de sauvegarde restaurer, en cas de besoin.

Puisqu'il faut conserver tous les fichiers WAL archivés depuis la dernière sauvegarde de base, l'intervalle entre les sauvegardes de base est habituellement choisi en fonction de l'espace de stockage qu'on accepte de consommer en fichiers d'archives WAL. Il faut également considérer le temps à dépenser pour la récupération, si celle-ci s'avère nécessaire -- le système doit rejouer tous les segments WAL et ceci peut prendre beaucoup de temps si la dernière sauvegarde de base est ancienne.

La fonction `pg_start_backup` crée un fichier nommé `backup_label` dans le répertoire du cluster de bases de données. Ce fichier est ensuite supprimé par `pg_stop_backup`. Ce fichier est archivé dans le fichier de sauvegarde. Le fichier de label de la sauvegarde inclut la chaîne de label passée à `pg_start_backup`, l'heure à laquelle `pg_start_backup` a été exécuté et le nom du fichier WAL initial. En cas de confusion, il est ainsi possible de regarder dans le fichier sauvegarde et de déterminer avec précision de quelle session de sauvegarde provient ce fichier.

Il est aussi possible de faire une sauvegarde alors que le serveur est arrêté. Dans ce cas, `pg_start_backup` et

pg_stop_backup ne peuvent pas être utilisés. L'utilisateur doit alors se débrouiller pour identifier les fichiers de sauvegarde et déterminer jusqu'où remonter avec les fichiers WAL associés. Il est généralement préférable de suivre la procédure d'archivage continu décrite ci-dessus.

24.3.3. Récupération à partir d'un archivage continu

Le pire est arrivé et il faut maintenant repartir d'une sauvegarde. Voici la procédure :

1. Arrêter le serveur s'il est en cours d'exécution.
2. Si la place nécessaire est disponible, copier le répertoire complet de données du cluster et tous les *tablespaces* dans un emplacement temporaire en prévision d'un éventuel besoin ultérieur. Cette précaution nécessite qu'un espace suffisant sur le système soit disponible pour contenir deux copies de la base de données existante. S'il n'y a pas assez de place disponible, il faut au minimum copier le contenu du sous-répertoire pg_xlog du répertoire des données du cluster car il peut contenir des journaux qui n'ont pas été archivés avant l'arrêt du serveur.
3. Effacer tous les fichiers et sous-répertoires existant sous le répertoire des données du cluster et sous les répertoires racines des *tablespaces*.
4. Restaurer les fichiers de la base de données à partir de la sauvegarde des fichiers. Il faut veiller à ce qu'ils soient restaurés avec le bon propriétaire (l'utilisateur système de la base de données, et non pas root !) et avec les bons droits. Si des *tablespaces* sont utilisés, il faut s'assurer que les liens symboliques dans pg_tblspc/ ont été correctement restaurés.
5. Supprimer tout fichier présent dans pg_xlog/ ; ils proviennent de la sauvegarde et sont du coup probablement obsolètes. Si pg_xlog/ n'a pas été archivé, il suffit de recréer ce répertoire en faisant attention à le créer en tant que lien symbolique, si c'était le cas auparavant.
6. Si des fichiers de segment WAL non archivés ont été sauvegardés dans l'étape 2, les copier dans pg_xlog/. Il est préférable de les copier plutôt que de les déplacer afin qu'une version non modifiée de ces fichiers soit toujours disponible si un problème survient et qu'il faille recommencer.
7. Créer un fichier de commandes de récupération recovery.conf dans le répertoire des données du cluster (voir Chapitre 26, Configuration de la récupération). Il peut, de plus, être judicieux de modifier temporairement le fichier pg_hba.conf pour empêcher les utilisateurs ordinaires de se connecter tant qu'il n'est pas certain que la récupération a réussi.
8. Démarrer le serveur. Le serveur se trouve alors en mode récupération et commence la lecture des fichiers WAL archivés dont il a besoin. Si la récupération se termine sur une erreur externe, le serveur peut tout simplement être relancé. Il continue alors la récupération. À la fin du processus de récupération, le serveur renomme recovery.conf en recovery.done (pour éviter de retourner accidentellement en mode de récupération), puis passe en mode de fonctionnement normal.
9. Inspecter le contenu de la base de données pour s'assurer que la récupération a bien fonctionné. Dans le cas contraire, retourner à l'étape 1. Si tout va bien, le fichier pg_hba.conf peut-être restauré pour autoriser les utilisateurs à se reconnecter.

Le point clé de tout ceci est l'écriture d'un fichier de configuration de récupération qui décrit comment et jusqu'où récupérer. Le fichier recovery.conf.sample (normalement présent dans le répertoire d'installation share/) peut être utilisé comme prototype. La seule chose qu'il faut absolument préciser dans recovery.conf, c'est restore_command qui indique à PostgreSQL™ comment récupérer les fichiers de segment WAL archivés. À l'instar d'archive_command, c'est une chaîne de commande shell. Elle peut contenir %f, qui est remplacé par le nom du journal souhaité, et %p, qui est remplacé par le chemin du répertoire où copier le journal. (Le nom du chemin est relatif au répertoire de travail du serveur, c'est-à-dire le répertoire des données du cluster.) Pour écrire le caractère % dans la commande, on utilise %%. La commande la plus simple ressemble à :

```
restore_command = 'cp /mnt/serveur/répertoire_archive/%f %p'
```

qui copie les segments WAL précédemment archivés à partir du répertoire /mnt/serveur/répertoire_archive. Il est toujours possible d'utiliser une commande plus compliquée, voire même un script shell qui demande à l'utilisateur de monter la cassette appropriée.

Il est important que la commande retourne un code de sortie différent de zéro en cas d'échec. Des fichiers absents de l'archive *seront* demandés à la commande ; elle doit renvoyer autre chose que zéro dans ce cas. Ce n'est pas une condition d'erreur. Tous les fichiers demandés ne seront pas des segmets WAL; vous pouvez aussi vous attendre à des demandes de fichiers suffixés par .backup or .history. Il faut également garder à l'esprit que le nom de base du chemin %p diffère de %f ; il ne sont pas interchangeables.

Les segments WAL qui ne se trouvent pas dans l'archive sont recherchés dans pg_xlog/ ; cela autorise l'utilisation de segments récents non archivés. Néanmoins, les segments disponibles dans l'archive sont utilisés de préférence aux fichiers contenus dans pg_xlog/. Le système ne surcharge pas le contenu de pg_xlog/ lors de la récupération des fichiers archivés.

Normalement, la récupération traite tous les segments WAL disponibles, restaurant du coup la base de données à l'instant présent (ou aussi proche que possible, en fonction des segments WAL disponibles). Une récupération normale se finit avec un message

« fichier non trouvé », le texte exact du message d'erreur dépendant du choix de `restore_command`. Un message d'erreur au début de la récupération peut également apparaître concernant un fichier nommé dont le nom ressemble à `00000001.history`. Ceci est aussi normal et n'indique par un problème dans les situations de récupération habituelles. Voir Section 24.3.4, « Lignes temporelles (*Timelines*) » pour plus d'informations.

Pour récupérer à un moment précis (avant que le DBA junior n'ait supprimé la table principale), il suffit d'indiquer le point d'arrêt requis dans `recovery.conf`. Le point d'arrêt, aussi nommé « recovery target » (cible de récupération), peut être précisé par une combinaison date/heure ou par le dernier identifiant de transaction. Actuellement, seule l'option date/heure est vraiment utilisable car il n'existe pas d'outils permettant d'identifier avec précision l'identifiant de transaction à utiliser.

> **Note**
>
> Le point d'arrêt doit être postérieur à la fin de la sauvegarde de la base (le moment où `pg_stop_backup` se termine). Une sauvegarde ne peut pas être utilisée pour repartir d'un instant où elle était encore en cours (pour ce faire, il faut récupérer la sauvegarde précédente et rejouer à partir de là).

Si la récupération fait face à une corruption des données WAL, elle se termine à ce point et le serveur ne démarre pas. Dans un tel cas, le processus de récupération peut alors être ré-exécuté à partir du début en précisant une « cible de récupération » antérieure au point de récupération pour permettre à cette dernière de se terminer correctement. Si la récupération échoue pour une raison externe (arrêt brutal du système ou archive WAL devenue inaccessible), la récupération peut être simplement relancée. Elle redémarre alors quasiment là où elle a échoué. Le redémarrage de la restauration fonctionne comme les points de contrôle du déroulement normal : le serveur force une écriture régulière de son état sur les disques et actualise alors le fichier `pg_control` pour indiquer que les données WAL déjà traitées n'ont plus à être parcourues.

24.3.4. Lignes temporelles (*Timelines*)

La possibilité de restaurer la base de données à partir d'un instantané crée une complexité digne des histoires de science-fiction traitant du voyage dans le temps et des univers parallèles.

Par exemple, dans l'historique original de la base de données, supposez qu'une table critique ait été supprimée à 17h15 mardi soir, mais personne n'a réalisé cette erreur avant mercredi midi. Sans stress, la sauvegarde est récupérée et restaurée dans l'état où elle se trouvait à 17h14 mardi soir. La base est fonctionnelle. Dans *cette* histoire de l'univers de la base de données, la table n'a jamais été supprimée. Or, l'utilisateur réalise peu après que ce n'était pas une si grande idée et veut revenir à un quelconque moment du mercredi matin. Cela n'est pas possible, si, alors que la base de données est de nouveau fonctionnelle, elle réutilise certaines séquences de fichiers WAL qui permettent de retourner à ce point. Il est donc nécessaire de pouvoir distinguer les séries d'enregistrements WAL engendrées après la récupération de l'instantané de celles issues de l'historique originel de la base.

Pour gérer ces difficultés, PostgreSQL™ inclut la notion de *lignes temporelles* (ou *timelines*). Quand une récupération d'archive est terminée, une nouvelle ligne temporelle est créée pour identifier la série d'enregistrements WAL produits après cette restauration. Le numéro d'identifiant de la timeline est inclus dans le nom des fichiers de segment WAL. De ce fait, une nouvelle timeline ne réécrit pas sur les données engendrées par des timelines précédentes. En fait, il est possible d'archiver plusieurs timelines différentes. Bien que cela semble être une fonctionnalité inutile, cela peut parfois sauver des vies. Dans une situation où l'instantané à récupérer n'est pas connu avec certitude, il va falloir tester les récupérations de différents instantanés jusqu'à trouver le meilleur. Sans les timelines, ce processus engendre vite un bazar ingérable. Avec les timelines, il est possible de récupérer *n'importe quel* état précédent, même les états de branches temporelles abandonnées.

Chaque fois qu'une nouvelle timeline est créée, PostgreSQL™ crée un fichier d'« historique des timelines » qui indique à quelle timeline il est attaché, et depuis quand. Ces fichiers d'historique sont nécessaires pour permettre au système de choisir les bons fichiers de segment WAL lors de la récupération à partir d'une archive qui contient plusieurs timelines. Ils sont donc archivés comme tout fichier de segment WAL. Puisque ce sont de simples fichiers texte, il est peu coûteux et même judicieux de les conserver indéfiniment (contrairement aux fichiers de segment, volumineux). Il est possible d'ajouter des commentaires au fichier d'historique expliquant comment et pourquoi cette timeline a été créée. De tels commentaires s'avèrent précieux lorsque l'expérimentation conduit à de nombreuses timelines.

Par défaut, la récupération s'effectue sur la timeline en vigueur au cours de la la sauvegarde. Si l'on souhaite effectuer la récupération dans une timeline fille (c'est-à-dire retourner à un état enregistré après une tentative de récupération), il faut préciser l'identifiant de la timeline dans `recovery.conf`. Il n'est pas possible de récupérer dans des timelines antérieures à la sauvegarde.

24.3.5. Conseils et exemples

Quelques conseils de configuration de l'archivage continue sont donnés ici.

24.3.5.1. Configuration de la récupération

Il est possible d'utiliser les capacités de sauvegarde de PostgreSQL™ pour produire des sauvegardes autonomes à chaud. Ce sont des sauvegardes qui ne peuvent pas être utilisées pour la récupération à un instant donné, mais ce sont des sauvegardes qui sont typiquement plus rapide à obtenir et à restaurer que ceux issus de pg_dump. (Elles sont aussi bien plus volumineuses qu'un export pg_dump, il se peut donc que l'avantage de rapidité soit négatif.)

En vue d'effectuer des sauvegardes à chaud autonomes, on positionne `wal_level` à `archive` (ou `hot_standby`), `archive_mode` à `on`, et on configure `archive_command` de telle sorte que l'archivage ne soit réalisé que lorsqu'un *fichier de bascule* existe. Par exemple :

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f
/var/lib/pgsql/archive/%f && cp %p /var/lib/pgsql/archive/%f)'
```

Cette commande réalise l'archivage dès lors que `/var/lib/pgsql/backup_in_progress` existe. Dans le cas contraire, elle renvoie silencieusement le code de statut zéro (permettant à PostgreSQL™ de recycler le journal de transactions non désiré).

Avec cette préparation, une sauvegarde peut être prise en utilisant un script comme celui-ci :

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

Le fichier de bascule, `/var/lib/pgsql/backup_in_progress`, est créé en premier, activant l'archivage des journaux de transactions pleins. Après la sauvegarde, le fichier de bascule est supprimé. Les journaux de transaction archivés sont ensuite ajoutés à la sauvegarde pour que la sauvegarde de base et les journaux requis fassent partie du même fichier tar. Rappelez vous d'ajouter de la gestion d'erreur à vos scripts.

24.3.5.2. Compression des journaux archivés

Si la taille du stockage des archives est un problème, utilisez pg_compresslog, *http://pglesslog.projects.postgresql.org*, afin d'enlever les inutiles full_page_writes et les espaces de fin des journaux de transactions. Vous pouvez utiliser gzip pour compresser encore davantage le résultat de pg_compresslog :

```
          archive_command = 'pg_compresslog %p - | gzip > /var/lib/pgsql/archive/%f'
```

Vous aurez alors besoin d'utiliser gunzip et pg_decompresslog pendant la récupération :

```
          restore_command = 'gunzip < /mnt/server/archivedir/%f | pg_decompresslog -
%p'
```

24.3.5.3. Scripts `archive_command`

Nombreux sont ceux qui choisissent d'utiliser des scripts pour définir leur `archive_command`, de sorte que leur `postgresql.conf` semble très simple :

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Utiliser un script séparé est conseillé à chaque fois qu'il est envisagé d'utiliser plusieurs commandes pour le processus d'archivage. Ainsi toute la complexité est gérée dans le script qui peut être écrit dans un langage de scripts populaires comme bash ou perl.

Quelques exemples de besoins résolus dans un script :

- copier des données vers un stockage distant ;
- copier les journaux de transaction en groupe pour qu'ils soient transférés toutes les trois heures plutôt qu'un à la fois ;
- s'interfacer avec d'autres outils de sauvegarde et de récupération ;
- s'interfacer avec un outil de surveillance pour y renvoyer les erreurs.

Astuce

Lors de l'utilisation du script `archive_command`, il est préférable d'activer logging_collector. Tout message envoyé dans `stderr` à partir du script apparaîtra dans les traces du serveur, permettant un diagnostic plus aisé de

configurations complexes en cas de problème.

24.3.6. Restrictions

Au moment où ces lignes sont écrites, plusieurs limitations de la technique d'achivage continu sont connues. Elles seront probablement corrigées dans une prochaine version :

- Les opérations sur les index de hachage ne sont pas tracées dans les journaux de transactions. Ces index ne sont donc pas actualisés lorsque la sauvegarde est rejouée. Cela signifie que toute nouvelle insertion sera ignorée par l'index, que les lignes mises à jour sembleront disparaître et que les lignes supprimées auront toujours leur pointeurs. En d'autres termes, si vous modifier une table disposant d'un index hash, alors vous obtiendrez des résultats erronés sur un serveur en attente. Lorsque la restauration se termine, il est recommandé de lancer manuellement la commande REINDEX(7) sur chacun des index à la fin de la récupération.
- Si une commande CREATE DATABASE(7) est exécutée alors qu'une sauvegarde est en cours, et que la base de données modèle utilisée par l'instruction **CREATE DATABASE** est à son tour modifiée pendant la sauvegarde, il est possible que la récupération propage ces modifications dans la base de données créée. Pour éviter ce risque, il est préférable de ne pas modifier les bases de données modèle lors d'une sauvegarde de base.
- Les commandes CREATE TABLESPACE(7) sont tracées dans les WAL avec le chemin absolu et sont donc rejouées en tant que créations de *tablespace* suivant le même chemin absolu. Cela n'est pas forcément souhaitable si le journal est rejouée sur une autre machine. De plus, cela peut s'avérer dangereux même lorsque le journal est rejoué sur la même machine, mais dans un répertoire différent : la ré-exécution surcharge toujours le contenu du *tablespace* original. Pour éviter de tels problèmes, la meilleure solution consiste à effectuer une nouvelle sauvegarde de la base après la création ou la suppression de *tablespace*.

Il faut de plus garder à l'esprit que le format actuel des WAL est extrêmement volumineux car il inclut de nombreuses images des pages disques. Ces images de page sont conçues pour supporter la récupération après un arrêt brutal, puisqu'il peut être nécessaire de corriger des pages disque partiellement écrites. En fonction du matériel et des logiciels composant le système, le risque d'écriture partielle peut être suffisamment faible pour être ignoré, auquel cas le volume total des traces archivées peut être considérablement réduit par la désactivation des images de page à l'aide du paramètre full_page_writes (lire les notes et avertissements dans Chapitre 29, Fiabilité et journaux de transaction avant de le faire). Désactiver les images de page n'empêche pas l'utilisation des traces pour les opérations PITR. Un piste éventuelle de développement futur consiste à compresser les données des WAL archivés en supprimant les copies inutiles de pages même si `full_page_writes` est actif. Entre temps, les administrateurs peuvent souhaiter réduire le nombre d'images de pages inclus dans WAL en augmentant autant que possible les paramètres d'intervalle entre les points de vérification.

24.4. Migration entre versions

Cette section discute de la façon de migrer les données de la base vers une version plus récente de PostgreSQL™. Le sujet de cette section ne concerne pas la procédure d'installation *per se* du logiciel ; ces détails sont dans le Chapitre 15, Procédure d'installation de PostgreSQL™ du code source.

Les versions majeures de PostgreSQL™ sont représentées par les deux premiers nombres dans le numéro de version, par exemple 8.4. Les versions mineures sont représentées par le troisième groupe. Par exemple, la 8.4.2 est la deuxième version mineure de la version majeure 8.4. Les versions mineures ne modifient jamais le format de stockage interne et sont toujours compatibles avec les versions mineures antérieures et ultérieures de la même version majeure. Autrement dit, la version 8.4.2 est compatible avec les versions 8.4, 8.4.1 et 8.4.6. Pour mettre à jour entre des versions compatibles, vous devez simplement remplacer les exécutables quand le serveur est arrêté, puis redémarré le serveur. Le répertoire des données reste inchangé -- les mises à jour de versions mineures sont aussi simples que ça.

Pour les versions *majeures* de PostgreSQL™, le format de stockage interne des données est sujet à modification, ceci compliquant les mises à jour. La méthode traditionnelle pour déplacer les données vers une nouvelle version majeure est de sauvegarder puis de recharger la base de données. Sinon, des possibilités moins bien testées sont disponibles, comme indiqué ci-dessous.

Les nouvelles versions majeures introduisent aussi typiquement quelques incompatibilités visibles par les utilisateurs, donc des modifications des applications pourraient se révéler nécessaires. Les utilisateurs consciencieux voudront tester les applications sur la nouvelle version avant de basculer complètement ; du coup, il est souvent préférable de configurer des installations parallèles des ancienne et nouvelle version. Lors du test d'une mise à jour majeure de PostgreSQL™, réfléchissez aux catégories suivantes de modifications possibles :

Administration
: Les capacités disponibles pour la suveillance et le contrôle du serveur par les administrateurs s'améliorent de version en version.

SQL
: Cela inclut typiquement les nouvelles commandes SQL et non pas les modifications de comportement, sauf mention explicite dans les notes de versions.

API des bibliothèques
: Habituellement, les bibliothèques comme libpq ne font qu'ajouter de nouvelles fonctionnalités. Sauf, encore une fois, mention explicite dans les notes de versions.

Catalogue système
: Les modifications du catalogue système n'affectent habituellement que les outils d'administration de bases de données.

API langage C du serveur
: Ceci concerne les modifications dans l'API des fonctions du serveur, qui est écrit en langage C. De telles modifications affectent le code qui référencent les fonctions internes du serveur.

24.4.1. Migrer des données pg_dump

Pour sauvegarder des données à partir d'une version majeure de PostgreSQL™ dans le but de les recharger dans une autre, vous devez utiliser pg_dump ; les méthodes au niveau système de fichiers ne fonctionneront pas. Il existe des vérifications mises en place pour vous empêcher d'utiliser un répertoire des données avec une version incompatible de PostgreSQL™, donc aucun dommage ne pourra être fait sur un répertoire de données si un utilisateur essaie d'y lancer un serveur d'une autre version.)

Il est recommandé d'utiliser les programmes pg_dump et pg_dumpall issus de la nouvelle version de PostgreSQL™. Ceci permet de tirer parti des améliorations de ces programmes. Les versions actuelles des programmes de sauvegarde peuvent lire des données sur des serveurs de versions anciennes, jusqu'à la 7.0.

La durée d'indisponibilité est minimisée par l'installation de nouveau serveur dans un répertoire différent et par le lancement en parallèle des deux serveurs (ancien et nouveau), sur des ports différents. On peut alors utiliser une commande telle que :

```
pg_dumpall -p 5432 | psql -d postgres -p 6543
```

pour transférer les données. On peut aussi utiliser un fichier intermédiaire. L'ancien serveur peut alors être éteint, et le nouveau démarré sur le port utilisé par l'ancien. Il est impératif que la base de données ne soit pas modifiée pendant l'exécution de pg_dumpall. Ces modifications seraient, sans cela, perdues. Le Chapitre 19, Authentification du client informe sur la façon d'interdire l'accès.

Si n'est pas souhaité, ou pas possible, de lancer les deux serveurs en parallèle, il faut réaliser l'étape de sauvegarde avant d'installer la nouvelle version, éteindre l'ancien serveur, déplacer l'ancienne version vers un autre endroit, installer la nouvelle, la démarrer et enfin restaurer les données. Par exemple :

```
pg_dumpall > sauvegarde.sql
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
# Renommer tous les répertoires des tablespaces
cd ~/postgresql-9.0.18
gmake install
initdb -D /usr/local/pgsql/data
postgres -D /usr/local/pgsql/data
psql -f sauvegarde.sql postgres
```

Toutes les méthodes pour arrêter et démarrer les serveurs, ainsi que d'autres détails sont présentés dans le Chapitre 17, Configuration du serveur et mise en place. Les instructions d'installation donnent des conseils sur les endroits stratégiques pour réaliser ces opérations.

> **Note**
>
> Lorsque « l'ancienne installation est déplacée », il se peut qu'elle ne soit plus correctement utilisable. En effet, certains exécutables contiennent des chemins absolus vers les différents programmes et fichiers de données installés. Ce n'est habituellement pas un problème insurmontable, mais pour utiliser deux installations en parallèle pendant un moment, il faut leur affecter des répertoires d'installation différents au moment de la construction. (Ce problème est rectifié pour PostgreSQL™ 8.0 et suivantes tant que tous les sous-répertoires contenant des fichiers installées sont déplacés ensemble ; par exemple si `/usr/local/postgres/bin/` va dans `/usr/local/postgres.old/bin/`, alors `/usr/local/postgres/share/` doit aller dans `/usr/local/postgres.old/share/`. Dans les versions antérieures à la 8.0, déplacer une installation comme ceci n'aurait pas fonctionné.)

24.4.2. Other data migration methods

Le programme `contrib` program pg_upgrade permet la migration en ligne d'une installation à partir d'une version majeure de PostgreSQL™ vers la suivante. Gardez en tête que cette méthode ne fournit aucun moyen pour exécuter les ancienne et nouvelle version en parallèle. De plus, pg_upgrade est beaucoup moins testé que pg_dump, donc avoir une sauvegarde à jour est très fortement recommandé au cas où l'opération se terminerait mal.

Il est aussi possible d'utiliser certaines méthodes de réplication, comme Slony™, pour créer un deuxième serveur avec la version mise à jour de PostgreSQL™. L'esclave en question peut être sur le même ordinateur ou sur un ordinateur différent. Une fois qu'il est synchronisé avec le serveur maître (utilisant toujours l'ancienne version de PostgreSQL™), vous pouvez basculer le maître et arrêter l'ancienne instance. Ce type de bascule met généralement quelques secondes seulement pour s'exécuter.

Chapitre 25. Haute disponibilité, répartition de charge et réplication

Des serveurs de bases de données peuvent travailler ensemble pour permettre à un serveur secondaire de prendre rapidement la main si le serveur principal échoue (haute disponibilité, ou *high availability*), ou pour permettre à plusieurs serveurs de servir les mêmes données (répartition de charge, ou *load balancing*). Idéalement, les serveurs de bases de données peuvent travailler ensemble sans jointure.

Il est aisé de faire coopérer des serveurs web qui traitent des pages web statiques en répartissant la charge des requêtes web sur plusieurs machines. Dans les faits, les serveurs de bases de données en lecture seule peuvent également coopérer facilement. Malheureusement, la plupart des serveurs de bases de données traitent des requêtes de lecture/écriture et, de ce fait, collaborent plus difficilement. En effet, alors qu'il suffit de placer une seule fois les données en lecture seule sur chaque serveur, une écriture sur n'importe quel serveur doit, elle, être propagée à tous les serveurs afin que les lectures suivantes sur ces serveurs renvoient des résultats cohérents.

Ce problème de synchronisation représente la difficulté fondamentale à la collaboration entre serveurs. Comme la solution au problème de synchronisation n'est pas unique pour tous les cas pratiques, plusieurs solutions co-existent. Chacune répond de façon différente et minimise cet impact au regard d'une charge spécifique.

Certaines solutions gèrent la synchronisation en autorisant les modifications des données sur un seul serveur. Les serveurs qui peuvent modifier les données sont appelés serveur en lecture/écriture, *maître* ou serveur *primaire*. Les serveurs qui suivent les modifications du maître sont appelés *standby*, ou serveurs *esclaves*. Un serveur en standby auquel on ne peut pas se connecter tant qu'il n'a pas été promu en serveur maître est appelé un serveur en *warm standby*, et un qui peut accepter des connections et répondre à des requêtes en lecture seule est appelé un serveur en *hot standby*.

Certaines solutions sont synchrones, ce qui signifie qu'une transaction de modification de données n'est pas considérée valide tant que tous les serveurs n'ont pas validé la transaction. Ceci garantit qu'un *failover* ne perd pas de données et que tous les serveurs en répartition de charge retournent des résultats cohérents, quel que soit le serveur interrogé. Au contraire, les solutions asynchrones autorisent un délai entre la validation et sa propagation aux autres serveurs. Cette solution implique une éventuelle perte de transactions lors de la bascule sur un serveur de sauvegarde, ou l'envoi de données obsolètes par les serveurs à charge répartie. La communication asynchrone est utilisée lorsque la version synchrone est trop lente.

Les solutions peuvent aussi être catégorisées par leur granularité. Certaines ne gèrent que la totalité d'un serveur de bases alors que d'autres autorisent un contrôle par table ou par base.

Il importe de considérer les performances dans tout choix. Il y a généralement un compromis à trouver entre les fonctionnalités et les performances. Par exemple, une solution complètement synchrone sur un réseau lent peut diviser les performances par plus de deux, alors qu'une solution asynchrone peut n'avoir qu'un impact minimal sur les performances.

Le reste de cette section souligne différentes solutions de *failover*, de réplication et de répartition de charge. Un *glossaire* est aussi disponible.

25.1. Comparaison de différentes solutions

Failover sur disque partagé

Le *failover* (ou bascule sur incident) sur disque partagé élimine la surcharge de synchronisation par l'existence d'une seule copie de la base de données. Il utilise un seul ensemble de disques partagé par plusieurs serveurs. Si le serveur principal échoue, le serveur en attente est capable de monter et démarrer la base comme s'il récupérait d'un arrêt brutal. Cela permet un *failover* rapide sans perte de données.

La fonctionnalité de matériel partagé est commune aux périphériques de stockage en réseau. Il est également possible d'utiliser un système de fichiers réseau bien qu'il faille porter une grande attention au système de fichiers pour s'assurer qu'il a un comportement POSIX complet (voir Section 17.2.1, « Systèmes de fichiers réseaux »). Cette méthode comporte une limitation significative : si les disques ont un problème ou sont corrompus, le serveur primaire et le serveur en attente sont tous les deux non fonctionnels. Un autre problème est que le serveur en attente ne devra jamais accéder au stockage partagé tant que le serveur principal est en cours d'exécution.

Réplication de système de fichiers (périphérique bloc)

Il est aussi possible d'utiliser cette fonctionnalité d'une autre façon avec une réplication du système de fichiers, où toutes les modifications d'un système de fichiers sont renvoyées sur un système de fichiers situé sur un autre ordinateur. La seule restriction est que ce miroir doit être construit de telle sorte que le serveur en attente dispose d'une version cohérente du système de fichiers -- spécifiquement, les écritures sur le serveur en attente doivent être réalisées dans le même ordre que celles sur le maître. DRBD™ est une solution populaire de réplication de systèmes de fichiers pour Linux.

Warm et Hot Standby en utilisant PITR

Les serveurs *warm et hot standby* (voir Section 25.2, « Serveurs de Standby par transfert de journaux ») peuvent conserver leur cohérence en lisant un flux d'enregistrements de WAL. Si le serveur principal échoue, le serveur *standby* contient pratiquement toutes les données du serveur principal et peut rapidement devenir le nouveau serveur maître. Ceci est asynchrone et ne peut se faire que pour le serveur de bases complet.

Un serveur de standby en PITR peut être implémenté en utilisant la recopie de journaux par fichier (Section 25.2, « Serveurs de Standby par transfert de journaux ») ou la streaming replication (réplication en continu, voir Section 25.2.5, « Streaming Replication »), ou une combinaison des deux. Pour des informations sur le hot standby, voyez Section 25.5, « Hot Standby »..

Réplication maître/esclave basé sur des triggers

Une configuration de réplication maître/esclave envoie toutes les requêtes de modification de données au serveur maître. Ce serveur envoie les modifications de données de façon asynchrone au serveur esclave. L'esclave peut répondre aux requêtes en lecture seule alors que le serveur maître est en cours d'exécution. Le serveur esclave est idéal pour les requêtes vers un entrepôt de données.

Slony-I™ est un exemple de ce type de réplication, avec une granularité par table et un support des esclaves multiples. Comme il met à jour le serveur esclave de façon asynchrone (par lots), il existe une possibilité de perte de données pendant un *failover*.

Middleware de réplication basé sur les instructions

Avec les *middleware* de réplication basés sur les instructions, un programme intercepte chaque requête SQL et l'envoie à un ou tous les serveurs. Chaque serveur opère indépendamment. Les requêtes en lecture/écriture sont envoyées à tous les serveurs alors que les requêtes en lecture seule ne peuvent être envoyées qu'à un seul serveur, ce qui permet de distribuer la charge de lecture.

Si les requêtes sont envoyées sans modification, les fonctions comme `random()`, `CURRENT_TIMESTAMP` ainsi que les séquences ont des valeurs différentes sur les différents serveurs. Cela parce que chaque serveur opère indépendamment alors que les requêtes SQL sont diffusées (et non les données modifiées). Si cette solution est inacceptable, le *middleware* ou l'application doivent demander ces valeurs à un seul serveur, et les utiliser dans des requêtes d'écriture. Une autre solution est d'utiliser cette solution de réplication avec une configuration maître-esclave traditionnelle, c'est à dire que les requêtes de modification de données ne sont envoyées qu'au maître et sont propagées aux esclaves via une réplication maître-esclave, pas par le middleware de réplication. Il est impératif que toute transaction soit validée ou annulée sur tous les serveurs, éventuellement par validation en deux phases (PREPARE TRANSACTION(7) et COMMIT PREPARED(7). Pgpool-II™ et Sequoia™ sont des exemples de ce type de réplication.

Réplication asynchrone multi-maîtres

Pour les serveurs qui ne sont pas connectés en permanence, comme les ordinateurs portables ou les serveurs distants, conserver la cohérence des données entre les serveurs est un challenge. L'utilisation de la réplication asynchrone multi-maîtres permet à chaque serveur de fonctionner indépendamment. Il communique alors périodiquement avec les autres serveurs pour identifier les transactions conflictuelles. La gestion des conflits est alors confiée aux utilisateurs ou à un système de règles de résolution. Bucardo est un exemple de ce type de réplication.

Réplication synchrone multi-maîtres

Dans les réplications synchrones multi-maîtres, tous les serveurs acceptent les requêtes en écriture. Les données modifiées sont transmises du serveur d'origine à tous les autres serveurs avant toute validation de transaction.

Une activité importante en écriture peut être la cause d'un verrouillage excessif et conduire à un effondrement des performances. Dans les faits, les performances en écriture sont souvent pis que celles d'un simple serveur.

Tous les serveurs acceptent les requêtes en lecture.

Certaines implantations utilisent les disques partagés pour réduire la surcharge de communication.

Les performances de la réplication synchrone multi-maîtres sont meilleures lorsque les opérations de lecture représentent l'essentiel de la charge, alors que son gros avantage est l'acceptation des requêtes d'écriture par tous les serveurs -- il n'est pas nécessaire de répartir la charge entre les serveurs maîtres et esclaves et, parce que les modifications de données sont envoyées d'un serveur à l'autre, les fonctions non déterministes, comme `random()`, ne posent aucun problème.

PostgreSQL™ n'offre pas ce type de réplication, mais la validation en deux phases de PostgreSQL™ (PREPARE TRANSACTION(7) et COMMIT PREPARED(7)) autorise son intégration dans une application ou un *middleware*.

Solutions commerciales

Parce que PostgreSQL™ est libre et facilement extensible, certaines sociétés utilisent PostgreSQL™ dans des solutions commerciales fermées (*closed-source*) proposant des fonctionnalités de bascule sur incident (*failover*), réplication et répartition de charge.

La Tableau 25.1, « Matrice de fonctionnalités : haute disponibilité, répartition de charge et réplication » résume les possibilités des

différentes solutions listées plus-haut.

Tableau 25.1. Matrice de fonctionnalités : haute disponibilité, répartition de charge et réplication

Fonctionnalité	**Bascule par disques partagés (*Shared Disk Failover*)**	**Réplication par système de fichiers**	**Secours semi-automatique (*Hot/Warm Standby*) par PITR**	**Réplication maître/esclave basé sur les triggers**	***Middleware* de réplication sur instructions**	**Réplication asynchrone multi-maîtres**	**Réplication synchrone multi-maîtres**
Exemple d'implémentation	NAS	DRBD	PITR	Slony	pgpool-II	Bucardo	
Méthode de communication	Disque partagé	Blocs disque	WAL	Lignes de tables	SQL	Lignes de tables	Lignes de tables et verrous de ligne
Ne requiert aucun matériel spécial		•	•	•	•	•	•
Autorise plusieurs serveurs maîtres					•	•	•
Pas de surcharge sur le serveur maître	•		•		•		
Pas d'attente entre serveurs	•		•	•		•	
Pas de perte de données en cas de panne du maître	•	•			•		•
Les esclaves acceptent les requêtes en lecture seule			Hot only	•	•	•	•
Granularité de niveau table				•		•	•
Ne nécessite pas de résolution de conflit	•	•	•	•			•

Certaines solutions n'entrent pas dans les catégories ci-dessus :

Partitionnement de données
: Le partitionnement des données divise les tables en ensembles de données. Chaque ensemble ne peut être modifié que par un seul serveur. Les données peuvent ainsi être partitionnées par bureau, Londres et Paris, par exemple, avec un serveur dans chaque bureau. Si certaines requêtes doivent combiner des données de Londres et Paris, il est possible d'utiliser une application qui requête les deux serveurs ou d'implanter une réplication maître/esclave pour conserver sur chaque serveur une copie en lecture seule des données de l'autre bureau.

Exécution de requêtes en parallèle sur plusieurs serveurs
: La plupart des solutions ci-dessus permettent à plusieurs serveurs de répondre à des requêtes multiples, mais aucune ne permet à une seule requête d'être exécutée sur plusieurs serveurs pour se terminer plus rapidement. Cette solution autorisent plusieurs serveurs à travailler ensemble sur une seule requête. Ceci s'accomplit habituellement en répartissant les données entre les serveurs, chaque serveur exécutant une partie de la requête pour renvoyer les résultats à un serveur central qui les combine et les renvoie à l'utilisateur. Pgpool-II™ offre cette possibilité. Cela peut également être implanté en utilisant les outils PL/Proxy™.

25.2. Serveurs de Standby par transfert de journaux

L'archivage en continu peut être utilisé pour créer une configuration de cluster en *haute disponibilité* (HA) avec un ou plusieurs *serveurs de standby* prêts à prendre la main sur les opérations si le serveur primaire fait défaut. Cette fonctionnalité est généralement appelée *warm standby* ou *log shipping*.

Les serveurs primaire et de standby travaillent de concert pour fournir cette fonctionnalité, bien que les serveurs ne soient que faiblement couplés. Le serveur primaire opère en mode d'archivage en continu, tandis que le serveur de standby opère en mode de récupération en continu, en lisant les fichiers WAL provenant du primaire. Aucune modification des tables de la base ne sont requises pour activer cette fonctionnalité, elle entraîne donc moins de travail d'administration par rapport à d'autres solutions de réplication. Cette configuration a aussi un impact relativement faible sur les performances du serveur primaire.

Déplacer directement des enregistrements de WAL d'un serveur de bases de données à un autre est habituellement appelé log shipping. PostgreSQL™ implémente le log shipping par fichier, ce qui signifie que les enregistrements de WAL sont transférés un fichier (segment de WAL) à la fois. Les fichiers de WAL (16Mo) peuvent être transférés facilement et de façon peu coûteuse sur n'importe quelle distance, que ce soit sur un système adjacent, un autre système sur le même site, ou un autre système à l'autre bout du globe. La bande passante requise pour cette technique varie en fonction du débit de transactions du serveur primaire. Le log shipping par enregistrement est aussi possible avec la streaming replication (voir Section 25.2.5, « Streaming Replication »).

Il convient de noter que le log shipping est asynchrone, c'est à dire que les enregistrements de WAL sont transférés après que la transaction ait été validée. Par conséquent, il y a un laps de temps pendant lequel une perte de données pourrait se produire si le serveur primaire subissait un incident majeur; les transactions pas encore transférées seront perdues. La taille de la fenêtre de temps de perte de données peut être réduite par l'utilisation du paramètre `archive_timeout`, qui peut être abaissé à des valeurs de quelques secondes. Toutefois, un paramètre si bas augmentera de façon considérable la bande passante nécessaire pour le transfert de fichiers. Si vous voulez une fenêtre de moins d'une minute environ, envisagez l'utilisation de la streaming replication (voir Section 25.2.5, « Streaming Replication »).

La performance de la récupération est suffisamment bonne pour que le standby ne soit en général qu'à quelques instants de la pleine disponibilité à partir du moment où il aura été activé. C'est pour cette raison que cette configuration de haute disponibilité est appelée warm standby. Restaurer un serveur d'une base de sauvegarde archivée, puis appliquer tous les journaux prendra largement plus de temps, ce qui fait que cette technique est une solution de 'disaster recovery' (reprise après sinistre), pas de haute disponibilité. Un serveur de standby peut aussi être utilisé pour des requêtes en lecture seule, dans quel cas il est appelé un serveur de Hot Standby. Voir Section 25.5, « Hot Standby » pour plus d'information.

25.2.1. Préparatifs

Il est habituellement préférable de créer les serveurs primaire et de standby de façon à ce qu'ils soient aussi similaires que possible, au moins du point de vue du serveur de bases de données. En particulier, les chemins associés avec les tablespaces seront passés d'un noeud à l'autre sans conversion, ce qui implique que les serveurs primaire et de standby doivent avoir les mêmes chemins de montage pour les tablespaces si cette fonctionnalité est utilisée. Gardez en tête que si CREATE TABLESPACE(7) est exécuté sur le primaire, tout nouveau point de montage nécessaire pour cela doit être créé sur le primaire et tous les standby avant que la commande ne soit exécutée. Le matériel n'a pas besoin d'être exactement le même, mais l'expérience monte que maintenir deux systèmes identiques est plus facile que maintenir deux différents sur la durée de l'application et du système. Quoi qu'il en soit, l'architecture hardware doit être la même -- répliquer par exemple d'un serveur 32 bits vers un 64 bits ne fonctionnera pas.

De manière générale, le log shipping entre serveurs exécutant des versions majeures différentes de PostgreSQL™ est impossible. La politique du PostgreSQL Global Development Group est de ne pas réaliser de changement sur les formats disques lors des mises à jour mineures, il est par conséquent probable que l'exécution de versions mineures différentes sur le primaire et le standby fonctionne correctement. Toutefois, il n'y a aucune garantie formelle de cela et il est fortement conseillé de garder le serveur primaire et celui de standby au même niveau de version autant que faire se peut. Lors d'une mise à jour vers une nouvelle version mineure, la politique la plus sûre est de mettre à jour les serveurs de standby d'abord -- une nouvelle version mineure est davantage susceptible de lire les enregistrements WAL d'une ancienne version mineure que l'inverse.

25.2.2. Fonctionnement du Serveur de Standby

En mode de standby, le serveur applique continuellement les WAL reçus du serveur maître. Le serveur de standby peut lire les WAL d'une archive WAL (voir restore_command) ou directement du maître via une connexion TCP (streaming replication). Le serveur de standby essaiera aussi de restaurer tout WAL trouvé dans le répertoire `pg_xlog` du cluster de standby. Cela se produit habituellement après un redémarrage de serveur, quand le standby rejoue à nouveau les WAL qui ont été reçu du maître avant le redémarrage, mais vous pouvez aussi copier manuellement des fichiers dans `pg_xlog` à tout moment pour qu'ils soient rejoués.

Au démarrage, le serveur de standby commence par restaurer tous les WAL disponibles à l'endroit où se trouvent les archives, en appelant la `restore_command`. Une fois qu'il a épuisé tous les WAL disponibles à cet endroit et que `restore_command` échoue, il essaye de restaurer tous les WAL disponibles dans le répertoire `pg_xlog`. Si cela échoue, et que la réplication en flux a été activée, le standby essaye de se connecter au serveur primaire et de démarrer la réception des WAL depuis le dernier enregistrement valide trouvé dans les archives ou `pg_xlog`. Si cela échoue ou que la streaming replication n'est pas configurée, ou que

la connexion est plus tard déconnectée, le standby retourne à l'étape 1 et essaye de restaurer le fichier à partir de l'archive à nouveau. Cette boucle de retentatives de l'archive, `pg_xlog` et par la streaming replication continue jusqu'à ce que le serveur soit stoppé ou que le failover (bascule) soit déclenché par un fichier trigger (déclencheur).

Le mode de standby est quitté et le serveur bascule en mode de fonctionnement normal quand un fichier de trigger est trouvé (`trigger_file`). Avant de basculer, tout WAL immédiatement disponible dans l'archive ou le `pg_xlog` sera restaurée, mais aucune tentative ne sera faite pour se connecter au maître.

25.2.3. Préparer le Maître pour les Serveurs de Standby

Mettez en place un archivage en continu sur le primaire vers un répertoire d'archivage accessible depuis le standby, comme décrit dans Section 24.3, « Archivage continu et récupération d'un instantané (PITR) ». La destination d'archivage devrait être accessible du standby même quand le maître est inaccessible, c'est à dire qu'il devrait se trouver sur le serveur de standby lui-même ou un autre serveur de confiance, pas sur le serveur maître.

Si vous voulez utiliser la streaming replication, mettez en place l'authentification sur le serveur primaire pour autoriser les connexions de réplication à partir du (des) serveur de standby; c'est à dire, mettez en place une ou des entrées appropriées dans `pg_hba.conf` avec le champ database positionné à `replication`. Vérifiez aussi que `max_wal_senders` est positionné à une valeur suffisamment grande dans le fichier de configuration du serveur primaire.

Effectuez une sauvegarde de base comme décrit dans Section 24.3.2, « Réaliser une sauvegarde de base » pour initialiser le serveur de standby.

25.2.4. Paramétrer un Serveur de Standby

Pour paramétrer le serveur de standby, restaurez la sauvegarde de base effectué sur le serveur primaire (voir (see Section 24.3.3, « Récupération à partir d'un archivage continu »). Créez un fichier de commande de récupération `recovery.conf` dans le répertoire de données du cluster de standby, et positionnez `standby_mode` à on. Positionnez `restore_command` à une simple commande qui recopie les fichiers de l'archive de WAL.

> **Note**
>
> N'utilisez pas pg_standby ou des outils similaires avec le mode de standby intégré décrit ici. `restore_command` devrait retourner immédiatement si le fichier n'existe pas; le serveur essayera la commande à nouveau si nécessaire. Voir Section 25.4, « Méthode alternative pour le log shipping » pour utiliser des outils tels que pg_standby.

Si vous souhaitez utiliser la streaming replication, renseignez `primary_conninfo` avec une chaîne de connexion libpq, contenant le nom d'hôte (ou l'adresse IP) et tout détail supplémentaire nécessaire pour se connecter au serveur primaire. Si le primaire a besoin d'un mot de passe pour l'authentification, le mot de passe doit aussi être spécifié dans `primary_conninfo`.

Si vous mettez en place le serveur de standby pour des besoins de haute disponibilité, mettez en place l'archivage de WAL, les connexions et l'authentification à l'identique du serveur primaire, parce que le serveur de standby fonctionnera comme un serveur primaire après la bascule. Vous aurez aussi besoin de positionner `trigger_file` pour rendre la bascule possible. Si vous mettez en place le serveur de standby pour des besoins de reporting, sans aucune intention de basculer dessus, `trigger_file` n'est pas nécessaire.

Si vous utilisez une archive WAL, sa taille peut être réduite en utilisant l'option archive_cleanup_command pour supprimer les fichiers qui ne sont plus nécessaires au serveur de standby. L'outil pg_archivecleanup est conçu spécifiquement pour être utilisé avec `archive_cleanup_command` dans des configurations typiques de standby, voir Section F.22, « pg_archivecleanup ». Notez toutefois que si vous utilisez l'archive à des fins de sauvegarde, vous avez besoin de garder les fichiers nécessaires pour restaurer à partir de votre dernière sauvegarde de base, même si ces fichiers ne sont plus nécessaires pour le standby.

If you're using a WAL archive, its size can be minimized using the parameter to remove files that are no longer required by the standby server. Note however, that if you're using the archive for backup purposes, you need to retain files needed to recover from at least the latest base backup, even if they're no longer needed by the standby.

Un simple exemple de `recovery.conf` est:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
trigger_file = '/path/to/trigger_file'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

Vous pouvez avoir n'importe quel nombre de serveurs de standby, mais si vous utilisez la streaming replication, assurez vous d'avoir positionné `max_wal_senders` suffisamment haut sur le primaire pour leur permettre de se connecter simultanément.

25.2.5. Streaming Replication

La streaming replication permet à un serveur de standby de rester plus à jour qu'il n'est possible avec l'envoi de journaux par fichiers. Le standby se connecte au primaire, qui envoie au standby les enregistrements de WAL dès qu'ils sont générés, sans attendre qu'un fichier de WAL soit rempli.

La streaming replication est asynchrone, il y a donc toujours un petit délai entre la validation d'une transaction sur le primaire et le moment où les changements sont visibles sur le standby. Le délai est toutefois beaucoup plus petit qu'avec l'envoi de fichiers, habituellement en dessous d'une seconde en partant de l'hypothèse que le standby est suffisamment puissant pour supporter la charge. Avec la streaming replication, `archive_timeout` n'est pas nécessaire pour réduire la fenêtre de perte de données.

Si vous utilisez la streaming replication sans archivage en continu des fichiers, vous devez positionner `wal_keep_segments` sur le maître à une valeur suffisamment grande pour garantir que les anciens segments de WAL ne sont pas recyclés trop tôt, alors que le standby pourrait toujours avoir besoin d'eux pour rattraper son retard. Si le standby prend trop de retard, il aura besoin d'être réinitialisé à partir d'une nouvelle sauvegarde de base. Si vous positionnez une archive de WAL qui est accessible du standby, `wal_keep_segments` n'est pas nécessaire, puisque le standby peut toujours utiliser l'archive pour rattraper son retard.

Pour utiliser la streaming replication, mettez en place un serveur de standby en mode fichier comme décrit dans Section 25.2, « Serveurs de Standby par transfert de journaux ». L'étape qui transforme un standby en mode fichier en standby en streaming replication est de faire pointer `primary_conninfo` dans le fichier `recovery.conf` vers le serveur primaire. Positionnez listen_addresses et les options d'authentification (voir `pg_hba.conf`) sur le primaire pour que le serveur de standby puisse se connecter à la pseudo-base `replication` sur le serveur primaire (voir Section 25.2.5.1, « Authentification »).

Sur les systèmes qui supportent l'option de keepalive sur les sockets, positionner tcp_keepalives_idle, tcp_keepalives_interval et tcp_keepalives_count aide le primaire à reconnaître rapidement une connexion interrompue.

Positionnez le nombre maximum de connexions concurrentes à partir des serveurs de standby (voir max_wal_senders pour les détails).

Quand le standby est démarré et que `primary_conninfo` est positionné correctement, le standby se connectera au primaire après avoir rejoué tous les fichiers WAL disponibles dans l'archive. Si la connexion est établie avec succès, vous verrez un processus walreceiver dans le standby, et un processus walsender correspondant sur le primaire.

25.2.5.1. Authentification

Il est très important que les privilèges d'accès pour la réplications soient paramétrés pour que seuls les utilisateurs de confiance puissent lire le flux WAL, parce qu'il est facile d'en extraire des informations privilégiées. Les serveurs de standby doivent s'authentifier sur le primaire avec un compte superutilisateur. Par conséquent, un rôle avec les privilèges `SUPERUSER` et `LOGIN` doit être créé sur le primaire.

L'authentification cliente pour la réplication est contrôlée par un enregistrement de `pg_hba.conf` spécifiant `replication` dans le champ *`database`*. Par exemple, si le standby s'exécute sur un hôte d'IP `192.168.1.100` et que le nom du superutilisateur pour la réplication est `foo`, l'administrateur peut ajouter la ligne suivante au fichier `pg_hba.conf` sur le primaire:

```
# Autoriser l'utilisateur "foo" de l'hôte 192.168.1.100 à se connecter au primaire
# en tant que standby de replication si le mot de passe de l'utilisateur est
correctement fourni
#
# TYPE  DATABASE        USER            CIDR-ADDRESS            METHOD
host    replication     foo             192.168.1.100/32        md5
```

Le nom d'hôte et le numéro de port du primaire, le nom d'utilisateur de la connexion, et le mot de passe sont spécifiés dans le fichier `recovery.conf`. Le mot de passe peut aussi être enregistré dans le fichier `~/.pgpass` sur le serveur en attente (en précisant `replication` dans le champ *`database`*). Par exemple, si le primaire s'exécute sur l'hôte d'IP `192.168.1.50`, port `5432`, que le nom du superutilisateur pour la réplication est `foo`, et que le mot de passe est `foopass`, l'administrateur peut ajouter la ligne suivante au fichier `recovery.conf` sur le standby:

```
# Le standby se connecte au primaire qui s'exécute sur l'hôte 192.168.1.50
# et port 5432 en tant qu'utilisateur "foo" dont le mot de passe est "foopass"
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

25.2.5.2. Supervision

Un important indicateur de santé de la streaming replication est le nombre d'enregistrements générés sur le primaire, mais pas encore appliqués sur le standby. Vous pouvez calculer ce retard en comparant le point d'avancement des écritures du WAL sur le primaire avec le dernier point d'avancement reçu par le standby. Ils peuvent être récupérés en utilisant

`pg_current_xlog_location` sur le primaire et `pg_last_xlog_receive_location` sur le standby, respectivement (voir Tableau 9.56, « Fonctions de contrôle de la sauvegarde » et Tableau 9.57, « Fonctions d'information sur la restauration » pour plus de détails). Le point d'avancement de la réception dans le standby est aussi affiché dans le statut du processus de réception des WAL (wal receiver), affiché par la commande **ps** (voyez Section 27.1, « Outils Unix standard » pour plus de détails).

25.3. Failover (bascule)

Si le serveur primaire plante alors le serveur de standby devrait commencer les procédures de failover.

Si le serveur de standby plante alors il n'est pas nécessaire d'effectuer un failover. Si le serveur de standby peut être redémarré, même plus tard, alors le processus de récupération peut aussi être redémarré au même moment, en bénéficiant du fait que la récupération sait reprendre où elle en était. Si le serveur de standby ne peut pas être redémarré, alors une nouvelle instance complète de standby devrait être créé.

Si le serveur primaire plante, que le serveur de standby devient le nouveau primaire, et que l'ancien primaire redémarre, vous devez avoir un mécanisme pour informer l'ancien primaire qu'il n'est plus primaire. C'est aussi quelquefois appelé STONITH (Shoot The Other Node In The Head, ou Tire Dans La Tête De L'Autre Noeud), qui est nécessaire pour éviter les situations où les deux systèmes pensent qu'ils sont le primaire, ce qui amènerait de la confusion, et finalement de la perte de données.

Beaucoup de systèmes de failover n'utilisent que deux systèmes, le primaire et le standby, connectés par un mécanisme de type ligne de vie (heartbeat) pour vérifier continuellement la connexion entre les deux et la viabilité du primaire. Il est aussi possible d'utiliser un troisième système (appelé un serveur témoin) pour éviter certains cas de bascule inappropriés, mais la complexité supplémentaire peut ne pas être justifiée à moins d'être mise en place avec suffisamment de précautions et des tests rigoureux.

PostgreSQL™ ne fournit pas le logiciel système nécessaire pour identifier un incident sur le primaire et notifier le serveur de base de standby. De nombreux outils de ce genre existent et sont bien intégrés avec les fonctionnalités du système d'exploitation nécessaires à la bascule, telles que la migration d'adresse IP.

Une fois que la bascule vers le standby se produit, il n'y a plus qu'un seul serveur en fonctionnement. C'est ce qu'on appelle un état dégradé. L'ancien standby est maintenant le primaire, mais l'ancien primaire est arrêté et pourrait rester arrêté. Pour revenir à un fonctionnement normal, un serveur de standby doit être recréé, soit sur l'ancien système primaire quand il redevient disponible, ou sur un troisième, peut être nouveau, système. Une fois que ceci est effectué, le primaire et le standby peuvent être considérés comme ayant changé de rôle. Certaines personnes choisissent d'utiliser un troisième serveur pour fournir une sauvegarde du nouveau primaire jusqu'à ce que le nouveau serveur de standby soit recréé, bien que ceci complique visiblement la configuration du système et les procédures d'exploitation.

Par conséquent, basculer du primaire vers le serveur de standby peut être rapide mais requiert du temps pour re-préparer le cluster de failobver. Une bascule régulière du primaire vers le standby est utile, car cela permet une période d'interruption de production sur chaque système pour maintenance. Cela vous permet aussi pour vous assurer que votre mécanisme de bascule fonctionnera réellement quand vous en aurez besoin. Il est conseillé que les procédures d'administration soient écrites.

Pour déclencher le failover d'un serveur de standby en log-shipping, créez un fichier trigger (déclencheur) avec le nom de fichier et le chemin spécifiés par le paramètre `trigger_file` de `recovery.conf`. Si `trigger_file` n'est pas fourni, il n'y a aucun moyen de sortir du mode de recovery sur le standby et de le promouvoir en maître. Ceci peut être utile, par exemple, pour des serveurs de reporting qui ne sont utilisés que pour décharger le primaire de requêtes en lecture seule, pas à des fins de haute disponibilité.

25.4. Méthode alternative pour le log shipping

Une alternative au mode de standby intégré décrit dans les sections précédentes est d'utiliser une `restore_command` qui scrute le dépôt d'archives. C'était la seule méthode disponible dans les versions 8.4 et inférieures. Dans cette configuration, positionnez `standby_mode` à off, parce que vous implémentez la scrutation nécessaire au fonctionnement standby vous-mêmes. Voyez contrib/pg_standby (Section F.28, « pg_standby ») pour une implémentation de référence de ceci.

Veuillez noter que dans ce mode, le serveur appliquera les WAL fichier par fichier, ce qui entraîne que si vous requêtez sur le serveur de standby (voir Hot Standby), il y a un délai entre une action sur le maître et le moment où cette action devient visible sur le standby, correspondant au temps nécessaire à remplir le fichier de WAL. `archive_timeout` peut être utilisé pour rendre ce délai plus court. Notez aussi que vous ne pouvez combiner la streaming replication avec cette méthode.

Les opérations qui se produisent sur le primaire et les serveurs de standby sont des opérations normales d'archivage et de recovery. Le seul point de contact entre les deux serveurs de bases de données est l'archive de fichiers WAL qu'ils partagent: le primaire écrivant dans l'archive, le secondaire lisant de l'archive. Des précautions doivent être prises pour s'assurer que les archives WAL de serveurs primaires différents ne soient pas mélangées ou confondues. L'archive n'a pas besoin d'être de grande taille si elle n'est utilisée que pour le fonctionnement de standby.

La magie qui permet aux deux serveurs faiblement couplés de fonctionner ensemble est une simple `restore_command` utilisée sur le standby qui quand on lui demande le prochain fichier de WAL, attend que le primaire le mette à disposition. La `res-`

tore_command est spécifiée dans le fichier recovery.conf sur le serveur de standby. La récupération normale demanderait un fichier de l'archive WAL, en retournant un échec si le fichier n'était pas disponible. Pour un fonctionnement en standby, il est normal que le prochain fichier WAL ne soit pas disponible, ce qui entraîne que le standby doive attendre qu'il apparaisse. Pour les fichiers se terminant en .backup ou .history il n'y a pas besoin d'attendre, et un code retour différent de zéro doit être retourné. Une restore_command d'attente peut être écrite comme un script qui boucle après avoir scruté l'existence du prochain fichier de WAL. Il doit aussi y avoir un moyen de déclencher la bascule, qui devrait interrompre la restore_command , sortir le la boucle et retourner une erreur file-not-found au serveur de standby. Cela met fin à la récupération et le standby démarrera alors comme un serveur normal.

Le pseudocode pour une restore_command appropriée est:

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

Un exemple fonctionnel de restore_command d'attente est fournie par le module contrib appelé pg_standby. Il devrait être utilisé en tant que référence, comme la bonne façon d'implémenter correctement la logique décrite ci-dessus. Il peut aussi être étendu pour supporter des configurations et des environnements spécifiques.

La méthode pour déclencher une bascule est une composante importante de la planification et de la conception. Une possibilité est d'utiliser la commande restore_command. Elle est exécutée une fois pour chaque fichier WAL, mais le processus exécutant la restore_command est créé et meurt pour chaque fichier, il n'y a donc ni démon ni processus serveur, et on ne peut utiliser ni signaux ni gestionnaire de signaux. Par conséquent, la restore_command n'est pas appropriée pour déclencher la bascule. Il est possible d'utiliser une simple fonctionnalité de timeout, particulièrement si utilisée en conjonction avec un paramètre archive_timeout sur le primaire. Toutefois, ceci est sujet à erreur, un problème réseau ou un serveur primaire chargé pouvant suffire à déclencher une bascule. Un système de notification comme la création explicite d'un fichier trigger est idéale, dans la mesure du possible.

25.4.1. Implémentation

La procédure simplifié pour configurer un serveur de test en utilisant cette méthode alternative est la suivante. Pour tous les détails sur chaque étape, référez vous aux sections précédentes suivant les indications.

1. Paramétrez les systèmes primaire et standby de façon aussi identique que possible, y compris deux copies identiques de PostgreSQL™ au même niveau de version.
2. Activez l'archivage en continu du primaire vers un répertoire d'archives WAL sur le serveur de standby. Assurez vous que archive_mode, archive_command et archive_timeout sont positionnés correctement sur le primaire (voir Section 24.3.1, « Configurer l'archivage WAL »).
3. Effectuez une sauvegarde de base du serveur primaire(voir Section 24.3.2, « Réaliser une sauvegarde de base »), , et chargez ces données sur le standby.
4. Commencez la récupération sur le serveur de standby à partir de l'archive WAL locale, en utilisant un recovery.conf qui spécifie une restore_command qui attend comme décrit précédemment (voir Section 24.3.3, « Récupération à partir d'un archivage continu »).

Le récupération considère l'archive WAL comme étant en lecture seule, donc une fois qu'un fichier WAL a été copié sur le système de standby il peut être copié sur bande en même temps qu'il est lu par le serveur de bases de données de standby. Ainsi, on peut faire fonctionner un serveur de standby pour de la haute disponibilité en même temps que les fichiers sont stockés pour de la reprise après sinistre.

À des fins de test, il est possible de faire fonctionner le serveur primaire et de standby sur le même système. Cela n'apporte rien en termes de robustesse du serveur, pas plus que cela ne pourrait être décrit comme de la haute disponibilité.
.

25.4.2. Log Shipping par Enregistrements

Il est aussi possible d'implémenter du log shipping par enregistrements en utilisant cette méthode alternative, bien qu'elle nécessite des développements spécifiques, et que les modifications ne seront toujours visibles aux requêtes de hot standby qu'après que le fi-

chier complet de WAL ait été recopié.

Un programme externe peut appeler la fonction `pg_xlogfile_name_offset()` (voir Section 9.24, « Fonctions d'administration système ») pour obtenir le nom de fichier et la position exacte en octets dans ce fichier de la fin actuelle du WAL. Il peut alors accéder au fichier WAL directement et copier les données de la fin précédente connue à la fin courante vers les serveurs de standby. Avec cette approche, la fenêtre de perte de données est la période de scrutation du programme de copie, qui peut être très petite, et il n'y a pas de bande passante gaspillée en forçant l'archivage de fichiers WAL partiellement remplis. Notez que les scripts `restore_command` des serveurs de standby ne peuvent traiter que des fichiers WAL complets, les données copiées de façon incrémentale ne sont donc d'ordinaire pas mises à disposition des serveurs de standby. Elles ne sont utiles que si le serveur primaire tombe -- alors le dernier fichier WAL partiel est fourni au standby avant de l'autoriser à s'activer. L'implémentation correcte de ce mécanisme requiert la coopération entre le script `restore_command` et le programme de recopie des données.

À partir de PostgreSQL™ version 9.0, vous pouvez utiliser la streaming replication (voir Section 25.2.5, « Streaming Replication ») pour bénéficier des mêmes fonctionnalités avec moins d'efforts.

25.5. Hot Standby

Hot Standby est le terme utilisé pour décrire la possibilité de se connecter et d'exécuter des requêtes en lecture seule alors que le serveur est en récupération d'archive or standby mode. C'est utile à la fois pour la réplication et pour restaurer une sauvegarde à un état désiré avec une grande précision. Le terme Hot Standby fait aussi référence à la capacité du serveur à passer de la récupération au fonctionnement normal tandis-que les utilisateurs continuent à exécuter des requêtes et/ou gardent leurs connexions ouvertes.

Exécuter des requêtes en mode hot standby est similaire au fonctionnement normal des requêtes, bien qu'il y ait quelques différences d'utilisation et d'administration notées ci-dessous.

25.5.1. Aperçu pour l'utilisateur

Quand le paramètre hot_standby est configuré à true sur un serveur en attente, le serveur commencera à accepter les connexions une fois que la restauration est parvenue à un état cohérent. Toutes les connexions qui suivront seront des connexions en lecture seule ; même les tables temporaires ne pourront pas être utilisées.

Les données sur le standby mettent un certain temps pour arriver du serveur primaire, il y aura donc un délai mesurable entre primaire et standby. La même requête exécutée presque simultanément sur le primaire et le standby pourrait par conséquent retourner des résultats différents. On dit que la donnée est *cohérente à terme* avec le primaire. Une fois que l'enregistrement de validation (COMMIT) d'une transaction est rejoué sur le serveur en attente, les modifications réalisées par cette transaction seront visibles par toutes les images de bases obtenues par les transactions en cours sur le serveur en attente. Ces images peuvent être prises au début de chaque requête ou de chaque transaction, suivant le niveau d'isolation des transactions utilisé à ce moment. Pour plus de détails, voir Section 13.2, « Isolation des transactions ».

Les transactions exécutées pendant la période de restauration sur un serveur en mode hotstandby peuvent inclure les commandes suivantes :

- Accès par requête - **SELECT**, **COPY TO**
- Commandes de curseur - **DECLARE**, **FETCH**, **CLOSE**
- Paramètres - **SHOW**, **SET**, **RESET**
- Commandes de gestion de transaction
  - **BEGIN**, **END**, **ABORT**, **START TRANSACTION**
  - **SAVEPOINT**, **RELEASE**, **ROLLBACK TO SAVEPOINT**
  - Blocs d'**EXCEPTION** et autres sous-transactions internes
- **LOCK TABLE**, mais seulement quand explicitement dans un de ces modes: `ACCESS SHARE`, `ROW SHARE` ou `ROW EXCLUSIVE`.
- Plans et ressources - **PREPARE**, **EXECUTE**, **DEALLOCATE**, **DISCARD**
- Plugins et extensions - **LOAD**

Les transactions lancées pendant la restauration d'un serveur en hotstandby ne se verront jamais affectées un identifiant de transactions et ne peuvent pas être écrites dans les journaux de transactions. Du coup, les actions suivantes produiront des messages d'erreur :

- Langage de Manipulation de Données (LMD ou DML) - **INSERT**, **UPDATE**, **DELETE**, **COPY FROM**, **TRUNCATE**. Notez qu'il n'y a pas d'action autorisée qui entraînerait l'exécution d'un trigger pendant la récupération. Cette restriction s'applique même pour les tables temporaires car les lignes de ces tables ne peuvent être lues et écrites s'il n'est pas possible d'affecter un identifiant de transactions, ce qui n'est actuellement pas possible dans un environnement Hot Standby.
- Langage de Définition de Données (LDD ou DDL) - **CREATE**, **DROP**, **ALTER**, **COMMENT**. Cette restriction s'applique aussi aux tables temporaires car, pour mener à bien ces opérations, cela nécessiterait de mettre à jour les catalogues systèmes.
- **SELECT ... FOR SHARE | UPDATE**, car les verrous de lignes ne peuvent pas être pris sans mettre à jour les fichiers de données.
- Rules sur des ordres **SELECT** qui génèrent des commandes LMD.
- **LOCK** qui demandent explicitement un mode supérieur à `ROW EXCLUSIVE MODE`.
- **LOCK** dans sa forme courte par défaut, puisqu'il demande `ACCESS EXCLUSIVE MODE`.
- Commandes de gestion de transaction qui positionnent explicitement un état n'étant pas en lecture-seule:
  - **BEGIN READ WRITE**, **START TRANSACTION READ WRITE**
  - **SET TRANSACTION READ WRITE**, **SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE**
  - **SET transaction_read_only = off**
- Commandes de two-phase commit **PREPARE TRANSACTION**, **COMMIT PREPARED**, **ROLLBACK PREPARED** parce que même les transactions en lecture seule ont besoin d'écrire dans le WAL durant la phase de préparation (la première des deux phases du two-phase commit).
- Mise à jour de séquence - `nextval()`, `setval()`
- **LISTEN**, **UNLISTEN**, **NOTIFY**

Dans le cadre normal, les transactions « en lecture seule » permettent la mise à jour des séquences et l'utilisation des instructions **LISTEN**, **UNLISTEN** et **NOTIFY**, donc les sessions Hot Standby ont des restrictions légèrement inférieures à celles de sessions en lecture seule ordinaires. Il est possible que certaines des restrictions soient encore moins importantes dans une prochaine version.

Lors du fonctionnement en serveur hotstandby, le paramètre `transaction_read_only` est toujours à true et ne peut pas être modifié. Tant qu'il n'y a pas de tentative de modification sur la base de données, les connexions sur un serveur en hotstandby se comportent de façon pratiquement identiques à celles sur un serveur normal. Quand une bascule (*failover* ou *switchover*) survient, la base de données bascule dans le mode de traitement normal. Les sessions resteront connectées pendant le changement de mode. Quand le mode hotstandby est terminé, il sera possible de lancer des transactions en lecture/écriture, y compris pour les sessions connectées avant la bascule.

Les utilisateurs pourront déterminer si leur session est en lecture seule en exécutant **SHOW transaction_read_only**. De plus, un jeu de fonctions (Tableau 9.57, « Fonctions d'information sur la restauration ») permettent aux utilisateurs d' accéder à des informations à propos du serveur de standby. Ceci vous permet d'écrire des programmes qui sont conscients de l'état actuel de la base. Vous pouvez vous en servir pour superviser l'avancement de la récupération, ou pour écrire des programmes complexes qui restaurent la base dans des états particuliers.

25.5.2. Gestion des conflits avec les requêtes

Les noeuds primaire et standby sont de bien des façons faiblement couplés. Des actions sur le primaire auront un effet sur le standby. Par conséquent, il y a un risque d'interactions négatives ou de conflits entre eux. Le conflit le plus simple à comprendre est la performance : si un gros chargement de données a lieu sur le primaire, il générera un flux similaire d'enregistrements WAL sur le standby, et les requêtes du standby pourrait entrer en compétition pour les ressources systèmes, comme les entrées-sorties.

Il y a aussi d'autres types de conflits qui peuvent se produire avec le Hot Standby. Ces conflits sont des *conflits durs* dans le sens où des requêtes pourraient devoir être annulées et, dans certains cas, des sessions déconnectées, pour les résoudre. L'utilisateur dispose de plusieurs moyens pour gérer ces conflits. Voici les différents cas de conflits possibles :

- Des verrous en accès exclusif pris sur le serveur maître, incluant à la fois les commandes **LOCK** exclusives et quelques actions de type DDL, entrent en conflit avec les accès de table des requêtes en lecture seule.
- La suppression d'un tablespace sur le serveur maître entre en conflit avec les requêtes sur le serveur standby qui utilisent ce tablespace pour les fichiers temporaires.
- La suppression d'une base de données sur le serveur maître entre en conflit avec les sessions connectées sur cette base de don-

nées sur le serveur en attente.

- La copie d'un enregistrement nettoyé par un VACUUM entre en conflit avec les transactions sur le serveur en attente qui peuvent toujours « voir » au moins une des lignes à supprimer.
- La copie d'un enregistrement nettoyé par un VACUUM entre en conflit avec les requêtes accédant à la page cible sur le serveur en attente, qu'elles voient ou non les données à supprimer.

Sur le serveur maître, ces cas résultent en une attente supplémentaire ; l'utilisateur peut choisir d'annuler une des actions en conflit. Néanmoins, sur le serveur en attente, il n'y a pas de choix possibles : l'action enregistrée dans les journaux de transactions est déjà survenue sur le serveur maître et le serveur en standby doit absolument réussir à l'appliquer. De plus, permettre que l'enregistrement de l'action attende indéfiniment pourrait avoir des effets fortement non désirables car le serveur en attente sera de plus en plus en retard par rapport au maître. Du coup, un mécanisme est fourni pour forcer l'annulation des requêtes sur le serveur en attente qui entreraient en conflit avec des enregistrements des journaux de transactions en attente.

Voici un exemple de problème type : un administrateur exécute un **DROP TABLE** sur une table du serveur maître qui est actuellement utilisé dans des requêtes du serveur en attente. Il est clair que la requête ne peut pas continuer à s'exécuter si l'enregistrement dans les journaux de transactions, correspondant au **DROP TABLE** est appliqué sur le serveur en attente. Si cette situation survient sur le serveur maître, l'instruction **DROP TABLE** attendra jusqu'à ce que l'autre requête se termine. Par contre, quand le **DROP TABLE** est exécuté sur le serveur maître, ce dernier ne sait pas les requêtes en cours d'exécution sur le serveur en attente, donc il n'attendra pas la fin de l'exécution des requêtes sur le serveur en attente. L'enregistrement de cette modification dans les journaux de transactions arrivera au serveur en attente alors que la requête sur le serveur en attente est toujours en cours d'exécution, causant un conflit. Le serveur en attente doit soit retarder l'application des enregistrements des journaux de transactions (et tous ceux qui sont après aussi) soit annuler la requête en conflit, pour appliquer l'instruction **DROP TABLE**.

Quand une requête en conflit est courte, il est généralement préférable d'attendre un peu pour l'application du journal de transactions. Mais un délai plus long n'est généralement pas souhaitable. Donc, le mécanisme d'annulation dans l'application des enregistrements de journaux de transactions dispose de deux paramètres, max_standby_archive_delay et max_standby_streaming_delay, qui définissent le délai maximum autorisé pour appliquer les enregistrements. Les requêtes en conflit seront annulées si l'application des enregistrements prend plus de temps que celui défini. Il existe deux paramètres pour que des délais différents puissent être observés suivant le cas : lecture des enregistrements à partir d'un journal archivé (par exemple lors de la restauration initiale à partir d'une sauvegarde ou lors d'un « rattrapage » si le serveur en attente accumulait du retard par rapport au maître) et lecture des enregistrements à partir de la réplication en flux.

Pour un serveur en attente dont le but principal est la haute-disponibilité, il est préférable de configurer des valeurs assez basses pour les paramètres de délai, de façon à ce que le serveur en attente ne soit pas trop en retard par rapport au serveur maître à cause des délais suivis à cause des requêtes exécutées sur le serveur en attente. Par contre, si le serveur en attente doit exécuter des requêtes longues, alors une valeur haute, voire infinie, du délai pourrait être préférable. Néanmoins, gardez en tête qu'une requête mettant du temps à s'exécuter pourrait empêcher les autres requêtes de voir les modifications récentes sur le serveur primaire si elle retarde l'application des enregistrements de journaux de transactions.

La raison la plus fréquente des conflits entre les requêtes en lecture seule et le rejeu des journaux de transactions est le « nettoyage avancé ». Habituellement, PostgreSQL™ permet le nettoyage des anciennes versions de lignes quand aucune transaction ne peut les voir pour s'assurer du respect des règles de MVCC. Néanmoins, cette règle peut seulement s'appliquer sur les transactions exécutées sur le serveur maître. Donc il est possible que le nettoyage effectué sur le maître supprime des versions de lignes toujours visibles sur une transaction exécutée sur le serveur en attente.

Les utilisateurs expérimentés peuvent noter que le nettoyage des versions de ligne ainsi que le gel des versions de ligne peuvent potentiellement avoir un conflit avec les requêtes exécutées sur le serveur en attente. L'exécution d'un **VACUUM FREEZE** manuel a de grandes chances de causer des conflits, y compris sur les tables sans lignes mises à jour ou supprimées.

Une fois que le délai spécifié par `max_standby_archive_delay` ou `max_standby_streaming_delay` a été dépassé, toutes les requêtes en conflit seront annulées. Ceci résulte habituellement en une erreur d'annulation, bien que certains cas, comme un **DROP DATABASE**, peuvent occassionner l'arrêt complet de la connexion. De plus, si le conflit intervient sur un verrou détenu par une transaction en attente, la session en conflit sera terminée (ce comportement pourrait changer dans le futur).

Les requêtes annulées peuvent être ré-exécutées immédiatement (après avoir commencé une nouvelle transaction, bien sûr). Comme l'annulation des requêtes dépend de la nature des enregistrements dans le journal de transactions, une requête annulée pourrait très bien réussir si elle est de nouveau exécutée.

Gardez en tête que les paramètres de délai sont comparés au temps passé depuis que la donnée du journal de transactions a été reçue par le serveur en attente. Du coup, la période de grâce accordée aux requêtes n'est jamais supérieur au paramètre de délai, et peut être considérablement inférieur si le serveur en attente est déjà en retard suite à l'attente de la fin de l'exécution de requêtes précédentes ou suite à son impossibilité de conserver le rythme d'une grosse mise à jour.

Les utilisateurs doivent s'attendre à ce que les tables fréquemment mises à jour sur le serveur primaire seront aussi fréquemment la cause de requêtes annulées sur le serveur en attente. Dans un tel cas, le paramétrage d'une valeur finie pour

max_standby_archive_delay ou max_standby_streaming_delay peut être considéré comme similaire à la configuration de statement_timeout.

Si le nombre d'annulations de requêtes sur le serveur en attente est jugé inadmissible, quelques solutions existent. La première option est de se connecter au serveur maître et de conserver une requête active assez longtemps pour que les requêtes s'exécutent sur le serveur en attente. Cela empêche **VACUUM** de supprimer les versions de lignes récemment morts, empêchant du même coup des conflits sur le nettoyage des lignes. Cela peut se faire en utilisant contrib/dblink et pg_sleep(), mais aussi via d'autres mécanismes. Si vous le faites, vous devez noter que cela retardera le nettoyage des versions de lignes mortes sur le serveur maître, ce qui pourrait résulter en une fragmentation non désirée de la table. Néanmoins, cette situation ne sera pas meilleure si les requêtes du serveur en attente s'exécutaient directement sur le serveur maître. Vous avez toujours le bénéfice de l'exécution sur un serveur distant. max_standby_archive_delay doit être configuré avec une valeur suffisamment large dans ce cas car les journaux de transactions en retard pourraient déjà contenir des entrées en conflit avec les requêtes sur le serveur en attente.

Une autre option revient à augmenter vacuum_defer_cleanup_age sur le serveur maître, pour que les lignes mortes ne soient pas nettoyées aussi rapidement que d'habitude. Cela donnera plus de temps aux requêtes pour s'exécuter avant d'être annulées sur le serveur en attente, sans voir à configurer une valeur importante de max_standby_streaming_delay. Néanmoins, il est difficile de garantir une fenêtre spécifique de temps d'exécution avec cette approche car vacuum_defer_cleanup_age est mesuré en nombre de transactions sur le serveur maître.

25.5.3. Aperçu pour l'administrateur

Si hot_standby est positionné à on dans postgresql.conf et qu'une fichier recovery.conf est présent, le serveur fonctionnera en mode Hot Standby. Toutefois, il pourrait s'écouler du temps avant que les connections en Hot Standby soient autorisées, parce que le serveur n'acceptera pas de connexions tant que la récupération n'aura pas atteint un point garantissant un état cohérent permettant aux requêtes de s'exécuter. Pendant cette période, les clients qui tentent de se connecter seront rejetés avec un message d'erreur. Pour confirmer que le serveur a démarré, vous pouvez soit tenter de vous connecter en boucle, ou rechercher ces messages dans les journaux du serveur:

```
LOG:  entering standby mode

... puis, plus loin ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

L'information sur la cohérence est enregistrée une fois par checkpoint sur le primaire. Il n'est pas possible d'activer le hot standby si on lit des WAL générés durant une période pendant laquelle wal_level n'était pas positionné à hot_standby sur le primaire. L'arrivée à un état cohérent peut aussi être retardée si ces deux conditions se présentent:

- Une transaction en écriture a plus de 64 sous-transactions
- Des transactions en écriture ont une durée très importante

Si vous effectuez du log shipping par fichier ("warm standby"), vous pourriez devoir attendre jusqu'à l'arrivée du prochain fichier de WAL, ce qui pourrait être aussi long que le paramètre archive_timeout du primaire.

Certains paramètres sur le standby vont devoir être revus si ils ont été modifiés sur le primaire. Pour ces paramètres, la valeur sur le standby devra être égale ou supérieure à celle du primaire. Si ces paramètres ne sont pas suffisamment élevés le standby refusera de démarrer. Il est tout à fait possible de fournir de nouvelles valeurs plus élevées et de redémarrer le serveur pour reprendre la récupération. Ces paramètres sont les suivants:

- max_connections
- max_prepared_transactions
- max_locks_per_transaction

Il est important que l'administrateur sélectionne le paramétrage approprié pour max_standby_archive_delay et max_standby_streaming_delay. Le meilleur choix varie les priorités. Par exemple, si le serveur a comme tâche principale d'être un serveur de haute-disponibilité, alors il est préférable d'avoir une configuration assez basse, voire à zéro, de ces paramètres. Si le serveur en attente est utilisé comme serveur supplémentaire pour des requêtes du type décisionnel, il sera acceptable de mettre les paramètres de délai à des valeurs allant jusqu'à plusieurs heures, voire même -1 (cette valeur signifiant qu'il est possible d'attendre que les requêtes se terminent d'elles-même).

Les "hint bits" (bits d'indices) écrits sur le primaire ne sont pas journalisés en WAL, il est donc probable que les les hint bits soient réécrits sur le standby. Ainsi, le serveur de standby fera toujours des écritures disques même si tous les utilisateurs sont en lecture seule; aucun changement ne se produira sur les données elles mêmes. Les utilisateurs écriront toujours les fichiers temporaires

pour les gros tris et re-génèreront les fichiers d'information relcache, il n'y a donc pas de morceau de la base qui soit réellement en lecture seule en mode hot standby. Notez aussi que les écritures dans des bases distantes en utilisant le module dblink , et d'autres opération en dehors de la base s'appuyant sur des fonctions PL seront toujours possibles, même si la transaction est en lecture seule localement.

Les types suivants de commandes administratives ne sont pas acceptées durant le mode de récupération:

- Langage de Définition de Données (LDD ou DDL) - comme **CREATE INDEX**
- Privilège et possession - **GRANT**, **REVOKE**, **REASSIGN**
- Commandes de maintenance - **ANALYZE**, **VACUUM**, **CLUSTER**, **REINDEX**

Notez encore une fois que certaines de ces commandes sont en fait autorisées durant les transactions en "lecture seule" sur le primaire.

Par conséquent, vous ne pouvez pas créer d'index supplémentaires qui existeraient uniquement sur le standby, ni des statistiques qui n'existeraient que sur le standby. Si ces commandes administratives sont nécessaires, elles doivent être exécutées sur le primaire, et ces modifications se propageront à terme au standby.

`pg_cancel_backend()` fonctionnera sur les processus utilisateurs, mais pas sur les processus de démarrage, qui effectuent la récupération. pg_stat_activity ne montre pas d'entrée pour le processus de démarrage, et les transactions de récupération ne sont pas affichées comme actives. Ainsi, pg_prepared_xacts est toujours vide durant la récupération. Si vous voulez traiter des transactions préparées douteuses, interrogez pg_prepared_xacts sur le primaire, et exécutez les commandes pour résoudre le problème à cet endroit.

pg_locks affichera les verrous possédés par les processus, comme en temps normal. pg_locks affiche aussi une transaction virtuelle gérée par le processus de démarrage qui possède tous les `AccessExclusiveLocks` possédés par les transactions rejouées par la récupération. Notez que le processus de démarrage n'acquiert pas de verrou pour effectuer les modifications à la base, et que par conséquent les verrous autre que `AccessExclusiveLocks` ne sont pas visibles dans pg_locks pour le processus de démarrage; ils sont simplement censés exister.

Le plugin Nagios™ check_pgsql™ fonctionnera, parce que les informations simples qu'il vérifie existent. Le script de supervision check_postgres™ fonctionnera aussi, même si certaines valeurs retournées pourraient être différentes ou sujettes à confusion. Par exemple, la date de dernier vacuum ne sera pas mise à jour, puisqu'aucun vacuum ne se déclenche sur le standby. Les vacuums s'exécutant sur le primaire envoient toujours leurs modifications au standby.

Les options de contrôle des fichiers de WAL ne fonctionneront pas durant la récupération, comme `pg_start_backup`, `pg_switch_xlog`, etc...

Les modules à chargement dynamique fonctionnent, comme pg_stat_statements.

Les verrous consultatifs fonctionnent normalement durant la récupération, y compris en ce qui concerne la détection des verrous mortels (deadlocks). Notez que les verrous consultatifs ne sont jamais tracés dans les WAL, il est donc impossible pour un verrou consultatif sur le primaire ou le standby d'être en conflit avec la ré-application des WAL. Pas plus qu'il n'est possible d'acquérir un verrou consultatif sur le primaire et que celui-ci initie un verrou consultatif similaire sur le standby. Les verrous consultatifs n'ont de sens que sur le serveur sur lequel ils sont acquis.

Les systèmes de réplications à base de triggers tels que Slony™, Londiste™ et Bucardo™ ne fonctionneront pas sur le standby du tout, même s'ils fonctionneront sans problème sur le serveur primaire tant que les modifications ne sont pas envoyées sur le serveur standby pour y être appliquées. Le rejeu de WAL n'est pas à base de triggers, vous ne pouvez donc pas utiliser le standby comme relai vers un système qui aurait besoin d'écritures supplémentaires ou utilise des triggers.

Il n'est pas possible d'assigner de nouveaux OID, bien que des générateurs d' UUID puissent tout de même fonctionner, tant qu'ils n'ont pas besoin d'écrire un nouveau statut dans la base.

À l'heure actuelle, la création de table temporaire n'est pas autorisée durant les transactions en lecture seule, certains scripts existants pourraient donc ne pas fonctionner correctement. Cette restriction pourrait être levée dans une version ultérieure. Il s'agit à la fois d'un problème de respect des standards et un problème technique.

DROP TABLESPACE ne peut réussir que si le tablespace est vide. Certains utilisateurs pourraient utiliser de façon active le tablespace via leur paramètre `temp_tablespaces`. S'il y a des fichiers temporaires dans le tablespace, toutes les requêtes actives sont annulées pour s'assurer que les fichiers temporaires sont supprimés, afin de supprimer le tablespace et de continuer l'application des WAL.

Exécuter **DROP DATABASE** ou **ALTER DATABASE ... SET TABLESPACE** sur le serveur maître générera un enregistrement dans les journaux de transactions qui causera la déconnexion de tous les utilisateurs actuellement connectés à cette base de données. Cette action survient immédiatement, quelque soit la valeur du paramètre `max_standby_streaming_delay`. Notez que **ALTER DATABASE ... RENAME** ne déconnecte pas les utilisateurs qui, dans la plupart des cas, ne s'en apercevront

pas. Cela peut néanmoins confondre un programme qui dépendrait du nom de la base.

En fonctionnement normal (pas en récupération), si vous exécutez **DROP USER** ou **DROP ROLE** pour un rôle ayant le privilège LOGIN alors que cet utilisateur est toujours connecté alors rien ne se produit pour cet utilisateur connecté - il reste connecté. L'utilisateur ne peut toutefois pas se reconnecter. Ce comportement est le même en récupération, un **DROP USER** sur le primaire ne déconnecte donc pas cet utilisateur sur le standby.

Le collecteur de statistiques est actif durant la récupération. Tous les parcours, lectures, utilisations de blocs et d'index, etc... seront enregistrés normalement sur le standby. Les actions rejouées ne dupliqueront pas leur effets sur le primaire, l'application d'insertions n'incrémentera pas la colonne Inserts de pg_stat_user_tables. Le fichier de statistiques est effacé au démarrage de la récupération, les statistiques du primaire et du standby différeront donc; c'est vu comme une fonctionnalité, pas un bug.

Autovacuum n'est pas actif durant la récupération, il démarrera normalement à la fin de la récupération.

Le processus d'écriture en arrière plan (background writer) est actif durant la récupération et effectuera les restartpoints (points de reprise) (similaires aux points de synchronisation ou checkpoints sur le primaire) et les activités normales de nettoyage de blocs. Ceci peut inclure la mise à jour des information de hint bit des données du serveur de standby. La commande **CHECKPOINT** est acceptée pendant la récupération, bien qu'elle déclenche un restartpoint et non un checkpoint.

25.5.4. Référence des paramètres de Hot Standby

De nombreux paramètres ont été mentionnés ci-dessus dans Section 25.5.2, « Gestion des conflits avec les requêtes » et Section 25.5.3, « Aperçu pour l'administrateur ».

Sur le primaire, les paramètres wal_level et vacuum_defer_cleanup_age peuvent être utilisés. max_standby_archive_delay et max_standby_streaming_delay n'ont aucun effet sur le primaire.

Sur le serveur en attente, les paramètres hot_standby, max_standby_archive_delay et max_standby_streaming_delay peuvent être utilisés. vacuum_defer_cleanup_age n'a pas d'effet tant que le serveur reste dans le mode standby, mais deviendra important quand le serveur en attente deviendra un serveur maître.

25.5.5. Avertissements

Il y a plusieurs limitations de Hot Standby. Elles peuvent et seront probablement résolues dans des versions ultérieures:

- Les opérations sur les index hash ne sont pas écrits dans la WAL à l'heure actuelle, la récupération ne mettra donc pas ces index à jour.
- Une connaissance complète des transactions en cours d'exécution est nécessaire avant de pouvoir déclencher des instantanés. Des transactions utilisant un grand nombre de sous-transactions (à l'heure actuelle plus de 64) retarderont le démarrage des connexions en lecture seule jusqu'à complétion de la plus longue transaction en écriture. Si cette situation se produit, des messages explicatifs seront envoyés dans la trace du serveur.
- Des points de démarrage valides pour les requêtes de standby sont générés à chaque checkpoint sur le maître. Si le standby est éteint alors que le maître est déjà éteint, il est tout à fait possible ne ne pas pouvoir repasser en Hot Standby tant que le primaire n'aura pas été redémarré, afin qu'il génère de nouveaux points de démarrage dans les journaux WAL. Cette situation n'est pas un problème dans la plupart des situations où cela pourrait se produire. Généralement, si le primaire est éteint et plus disponible, c'est probablement en raison d'un problème sérieux qui va de toutes façons forcer la conversion du standby en primaire. Et dans des situations où le primaire est éteint intentionnellement, la procédure standard est de promouvoir le maître.
- À la fin de la récupération, les `AccessExclusiveLocks` possédés par des transactions préparées nécessiteront deux fois le nombre d'entrées normal dans la table de verrous. Si vous pensez soit exécuter un grand nombre de transactions préparées prenant des `AccessExclusiveLocks`, ou une grosse transaction prenant beaucoup de `AccessExclusiveLocks`, il est conseillé d'augmenter la valeur de `max_locks_per_transaction`, peut-être jusqu'à une valeur double de celle du serveur primaire. Vous n'avez pas besoin de prendre ceci en compte si votre paramètre `max_prepared_transactions` est `0`.

Chapitre 26. Configuration de la récupération

Ce chapitre décrit les paramètres disponibles dans le fichier `recovery.conf`. Ils ne s'appliquent que pendant la durée de la récupération. Ils doivent être repositionnés pour toute récupération ultérieure que vous souhaiterez effectuer. Ils ne peuvent pas être modifiés une fois que la récupération a commencé.

Les paramètres de `recovery.conf` sont spécifiés dans le format `nom = 'valeur'`. Un paramètre est déclaré par ligne. Les caractères dièse (#) indiquent que le reste de la ligne est un commentaire. Pour inclure un guillemet dans une valeur de paramètre, écrivez deux guillemets (`''`).

Un fichier d'exemple , `share/recovery.conf.sample`, est fourni dans le répertoire `share/` de l'installation.

26.1. Paramètres de récupération de l'archive

`restore_command` (chaîne de caractères)

La commande d'interpréteur à exécuter pour récupérer un segment de la série de fichiers WAL archivés. Ce paramètre est nécessaire pour la récupération à partir de l'archive, mais optionnel pour la réplication à flux continu. Tout `%f` dans la chaîne est remplacé par le nom du fichier à récupérer de l'archive, et tout `%p` est remplacé par le chemin de destination de la copie sur le serveur. (Le chemin est relatif au répertoire courant de travail, c'est à dire le répertoire de données de l'instance.) Tout `%r` est remplacé par le nom du fichier contenant le dernier point de reprise (restartpoint) valide. Autrement dit, le fichier le plus ancien qui doit être gardé pour permettre à la récupération d'être redémarrable. Cette information peut donc être utilisée pour tronquer l'archive au strict minimum nécessaire pour permettre de reprendre la restauration en cours. `%r` n'est typiquement utilisé que dans des configurations de warn-standby. (voir Section 25.2, « Serveurs de Standby par transfert de journaux »). Écrivez `%%` pour inclure un vrai caractère `%`.

Il est important que la commande ne retourne un code retour égal à zéro que si elle réussit. La commande *recevra* des demandes concernant des fichiers n'existant pas dans l'archive; elle doit avoir un code retour différent de zéro dans ce cas. Par exemple:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

`archive_cleanup_command` (string)

Ce paramètre optionel spécifie une commande d'interpréteur qui sera exécuté à chaque point de reprise. Le but de `archive_cleanup_command` est de fournir un mécanisme de nettoyage des vieux fichiers WAL archivés qui ne sont plus nécessaires au serveur de standby. Tout `%r` est remplacé par le nom du fichier contenant le dernier point de reprise (restartpoint) valide. Autrement dit, le fichier le plus ancien qui doit être *conservé* pour permettre à la récupération d'être redémarrable. Du coup, tous les fichiers créés avant `%r` peuvent être supprimés sans problème. Cette information peut être utilisée pour tronquer les archives au minimum nécessaire pour redémarrer à partir de la restauration en L'outil pg_archivecleanup fourni en module `contrib` (voir Section F.22, « pg_archivecleanup ») sert de configuration simple pour `archive_cleanup_command` dans des configurations de standby seuls. Par exemple :

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

Notez néanmoins que si plusieurs serveurs en standby sont mis à jour à partir du même répertoire d'archives, vous devez vous assurer que vous ne supprimez que les journaux de transactions qui ne sont plus utiles à tous les serveurs. `archive_cleanup_command` n'est typiquement utilisé que dans des configurations de warm-standby (voir Section 25.2, « Serveurs de Standby par transfert de journaux »). Écrivez `%%` pour inclure un vrai caractère `%`.

Si la commande retourne un code de retour différent de zéro alors um message de journal WARNING sera écrit.

`recovery_end_command` (chaîne de caractéres)

Ce paramètre spécifie une commande d'interpréteur qui sera exécutée une fois seulement, à la fin de la récupération. Ce paramètre est optionnel. Le but de `recovery_end_command` est de fournir un mécanisme pour un nettoyage à la fin de la réplication ou de la récupération. Tout `%r` est remplacé par le nom du fichier contenant le dernier point de reprise valide, comme dans archive_cleanup_command.

Si la commande retourne un code de retour différent de zéro alors um message de journal WARNING sera écrit et la base continuera son démarrage malgré tout. Par contre, si la commande a été terminée par un signal, la base n'effectuera pas son démarrage.

26.2. Paramètres de cible de récupération

`recovery_target_time` (timestamp)
Ce paramètre spécifie l'horodatage (timestamp) jusqu'auquel la récupération se poursuivra. On ne peut spécifier qu'un seul des paramètres `recovery_target_time` et recovery_target_xid au plus. Par défaut, la récupération se poursuit jusqu'à la fin du journal WAL. Le point précis d'arrêt dépend aussi de recovery_target_inclusive.

`recovery_target_xid` (chaîne de caractères)
Ce paramètre spécifie l'identifiant de transaction jusqu'auquel la récupération se poursuivra. Gardez à l'esprit que, bien que les identifiants de transactions sont assignés séquentiellement au démarrage des transactions, elles peuvent se terminer dans un ordre numérique différent. Les transactions qui seront récupérées sont celles qui auront réalisé leur COMMIT avant la transaction spécifiée (optionnellement incluse). On ne peut spécifier qu'un seul des paramètres `recovery_target_time` et recovery_target_xid au plus. Par défaut, la récupération se poursuit jusqu'à la fin du journal WAL. Le point précis d'arrêt dépend aussi de recovery_target_inclusive.

`recovery_target_inclusive` (booléen)
Spécifie si il faut s'arrêter juste après la cible de récupération spécifiée (`true`), ou juste avant la cible de récupération (`false`). S'applique à recovery_target_time comme à recovery_target_xid, suivant celui qui est spécifié pour cette récupération. Ceci indique si les transaction qui ont exactement le même horodatage ou le même identifiant de commit, respectivement, seront inclues dans la récupération. La valeur par défaut est (`true`).

`recovery_target_timeline` (chaîne de caractères)
Spécifie la ligne de temps (timeline) précise sur laquelle effectuer la récupération. Le comportement par défaut est de récupérer sur la même timeline que celle en cours lorsque la sauvegarde de base a été effectuée. Vous n'aurez besoin de positionner ce paramètre que dans des cas complexes de re-récupération, où vous aurez besoin d'atteindre un état lui même atteint après une récupération à un moment dans le temps (point-in-time recovery). Voir Section 24.3.4, « Lignes temporelles (*Timelines*) » pour plus d'informations.

26.3. Paramètres de serveur de Standby

`standby_mode` (booléen)
Spécifie s'il faut démarrer le serveur PostgreSQL™ en tant que standby. Si ce paramètre est à `on`, le serveur n'arrête pas la récupération quand la fin du WAL archivé est atteinte, mais continue d'essayer de poursuivre la récupération en récupérant de nouveaux segments en utilisant `restore_command` et/ou en se connectant au serveur primaire comme spécifié par le paramètre `primary_conninfo`.

`primary_conninfo` (chaîne de caractères)
Spécifie au serveur de standby la chaîne de connexion à utiliser pour atteindre le primaire. Cette chaîne est dans le format accepté par la fonction `PQconnectdb` de la libpq, décrite dans Section 31.1, « Fonctions de contrôle de connexion à la base de données ». Si une option n'est pas spécifiée dans cette chaîne, alors la variable d'environnement correspondante (voir Section 31.13, « Variables d'environnement ») est examinée. Si la variable d'environnement n'est pas positionnée non plus, la valeur par défaut est utilisée.

La chaîne de connexion devra spécifier le nom d'hôte (ou adresse) du serveur primaire, ainsi que le numéro de port si ce n'est pas le même que celui par défaut du serveur de standby. Spécifiez aussi un nom d'utilisateur correspondant à un rôle qui a les privilèges `SUPERUSER` et `LOGIN` sur le primaire (voir Section 25.2.5.1, « Authentification »). Un mot de passe devra aussi être fourni, si le primaire demande une authentification par mot de passe. Il peut être fourni soit dans la chaîne `primary_conninfo` soit séparément dans un fichier `~/.pgpass` sur le serveur de standby (utilisez `replication` comme nom de base de données). Ne spécifiez pas de nom de base dans la chaîne`primary_conninfo`

Ce paramètre n'a aucun effet si `standby_mode` vaut `off`.

`trigger_file` (chaîne de caractères)
Spécifie un fichier trigger dont la présence met fin à la récupération du standby. Si aucun fichier trigger n'est spécifié, le standby ne quitte jamais la récupération. Ce paramètre n'a aucun effet si `standby_mode` vaut `off`.

Chapitre 27. Surveiller l'activité de la base de données

Un administrateur de bases de données se demande fréquemment : « Que fait le système en ce moment ? » Ce chapitre discute de la façon de le savoir.

Plusieurs outils sont disponibles pour surveiller l'activité de la base de données et pour analyser les performances. Une grande partie de ce chapitre concerne la description du récupérateur de statistiques de PostgreSQL™ mais personne ne devrait négliger les programmes de surveillance Unix standards tels que **ps**, **top**, **iostat** et **vmstat**. De plus, une fois qu'une requête peu performante a été identifiée, des investigations supplémentaires pourraient être nécessaires en utilisant la commande EXPLAIN(7) de PostgreSQL™. La Section 14.1, « Utiliser **EXPLAIN** » discute de **EXPLAIN** et des autres méthodes pour comprendre le comportement d'une seule requête.

27.1. Outils Unix standard

Sur la plupart des plateformes Unix, PostgreSQL™ modifie son titre de commande reporté par **ps** de façon à ce que les processus serveur individuels puissent être rapidement identifiés. Voici un affichage d'exemple :

```
$ ps auxww | grep ^postgres
postgres   960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00 postgres -i
postgres   963  0.0  1.1  7084 1472 pts/1    SN   13:17   0:00 postgres: writer
process
postgres   965  0.0  1.1  6152 1512 pts/1    SN   13:17   0:00 postgres: stats
collector process
postgres   998  0.0  2.3  6532 2992 pts/1    SN   13:18   0:00 postgres: tgl runbogue
127.0.0.1 idle
postgres  1003  0.0  2.4  6532 3128 pts/1    SN   13:19   0:00 postgres: tgl
regression [local] SELECT waiting
postgres  1016  0.1  2.4  6532 3080 pts/1    SN   13:19   0:00 postgres: tgl
regression [local] idle in transaction
```

(L'appel approprié de **ps** varie suivant les différentes plateformes, de même que les détails affichés. Cet exemple est tiré d'un système Linux récent.) Le premier processus affiché ici est le processus serveur maître, le processus serveur maître. Les arguments affichés pour cette commande sont les mêmes qu'à son lancement. Les deux processus suivant sont des processus en tâche de fond lancés automatiquement par le processus maître (le processus « stats collector » n'est pas présent si vous avez configuré le système pour qu'il ne lance pas le récupérateur de statistiques). Chacun des autres processus est un processus serveur gérant une connexion cliente. Tous ces processus restant initialisent l'affichage de la ligne de commande de la forme

```
postgres: utilisateur base_de_données hôte activité
```

L'utilisateur, la base de données et les éléments de l'hôte (client) restent identiques pendant toute la vie de connexion du client mais l'indicateur d'activité change. L'activité pourrait être `idle` (c'est-à-dire en attente d'une commande du client), `idle in transaction` (en attente du client à l'intérieur d'un bloc de **BEGIN**/**COMMIT**) ou un nom de commande du type `SELECT`. De plus, `waiting` est ajouté si le processus serveur est en attente d'un verrou détenu par une autre session. Dans l'exemple ci-dessus, nous pouvons supposer que le processus 1003 attend que le processus 1016 ait terminé sa transaction et, du coup, libère un verrou.

Si vous avez désactivé update_process_title, alors l'indicateur d'activité n'est pas mis à jour ; le titre du processus est configuré une seule fois quand un nouveau processus est lancé. Sur certaines plateformes, ceci permet d'économiser du temps. Sur d'autres, cette économie est insignifiante.

Astuce

Solaris™ requiert une gestion particulière. Vous devez utiliser **/usr/ucb/ps** plutôt que **/bin/ps**. Vous devez aussi utiliser deux options `w` et non pas seulement une. En plus, votre appel original de la commande **postgres** doit avoir un affichage de statut dans **ps** plus petit que celui fourni par les autres processus serveur. Si vous échouez dans les trois, l'affichage de **ps** pour chaque processus serveur sera la ligne de commande originale de **postgres**.

27.2. Le récupérateur de statistiques

Le *récupérateur de statistiques* de PostgreSQL™ est un sous-système qui supporte la récupération et les rapports d'informations sur l'activité du serveur. Actuellement, le récupérateur peut compter les accès aux tables et index à la fois en terme de blocs disque et de lignes individuelles. Il conserve aussi la trace du nombre total de lignes dans chaque table ainsi que la date et

l'heure du dernier VACUUM et du dernier ANALYZE pour chaque table. Il peut aussi compter le nombre d'appels aux fonctions définies par l'utilisateur ainsi que le temps total dépensé par chacune.

PostgreSQL™ supporte aussi la détermination de la commande exacte en cours d'exécution par les autres processus serveur. Cette fonctionnalité indépendante ne dépend pas du récupérateur de statistiques.

27.2.1. Configuration de la récupération de statistiques

Comme la récupération de statistiques ajoute un temps supplémentaire à l'exécution de la requête, le système peut être configuré pour récupérer ou non des informations. Ceci est contrôlé par les paramètres de configuration qui sont normalement initialisés dans `postgresql.conf` (voir Chapitre 18, Configuration du serveur pour plus de détails sur leur initialisation).

Le paramètre track_counts contrôle si les statistiques sont récupérées pour les accès aux tables et index.

Le paramètre track_functions active le calcul de statistiques sur l'utilisation des fonctions définies par l'utilisateur.

Le paramètre track_activities active la surveillance de la commande en cours d'exécution par un processus serveur.

Normalement, ces paramètres sont configurés dans `postgresql.conf` de façon à ce qu'ils s'appliquent à tous les processus serveur mais il est possible de les activer/désactiver sur des sessions individuelles en utilisant la commande SET(7) (pour empêcher les utilisateurs ordinaires de cacher leur activité à l'administrateur, seuls les superutilisateurs sont autorisés à modifier ces paramètres avec **SET**).

Le collecteur de statistiques communique avec les processus serveur (ainsi que l'autovacuum) via des fichiers temporaires. Ces fichiers sont stockées dans le sous-répertoire `pg_stat_tmp`. Quand le processus père est arrêté, une copie permanente des données statistiques est stockée dans le sous-répertoire `global`. Pour des performances accrues, le paramètre stats_temp_directory peut être pointé vers un système de fichiers en RAM, diminuant fortement les besoins en entrées/sorties.

27.2.2. Visualiser les statistiques récupérées

Plusieurs vues prédéfinies, listées dans le Tableau 27.1, « Vues statistiques standards », sont disponibles pour afficher les résultats de la récupération de statistiques. Autrement, vous pouvez construire des vues personnalisées en utilisant les fonctions statistiques existantes.

En utilisant les statistiques pour surveiller l'activité en cours, il est important de réaliser que l'information n'est pas mise à jour instantanément. Chaque processus serveur individuel transmet les nouvelles statistiques au récupérateur juste avant l'attente d'une nouvelle commande du client ; donc une requête toujours en cours n'affecte pas les totaux affichés. De plus, le récupérateur lui-même émet un nouveau rapport une fois par `PGSTAT_STAT_INTERVAL` millisecondes (500, sauf si cette valeur a été modifiée lors de la construction du serveur). Donc, les totaux affichés sont bien derrière l'activité réelle. Néanmoins, l'information sur la requête en cours récupérée par `track_activities` est toujours à jour.

Un autre point important est que, lorsqu'un processus serveur se voit demander d'afficher une des statistiques, il récupère tout d'abord le rapport le plus récent émis par le processus de récupération, puis continue d'utiliser cette image de toutes les vues et fonctions statistiques jusqu'à la fin de sa transaction en cours. De façon similaire, les informations sur les requêtes en cours, quelque soit le processus, sont récupérées quand une telle information est demandée dans une transaction, et cette même information sera affichée lors de la transaction. Donc, les statistiques afficheront des informations statiques tant que vous restez dans la même transaction. Ceci est une fonctionnalité, et non pas un bogue, car il vous permet de traiter plusieurs requêtes sur les statistiques et de corréler les résultats sans vous inquiéter que les nombres aient pu changer. Mais si vous voulez voir les nouveaux résultats pour chaque requête, assurez-vous de lancer les requêtes en dehors de tout bloc de transaction. Autrement, vous pouvez appeler `pg_stat_clear_snapshot()`, qui annulera l'image statistique de la transaction en cours. L'utilisation suivante des informations statistiques causera la récupération d'une nouvelle image.

Tableau 27.1. Vues statistiques standards

Nom de la vue	Description
pg_stat_activity	Une ligne par processus serveur, affichant l'OID de la base de données, le nom de la base, l'ID du processus, l'OID de l'utilisateur, son nom, le nom de l'application, l'adresse et le port du client, la date et l'heure à laquelle le client s'est connecté, de début de la transaction, et de début de la requête, le statut d'attente de verrou du processus et le texte de la requête en cours d'exécution. Les colonnes renvoyant des données sur la requête en cours sont disponibles sauf si le paramètre `track_activities` a été désactivé. De plus, ces colonnes sont seulement visibles si l'utilisateur examinant cette vue est un superutilisateur ou est le propriétaire du processus en cours de rapport.
pg_stat_bgwriter	Une seule ligne indiquant des statistiques du cluster complet provenant du processus d'écriture en tâche de fond : nombre de points de vérification planifiés, points de vérification demandés, tampons écrits par les points de vérification et parcours de nettoyage, et le nombre de fois où le processus d'écriture en tâche de fond a stoppé un parcours de nettoyage parce qu'il a écrit trop de tampons. Cela inclut aussi

Nom de la vue	Description
	des statistiques sur les tampons partagés dont le nombre de tampons écrit par les processus serveur (c'est-à-dire par autre chose que le processus d'écriture en tâche de fond) et le nombre total de tampons alloués.
pg_stat_database	Une ligne par base de données, affichant l'OID de la base de données, son nom, le nombre de processus serveur actifs connectés à cette base, le nombre total de transactions validées et le nombre de celles qui ont été annulées, le nombre total de blocs disque lus, le nombre total de succès du tampon (c'est-à-dire le nombre de lectures de blocs évitées en trouvant déjà le bloc dans le cache du tampon), le nombre de lignes renvoyées, récupérées, insérées, mises à jour et supprimées.
pg_stat_all_tables	Pour chaque table dans la base de données en cours (ceci incluant les tables TOAST), l'OID de la table, le nom du schéma et de la table, le nombre de parcours séquentiels réalisés, le nombre de lignes vivantes récupérées par des parcours séquentiels, le nombre de lignes vivantes récupérées par des parcours séquentiels, le nombre de parcours d'index réalisés (pour tous les index appartenant à cette table), le nombre de lignes vivantes récupérées par les parcours d'index, le nombre d'insertions, de modifications et de suppressions de ligne, le nombre de mises à jour de ligne via HOT (donc sans mise à jour séparée des index), le nombre de lignes vivantes et mortes, la dernière fois que la table a été la cible d'un VACUUM manuel, la dernière fois qu'elle a été la cible d'un VACUUM exécuté par le démon autovacuum, la dernière fois que la table a été la cible d'un ANALYZE manuel, la dernière fois qu'elle a été la cible d'un ANALYZE exécuté par le démon autovacuum.
pg_stat_sys_tables	Identique à pg_stat_all_tables, sauf que seules les tables systèmes sont affichées.
pg_stat_user_tables	Identique à pg_stat_all_tables, sauf que seules les tables utilisateurs sont affichées.
pg_stat_all_indexes	Pour chaque index de la base de données en cours, l'OID de la table et de l'index, le nom du schéma, de la table et de l'index, le nombre de parcours d'index initiés sur cet index, le nombre d'entrées de l'index renvoyées par les parcours d'index, et le nombre de lignes actives de table récupérées par de simples parcours d'index utilisant cet index.
pg_stat_sys_indexes	Identique à pg_stat_all_indexes, sauf que seules les tables systèmes sont affichées.
pg_stat_user_indexes	Identique à pg_stat_all_indexes, sauf que seules les tables utilisateurs sont affichées.
pg_statio_all_tables	Pour chaque table de la base de données en cours (ceci incluant les tables TOAST), l'OID de la table, le nom du schéma et de la table, le nombre de blocs disque lus à partir de cette table, le nombre de lectures tampon réussies dans tous les index de cette table, le nombre de blocs disque lus et de lectures tampon réussies à partir de la table TOAST (si elle existe), et, enfin, le nombre de blocs disque lus et le nombre de lectures tampon réussies à partir de l'index de la table TOAST.
pg_statio_sys_tables	Identique à pg_statio_all_tables, sauf que seules les tables systèmes sont affichées.
pg_statio_user_tables	Identique à pg_statio_all_tables, sauf que seules les tables utilisateur sont affichées.
pg_statio_all_indexes	Pour chaque index de la base de données en cours, l'OID de la table et de l'index, le nom du schéma, de la table et de l'index, le nombre de blocs disque lus et le nombre de lectures tampon réussies pour cet index.
pg_statio_sys_indexes	Identique à pg_statio_all_indexes, sauf que seuls les index systèmes sont affichés.
pg_statio_user_indexes	Identique à pg_statio_all_indexes, sauf que seuls les index utilisateur sont affichés.
pg_statio_all_sequences	Pour chaque séquence de la base de données en cours, l'OID de la séquence, le nom du schéma et de la séquence, le nombre de blocs disque lus et le nombre de lectures réussies du tampon pour cette séquence.
pg_statio_sys_sequences	Identique à pg_statio_all_sequences, sauf que seules les séquences système sont affichées (actuellement, aucune séquence système n'est définie, donc cette vue est toujours vide)
pg_statio_user_sequences	Identique à pg_statio_all_sequences, sauf que seules les séquences utilisateur sont affichées.
pg_stat_user_functions	Pour les fonctions tracées, OID de la fonction, schéma, nom, nombre d'appels, temps total et temps de la fonction. Ce dernier est le temps passé uniquement dans la fonction. Le temps total inclus le temps passé dans les fonctions appelées. Les valeurs sont en millisecondes.

Les statistiques par index sont particulièrement utiles pour déterminer les index utilisés et leur efficacité.

À partir de la version 8.1 de PostgreSQL™, les index peuvent être utilisés soit directement soit via des « parcours de bitmap ». Dans un parcours de bitmap, les résultats de plusieurs index peuvent être combinés via des règles AND ou OR ; donc il est difficile d'associer des récupérations de lignes d'en-têtes individuelles avec des index spécifiques quand un parcours de bitmap est uti-

lisé. Du coup, un parcours de bitmap incrémente le nombre dans pg_stat_all_indexes.*idx_tup_read* pour les index qu'il utilise et il incrémente le nombre pg_stat_all_tables.*idx_tup_fetch* pour la table, mais il n'affecte pas pg_stat_all_indexes.*idx_tup_fetch*.

Note

Avant PostgreSQL™ 8.1, les totaux *idx_tup_read* et *idx_tup_fetch* étaient pratiquement toujours égaux. Maintenant, ils peuvent être différents même sans considérer les parcours de bitmap parce que *idx_tup_read* compte les entrées d'index récupérées à partir de l'index alors que *idx_tup_fetch* compte les lignes actives récupérées à partir de la table ; ce dernier sera moindre si des lignes mortes ou pas-encore-validées sont récupérées en utilisant l'index.

Les vues pg_statio_ sont principalement utiles pour déterminer l'efficacité du cache tampon. Quand le nombre de lectures disques réelles est plus petit que le nombre de récupérations valides par le tampon, alors le cache satisfait la plupart des demandes de lecture sans faire appel au noyau. Néanmoins, ces statistiques ne nous donnent pas l'histoire complète : à cause de la façon dont PostgreSQL™ gère les entrées/sorties disque, les données qui ne sont pas dans le tampon de PostgreSQL™ pourraient toujours résider dans le tampon d'entrées/sorties du noyau et pourraient, du coup, être toujours récupérées sans nécessiter une lecture physique. Les utilisateurs intéressés pour obtenir des informations plus détaillées sur le comportement des entrées/sorties dans PostgreSQL™ sont invités à utiliser le récupérateur de statistiques de PostgreSQL™ avec les outils du système d'exploitation permettant une vue de la gestion des entrées/sorties par le noyau.

Il existe d'autres façons de regarder les statistiques. Cela se fait en écrivant des requêtes qui utilisent les mêmes fonctions d'accès aux statistiques que les vues standards. Ces fonctions sont listées dans le Tableau 27.2, « Fonctions d'accès aux statistiques ». Les fonctions d'accès par base de données prennent un OID de la base de données comme argument pour identifier la base de données du rapport. Les fonctions par table et par index prennent l'OID de la table ou de l'index. Les fonctions des statistiques pour les appels de fonctions prennent un OID. (notez que seuls les tables et les index de la base de données en cours peuvent être vus par ces fonctions). Les fonctions d'accès au processus prennent le numéro d'identifiant du processus.

Tableau 27.2. Fonctions d'accès aux statistiques

Fonction	Code de retour	Description
`pg_stat_get_db_numbackends(oid)`	integer	Nombre de processus actifs pour la base de données
`pg_stat_get_db_xact_commit(oid)`	bigint	Nombre de transactions validées dans la base de données
`pg_stat_get_db_xact_rollback(oid)`	bigint	Nombre de transactions annulées dans la base de données
`pg_stat_get_db_blocks_fetched(oid)`	bigint	Nombre de demandes de récupérations de blocs disque pour la base de données
`pg_stat_get_db_blocks_hit(oid)`	bigint	Nombre de demandes de récupérations de blocs disque trouvés dans le tampon pour la base de données
`pg_stat_get_db_tuples_returned(oid)`	bigint	Nombre de lignes renvoyées pour la base
`pg_stat_get_db_tuples_fetched(oid)`	bigint	Nombre de lignes récupérées pour la base
`pg_stat_get_db_tuples_inserted(oid)`	bigint	Nombre de lignes insérées dans la base
`pg_stat_get_db_tuples_updated(oid)`	bigint	Nombre de lignes mises à jour dans la base
`pg_stat_get_db_tuples_deleted(oid)`	bigint	Nombre de lignes supprimées dans la base
`pg_stat_get_numscans(oid)`	bigint	Nombre de parcours séquentiels réalisés lorsque l'argument est une table, ou nombre de parcours d'index lorsque l'argument est un index
`pg_stat_get_tuples_returned(oid)`	bigint	Nombre de lignes lues par les parcours séquentiels lorsque l'argument est une table, ou nombre de lignes d'index lues lorsque l'argument est un index
`pg_stat_get_tuples_fetched(oid)`	bigint	Le nombre de lignes de table récupérées par des parcours de bitmap quand

Fonction	Code de retour	Description
		l'argument est une table, ou les lignes de table récupérées par de simples parcours d'index en utilisant cet index quand l'argument est un index.
pg_stat_get_tuples_inserted(oid)	bigint	Nombre de lignes insérées dans la table
pg_stat_get_tuples_updated(oid)	bigint	Nombre de lignes mises à jour dans la table (incluant les mises à jour via HOT)
pg_stat_get_tuples_deleted(oid)	bigint	Nombre de lignes supprimées dans la table
pg_stat_get_tuples_hot_updated(oid)	bigint	Nombre de lignes mises à jour via HOT dans la table
pg_stat_get_live_tuples(oid)	bigint	Nombre de lignes vivantes dans la table
pg_stat_get_dead_tuples(oid)	bigint	Nombre de lignes mortes dans la table
pg_stat_get_blocks_fetched(oid)	bigint	Nombre de demandes de récupération de blocs disques pour la table ou l'index
pg_stat_get_blocks_hit(oid)	bigint	Nombre de demandes de blocs disque récupérés dans le tampon pour la table ou l'index
pg_stat_get_tuples_deleted(oid)	bigint	Nombre de lignes supprimées dans la table
pg_stat_get_blocks_fetched(oid)	bigint	Nombre de requêtes de récupération de blocs disque pour les tables ou index
pg_stat_get_blocks_hit(oid)	bigint	Nombre de requêtes de blocs disque trouvés en cache pour les tables ou index
pg_stat_get_last_vacuum_time(oid)	timestamptz	Date/heure du dernier VACUUM survenu sur cette table à la demande de l'utilisateur
pg_stat_get_last_autovacuum_time(oid)	timestamptz	Date/heure du dernier ANALYZE lancé par le démon autovacuum survenu sur cette table.
pg_stat_get_last_analyze_time(oid)	timestamptz	Date/heure du dernier VACUUM survenu sur cette table à la demande de l'utilisateur
pg_stat_get_last_autoanalyze_time(oid)	timestamptz	Date/heure du dernier ANALYZE lancé par le démon autovacuum survenu sur cette table.
pg_backend_pid()	integer	ID du processus pour le processus serveur attaché à la session en cours
pg_stat_get_activity(integer)	setof record	Renvoie un ensemble d'informations sur le processus serveur correspondant au PID indiqué, ou un enregistrement pour chaque processus serveur actif sur le système si le PID vaut NULL. Les champs renvoyés sont un sous-ensemble de ceux proposés par la vue pg_stat_activity
pg_stat_get_function_calls(oid)	bigint	Nombre d'appels de la fonction.
pg_stat_get_function_time(oid)	bigint	Temps passé dans la fonction, en microsecondes. Inclut le temps passé dans les fonctions appelées par cette fonction.
pg_stat_get_function_self_time(oid)	bigint	Temps passé uniquement dans cette fonction. Le temps passé dans les fonctions appelantes est exclu.
pg_stat_get_backend_idset()	setof integer	PID des processus serveurs actifs à ce moment (de 1 au nombre de processus serveurs actifs). Voir un exemple d'utilisation dans le texte.

Fonction	Code de retour	Description
`pg_stat_get_backend_pid(integer)`	integer	ID du processus pour le processus serveur donné
`pg_stat_get_backend_dbid(integer)`	oid	ID de la base de données pour le processus serveur en cours
`pg_stat_get_backend_userid(integer)`	oid	ID de l'utilisateur pour le processus serveur en cours
`pg_stat_get_backend_activity(integer)`	text	Commande active du processus serveur indiqué mais seulement si l'utilisateur courant est un superutilisateur ou le même utilisateur dont vient la commande (et que `track_activities` est activé)
`pg_stat_get_backend_waiting(integer)`	boolean	True si le processus serveur indiqué attend un verrou mais seulement si l'utilisateur courant est un superutilisateur ou le même utilisateur dont vient la commande (et que `track_activities` est activé)
`pg_stat_get_backend_activity_start(integer)`	timestamp with time zone	Date/heure du lancement de la requête en cours d'exécution sur le processus serveur indiqué, mais seulement si l'utilisateur courant est un superutilisateur ou le même utilisateur dont vient la commande (et que `track_activities` est activé)
`pg_stat_get_backend_xact_start(integer)`	timestamp with time zone	Le moment auquel le processus serveur indiqué a été exécuté. Seulement si l'utilisateur est un superutilisateur ou le même utilisateur que celui qui a lancé la session (et que `track_activities` est activé)
`pg_stat_get_backend_start(integer)`	timestamp with time zone	L'heure à laquelle le processus serveur donné a été lancé ou NULL si l'utilisateur en cours n'est ni un superutilisateur ni l'utilisateur de la session requêtée
`pg_stat_get_backend_client_addr(integer)`	inet	L'adresse IP du client connecté au processus serveur donné. NULL si la connexion est établie sur un socket de domaine Unix. Aussi NULL si l'utilisateur en cours n'est ni un superutilisateur ni l'utilisateur de la session requêtée
`pg_stat_get_backend_client_port(integer)`	integer	Le numéro de port TCP du client connecté au processus serveur donné. -1 si la connexion est établie sur un socket de domaine Unix. NULL si l'utilisateur en cours n'est ni un superutilisateur ni l'utilisateur de la session requêtée
`pg_stat_get_bgwriter_timed_checkpoints()`	bigint	Le nombre de fois où le processus d'écriture en tâche de fond a lancé des points de vérification planifiés (donc parce que `checkpoint_timeout` est arrivé à expiration)
`pg_stat_get_bgwriter_requested_checkpoints()`	bigint	Nombre de fois où le processus d'écriture en tâche de fond a lancé des points de vérification en se basant sur les demandes des processus serveur parce que `checkpoint_segments` a été dépassé ou parce que la commande **CHECKPOINT** a été lancée

Fonction	Code de retour	Description
`pg_stat_get_bgwriter_buf_written_checkpoints()`	bigint	Nombre de tampons écrits par le processus d'écriture en tâche de fond lors de points de vérification
`pg_stat_get_bgwriter_buf_written_clean()`	bigint	Nombre de tampons écrits par le processus d'écriture en tâche de fond pour le nettoyage de routine des pages sales
`pg_stat_get_bgwriter_maxwritten_clean()`	bigint	Nombre de fois où le processus d'écriture en tâche de fond est arrêté son parcours de nettoyage parce qu'il a écrit plus de tampons que ce qui est spécifié par le paramètre `bgwriter_lru_maxpages`
`pg_stat_get_buf_written_backend()`	bigint	Nombre de tampons écrits par les processus serveur parce qu'ils ont besoin d'allouer un nouveau tampon
`pg_stat_get_buf_alloc()`	bigint	Le nombre total d'allocations de tampons
`pg_stat_clear_snapshot()`	void	Annule l'image statistique actuelle
`pg_stat_reset()`	void	Réinitialise à zéro tous les compteurs statistiques pour la base de données actuelle (nécessite les droits superutilisateur)
`pg_stat_reset_shared(text)`	void	Réinitialise des compteurs de statistiques partagés pour le cluster de base de données à zéro (nécessite les droits du superutilisateur). Appeler `pg_stat_reset_shared('bgwriter')` mettra à zéro toutes les valeurs affichées par pg_stat_bgwriter.
`pg_stat_reset_single_table_counters(oid)`	void	Réinitialise les statistiques pour une table ou un index particulier dans la base de données en cours (nécessite les droits du superutilisateur).
`pg_stat_reset_single_function_counters(oid)`	void	Réinitialise les statistiques pour une fonction particulière dans la base de données en cours (nécessite les droits du superutilisateur).

Note

`pg_stat_get_blocks_fetched` moins `pg_stat_get_blocks_hit` donne le nombre d'appels lancés pour la table, l'index ou la base de données ; mais le nombre de lectures physiques réelles est habituellement moindre à cause des tampons du noyau. Les colonnes statistiques `*_blks_read` utilisent cette soustraction, c'est-à-dire lus en cache soustrait aux lus sur disque.

Toutes les fonctions accédant aux informations sur les processus sont indexées par numéro d'identifiant, sauf que `pg_stat_get_activity` est indexé par PID. La fonction `pg_stat_get_backend_idset` fournit un moyen agréable de générer une ligne pour chaque processus serveur actif. Par exemple, pour afficher les PID et les requêtes en cours pour tous les processus serveur :

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

27.3. Visualiser les verrous

Un autre outil utile pour surveiller l'activité des bases de données est la table système pg_locks. Elle permet à l'administrateur système de visualiser des informations sur les verrous restant dans le gestionnaire des verrous. Par exemple, cette fonctionnalité peut

être utilisée pour :

- Visualiser tous les verrous en cours, tous les verrous sur les relations d'une base de données particulière ou tous les verrous détenus par une session PostgreSQL™ particulière.
- Déterminer la relation de la base de données disposant de la plupart des verrous non autorisés (et qui, du coup, pourraient être une source de contention parmi les clients de la base de données).
- Déterminer l'effet de la contention des verrous sur les performances générales des bases de données, ainsi que l'échelle dans laquelle varie la contention sur le trafic de la base de données.

Les détails sur la vue pg_locks apparaissent dans la Section 45.50, « pg_locks ». Pour plus d'informations sur les verrous et la gestion des concurrences avec PostgreSQL™, référez-vous au Chapitre 13, Contrôle d'accès simultané.

27.4. Traces dynamiques

PostgreSQL™ fournit un support pour les traces dynamiques du serveur de bases de données. Ceci permet l'appel à un outil externe à certains points du code pour tracer son exécution.

Un certain nombre de sondes et de points de traçage sont déjà insérés dans le code source. Ces sondes ont pour but d'être utilisées par des développeurs et des administrateurs de base de données. Par défaut, les sondes ne sont pas compilées dans PostgreSQL™ ; l'utilisateur a besoin de préciser explicitement au script configure de rendre disponible les sondes.

Actuellement, seul l'outil *DTrace*, disponible sur OpenSolaris, Solaris 10 et Mac OS X Leopard, est supporté. DTrace devrait être disponible pour FreeBSD dans le futur et peut-être pour d'autres systèmes d'exploitation.Le projet *SystemTap* pour Linux fournit aussi un équivalent DTrace. Le support d'autres outils de traces dynamiques est possible théoriquement en modifiant les définitions des macros dans `src/include/utils/probes.h`.

27.4.1. Compiler en activant les traces dynamiques

Par défaut, les sondes ne sont pas disponibles, donc vous aurez besoin d'indiquer explicitement au script configure de les activer dans PostgreSQL™. Pour inclure le support de DTrace, ajoutez `--enable-dtrace` aux options de configure. Lire Section 15.5, « Procédure d'installation » pour plus d'informations.

27.4.2. Sondes disponibles

Un certain nombre de sondes standards sont fournies dans le code source, comme indiqué dans Tableau 27.3, « Sondes disponibles pour DTrace ». D'autres peuvent être ajoutées pour améliorer la surveillance de PostgreSQL™.

Tableau 27.3. Sondes disponibles pour DTrace

Nom	Paramètres	Aperçu
transaction-start	(LocalTransactionId)	Sonde qui se déclenche au lancement d'une nouvelle transaction. arg0 est l'identifiant de transaction.
transaction-commit	(LocalTransactionId)	Sonde qui se déclenche quand une transaction se termine avec succès. arg0 est l'identifiant de transaction.
transaction-abort	(LocalTransactionId)	Sonde qui se déclenche quand une transaction échoue. arg0 est l'identifiant de transaction.
query-start	(const char *)	Sonde qui se déclenche lorsque le traitement d'une requête commence. arg0 est la requête.
query-done	(const char *)	Sonde qui se déclenche lorsque le traitement d'une requête se termine. arg0 est la requête.
query-parse-start	(const char *)	Sonde qui se déclenche lorsque l'analyse d'une requête commence. arg0 est la requête.
query-parse-done	(const char *)	Sonde qui se déclenche lorsque l'analyse d'une requête se termine. arg0 est la re-

Nom	Paramètres	Aperçu
		quête.
query-rewrite-start	(const char *)	Sonde qui se déclenche lorsque la réécriture d'une requête commence. arg0 est la requête.
query-rewrite-done	(const char *)	Sonde qui se déclenche lorsque la réécriture d'une requête se termine. arg0 est la requête.
query-plan-start	()	Sonde qui se déclenche lorsque la planification d'une requête commence.
query-plan-done	()	Sonde qui se déclenche lorsque la planification d'une requête se termine.
query-execute-start	()	Sonde qui se déclenche lorsque l'exécution d'une requête commence.
query-execute-done	()	Sonde qui se déclenche lorsque l'exécution d'une requête se termine.
statement-status	(const char *)	Sonde qui se déclenche à chaque fois que le processus serveur met à jour son statut dans pg_stat_activity.`current_query`. arg0 est la nouvelle chaîne de statut.
checkpoint-start	(int)	Sonde qui se déclenche quand un point de retournement commence son exécution. arg0 détient les drapeaux bit à bit utilisés pour distingurer les différents types de points de retournement, comme un point suite à un arrêt, un point immédiat ou un point forcé.
checkpoint-done	(int, int, int, int, int)	Sonde qui se déclenche quand un point de retournement a terminé son exécution (les sondes listées après se déclenchent en séquence lors du traitement d'un point de retournement). arg0 est le nombre de tampons mémoires écrits. arg1 est le nombre total de tampons mémoires. arg2, arg3 et arg4 contiennent respectivement le nombre de journaux de transactions ajoutés, supprimés et recyclés.
clog-checkpoint-start	(bool)	Sonde qui se déclenche quand la portion CLOG d'un point de retournement commence. arg0 est true pour un point de retournement normal, false pour un point de retournement suite à un arrêt.
clog-checkpoint-done	(bool)	Sonde qui se déclenche quand la portion CLOG d'un point de retournement commence. arg0 a la même signification que pour clog-checkpoint-start.
subtrans-checkpoint-start	(bool)	Sonde qui se déclenche quand la portion SUBTRANS d'un point de retournement commence. arg0 est true pour un point de retournement normal, false pour un point de retournement suite à un arrêt.
subtrans-checkpoint-done	(bool)	Sonde qui se déclenche quand la portion SUBTRANS d'un point de retournement se termine. arg0 a la même signification que pour subtrans-checkpoint-start.
multixact-checkpoint-start	(bool)	Sonde qui se déclenche quand la portion MultiXact d'un point de retournement

Nom	Paramètres	Aperçu
		commence. arg0 est true pour un point de retournement normal, false pour un point de retournement suite à un arrêt.
multixact-checkpoint-done	(bool)	Sonde qui se déclenche quand la portion MultiXact d'un point de retournement se termine. arg0 a la même signification que pour multixact-checkpoint-start.
buffer-checkpoint-start	(int)	Sonde qui se déclenche quand la portion d'écriture de tampons d'un point de retournement commence. arg0 contient les drapeaux bit à bit pour distinguer différents types de point de retournement comme le point après arrêt, un point immédiat, un point forcé.
buffer-sync-start	(int, int)	Sonde qui se déclenche quand nous commençons d'écrire les tampons modifiés pendant un point de retournement (après identification des tampons qui doivent être écrits). arg0 est le nombre total de tampons. arg1 est le nombre de tampons qui sont modifiés et n'ont pas besoin d'être écrits.
buffer-sync-written	(int)	Sonde qui se déclenche après chaque écriture d'un tampon lors d'un point de retournement. arg0 est le numéro d'identifiant du tampon.
buffer-sync-done	(int, int, int)	Sonde qui se déclenche quand tous les tampons modifiés ont été écrits. arg0 est le nombre total de tampons. arg1 est le nombre de tampons réellement écrits par le processus de point de retournement. arg2 est le nombre attendu de tampons à écrire (arg1 de buffer-sync-start) ; toute différence reflète d'autres processus écrivant des tampons lors du point de retournement.
buffer-checkpoint-sync-start	()	Sonde qui se déclenche une fois les tampons modifiés écrits par le noyau et avant de commencer à lancer des requêtes fsync.
buffer-checkpoint-done	()	Sonde qui se déclenche après la fin de la synchronisation des tampons sur le disque.
twophase-checkpoint-start	()	Sonde qui se déclenche quand la portion deux-phases d'un point de retournement est commencée.
twophase-checkpoint-done	()	Sonde qui se déclenche quand la portion deux-phases d'un point de retournement est terminée.
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, bool, bool)	Sonde qui se déclenche quand la lecture d'un tampon commence. arg0 et arg1 contiennent les numéros de fork et de bloc de la page (arg1 vaudra -1 s'il s'agit de demande d'extension de la relation). arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est true pour un

Nom	Paramètres	Aperçu
		tampon local, false pour un tampon partagé. arg6 est true pour une demande d'extension de la relation, false pour une lecture ordinaire.
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, bool, bool, bool)	Sonde qui se déclenche quand la lecture d'un tampon se termine. arg0 et arg1 contiennent les numéros de fork et de bloc de la page (arg1 contient le numéro de bloc du nouveau bloc ajouté s'il s'agit de demande d'extension de la relation). arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est true pour un tampon local, false pour un tampon partagé. arg6 est true pour une demande d'extension de la relation, false pour une lecture ordinaire. arg7 est true si la tampon était disponible en mémoire, false sinon.
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Sonde qui se déclenche avant de lancer une demande d'écriture pour un bloc partagé. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation.
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Sonde qui se déclenche quand une demande d'écriture se termine. (Notez que ceci ne reflète que le temps passé pour fournir la donnée au noyau ; ce n'est habituellement pas encore écrit sur le disque.) Les arguments sont identiques à ceux de buffer-flush-start.
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Sonde qui se déclenche quand un processus serveur commence à écrire un tampon modifié sur disque. Si cela arrive souvent, cela implique que shared_buffers est trop petit ou que les paramètres de contrôle de bgwriter ont besoin d'un ajustement.) arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation.
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Sonde qui se déclenche quand l'écriture d'un tampon modifié est terminé. Les arguments sont identiques à ceux de buffer-write-dirty-start.
wal-buffer-write-dirty-start	()	Sonde qui se déclenche quand un processus serveur commence à écrire un tampon modifié d'un journal de transactions parce qu'il n'y a plus d'espace disponible dans le cache des journaux de transactions. (Si cela arrive souvent, cela implique que wal_buffers est trop petit.)
wal-buffer-write-dirty-done	()	Sonde qui se déclenche quand l'écriture d'un tampon modifié d'un journal de transactions est terminé.

Nom	Paramètres	Aperçu
xlog-insert	(unsigned char, unsigned char)	Sonde qui se déclenche quand un enregistrement est inséré dans un journal de transactions. arg0 est le gestionnaire de ressource (rmid) pour l'enregistrement. arg1 contient des informations supplémentaires.
xlog-switch	()	Sonde qui se déclenche quand une bascule du journal de transactions est demandée.
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Sonde qui se déclenche au début de la lecture d'un bloc d'une relation. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation.
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int)	Sonde qui se déclenche à la fin de la lecture d'un bloc. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est le nombre d'octets réellement lus, alors qu'arg6 est le nombre d'octets demandés (s'il y a une différence, il y a un problème).
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Sonde qui se déclenche au début de l'écriture d'un bloc dans une relation. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation.
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int)	Sonde qui se déclenche à la fin de l'écriture d'un bloc. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est le nombre d'octets réellement écrits, alors qu'arg6 est le nombre d'octets demandés (si ces nombres sont différents, cela indique un problème).
sort-start	(int, bool, int, int, bool)	Sonde qui se déclenche quand une opération de tri est démarré. arg0 indique un tri de la table, de l'index ou d'un datum. arg1 est true si on force les valeurs uniques. arg2 est le nombre de colonnes clés. arg3 est le nombre de Ko de mémoire autorisé pour ce travail. arg4 est true si un accès aléatoire au résultat du tri est requis.
sort-done	(bool, long)	Sonde qui se déclenche quand un tri est terminé. arg0 est true pour un tri externe, false pour un tri interne. arg1 est le nombre de blocs disque utilisés pour un tri externe, ou le nombre de Ko de mémoire utilisés pour un tri interne.

Nom	Paramètres	Aperçu
lwlock-acquire	(LWLockId, LWLockMode)	Sonde qui se déclenche quand un LWLock a été acquis. arg0 est l'identifiant du LWLock. arg1 est le mode de verrou demandé, soit exclusif soit partagé.
lwlock-release	(LWLockId)	Sonde qui se déclenche quand un LWLock a été relâché (mais notez que tout processus en attente n'a pas encore été réveillé). arg0 est l'identifiant du LWLock.
lwlock-wait-start	(LWLockId, LWLockMode)	Sonde qui se déclenche quand un LWLock n'était pas immédiatement disponible et qu'un processus serveur a commencé à attendre la disponibilité du verrou. arg0 est l'identifiant du LWLock. arg1 est le mode de verrou demandé, soit exclusif soit partagé.
lwlock-wait-done	(LWLockId, LWLockMode)	Sonde qui se déclenche quand un processus serveur n'est plus en attente d'un LWLock (il n'a pas encore le verrou). arg0 est l'identifiant du LWLock. arg1 est le mode de verrou demandé, soit exclusif soit partagé.
lwlock-condacquire	(LWLockId, LWLockMode)	Sonde qui se déclenche quand un LWLock a été acquis avec succès malgré le fait que l'appelant ait demandé de ne pas attendre. arg0 est l'identifiant du LWLock. arg1 est le mode de verrou demandé, soit exclusif soit partagé.
lwlock-condacquire-fail	(LWLockId, LWLockMode)	Sonde qui se déclenche quand un LWLock, demandé sans attente, n'est pas accepté. arg0 est l'identifiant du LWLock. arg1 est le mode de verrou demandé, soit exclusif soit partagé.
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Sonde qui se déclenche quand une demande d'un gros verrou (*lmgr lock*) a commencé l'attente parce que le verrou n'était pas disponible. arg0 à arg3 sont les chmps identifiant l'objet en cours de verrouillage. arg4 indique le type d'objet à verrouiller. arg5 indique le type du verrou demandé.
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Sonde qui se déclenche quand une demande d'un gros verrou (*lmgr lock*) a fini d'attendre (c'est-à-dire que le verrou a été accepté). Les arguments sont identiques à ceux de lock-wait-start.
deadlock-found	()	Sonde qui se déclenche quand un verrou mortel est trouvé par le détecteur.

Tableau 27.4. Types définis utilisés comme paramètres de sonde

Type	Definition
LocalTransactionId	unsigned int
LWLockId	int
LWLockMode	int
LOCKMODE	int

Type	Definition
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	char

27.4.3. Utiliser les sondes

L'exemple ci-dessous montre un script DTrace pour l'analyse du nombre de transactions sur le système, comme alternative à l'interrogation régulière de pg_stat_database avant et après un test de performance :

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

À son exécution, le script de l'exemple D donne une sortie comme :

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                          2312105013
```

> **Note**
>
> SystemTap utilise une notation différente de DTrace pour les scripts de trace, même si les points de trace sont compatibles. Il est intéressant de noter que, lorsque nous avons écrit ce texte, les scripts SystemTap doivent référencer les noms des sondes en utilisant des tirets bas doubles à la place des tirets simples. Il est prévu que les prochaines versions de SystemTap corrigent ce problème.

Vous devez vous rappeler que les programmes DTrace doivent être écrits soigneusement, sinon les informations récoltées pourraient ne rien valoir. Dans la plupart des cas où des problèmes sont découverts, c'est l'instrumentation qui est erronée, pas le système sous-jacent. En discutant des informations récupérées en utilisant un tel système, il est essentiel de s'assurer que le script utilisé est lui-aussi vérifié et discuter.

D'autres exemples de scripts sont disponibles dans le *projet dtrace* de PgFoundry.

27.4.4. Définir de nouvelles sondes

De nouvelles sondes peuvent être définies dans le code partout où le développeur le souhaite bien que cela nécessite une nouvelle compilation. Voici les étapes nécessaires pour insérer de nouvelles sondes :

1. Décider du nom de la sonde et des données nécessaires pour la sonde
2. Ajoutez les définitions de sonde dans `src/backend/utils/probes.d`

3. Inclure `pg_trace.h` s'il n'est pas déjà présent dans le module contenant les points de sonde, et insérer les macros `TRACE_POSTGRESQL` aux emplacements souhaités dans le code source
4. Recompiler et vérifier que les nouvelles sondes sont disponibles

Exemple : Voici un exemple d'ajout d'une sonde pour tracer toutes les nouvelles transactions par identifiant de transaction.

1. La sonde sera nommée `transaction-start` et nécessite un paramètre de type LocalTransactionId
2. Ajout de la définition de la sonde dans `src/backend/utils/probes.d` :

   ```
   probe transaction__start(LocalTransactionId);
   ```

   Notez l'utilisation du double tiret bas dans le nom de la sonde. Dans un script DTrace utilisant la sonde, le double tiret bas doit être remplacé par un tiret, donc `transaction-start` est le nom à documenter pour les utilisateurs.
3. Au moment de la compilation, `transaction__start` est converti en une macro appelée `TRACE_POSTGRESQL_TRANSACTION_START` (notez que les tirets bas ne sont plus doubles ici), qui est disponible en incluant le fichier `pg_trace.h`. Il faut ajouter l'appel à la macro aux bons emplacements dans le code source. Dans ce cas, cela ressemble à :

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```

4. Après une nouvelle compilation et l'exécution du nouveau binaire, il faut vérifier que la nouvelle sonde est disponible en exécutant la commande DTrace suivante. Vous deviez avoir cette sortie :

   ```
   # dtrace -ln transaction-start
      ID    PROVIDER          MODULE           FUNCTION NAME
   18705 postgresql49878     postgres     StartTransactionCommand transaction-start
   18755 postgresql49877     postgres     StartTransactionCommand transaction-start
   18805 postgresql49876     postgres     StartTransactionCommand transaction-start
   18855 postgresql49875     postgres     StartTransactionCommand transaction-start
   18986 postgresql49873     postgres     StartTransactionCommand transaction-start
   ```

Il faut faire attention à d'autres choses lors de l'ajout de macros de trace dans le code C :

- Vous devez faire attention au fait que les types de données indiqués pour les paramètres d'une sonde correspondent aux types de données des variables utilisées dans la macro. Dans le cas contraire, vous obtiendrez des erreurs de compilation.
- Sur la plupart des platformes, si PostgreSQL™ est construit avec `--enable-dtrace`, les arguments pour une macro de trace seront évalués à chaque fois que le contrôle passe dans la macro, *même si aucun traçage n'est réellement en cours*. Cela a généralement peu d'importance si vous rapportez seulement les valeurs de quelques variables locales mais faites bien attention à l'utilisation de fonctions coûteuses. Si vous devez le faire, pensez à protéger la macro avec une vérification pour vous assurer que la trace est bien activée :

  ```
  if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
      TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
  ```

  Chaque macro de trace a une macro `ENABLED` correspondante.

Chapitre 28. Surveiller l'utilisation des disques

Ce chapitre explique comment surveiller l'utilisation que fait PostgreSQL™ des disques.

28.1. Déterminer l'utilisation des disques

Chaque table possède un fichier principal dans lequel la majorité des données sont stockées. Si la table contient des colonnes pouvant recevoir des valeurs étendues, il pourrait aussi y avoir un fichier TOAST associé à la table. Ce fichier permet de stocker les valeurs trop larges pour tenir dans la table principale (voir la Section 54.2, « TOAST »). Si la table TOAST existe, un index lui est associé. Des index peuvent également être associés à la table de base. Chaque table ou index est stocké dans un fichier distinct -- ou plusieurs si la taille du fichier dépasse 1 Go. Les conventions de nommage de ces fichiers sont décrites dans la Section 54.1, « Emplacement des fichiers de la base de données ».

L'espace disque peut être surveillé de trois façons différentes : en utilisant les fonctions SQL listées dans Tableau 9.58, « Fonctions de calcul de la taille des objets de la base de données », en utilisant les outils de `contrib/oid2name` ou en inspectant manuellement les catalogues système. Les fonctions SQL sont les plus simples à utiliser et sont généralement recommandées. `contrib/oid2name` est décrit dans Section F.19, « oid2name ». Le reste de cette section montre comment le faire en inspectant les catalogues système.

L'utilisation de psql sur une base de données récemment « nettoyée » (**VACUUM**) ou « analysée » (**ANALYZE**) permet de lancer des requêtes pour connaître l'occupation disque d'une table :

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Chaque page a une taille de 8 Ko, typiquement. (Rappelez-vous que *`relpages`* est seulement mis à jour par **VACUUM**, **ANALYZE** et quelques commandes DDL telles que **CREATE INDEX**.) Le chemin du fichier n'a d'intérêt que si vous voulez examiner directement le fichier de la table.

Pour connaître l'espace disque utilisé par les tables TOAST, on utilise une requête similaire à la suivante :

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT reltoastidxid
             FROM pg_class
             WHERE oid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

On peut aussi facilement afficher la taille des index :

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

Les tables et les index les plus volumineux sont repérés à l'aide de la requête suivante :

```
SELECT relname, relpages
FROM pg_class
```

```
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

28.2. Panne pour disque saturé

La tâche la plus importante d'un administrateur de base de données, en ce qui concerne la surveillance des disques, est de s'assurer que les disques n'arrivent pas à saturation. Un disque de données plein ne corrompt pas les données mais peut empêcher toute activité. S'il s'agit du disque contenant les fichier WAL, une alerte PANIC et un arrêt du serveur peuvent survenir.

S'il n'est pas possible de libérer de la place sur le disque, il faut envisager le déplacement de quelques fichiers vers d'autres systèmes de fichiers à l'aide des *tablespaces*. Voir la Section 21.6, « Tablespaces » pour plus d'informations.

Astuce

Certains systèmes de fichiers réagissent mal à proximité des limites de remplissage. Il est donc préférable de ne pas attendre ce moment pour réagir.

Si le système supporte les quotas disque par utilisateur, la base de données est alors soumise au quota de l'utilisateur qui exécute le serveur de base de données. Dépasser ce quota a les mêmes conséquences néfastes qu'un disque plein.

Chapitre 29. Fiabilité et journaux de transaction

Ce chapitre explique comment les journaux de transaction sont utilisés pour obtenir des traitements efficaces et fiables.

29.1. Fiabilité

La fiabilité est une propriété importante de tout système de base de données sérieux. PostgreSQL™ fait tout ce qui est en son pouvoir pour garantir une fiabilité à toute épreuve. Un des aspects de cette fiabilité est que toutes les données enregistrées par une transaction validée doivent être stockées dans un espace non volatile, un espace non sensible aux coupures de courant, aux bogues du système d'exploitation et aux problèmes matériels (sauf en cas de problème sur l'espace non volatile, bien sûr). Écrire avec succès les données sur le stockage permanent de l'ordinateur (disque dur ou un équivalent) est habituellement suffisant pour cela. En fait, même si un ordinateur est vraiment endommagé, si le disque dur survit, il peut être placé dans un autre ordinateur avec un matériel similaire et toutes les transactions validées resteront intactes.

Bien que forcer l'enregistrement des données périodiquement sur le disque semble être une opération simple, ce n'est pas le cas. Comme les disques durs sont beaucoup plus lents que la mémoire principale et les processeurs, plusieurs niveaux de cache existent entre la mémoire principale de l'ordinateur et les disques. Tout d'abord, il existe le tampon cache du système d'exploitation, qui met en cache les blocs disque fréquemment utilisés et combine les écritures sur le disque. Heureusement, tous les systèmes d'exploitation donne un moyen de forcer les écritures du cache disque vers le disque et PostgreSQL™ utilise ces fonctions (voir le paramètre wal_sync_method pour voir comment cela se fait).

Ensuite, il pourrait y avoir un cache dans le contrôleur du disque dur ; ceci est assez commun sur les cartes contrôleur RAID. Certains de ces caches sont *write-through*, signifiant que les écritures sont envoyées au lecteur dès qu'elles arrivent. D'autres sont *write-back*, signifiant que les données sont envoyées au lecteur un peu après. De tels caches peuvent apporter une faille dans la fiabilité car la mémoire du cache du disque contrôleur est volatile et qu'elle perdra son contenu à la prochaine coupure de courant. Des cartes contrôleur de meilleure qualité ont des caches *avec batterie* (BBU), signifiant que la carte dispose d'une batterie qui maintient le courant dans le cache en cas de perte de courant. Une fois le courant revenu, les données seront écrites sur les disques durs.

Et enfin, la plupart des disques durs ont des caches. Certains sont « write-through » alors que d'autres sont « write-back ». Les mêmes soucis sur la perte de données existent pour ces deux types de cache. Les lecteurs IDE ont principalement des caches « write-back » qui ne survivront pas à une perte de courantMany solid-state drives (SSD) also have volatile write-back caches.

Ces caches peuvent typiquement être désactivés. Néanmoins, la méthode pour le faire dépend du système d'exploitation et du type de disque :

- Sur Linux™, les disques IDE peuvent être vérifiés avec la commande **hdparm -I** ; le cache en écriture est activé si une étoile (`*`) se trouve derrière le texte `Write cache`. **hdparm -W** peut être utilisé pour désactiver le cache en écriture. Les disques SCSI peuvent être vérifiés en utilisant *sdparm*. Utilisez **sdparm --get=WCE** pour vérifier si le cache en écriture est activé et **sdparm --clear=WCE** pour le désactiver.
- Sur FreeBSD™, les disques IDE peuvent être vérifiés avec **atacontrol**, et les disques SCSI avec **sdparm**.
- Sur Solaris™, le cache disque en écriture est contrôlé par `format -e`. (Le système de fichiers Solaris ZFS est sûr, y compris quand le cache disque en écriture est activé car il exécute ses propres commandes de vidage du cache.)
- Sur Windows™, si `wal_sync_method` vaut `open_datasync` (la valeur par défaut), le cache en écriture peut être désactivé en décochant `My Computer\Open\`*`disk drive`*`\Properties\Hardware\Properties\Policies\Enable write caching on the disk`. Sinon configurez `wal_sync_method` à `fsync` ou `fsync_writethrough` pour désactiver le cache en écriture.
- Sur Mac OS X™, le cache en écriture peut être évité en configurant `wal_sync_method` à `fsync_writethrough`.

Les disques SATA récents (ceux compatibles ATAPI-6 ou supérieurs) proposent une commande pour vider le cache sur le disque (**FLUSH CACHE EXT**) alors que les disques SCSI proposent depuis longtemps une commande similaire, **SYNCHRONIZE CACHE**. Ces commandes ne sont pas directement accessibles à PostgreSQL™, mais certains systèmes de fichiers (comme ZFS, ext4) peuvent les utiliser pour vider les données sur disque pour les disques dont le cache en écriture est activé. Malheureusement, ces systèmes de fichiers se comportent de façon non optimale avec des contrôleurs disque équipés de batterie (BBU, acronyme de *Battery-Backed Unit*). Dans ce type de configuration, la commande de synchronisation force l'écriture de toutes les données comprises dans le cache sur les disques, éliminant ainsi tout l'intérêt d'un cache protégé par une batterie. Vous pouvez lancer l'outil `src/tools/fsync`, disponible dans le code source de PostgreSQL, pour vérifier si vous êtes affecté. Si vous l'êtes, les améliorations de performance du cache BBU peuvent être de nouveaux obtenues en désactivant les barrières d'écriture dans la configuration du système de fichiers ou en reconfigurant le contrôleur de disque, si cela est possible. Si les barrières d'écriture sont désactivées, assurez-vous que la batterie reste active. Une batterie défectueuse peut être une cause de perte de données. Il reste à espérer que les concepteurs de systèmes de fichiers et de contrôleurs disque finissent par s'attaquer à ce

comportement gênant.

Quand le système d'exploitation envoie une demande d'écriture au système de stockage, il ne peut pas faire grand chose pour s'assurer que les données sont arrivées dans un espace de stockage non volatile. Ce travail incombe à l'administrateur : ce dernier doit s'assurer que tous les composants de stockage assurent l'intégrité des données. Évitez les contrôleurs disques ne disposant pas de caches protégés par batterie. Au niveau du disque, désactivez le cache « write-back » si le disque ne garantit pas que les données seront écrites avant un arrêt. Si vous utilisez des disques SSD, sachez que beaucoup n'honorent pas les commandes de vidage de cache par défaut. Vous pouvez tester la fiabilité du comportement du système disque en utilisant `diskchecker.pl`.

Un autre risque concernant la perte des données est dû aux opérations d'écriture sur les plateaux du disque. Les plateaux sont divisés en secteur de 512 octets généralement. Chaque opération de lecture ou écriture physique traite un secteur entier. Quand la demande d'écriture arrive au lecteur, elle pourrait contenir des multiples de 512 octets (PostgreSQL™ écrit généralement 8192 octets, soit 16 secteurs, à la fois) et le processus d'écriture pourrait échouer à cause d'une perte de courant à tout moment signifiant que certains octets pourraient être écrits et les autres perdus. Pour se prévenir contre ce type d'échec, PostgreSQL™ écrit périodiquement des images de page complète sur le stockage permanent des journaux de transactions *avant* de modifier la page réelle sur disque. En effectuant ceci, lors d'une récupération après un arrêt brutal, PostgreSQL™ peut restaurer des pages écrites partiellement à partir des journaux de transactions. Si vous avez un système de fichiers qui vous protège contre les écritures de pages incomplètes (par exemple ZFS), vous pouvez désactiver la création des images de page en utilisant le paramètre full_page_writes. Les contrôleurs disques disposant d'une batterie (BBU pour *Battery-Backed Unit*) n'empêchent pas les écritures de pages partielles sauf s'ils garantissent que les données sont écrites par pages complètes de 8 Ko.

29.2. Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) est une méthode conventionnelle pour s'assurer de l'intégrité des données. Une description détaillée peut être trouvée dans la plupart des livres sur le traitement transactionnel. Brièvement, le concept central du WAL est d'effectuer les changements des fichiers de données (donc les tables et les index) uniquement après que ces changements ont été écrits de façon sûr dans un journal, appelé journal des transactions. Si nous suivons cette procédure, nous n'avons pas besoin d'écrire les pages de données vers le disque à chaque validation de transaction car nous savons que, dans l'éventualité d'une défaillance, nous serons capables de récupérer la base de données en utilisant le journal : chaque changement qui n'a pas été appliqué aux pages de données peut être ré-exécuté depuis les enregistrements du journal (ceci est une récupération roll-forward, aussi connue sous le nom de REDO).

Astuce

Comme les journaux de transaction permettent de restaurer le contenu des fichiers de base de données après un arrêt brutal, les systèmes de fichiers journalisés ne sont pas nécessaires pour stocker avec fiabilité les fichiers de données ou les journaux de transactions. En fait, la surcharge causée par la journalisation peut réduire les performances, tout spécialement si la journalisation fait que les *données* du système de fichiers sont envoyées sur disque. Heureusement, l'envoi des données lors de la journalisation peut souvent être désactivé avec une option de montage du système de fichiers, par exemple `data=writeback` sur un système de fichiers Linux ext3. Par contre, les systèmes de fichiers journalisés améliorent la rapidité au démarrage après un arrêt brutal.

Utiliser les journaux de transaction permet de réduire de façon significative le nombre d'écritures sur le disque puisque seul le journal a besoin d'être écrit sur le disque pour garantir qu'une transaction a été validée plutôt que d'écrire dans chaque fichier de données modifié par la transaction. Ce journal est écrit séquentiellement ce qui fait que le coût de synchronisation du journal est largement moindre que le coût d'écriture des pages de données. Ceci est tout spécialement vrai pour les serveurs gérant beaucoup de petites transactions touchant différentes parties du stockage de données. De plus, quand le serveur traite plein de petites transactions en parallèle, un `fsync` du journal des transactions devrait suffire pour enregistrer plusieurs transactions.

Les journaux de transaction rendent possible le support de sauvegarde en ligne et de récupération à un moment, comme décrit dans la Section 24.3, « Archivage continu et récupération d'un instantané (PITR) ». En archivant les journaux de transaction, nous pouvons supporter le retour à tout instant couvert par les données disponibles dans les journaux de transaction : nous installons simplement une ancienne sauvegarde physique de la base de données et nous rejouons les journaux de transaction jusqu'au moment désiré. Qui plus est, la sauvegarde physique n'a pas besoin d'être une image instantanée de l'état de la base de données -- si elle a été faite pendant une grande période de temps, alors rejouer les journaux de transaction pour cette période corrigera toute incohérence interne.

29.3. Validation asynchrone (Asynchronous Commit)

La *validation asynchrone* est une option qui permet aux transactions de se terminer plus rapidement. Le risque encouru est de perdre les transactions les plus récentes dans le cas où le serveur s'arrête brutalement. Dans beaucoup d'applications, le compromis est acceptable.

Comme le décrit la section précédente, la validation d'une transaction est habituellement *synchrone* : le serveur attend que les en-

registrements des journaux de transaction soient bien sauvegardés sur un disque avant de renvoyer l'information du succès de l'opération au client. Ce dernier a donc la garantie qu'une transaction validée est stockée de façon sûre, donc même en cas d'arrêt brutal immédiatement après. Néanmoins, pour les petites transactions, ce délai est une partie importante de la durée totale d'exécution de la transaction. Sélectionner le mode de validation asynchrone signifie que le serveur renvoie le succès de l'opération dès que la transaction est terminée logiquement, donc avant que les enregistrements du journal de transaction que cette transaction a généré ne soient réellement stockées sur disque. Ceci peut apporter une accélération importante pour les petites transactions.

La validation asynchrone introduit le risque des pertes de données. Il existe un petit délai entre le moment où le rapport de la fin d'une transaction est envoyé au client et celui où la transaction est réellement enregistrée (c'est-à-dire le moment où le résultat de cette transaction ne pourra pas être perdu même en cas d'arrêt brutal du serveur). Du coup, la validation asynchrone ne devrait pas être utilisée si le client se base sur le fait que la transaction est enregistrée de façon sûre. Par exemple, une banque ne devra pas utiliser la validation asynchrone pour l'enregistrement d'une transaction sur les opérations sur un compte bancaire. Dans de nombreux autres scénarios, comme la trace d'événements, il n'y a pas de garantie forte de ce type.

Le risque pris avec l'utilisation de la validation asynchrone concerne la perte de données, pas la corruption de données. Si le serveur s'arrête brutalement, il récupèrera en rejouant les journaux de transaction jusqu'au dernier enregistrement qui a été envoyé au disque. La base de données sera donc dans un état cohérent mais toutes les transactions qui n'auront pas été enregistrées sur disque n'apparaîtront pas. L'effet immédiat est donc la perte des dernières transactions. Comme les transactions sont rejouées dans l'ordre de validation, aucune incohérence ne sera introduite -- par exemple, si la transaction B fait des modifications sur les effets d'une précédente transaction A, il n'est pas possible que les effets de A soient perdus alors que les effets de B sont préservés.

L'utilisateur peut sélectionner le mode de validation de chaque transaction, donc il est possible d'avoir en même temps des transactions validées en synchrone et en asynchrone. Une grande flexibilité est permise entre performance et durabilité de certaines transactions. Le mode de validation est contrôlé par le paramètre utilisateur synchronous_commit, qui peut être modifié comme tout autre paramètre utilisateur. Le mode utilisé pour toute transaction dépend de la valeur de `synchronous_commit` au début de la transaction.

Certaines commandes, par exemple **DROP TABLE**, sont forcées en mode synchrone quelque soit la valeur du paramètre `synchronous_commit`. Ceci a pour but de s'assurer de la cohérence entre le système de fichiers du serveur et l'état logique de la base de données. Les commandes gérant la validation en deux phases, comme **PREPARE TRANSACTION**, sont aussi toujours synchrones.

Si la base de données s'arrête brutalement lors du délai entre une validation asynchrone et l'écriture des enregistrements dans le journal des transactions, les modifications réalisées lors de cette transaction *seront* perdues. La durée de ce délai est limitée car un processus en tâche de fond (le « wal writer ») envoie les enregistrements non écrits des journaux de transaction sur le disque toutes les wal_writer_delay millisecondes. La durée maximum actuelle de ce délai est de trois fois `wal_writer_delay` car le processus d'écriture des journaux de transaction est conçu pour favoriser l'écriture de pages complètes lors des périodes de grosses activités.

Attention

Un arrêt en mode immédiat est équivalent à un arrêt brutal et causera du coup la perte des validations asynchrones.

La validation asynchrone fournit un comportement différent de la simple désactivation de fsync. `fsync` est un paramètre pour le serveur entier qui modifie le comportement de toutes les transactions. Cela désactive toute logique de PostgreSQL™ qui tente de synchroniser les écritures aux différentes parties de la base de données (c'est-à-dire l'arrêt brutal du matériel ou du système d'exploitation, par un échec de PostgreSQL™ lui-même) pourrait résulter en une corruption arbitraire de l'état de la base de données. Dans de nombreux scénarios, la validation asynchrone fournit la majorité des améliorations de performances obtenues par la désactivation de `fsync`, mais sans le risque de la corruption de données.

commit_delay semble aussi très similaire à la validation asynchrone mais il s'agit en fait d'une méthode de validation synchrone (en fait, `commit_delay` est ignoré lors d'une validation asynchrone). `commit_delay` a pour effet l'application d'un délai juste avant qu'une validation synchrone tente d'enregistrement les journaux de transaction sur disque, dans l'espoir que la demande de synchronisation occasionnée par les écritures puissent aussi servir à d'autres transactions validées à peu près en même temps. Configurer `commit_delay` sert seulement quand beaucoup de transactions sont validées en parallèle et il est difficile de configurer correctement ce paramètre avec une valeur qui aide plus qu'elle ne dessert.

29.4. Configuration des journaux de transaction

Il y a plusieurs paramètres de configuration associés aux journaux de transaction qui affectent les performances de la base de données. Cette section explique leur utilisation. Consultez le Chapitre 18, Configuration du serveur pour des détails sur la mise en place de ces paramètres de configuration.

Dans la séquence des transactions, les *points de contrôles* (appelés *checkpoints*) sont des points qui garantissent que les fichiers de données table et index ont été mis à jour avec toutes les informations enregistrées dans le journal avant le point de contrôle. Au

moment du point de contrôle, toutes les pages de données non propres sont écrites sur le disque et une entrée spéciale, pour le point de contrôle, est écrite dans le journal. (Les modifications étaient déjà envoyées dans les journaux de transactions.) En cas de défaillance, la procédure de récupération recherche le dernier enregistrement d'un point de vérification dans les traces (enregistrement connus sous le nom de « redo log ») à partir duquel il devra lancer l'opération REDO. Toute modification effectuée sur les fichiers de données avant ce point est garantie d'avoir été enregistrée sur disque. Du coup, après un point de vérification, tous les segments représentant des journaux de transaction précédant celui contenant le « redo record » ne sont plus nécessaires et peuvent être soit recyclés soit supprimés (quand l'archivage des journaux de transaction est activé, ces derniers doivent être archivés avant d'être recyclés ou supprimés).

CHECKPOINT doit écrire toutes les pages de données modifiées sur disque, ce qui peut causer une charge disque importante. Du coup, l'activité des CHECKPOINT est diluée de façon à ce que les entrées/sorties disque commencent au début du CHECKPOINT et se termine avant le démarrage du prochain CHECKPOINT ; ceci minimise la dégradation des performances lors des CHECKPOINT.

Le processus d'écriture en tâche de fond lance automatiquement un point de contrôle de temps en temps : tous les checkpoint_segments journaux de transaction ou dès que checkpoint_timeout secondes se sont écoulées. Les paramètres par défaut sont respectivement de trois journaux et de 300 secondes (5 minutes). Il est également possible de forcer la création d'un point de contrôle en utilisant la commande SQL **CHECKPOINT**.

La réduction de `checkpoint_segments` et/ou `checkpoint_timeout` implique des points de contrôle plus fréquents. Cela permet une récupération plus rapide après défaillance puisqu'il y a moins d'écritures à synchroniser. Cependant, il faut équilibrer cela avec l'augmentation du coût d'écriture des pages de données modifiées. Si full_page_writes est configuré (ce qui est la valeur par défaut), il reste un autre facteur à considérer. Pour s'assurer de la cohérence des pages de données, la première modification d'une page de données après chaque point de vérification résulte dans le traçage du contenu entier de la page. Dans ce cas, un intervalle de points de vérification plus petit augmentera le volume en sortie des journaux de transaction, diminuant légèrement l'intérêt d'utiliser un intervalle plus petit et impliquant de toute façon plus d'entrées/sorties au niveau disque.

Les points de contrôle sont assez coûteux, tout d'abord parce qu'ils écrivent tous les tampons utilisés, et ensuite parce que cela suscite un trafic supplémentaire dans les journaux de transaction, comme indiqué ci-dessus. Du coup, il est conseillé de configurer les paramètres en relation assez haut pour que ces points de contrôle ne surviennent pas trop fréquemment. Pour une vérification rapide de l'adéquation de vos paramètres, vous pouvez configurer le paramètre checkpoint_warning. Si les points de contrôle arrivent plus rapidement que `checkpoint_warning` secondes, un message est affiché dans les journaux applicatifs du serveur recommandant d'accroître `checkpoint_segments`. Une apparition occasionnelle d'un message ne doit pas vous alarmer mais, s'il apparaît souvent, alors les paramètres de contrôle devraient être augmentés. Les opérations en masse, comme les transferts importants de données via **COPY**, pourraient être la cause de l'apparition d'un tel nombre de messages d'avertissements si vous n'avez pas configuré `checkpoint_segments` avec une valeur suffisamment haute.

Pour éviter de remplir le système disque avec de très nombreuses écritures de pages, l'écriture des pages modifiés pendant un point de vérification est étalée sur une période de temps. Cette période est contrôlée par checkpoint_completion_target, qui est donné comme une fraction de l'intervalle des points de vérification. Le taux d'entrées/sorties est ajusté pour que le point de vérification se termine quand la fraction donnée de `checkpoint_segments` journaux de transaction a été consommée ou quand la fraction donné de `checkpoint_timeout` secondes s'est écoulée (la condition que se verra vérifiée la première). Avec une valeur par défaut de 0,5, PostgreSQL™ peut s'attendre à terminer chaque point de vérification en moitié moins de temps qu'il ne faudra pour lancer le prochain point de vérification. Sur un système très proche du taux maximum en entrée/sortie pendant des opérations normales, vous pouvez augmenter `checkpoint_completion_target` pour réduire le chargement en entrée/sortie dû aux points de vérification. L'inconvénient de ceci est que prolonger les points de vérification affecte le temps de récupération parce qu'il faudra conserver plus de journaux de transaction si une récupération est nécessaire. Bien que `checkpoint_completion_target` puisse valoir 1.0, il est bien mieux de la configurer à une valeur plus basse que ça (au maximum 0,9) car les points de vérification incluent aussi d'autres activités en dehors de l'écriture des pages modifiées. Une valeur de 1,0 peut avoir pour résultat des points de vérification qui ne se terminent pas à temps, ce qui aurait pour résultat des pertes de performance à cause de variation inattendue dans le nombre de journaux nécessaires.

Il y aura toujours au moins un journal de transaction et normalement pas plus de fichiers que wal_keep_segments ou (2 + `checkpoint_completion_target`) * `checkpoint_segments` + 1 journaux. Chaque journal fait normalement 16 Mo (bien que cette taille puisse être modifiée lors de la compilation du serveur). Vous pouvez utiliser cela pour estimer l'espace disque nécessaire au stockage des journaux de transaction. D'habitude, quand les vieux journaux ne sont plus nécessaires, ils sont recyclés (renommés pour devenir les prochains segments dans une séquence numérotée). S'il y a plus de 3 * `checkpoint_segments` + 1 fichiers à cause d'un pic temporaire du taux d'écriture des journaux, ceux inutilisés seront effacés au lieu d'être recyclés jusqu'à ce que le système soit en-dessous de cette limite.

Dans le mode de restauration d'archive et dans le mode standby, le serveur réalise périodiquement des *restartpoints* (points de redémarrage). C'est similaire aux checkpoints lors du fonctionnement normal : le serveur force l'écriture de son état sur disque, met à jour le fichier `pg_control` pour indiquer que les données déjà traitées des journaux de transactions n'ont plus besoin d'être parcourues de nouveau, puis recycle les anciens journaux de transactions trouvés dans le répertoire `pg_xlog`. Un restartpoint est déclenché si au moins un enregistrement checkpoint a été rejoué et à chaque fois que `checkpoint_timeout` secondes se sont écoulées depuis le dernier restartpoint. Dans le mode standby, un restartpoint est aussi déclenché si `checkpoint_segments`

journaux de transactions ont été rejoués depuis le dernier restartpoint et qu'au moins un enregistrement checkpoint a été rejoué. Les restartpoints ne peuvent être réalisés plus fréquemment que les checkpoints du maître car les restartpoints peuvent seulement être réalisés aux enregistrements de checkpoint.

Il existe deux fonctions WAL internes couramment utilisées : `LogInsert` et `LogFlush`. `LogInsert` est utilisée pour placer une nouvelle entrée à l'intérieur des tampons WAL en mémoire partagée. S'il n'y a plus d'espace pour une nouvelle entrée, `LogInsert` devra écrire (autrement dit, déplacer dans le cache du noyau) quelques tampons WAL remplis. Ceci n'est pas désirable parce que `LogInsert` est utilisée lors de chaque modification bas niveau de la base (par exemple, lors de l'insertion d'une ligne) quand un verrou exclusif est posé sur des pages de données affectées. À cause de ce verrou, l'opération doit être aussi rapide que possible. Pire encore, écrire des tampons WAL peut forcer la création d'un nouveau journal, ce qui peut prendre beaucoup plus de temps. Normalement, les tampons WAL doivent être écrits et vidés par une requête de `LogFlush` qui est faite, la plupart du temps, au moment de la validation d'une transaction pour assurer que les entrées de la transaction sont écrites vers un stockage permanent. Sur les systèmes avec une importante écriture de journaux, les requêtes de `LogFlush` peuvent ne pas arriver assez souvent pour empêcher `LogInsert` d'avoir à écrire lui-même sur disque. Sur de tels systèmes, on devrait augmenter le nombre de tampons WAL en modifiant le paramètre de configuration wal_buffers. Par défaut, le nombre de tampons est de 8. Réhausser cette valeur augmente considérablement l'utilisation de la mémoire partagée. Quand full_page_writes est configuré et que le système est très occupé, configurer cette variable avec une valeur plus importante aide à avoir des temps de réponse plus réguliers lors de la période suivant chaque point de vérification.

Le paramètre commit_delay définit la durée d'endormissement en micro-secondes du processus serveur après l'écriture d'une entrée de validation dans le journal avec `LogInsert` avant d'exécuter un `LogFlush`. Ce délai permet aux autres processus du serveur d'ajouter leurs entrées de validation dans le journal afin de tout écrire vers le disque avec une seule synchronisation du journal. Aucune mise en sommeil n'aura lieu si fsync n'est pas disponible ou si moins de commit_siblings autres sessions sont, à ce moment, dans des transactions actives ; cela évite de dormir quand il est improbable qu'une autre session fasse bientôt une validation. Notez que dans la plupart des plate-formes, la résolution d'une requête d'endormissement est de 10 millisecondes, donc un `commit_delay` différent de zéro et configuré entre 1 et 10000 micro-secondes a le même effet. Les bonnes valeurs pour ce paramètre ne sont pas encore claires ; les essais sont encouragés.

Le paramètre wal_sync_method détermine la façon dont PostgreSQL™ demande au noyau de forcer les mises à jour des journaux de transaction sur le disque. Toutes les options ont un même comportement avec une exception, `fsync_writethrough`, qui peut parfois forcer une écriture du cache disque même quand d'autres options ne le font pas. Néanmoins, dans la mesure où la fiabilité ne disparaît pas, c'est avec des options spécifiques à la plate-forme que la rapidité la plus importante sera observée ; vous pouvez tester la performance de chaque option en utilisant l'outil `src/tools/fsync` disponible dans les sources de PostgreSQL. Notez que ce paramètre est ignoré si `fsync` a été désactivé.

Activer le paramètre de configuration wal_debug (à supposer que PostgreSQL™ ait été compilé avec le support de ce paramètre) permet d'enregistrer chaque appel WAL à `LogInsert` et `LogFlush` dans les journaux applicatifs du serveur. Cette option pourrait être remplacée par un mécanisme plus général dans le futur.

29.5. Vue interne des journaux de transaction

Le mécanisme WAL est automatiquement disponible ; aucune action n'est requise de la part de l'administrateur excepté de s'assurer que l'espace disque requis par les journaux de transaction soit présent et que tous les réglages soient faits (regardez la Section 29.4, « Configuration des journaux de transaction »).

Les journaux de transaction sont stockés dans le répertoire `pg_xlog` sous le répertoire de données, comme un ensemble de fichiers, chacun d'une taille de 16 Mo généralement (cette taille pouvant être modifiée en précisant une valeur pour l'option `--with-wal-segsize` de configure lors de la construction du serveur). Chaque fichier est divisé en pages de généralement 8 Ko (cette taille pouvant être modifiée en précisant une valeur pour l'option `--with-wal-blocksize` de configure). Les entêtes de l'entrée du journal sont décrites dans `access/xlog.h` ; le contenu de l'entrée dépend du type de l'événement qui est enregistré. Les fichiers sont nommés suivant un nombre qui est toujours incrémenté et qui commence à `000000010000000000000000`. Les nombres ne bouclent pas, mais cela prendra beaucoup de temps pour épuiser le stock de nombres disponibles.

Il est avantageux que les journaux soient situés sur un autre disque que celui des fichiers principaux de la base de données. Cela peut se faire en déplaçant le répertoire `pg_xlog` vers un autre emplacement (alors que le serveur est arrêté) et en créant un lien symbolique de l'endroit d'origine dans le répertoire principal de données au nouvel emplacement.

Le but de WAL est de s'assurer que le journal est écrit avant l'altération des entrées dans la base, mais cela peut être mis en échec par les disques qui rapportent une écriture réussie au noyau quand, en fait, ils ont seulement mis en cache les données et ne les ont pas encore stockés sur le disque. Une coupure de courant dans ce genre de situation peut mener à une corruption irrécupérable des données. Les administrateurs devraient s'assurer que les disques contenant les journaux de transaction de PostgreSQL™ ne produisent pas ce genre de faux rapports. (Voir Section 29.1, « Fiabilité ».)

Après qu'un point de contrôle ait été fait et que le journal ait été écrit, la position du point de contrôle est sauvegardée dans le fichier `pg_control`. Donc, au début de la restauration, le serveur lit en premier `pg_control` et ensuite l'entrée du point de

contrôle ; ensuite, il exécute l'opération REDO en parcourant vers l'avant à partir de la position du journal indiquée dans l'entrée du point de contrôle. Parce que l'ensemble du contenu des pages de données est sauvegardé dans le journal à la première modification de page après un point de contrôle (en supposant que full_page_writes n'est pas désactivé), toutes les pages changées depuis le point de contrôle seront restaurées dans un état cohérent.

Pour gérer le cas où `pg_control` est corrompu, nous devons permettre le parcours des segments de journaux existants en ordre inverse -- du plus récent au plus ancien -- pour trouver le dernier point de vérification. Ceci n'a pas encore été implémenté. `pg_control` est assez petit (moins d'une page disque) pour ne pas être sujet aux problèmes d'écriture partielle et, au moment où ceci est écrit, il n'y a eu aucun rapport d'échecs de la base de données uniquement à cause de son incapacité à lire `pg_control`. Donc, bien que cela soit théoriquement un point faible, `pg_control` ne semble pas être un problème en pratique.

Chapitre 30. Tests de régression

Les tests de régression composent un ensemble exhaustif de tests pour l'implémentation SQL dans PostgreSQL™. Ils testent les opérations SQL standards ainsi que les fonctionnalités étendues de PostgreSQL™.

30.1. Lancer les tests

Les tests de régression peuvent être lancés sur un serveur déjà installé et fonctionnel ou en utilisant une installation temporaire à l'intérieur du répertoire de construction. De plus, ils peuvent être lancés en mode « parallèle » ou en mode « séquentiel ». Le mode séquentiel lance les scripts de test en série, alors que le mode parallèle lance plusieurs processus serveurs pour parallèliser l'exécution des groupes de tests. Les tests parallèles permettent de s'assurer du bon fonctionnement des communications inter-processus et du verrouillage.

Pour lancer les tests de régression en parallèle après la construction mais avant l'installation, il suffit de saisir

```
gmake check
```

dans le répertoire de premier niveau (on peut aussi se placer dans le répertoire `src/test/regress` et y lancer la commande). En premier lieu seront construits différents fichiers auxiliaires, tels des exemples de fonctions de déclencheurs utilisateur, puis le script de pilotage des tests sera exécuté. Au final, la sortie devrait ressembler à quelque chose comme

```
=====================
 All 100 tests passed.
=====================
```

ou une note indiquant l'échec des tests. Voir la Section 30.2, « Évaluation des tests » avant de supposer qu'un « échec » représente un problème sérieux.

> **Avertissement**
>
> Sur les systèmes ne disposant pas de sockets de domaine Unix, la méthode de tests démarre un serveur temporaire configuré pour accepter toute connexion provenant de la machine locale. Tout utilisateur local peut obtenir les droits superutilisateur de la base de données en se connectant à ce serveur et pourrait en principe exploiter tous les droits de l'utilisateur du système d'exploitation exécutant les tests. De ce fait, il n'est pas recommandé d'utiliser `gmake check` sur un système partagé avec d'autres utilisateurs. À la place, lancez les tests après avoir terminé l'installation, comme indiqué dans la section suivante.

Comme cette méthode de tests exécute un serveur temporaire, cela ne fonctionnera pas si vous avez construit le serveur en tant que root, étant donné que le serveur ne démarre pas en tant que root. La procédure recommandée est de ne pas construire en tant que root ou de réaliser les tests après avoir terminé l'installation.

Si vous avez configuré PostgreSQL™ pour qu'il s'installe dans un emplacement où existe déjà une ancienne installation de PostgreSQL™ et que vous lancez `gmake check` avant d'installer la nouvelle version, vous pourriez trouver que les tests échouent parce que les nouveaux programmes essaient d'utiliser les bibliothèques partagées déjà installées (les symptômes typiques sont des plaintes concernant des symboles non définis). Si vous souhaitez lancer les tests avant d'écraser l'ancienne installation, vous devrez construire avec `configure --disable-rpath`. Néanmoins, il n'est pas recommandé d'utiliser cette option pour l'installation finale.

Les tests de régression en parallèle lancent quelques processus avec votre utilisateur. Actuellement, le nombre maximum est de vingt scripts de tests en parallèle, ce qui signifie 40 processus : il existe un processus serveur, un psql et habituellement un processus parent pour le psql de chaque script de tests. Si votre système force une limite par utilisateur sur le nombre de processus, assurez-vous que cette limite est d'au moins 50, sinon vous pourriez obtenir des échecs hasardeux dans les tests en parallèle. Si vous ne pouvez pas augmenter cette limite, vous pouvez diminuer le degré de parallélisme en initialisant le paramètre `MAX_CONNECTIONS`. Par exemple,

```
gmake MAX_CONNECTIONS=10 check
```

ne lance pas plus de dix tests en même temps.

Pour lancer les tests après l'installation (voir le Chapitre 15, Procédure d'installation de PostgreSQL™ du code source), initialisez un espace de données et lancez le serveur comme expliqué dans le Chapitre 17, Configuration du serveur et mise en place, puis lancez

```
gmake installcheck
```

ou pour un test parallèle

```
gmake installcheck-parallel
```

Les tests s'attendront à contacter le serveur sur l'hôte local et avec le numéro de port par défaut, sauf en cas d'indication contraire avec les variables d'environnement `PGHOST` et `PGPORT`.

La distribution source contient aussi les tests de régression pour les langages de procédures et pour certains des modules de `contrib`. Actuellement, ces tests peuvent seulement être utilisés avec un serveur déjà installé. Pour exécuter les tests pour tous les langages de procédure qui ont été construits et installés, placez-vous dans le sous-répertoire `src/pl` du répertoire de construction et lancez

```
gmake installcheck
```

Vous pouvez aussi le faire dans tous les sous-répertoires de `src/pl` pour lancer les tests pour un seul langage de procédure. Pour lancer les tests sur tous les modules `contrib` qui les ont, placez-vous dans le répertoire `contrib` du répertoire de construction et exécutez

```
gmake installcheck
```

Les modules `contrib` doivent avoir été construits et installés tout d'abord. Vous pouvez aussi le faire dans un sous-répertoire de `contrib` pour exécuter les tests pour un seul module.

La distribution des sources contient aussi des tests de régression du comportement statique du Hot Standby. Ces tests requièrent un serveur primaire et un serveur en attente, les deux en cours d'exécution, le dernier acceptant les modifications des journaux de transactions du primaire en utilisant soit l'envoi des fichiers soit la réplication en flux. Ces serveurs ne sont pas automatiquement créés pour vous, pas plus que la configuration n'est documentée ici. Merci de vérifier les différentes sections de la documentation qui sont déjà dévolues aux commandes requises et aux problèmes associés.

Tout d'abord, créez une base de données appelée « regression » sur le primaire.

```
psql -h primary -c "CREATE DATABASE regression"
```

Ensuite, exécutez un script préparatoire sur le primaire dans la base de données de regréssion : `src/test/regress/sql/hs_primary_setup.sql`, puis autorisez la propagation des modifications vers le serveur en attente. Par exemple :

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

Maintenant, confirmez que la connexion par défaut du testeur est le serveur en attente sous test, puis lancez l'action `standby-check` à partir du répertoire de la suite de tests de régression.

```
cd src/test/regress
gmake standbycheck
```

Certains comportements extrêmes peuvent aussi être créés sur le primaire en utilisant le script `src/test/regress/sql/hs_primary_extremes.sql` pour permettre le test du comportement du serveur en attente.

Des tests automatisés supplémentaires pourraient être disponibles avec les prochaines versions.

30.2. Évaluation des tests

Quelques installations de PostgreSQL™ proprement installées et totalement fonctionnelles peuvent « échouer » sur certains des tests de régression à cause de certains points spécifiques à la plateforme comme une représentation de nombres à virgules flottantes ou « message wording ». Les tests sont actuellement évalués en utilisant une simple comparaison **diff** avec les sorties générées sur un système de référence, donc les résultats sont sensibles aux petites différences système. Quand un test est rapporté comme « échoué », toujours examiner les différences entre les résultats attendus et ceux obtenus ; vous pourriez très bien trouver que les différences ne sont pas significatives. Néanmoins, nous nous battons toujours pour maintenir des fichiers de références précis et à jour pour toutes les plateformes supportés de façon à ce que tous les tests puissent réussir.

Les sorties actuelles des tests de régression sont dans les fichiers du répertoire `src/test/regress/results`. Le script de test utilise **diff** pour comparer chaque fichier de sortie avec les sorties de référence stockées dans le répertoire `src/test/regress/expected`. Toutes les différences sont conservées pour que vous puissiez les regarder dans `src/test/regress/regression.diffs` (ou vous pouvez lancer **diff** vous-même, si vous préférez).

Si, pour certaines raisons, une plateforme particulière génère un « échec » pour un test donné mais qu'une revue de la sortie vous convaint que le résultat est valide, vous pouvez ajouter un nouveau fichier de comparaison pour annuler le rapport d'échec pour les prochains lancements du test. Voir la Section 30.3, « Fichiers de comparaison de variants » pour les détails.

30.2.1. Différences dans les messages d'erreurs

Certains des tests de régression impliquent des valeurs en entrée intentionnellement invalides. Les messages d'erreur peuvent provenir soit du code de PostgreSQL™ soit des routines système de la plateforme hôte. Dans ce dernier cas, les messages pourraient

varier entre plateformes mais devraient toujours refléter des informations similaires. Ces différences dans les messages résulteront en un échec du test de régression qui pourrait être validé après vérification.

30.2.2. Différences au niveau des locales

Si vous lancez des tests sur un serveur initialisé avec une locale autre que C, alors il pourrait y avoir des différences dans les ordres de tris. La suite de tests de régression est initialisée pour gérer ce problème en fournissant des fichiers de résultats alternatifs qui gèrent ensemble un grand nombre de locales.

Pour exécuter les tests dans une locale différente lors de l'utilisation de la méthode d'installation temporaire, passez les variables d'environnement relatives à la locale sur la ligne de commande de **make**, par exemple :

```
gmake check LANG=de_DE.utf8
```

(Le pilote de tests des régressions déconfigure `LC_ALL`, donc choisir la locale par cette variable ne fonctionne pas.) Pour ne pas utiliser de locale, vous devez soit déconfigurer toutes les variables d'environnement relatives aux locales (ou les configurer à `C`) ou utiliser une option spéciale :

```
gmake check NO_LOCALE=1
```

Lors de l'exécution des tests sur une installation existante, la configuration de la locale est déterminée d'après l'installation existante. Pour la modifier, initialiser le cluster avec une locale différente en passant les options appropriées à **initdb**.

En général, il est conseillé d'essayer l'exécution des tests de régression dans la configuration de locale souhaitée pour l'utilisation en production, car cela testera aussi les portions de code relatives à l'encodage et à la locale qui pourront être utilisées en production. Suivant l'environnement du système d'exploitation, vous pourrez obtenir des échecs, mais vous saurez au moins le comportement à attendre sur la locale lorsque vous utiliserez vos vraies applications.

30.2.3. Différences au niveau des dates/heures

La plupart des résultats date/heure sont dépendants de l'environnement de zone horaire. Les fichiers de référence sont générés pour la zone horaire `PST8PDT` (Berkeley, Californie), et il y aura des échecs apparents si les tests ne sont pas lancés avec ce paramétrage de fuseau horaire. Le pilote des tests de régression initialise la variable d'environnement `PGTZ` à `PST8PDT` ce qui nous assure normalement de bons résultats.

30.2.4. Différences sur les nombres à virgules flottantes

Quelques tests impliquent des calculs sur des nombres flottants à 64 bits (double precision) à partir de colonnes de tables. Des différences dans les résultats appliquant des fonctions mathématiques à des colonnes double precision ont été observées. Les tests de `float8` et `geometry` sont particulièrement sensibles aux différences entre plateformes, voire aux différentes options d'optimisation des compilateurs. L'œil humain est nécessaire pour déterminer la véritable signification de ces différences, habituellement situées après la dixième décimale.

Certains systèmes affichent moins zéro comme `-0` alors que d'autres affichent seulement `0`.

Certains systèmes signalent des erreurs avec `pow()` et `exp()` différemment suivant le mécanisme attendu du code de PostgreSQL™.

30.2.5. Différences dans l'ordre des lignes

Vous pourriez voir des différences dans lesquelles les mêmes lignes sont affichées dans un ordre différent de celui qui apparaît dans le fichier de référence. Dans la plupart des cas, ce n'est pas à strictement parlé un bogue. La plupart des scripts de tests de régression ne sont pas assez stricts pour utiliser un `ORDER BY` sur chaque `SELECT` et, du coup, l'ordre des lignes pourrait ne pas être correctement défini suivant la spécification SQL. En pratique, comme nous sommes avec les mêmes requêtes sur les mêmes données avec le même logiciel, nous obtenons habituellement le même résultat sur toutes les plateformes et le manque d'`ORDER BY` n'est pas un problème. Quelques requêtes affichent des différences d'ordre entre plateformes. Lors de tests avec un serveur déjà installé, les différences dans l'ordre des lignes peuvent aussi être causées par un paramètrage des locales à une valeur différente de C ou par un paramètrage personnalisé, comme des valeurs personnalisées de `work_mem` ou du coût du planificateur.

Du coup, si vous voyez une différence dans l'ordre, vous n'avez pas à vous inquiéter sauf si la requête possède un `ORDER BY` que votre résultat ne respecte pas. Néanmoins, rapportez tout de même ce problème que nous ajoutions un `ORDER BY` à cette requête pour éliminer les faux « échecs » dans les versions suivantes.

Vous pourriez vous demander pourquoi nous n'ordonnons pas toutes les requêtes des tests de régression explicitement pour supprimer ce problème une fois pour toutes. La raison est que cela rendrait les tests de régression moins utiles car ils tendraient à exercer des types de plans de requêtes produisant des résultats ordonnés à l'exclusion de celles qui ne le font pas.

30.2.6. Profondeur insuffisante de la pile

Si les tests d'`erreurs` se terminent avec un arrêt brutal du serveur pendant la commande `select infinite_recurse()`, cela signifie que la limite de la plateforme pour la taille de pile du processus est plus petite que le paramètre `max_stack_depth` max_stack_depth ne l'indique. Ceci est corrigeable en exécutant le postmaster avec une limite pour la taille de pile plus importante (4 Mo est recommandé avec la valeur par défaut de `max_stack_depth`). Si vous n'êtes pas capables de le faire, une alternative est de réduire la valeur de `max_stack_depth`.

30.2.7. Test « random »

Le script de tests `random` a pour but de produire des résultats aléatoires. Dans de rares cas, ceci fait échouer random aux tests de régression. Saisir :

```
diff results/random.out expected/random.out
```

ne devrait produire au plus que quelques lignes différentes. Cela est normal et ne devient préoccupant que si les tests random échouent en permanence lors de tests répétés

30.3. Fichiers de comparaison de variants

Comme certains de ces tests produisent de façon inhérente des résultats dépendants de l'environnement, nous avons fourni des moyens de spécifier des fichiers résultats alternatifs « attendus ». Chaque test de régression peut voir plusieurs fichiers de comparaison affichant les résultats possibles sur différentes plateformes. Il existe deux mécanismes indépendants pour déterminer quel fichier de comparaison est utilisé pour chaque test.

Le premier mécanisme permet de sélectionner les fichiers de comparaison suivant des plateformes spécifiques. Le fichier de correspondance `src/test/regress/resultmap` définit le fichier de comparaison à utiliser pour chaque plateforme. Pour éliminer les tests « échoués » par erreur pour une plateforme particulière, vous choisissez ou vous créez un fichier variant de résultat, puis vous ajoutez une ligne au fichier `resultmap`.

Chaque ligne du fichier de correspondance est de la forme

```
nomtest:sortie:modeleplateform=fichiercomparaison
```

Le nom de tests est juste le nom du module de tests de régression particulier. La valeur en sortie indique le fichier à vérifier. Pour les tests de régression standards, c'est toujours `out`. La valeur correspond à l'extension de fichier du fichier en sortie. Le modèle de plateforme est un modèle dans le style des outils Unix **expr** (c'est-à-dire une expression rationnelle avec une ancre implicite `^` au début). Il est testé avec le nom de plateforme affiche par **config.guess**. Le nom du fichier de comparaison est le nom de base du fichier de comparaison substitué.

Par exemple : certains systèmes interprètent les très petites valeurs en virgule flottante comme zéro, plutôt que de rapporter une erreur. Ceci fait quelques petites différences dans le test de régression `float8`. Du coup, nous fournissons un fichier de comparaison variable, `float8-small-is-zero.out`, qui inclut les résultats attendus sur ces systèmes. Pour faire taire les messages d'« échec » erronés sur les plateformes `OpenBSD`, `resultmap` inclut

```
float8:out:i.86-.*-openbsd=float8-small-is-zero.out
```

qui se déclenche sur toute machine où la sortie de **config.guess** correspond à `i.86-.*-openbsd`. D'autres lignes dans `resultmap` sélectionnent le fichier de comparaison variable pour les autres plateformes si c'est approprié.

Le second mécanisme de sélection des fichiers de comapraison variants est bien plus automatique : il utilise simplement la « meilleure correspondance » parmi les différents fichiers de comparaison fournis. Le script pilote des tests de régression considère le fichier de comparaison standard pour un test, *`nomtest`*`.out`, et les fichiers variants nommés *`nomtest_chiffre`*`.out` (où *`chiffre`* est un seul chiffre compris entre `0` et `9`). Si un tel fichier établit une correspondance exacte, le test est considéré réussi ; sinon, celui qui génère la plus petite différence est utilisé pour créer le rapport d'échec. (Si `resultmap` inclut une entrée pour le test particulier, alors le *`nomtest`* de base est le nom de substitut donné dans `resultmap`.)

Par exemple, pour le test `char`, le fichier de comparaison `char.out` contient des résultats qui sont attendus dans les locales `C` et `POSIX`, alors que le fichier `char_1.out` contient des résultats triés comme ils apparaissent dans plusieurs autres locales.

Le mécanisme de meilleure correspondance a été conçu pour se débrouiller avec les résultats dépendant de la locale mais il peut être utilisé dans toute situation où les résultats des tests ne peuvent pas être prédits facilement à partir de la plateforme seule. Une limitation de ce mécanisme est que le pilote test ne peut dire quelle variante est en fait « correcte » dans l'environnement en cours ; il récupèrera la variante qui semble le mieux fonctionner. Du coup, il est plus sûr d'utiliser ce mécanisme seulement pour les résultats variants que vous voulez considérer comme identiquement valides dans tous les contextes.

30.4. Examen de la couverture du test

Le code source de PostgreSQL peut être compilé avec des informations supplémentaire sur la couverture des tests, pour qu'il devienne possible d'examiner les parties du code couvertes par les tests de régression ou par toute suite de tests exécutée avec le code. Cette fonctionnalité est supportée en compilant avec GCC et nécessite les programmes **gcov** et **lcov**.

La suite typique de commandes ressemble à ceci :

```
./configure --enable-coverage ... OTHER OPTIONS ...
gmake
gmake check # or other test suite
gmake coverage-html
```

Puis pointez votre navigateur HTML vers `coverage/index.html`. Les commandes **gmake** travaillent aussi dans les sous-répertoires.

Pour réinitialiser le compteur des exécutions entre chaque test, exécutez :

```
gmake coverage-clean
```

Partie IV. Interfaces client

Cette partie décrit les interfaces de programmation client distribuées avec PostgreSQL™. Chacun de ces chapitres peut être lu indépendamment. On trouve beaucoup d'autres interfaces de programmation de clients, chacune distribuée séparément avec sa propre documentation. Les lecteurs de cette partie doivent être familiers de l'utilisation des requêtes SQL de manipulation et d'interrogation d'une base (voir la Partie II, « Langage SQL ») et surtout du langage de programmation utilisé par l'interface.

Chapitre 31. libpq - Bibliothèque C

libpq est l'interface de programmation pour les applications C avec PostgreSQL™. libpq est un ensemble de fonctions permettant aux programmes clients d'envoyer des requêtes au serveur PostgreSQL™ et de recevoir les résultats de ces requêtes.

libpq est aussi le moteur sous-jacent de plusieurs autres interfaces de programmation de PostgreSQL™, comme ceux écrits pour C++, Perl, Python, Tcl et ECPG. Donc, certains aspects du comportement de libpq seront importants pour vous si vous utilisez un de ces paquetages. En particulier, la Section 31.13, « Variables d'environnement », la Section 31.14, « Fichier de mots de passe » et la Section 31.17, « Support de SSL » décrivent le comportement que verra l'utilisateur de toute application utilisant libpq.

Quelques petits programmes sont inclus à la fin de ce chapitre (Section 31.20, « Exemples de programmes ») pour montrer comment écrire des programmes utilisant libpq. Il existe aussi quelques exemples complets d'applications libpq dans le répertoire `src/test/examples` venant avec la distribution des sources.

Les programmes clients utilisant libpq doivent inclure le fichier d'en-tête `libpq-fe.h` et doivent être lié avec la bibliothèque libpq.

31.1. Fonctions de contrôle de connexion à la base de données

Les fonctions suivantes concernent la réalisation d'une connexion avec un serveur PostgreSQL™. Un programme peut avoir plusieurs connexions ouvertes sur des serveurs à un même moment (une raison de la faire est d'accéder à plusieurs bases de données). Chaque connexion est représentée par un objet PGconn, obtenu avec la fonction `PQconnectdb`, `PQconnectdbParams`, ou `PQsetdbLogin`. Notez que ces fonctions renverront toujours un pointeur d'objet non nul, sauf peut-être dans un cas de manque de mémoire pour l'allocation de l'objet PGconn. La fonction `PQstatus` doit être appelée pour vérifier si la connexion s'est bien effectuée avant de lancer des requêtes via l'objet de connexion.

> **Avertissement**
>
> Sur Unix, la création d'un processus via l'appel système fork() avec des connexions libpq ouvertes peut amener à des résultats imprévisibles car les processus parent et enfants partagent les même sockets et les mêmes ressources du système d'exploitation. Pour cette raison, un tel usage n'est pas recommandé, alors qu'exécuter un `exec` à partir du processus enfant pour charger un nouvel exécutable est sûr.

> **Note**
>
> Sur Windows, il existe un moyen pour améliorer les performances si une connexion seule à la base de données est ouverte puis fermée de façon répétée. En interne, libpq appelle `WSAStartup()` et `WSACleanup()` respectivement pour le début et la fin de la transaction. `WSAStartup()` incrémente un compteur de référence interne à la bibliothèque Windows. Ce compteur est décrémenté par `WSACleanup()`. Quand le compteur arrive à un, appeler `WSACleanup()` libère toutes les ressources et toutes les DLL associées. C'est une opération coûteuse. Pour éviter cela, une application peut appeler manuellement `WSAStartup()` afin que les ressources ne soient pas libérées quand la dernière connexion est fermée.

`PQconnectdbParams`

Établit une nouvelle connexion au serveur de base de données.

```
PGconn *PQconnectdbParams(const char **keywords, const char **values, int expand_dbname);
```

Cette fonction ouvre une nouvelle connexion à la base de données en utilisant les paramètres à partir des deux tableaux terminés par un NULL. Le premier, `keywords`, est défini comme un tableau de chaînes, chacune étant un mot-clé. Le second, `values`, donne la valeur pour chaque mot-clé. Contrairement à `PQsetdbLogin` ci-dessous, l'ensemble des paramètres peut être étendu sans changer la signature de la fonction donc son utilisation (ou ses versions non bloquantes, à savoir `PQconnectStartParams` et `PQconnectPoll`) est recommendée pour les nouvelles applications.

Quand `expand_dbname` est différent de zéro, la valeur du mot-clé *`dbname`* peut être reconnue comme une chaîne *`conninfo`*. Voir ci-dessous pour les détails.

Les tableaux fournis peuvent être vides pour utiliser tous les paramètres par défaut ou peuvent contenir un ou plusieurs paramètres. Ils doivent avoir la même longueur. Le traitement stoppera au premier élément NULL découvert dans le tableau

`keywords`.

Les mots clés actuellement reconnus sont :

`host`
Nom de l'hôte sur lequel se connecter. S'il commence avec un slash, il spécifie une communication par domaine Unix plutôt qu'une communication TCP/IP ; la valeur est le nom du répertoire où le fichier socket est stocké. Par défaut, quand `host` n'est pas spécifié, il s'agit d'une communication par socket de domaine Unix dans `/tmp` (ou tout autre répertoire de socket spécifié lors de la construction de PostgreSQL™). Sur les machines sans sockets de domaine Unix, la valeur par défaut est de se connecter à `localhost`.

`hostaddr`
Adresse IP numérique de l'hôte de connexion. Elle devrait être au format d'adresse standard IPv4, c'est-à-dire `172.28.40.9`. Si votre machine supporte IPv6, vous pouvez aussi utiliser ces adresses. La communication TCP/IP est toujours utilisée lorsqu'une chaîne non vide est spécifiée pour ce paramètre.

Utiliser `hostaddr` au lieu de `host` permet à l'application d'éviter une recherche de nom d'hôte, qui pourrait être importante pour les applications ayant des contraintes de temps. Un nom d'hôte est requis pour les authentifications Kerberos, GSSAPI et SSPI, ainsi que pour les vérifications complètes de certificats SSL. Les règles suivantes sont utilisées : : si `host` est spécifié sans `hostaddr`, une recherche de nom d'hôte a lieu. Si `hostaddr` est spécifié sans `host`, la valeur de `hostaddr` donne l'adresse du serveur. La tentative de connexion échouera dans tous les cas où un nom d'hôte est requis. Si à la fois `host` et `hostaddr` sont spécifiés, la valeur de `hostaddr` donne l'adresse du serveur ; la valeur de `host` est ignorée, sauf si Kerberos est utilisé, auquel cas cette valeur est utilisée pour l'authentification Kerberos. Notez que l'authentification a des risques d'échouer, `host` est le nom de la machine sur `hostaddr`). De même, `host` plutôt que `hostaddr` est utilisé pour identifier la connexion dans `~/.pgpass` (voir la Section 31.14, « Fichier de mots de passe »).

Sans un nom ou une adresse d'hôte, libpq se connectera en utilisant un socket local de domaine Unix. Sur des machines sans sockets de domaine Unix, il tentera une connexion sur `localhost`.

`port`
Numéro de port pour la connexion au serveur ou extension du nom de fichier pour des connexions de domaine Unix.

`dbname`
Nom de la base de données. Par défaut, la même que le nom utilisateur.

`user`
Nom de l'utilisateur PostgreSQL™ qui se connecte. Par défaut, il s'agit du nom de l'utilisateur ayant lancé l'application.

`password`
Mot de passe à utiliser si le serveur demande une authentification par mot de passe.

`connect_timeout`
Attente maximum pour une connexion, en secondes (saisie comme une chaîne d'entier décimaux). Zéro ou non spécifié signifie une attente indéfinie. Utiliser un décompte de moins de deux secondes n'est pas recommandé.

`options`
Ajout d'options en ligne de commande à envoyer au serveur à l'exécution. Par exemple, en le configurant à `-c geqo=off`, cela configure la valeur de la session pour le paramètre `geqo` à `off`. Pour une discussion détaillée des options disponibles, voir Chapitre 18, Configuration du serveur.

`application_name`
Précise une valeur pour le paramètre de configuration application_name.

`fallback_application_name`
Indique une valeur de secours pour le paramètre de configuration application_name. Cette valeur sera utilisée si aucune valeur n'est donnée à `application_name` via un paramètre de connexion ou la variable d'environnement. L'indication d'un nom de secours est utile pour les programmes outils génériques qui souhaitent configurer un nom d'application par défaut mais permettrait sa surcharge par l'utilisateur.

`keepalives`
Contrôle si les paramètres TCP keepalives côté client sont utilisés. La valeur par défaut est de 1, signifiant ainsi qu'ils sont utilisés. Vous pouvez le configurer à 0, ce qui aura pour effet de les désactiver si vous n'en voulez pas. Ce paramètre est ignoré pour les connexions réalisées via un socket de domaine Unix.

`keepalives_idle`
Contrôle le nombre de secondes d'inactivité après lequel TCP doit envoyer un message keepalive au server. Une valeur de zéro utilise la valeur par défaut du système. Ce paramètre est ignoré pour les connexions réalisées via un socket de domaine Unix ou si les paramètres keepalives sont désactivés. Ce paramètre est uniquement supporté sur les systèmes où les options TCP_KEEPIDLE ou TCP_KEEPALIVE sont disponibles et sur Windows ; pour les autres systèmes, ce paramètre n'a pas d'effet.

`keepalives_interval`
Contrôle le nombre de secondes après lequel un message TCP keepalive doit être retransmis si le serveur ne l'a pas acquitté. Une valeur de zéro utilise la valeur par défaut du système. Ce paramètre est uniquement supporté sur les systèmes où l'option TCP_KEEPINTVL est disponible et sur Windows ; pour les autres systèmes, ce paramètre n'a pas d'effet.

`keepalives_count`
Contrôle le nombre de messages TCP keepalives pouvant être perdus avant que la connexion du client au serveur ne soit considérée comme perdue. Une valeur de zéro utilise la valeur par défaut du système. Ce paramètre est uniquement supporté sur les systèmes où l'option TCP_KEEPCNT est disponible et sur Windows ; pour les autres systèmes, ce paramètre n'a pas d'effet.

`tty`
Ignoré (auparavant, ceci indiquait où envoyer les traces de débogage du serveur).

`sslmode`
Cette option détermine si ou avec quelle priorité une connexion TCP/IP SSL sécurisée sera négociée avec le serveur. Il existe six modes :

Tableau 31.1. Options de `sslmode`

Option	Description
`disable`	essaie seulement une connexion non SSL
`allow`	essaie en premier lieu une connexion non SSL ; si cette tentative échoue, essaie une connexion SSL
`prefer` (default)	essaie en premier lieu une connexion SSL ; si cette tentative échoue, essaie une connexion non SSL
`require`	essaie seulement une connexion SSL. Si un certificat racine d'autorité est présent, vérifie le certificat de la même façon que si `verify-ca` était spécifié
`verify-ca`	essaie seulement une connexion SSL et vérifie que le certificat client est créé par un CA de confiance.
`verify-full`	essaie seulement une connexion SSL, vérifie que le certificat client est créé par un CA de confiance et que le nom du serveur correspond bien à celui du certificat.

Voir Section 31.17, « Support de SSL » pour une description détaillée de comment ces options fonctionnent.

`sslmode` est ignoré pour la communication par socket de domaine Unix. Si PostgreSQL™ est compilé sans le support de SSL, l'utilisation des options `require`, `verify-ca` et `verify-full` causera une erreur alors que les options `allow` et `prefer` seront acceptées mais libpq ne sera pas capable de négocier une connexion SSL.

Cette option est obsolète et remplacée par l'option `sslmode`.

Si initialisée à 1, une connexion SSL au serveur est requise (ce qui est équivalent à un `sslmode require`). libpq refusera alors de se connecter si le serveur n'accepte pas une connexion SSL. Si initialisée à 0 (la valeur par défaut), libpq négociera le type de connexion avec le serveur (équivalent à un `sslmode prefer`). Cette option est seulement disponible si PostgreSQL™ est compilé avec le support SSL.

`sslcert`
Ce paramètre indique le nom du fichier du certificat SSL client, remplaçant le fichier par défaut, `~/.postgresql/postgresql.crt`. Ce paramètre est ignoré si la connexion n'utilise pas SSL.

`sslkey`
Ce paramètre indique l'emplacement de la clé secrète utilisée pour le certificat client. Il peut soit indiquer un nom de fichier qui sera utilisé à la place du fichier `~/.postgresql/postgresql.key` par défaut, soit indiquer un clé obtenue par un moteur externe (les moteurs sont des modules chargeables d'OpenSSL™). La spécification d'un moteur externe devrait consister en un nom de moteur et un identifiant de clé spécifique au moteur, les deux séparés par une virgule. Ce paramètre est ignoré si la connexion n'utilise pas SSL.

`sslrootcert`
Ce paramètre indique le nom d'un fichier contenant le ou les certificats de l'autorité de certificats SSL (CA). Si le fichier existe, le certificat du serveur sera vérifié. La signature devra appartenir à une de ces autorités. La valeur par défaut de ce paramètre est `~/.postgresql/root.crt`.

`sslcrl`
Ce paramètre indique le nom du fichier de la liste de révocation du certificat SSL. Les certificats listés dans ce fichier, s'il existe bien, seront rejetés lors d'une tentative d'authentification avec le certificat du serveur. La valeur par défaut de ce paramètre est `~/.postgresql/root.crl`.

`krbsrvname`
Nom du service Kerberos à utiliser lors de l'authentification avec Kerberos 5 et GSSAPI. Il doit correspondre avec le nom du service spécifié dans la configuration du serveur pour que l'authentification Kerberos puisse réussir (voir aussi la Section 19.3.5, « Authentification Kerberos » et Section 19.3.3, « Authentification GSSAPI ».)

`gsslib`
Bibliothèque GSS à utiliser pour l'authentification GSSAPI. Utilisée seulement sur Windows. Configurer à `gssapi` pour forcer libpq à utiliser la bibliothèque GSSAPI pour l'authentification au lieu de SSPI par défaut.

`service`
Nom du service à utiliser pour des paramètres supplémentaires. Il spécifie un nom de service dans `pg_service.conf` contenant des paramètres de connexion supplémentaires. Ceci permet aux applications de spécifier uniquement un nom de service, donc les paramètres de connexion peuvent être maintenus de façon centrale. Voir Section 31.15, « Fichier des connexions de service ».
Si un paramètre manque, alors la variable d'environnement correspondante est vérifiée (voir la Section 31.13, « Variables d'environnement »). Si elle n'est pas disponible, alors la valeur par défaut indiquée est utilisée.

Si `expand_dbname` est différent de zéro et que *`dbname`* contient un signe =, il est pris en tant que chaîne *`conninfo`* exactement de la même façon qu'il a été passé à `PQconnectdb` (voir ci-dessous). Les mots-clés traités précédemment seront surchargés par les mots-clés de la chaîne *`conninfo`*.

En général, les mots-clés sont traités à partir du début de ces tableaux dans l'ordre de l'index. L'effet qui en découle est que, quand les mots-clés sont répétés, la valeur correspondant au dernier traitement est conservée. Du coup, via un placement attentionné du mot-clé *`dbname`*, il est possible de déterminer ce qui pourrait être surchargé par une chaîne *`conninfo`* et ce qui ne le sera pas.

`PQconnectdb`
Établit une nouvelle connexion à un serveur de bases de données.

```
PGconn *PQconnectdb(const char *conninfo);
```

Cette fonction ouvre une nouvelle connexion à la base de données en utilisant les paramètres pris à partir de la chaîne `conninfo`.

La chaîne passée peut être vide pour utiliser tous les paramètres par défaut ou elle peut contenir un ou plusieurs paramètres, séparés par des espaces blancs. Chaque configuration est de la forme `motclé = valeur`. Les espaces autour du signe égal sont optionnels. Pour écrire une valeur vide ou une valeur contenant des espaces, il est nécessaire de la placer entre guillemets simples, par exemple `motclé = 'une valeur'`. Les guillemets simples et les antislashs dans la valeur doivent être échappés avec un antislash, donc `\'` et `\\`.

Les mots-clés actuellement reconnus sont identiques à ceux indiqués ci-dessus.

`PQsetdbLogin`
Crée une nouvelle connexion sur le serveur de bases de données.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

C'est le prédécesseur de `PQconnectdb` avec un ensemble fixe de paramètres. Cette fonction a les mêmes fonctionnalités sauf que les paramètres manquants seront toujours initialisés avec leur valeurs par défaut. Écrire NULL ou une chaîne vide pour un de ces paramètres fixes dont vous souhaitez utiliser la valeur par défaut.

Si *`dbName`* contient un signe =, il est pris pour une chaîne *`conninfo`* exactement de la même façon que si elle était passée à `PQconnectdb`, et le reste des paramètres est ensuite appliqué comme ci-dessus.

`PQsetdb`
Crée une nouvelle connexion sur le serveur de bases de données.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

C'est une macro faisant appel à `PQsetdbLogin` avec des pointeurs nuls pour les paramètres *`login`* et *`pwd`*. Elle est fournie pour une compatibilité ascendante des très vieux programmes.

`PQconnectStartParams`, `PQconnectStart`, `PQconnectPoll`

Crée une connexion au serveur de bases de données d'une façon non bloquante.

```
PGconn *PQconnectStartParams(const char **keywords, const char **values, int
expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Ces trois fonctions sont utilisées pour ouvrir une connexion au serveur de bases de données d'une façon telle que le thread de votre application n'est pas bloqué sur les entrées/sorties distantes en demandant la connexion. Le but de cette approche est que l'attente de la fin des entrées/sorties peut se faire dans la boucle principale de l'application plutôt qu'à l'intérieur de `PQconnectdbParams` ou `PQconnectdb`, et donc l'application peut gérer des opérations en parallèle à d'autres activités.

Avec `PQconnectStartParams`, la connexion à la base de données est faite en utilisant les paramètres à partir des tableaux `keywords` et `values`, et contrôlée par `expand_dbname`, comme décrit ci-dessus pour `PQconnectdbParams`.

Avec `PQconnectStart`, la connexion à la base de données est faite en utilisant les paramètres provenant de la chaîne `conninfo` comme décrit ci-dessus pour `PQconnectdb`.

Ni `PQconnectStartParams` ni `PQconnectStart` ni `PQconnectPoll` ne bloqueront, aussi longtemps qu'un certain nombre de restrictions est respecté :

- Les paramètres `hostaddr` et `host` sont utilisés de façon appropriée pour vous assurer que la requête de nom et la requête inverse ne soient pas lancées. Voir la documentation de ces paramètres avec `PQconnectdbParams` ci-dessus pour les détails.
- Si vous appelez `PQtrace`, assurez-vous que l'objet de flux dans lequel vous enregistrez les traces ne bloquera pas.
- Assurez-vous que le socket soit dans l'état approprié avant d'appeler `PQconnectPoll`, comme décrit ci-dessous.

Note : l'utilisation de `PQconnectStartParams` est analogue à `PQconnectStart` affichée ci-dessous.

Pour commencer une demande de connexion non bloquante, appelez `conn = PQconnectStart("`*`connection_info_string`*`")`. Si `conn` est nul, alors libpq a été incapable d'allouer une nouvelle structure PGconn. Sinon, un pointeur valide vers une structure PGconn est renvoyé (bien qu'il ne représente pas encore une connexion valide vers la base de données). Au retour de `PQconnectStart`, appelez `status = PQstatus(conn)`. Si `status` vaut CONNECTION_BAD, `PQconnectStart` a échoué.

Si `PQconnectStart` réussit, la prochaine étape est d'appeler souvent libpq de façon à ce qu'il continue la séquence de connexion. Utilisez `PQsocket(conn)` pour obtenir le descripteur de socket sous la connexion à la base de données. Du coup, une boucle : si le dernier retour de `PQconnectPoll(conn)` est PGRES_POLLING_READING, attendez que la socket soit prête pour lire (comme indiqué par `select()`, `poll()` ou une fonction système similaire). Puis, appelez de nouveau `PQconnectPoll(conn)`. En revanche, si le dernier retour de `PQconnectPoll(conn)` est PGRES_POLLING_WRITING, attendez que la socket soit prête pour écrire, puis appelez de nouveau `PQconnectPoll(conn)`. Si vous devez encore appeler `PQconnectPoll`, c'est-à-dire juste après l'appel de `PQconnectStart`, continuez comme s'il avait renvoyé PGRES_POLLING_WRITING. Continuez cette boucle jusqu'à ce que `PQconnectPoll(conn)` renvoie PGRES_POLLING_FAILED, indiquant que la procédure de connexion a échoué ou PGRES_POLLING_OK, indiquant le succès de la procédure de connexion.

À tout moment pendant la connexion, le statut de cette connexion pourrait être vérifié en appelant `PQstatus`. Si le résultat est CONNECTION_BAD, alors la procédure de connexion a échoué ; si, au contraire, elle renvoie `CONNECTION_OK`, alors la connexion est prête. Ces deux états sont détectables à partir de la valeur de retour de `PQconnectPoll`, décrite ci-dessus. D'autres états pourraient survenir lors (et seulement dans ce cas) d'une procédure de connexion asynchrone. Ils indiquent l'état actuel de la procédure de connexion et pourraient être utile pour fournir un retour à l'utilisateur. Ces statuts sont :

CONNECTION_STARTED
Attente de la connexion à réaliser.

CONNECTION_MADE
Connexion OK ; attente d'un envoi.

CONNECTION_AWAITING_RESPONSE
Attente d'une réponse du serveur.

CONNECTION_AUTH_OK
Authentification reçue ; attente de la fin du lancement du moteur.

CONNECTION_SSL_STARTUP
Négociation du cryptage SSL.

CONNECTION_SETENV
Négociation des paramétrages de l'environnement.

Notez que, bien que ces constantes resteront (pour maintenir une compatibilité), une application ne devrait jamais se baser sur un ordre pour celles-ci ou sur tout ou sur le fait que le statut fait partie de ces valeurs documentés. Une application pourrait faire quelque chose comme ça :

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connexion en cours...";
        break;

    case CONNECTION_MADE:
        feedback = "Connecté au serveur...";
        break;
.
.
.
    default:
        feedback = "Connexion...";
}
```

Le paramètre de connexion `connect_timeout` est ignoré lors de l'utilisation `PQconnectPoll` ; c'est de la responsabilité de l'application de décider quand une période de temps excessive s'est écoulée. Sinon, `PQconnectStart` suivi par une boucle `PQconnectPoll` est équivalent à `PQconnectdb`.

Notez que si `PQconnectStart` renvoie un pointeur non nul, vous devez appeler `PQfinish` lorsque vous en avez terminé avec lui, pour supprimer la structure et tous les blocs mémoires qui lui sont associés. Ceci doit être fait même si la tentative de connexion échoue ou est abandonnée.

31.2. Fonctions de statut de connexion

Ces fonctions sont utilisées pour interroger le statut d'un objet de connexion existant.

> **Astuce**
>
> Les développeurs d'application libpq devraient être attentif au maintien de leur abstraction PGconn. Utilisez les fonctions d'accès décrites ci-dessous pour obtenir le contenu de PGconn. Référence les champs internes de PGconn en utilisant `libpq-int.h` n'est pas recommandé parce qu'ils sont sujets à modification dans le futur.

Les fonctions suivantes renvoient les valeurs des paramètres utilisés pour la connexion. Ces valeurs sont fixes pour la durée de vie de l'objet PGconn.

`PQdb`
Renvoie le nom de la base de données de la connexion.

```
char *PQdb(const PGconn *conn);
```

`PQuser`
Renvoie le nom d'utilisateur utilisé pour la connexion.

```
char *PQuser(const PGconn *conn);
```

`PQpass`

Renvoie le mot de passe utilisé pour la connexion.

```
char *PQpass(const PGconn *conn);
```

`PQhost`

Renvoie le nom d'hôte du serveur utilisé pour la connexion.

```
char *PQhost(const PGconn *conn);
```

`PQport`

Renvoie le numéro de port utilisé pour la connexion.

```
char *PQport(const PGconn *conn);
```

`PQtty`

Renvoie le TTY de débogage pour la connexion (ceci est obsolète car le serveur ne fait plus attention au paramétrage du TTY mais les fonctions restent pour des raisons de compatibilité ascendante).

```
char *PQtty(const PGconn *conn);
```

`PQoptions`

Renvoie les options en ligne de commande passées lors de la demande de connexion.

```
char *PQoptions(const PGconn *conn);
```

Les fonctions suivantes renvoient le statut car il peut changer suite à l'exécution d'opérations sur l'objet PGconn.

`PQstatus`

Renvoie l'état de la connexion.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Le statut peut faire partie d'un certain nombre de valeurs. Néanmoins, seules deux ne concernent pas les procédures de connexion asynchrone : `CONNECTION_OK` et `CONNECTION_BAD`. Une bonne connexion de la base de données a l'état `CONNECTION_OK`. Une tentative échouée de connexion est signalée par le statut `CONNECTION_BAD`. D'habitude, un état OK restera ainsi jusqu'à `PQfinish` mais un échec de communications pourrait résulter en un statut changeant prématurément `CONNECTION_BAD`. Dans ce cas, l'application pourrait essayer de récupérer en appelant `PQreset`.

Voir l'entrée de `PQconnectStartParams`, `PQconnectStart` et de `PQconnectPoll` en regard aux autres codes de statut, qui pourraient être vus.

`PQtransactionStatus`

Renvoie l'état actuel de la transaction du serveur.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

Le statut peut être `PQTRANS_IDLE` (actuellement inactif), `PQTRANS_ACTIVE` (une commande est en cours), `PQTRANS_INTRANS` (inactif, dans un bloc valide de transaction) ou `PQTRANS_INERROR` (inactif, dans un bloc de transaction échoué). `PQTRANS_UNKNOWN` est reporté si la connexion est mauvaise. `PQTRANS_ACTIVE` est reporté seulement quand une requête a été envoyée au serveur mais qu'elle n'est pas terminée.

Attention

`PQtransactionStatus` donnera des résultats incorrects lors de l'utilisation d'un serveur PostgreSQL™ 7.3 qui a désactivé le paramètre `autocommit`. La fonctionnalité autocommit, côté serveur, est obsolète et n'existe pas dans les versions serveur ultérieures.

`PQparameterStatus`

Recherche un paramétrage actuel du serveur.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Certaines valeurs de paramètres sont reportées par le serveur automatiquement ou lorsque leur valeurs changent. `PQparameterStatus` peut être utilisé pour interroger ces paramétrages. Il renvoie la valeur actuelle d'un paramètre s'il est connu et NULL si le paramètre est inconnu.

Les paramètres reportés pour la version actuelle incluent `server_version`, `server_encoding`, `client_encoding`, `application_name`, `is_superuser`, `session_authorization`, `datestyle`, `IntervalStyle`, `TimeZone`,

integer_datetimes et standard_conforming_strings. (server_encoding, TimeZone et integer_datetimes n'étaient pas rapportés dans les versions antérieures à la 8.0 ; standard_conforming_strings n'était pas rapporté dans les versions antérieures à la 8.1; IntervalStyle n'était pas rapporté dans les versions antérieures à la 8.4; application_name n'était pas rapporté dans les versions antérieures à la 9.0). Notez que server_version, server_encoding et integer_datetimes ne peuvent pas changer après le lancement du serveur.

Les serveurs utilisant un protocole antérieur à la 3.0 ne reportent pas la configuration des paramètres mais libpq inclut la logique pour obtenir des valeurs pour server_version et client_encoding. Les applications sont encouragées à utiliser PQparameterStatus plutôt qu'un code *ad-hoc* modifiant ces valeurs (néanmoins, attention, les connexions pré-3.0, changeant client_encoding via **SET** après le lancement de la connexion, ne seront pas reflétées par PQparameterStatus). Pour server_version, voir aussi PQserverVersion, qui renvoie l'information dans un format numérique qui est plus facile à comparer.

Si aucune valeur n'est indiquée pour standard_conforming_strings, les applications pourraient supposer qu'elle vaut off, c'est-à-dire que les antislashs sont traités comme des échappements dans les chaînes littérales. De plus, la présence de ce paramètre pourrait être pris comme une indication que la syntaxe d'échappement d'une chaîne (E'...') est acceptée.

Bien que le pointeur renvoyé est déclaré const, il pointe en fait vers un stockage mutable associé avec la structure PGconn. Il est déconseillé de supposer que le pointeur restera valide pour toutes les requêtes.

PQprotocolVersion

Interroge le protocole interface/moteur lors de son utilisation.

```
int PQprotocolVersion(const PGconn *conn);
```

Les applications souhaitent utiliser ceci pour déterminer si certaines fonctionnalités sont supportées. Actuellement, les seules valeurs possible sont 2 (protocole 2.0), 3 (protocole 3.0) ou zéro (mauvaise connexion). Ceci ne changera pas après la fin du lancement de la connexion mais cela pourrait être changé théoriquement avec une réinitialisation de la connexion. Le protocole 3.0 sera normalement utilisé lors de la communication avec les serveurs PostgreSQL™ 7.4 ou ultérieures ; les serveurs antérieurs à la 7.4 supportent uniquement le protocole 2.0 (le protocole 1.0 est obsolète et non supporté par libpq).

PQserverVersion

Renvoie un entier représentant la version du moteur.

```
int PQserverVersion(const PGconn *conn);
```

Les applications pourraient utiliser ceci pour déterminer la version du serveur de la base de données auquel ils sont connectés. Le numéro est formé en convertissant les nombres majeur, mineur et de révision en un nombre à deux chiffres décimaux et en leur assemblant. Par exemple, la version 8.1.5 sera renvoyée en tant que 80105 et la version 8.2 sera renvoyée en tant que 80200 (les zéros au début ne sont pas affichés). Zéro est renvoyée si la connexion est mauvaise.

PQerrorMessage

Renvoie le dernier message d'erreur généré par une opération sur la connexion.

```
char *PQerrorMessage(const PGconn* conn);
```

Pratiquement toutes les fonctions libpq initialiseront un message pour PQerrorMessage en cas d'échec. Notez que, par la convention libpq, un résultat non vide de PQerrorMessage peut être sur plusieurs lignes et contiendra un retour chariot à la fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée PGconn associée est passée à PQfinish. Vous ne devriez pas supposer que la chaîne résultante reste identique suite à toutes les opérations sur la structure PGconn.

PQsocket

Obtient le descripteur de fichier du socket de la connexion au serveur. Un descripteur valide sera plus grand ou égal à 0 ; un résultat de -1 indique qu'aucune connexion au serveur n'est actuellement ouverte (ceci ne changera pas lors de l'opération normale mais pourra changer lors d'une configuration de l'initialisation ou lors d'une réinitialisation).

```
int PQsocket(const PGconn *conn);
```

PQbackendPID

Renvoie l'identifiant du processus (PID) du serveur gérant cette connexion.

```
int PQbackendPID(const PGconn *conn);
```

Le PID du moteur est utile pour des raisons de débogage et pour la comparaison avec les messages **NOTIFY** (qui incluent le PID du processus serveur lançant la notification). Notez que le PID appartient à un processus exécuté sur l'hôte du serveur de bases de données et non pas sur l'hôte local !

PQconnectionNeedsPassword

Renvoie true (1) si la méthode d'authentification de la connexion nécessite un mot de passe, mais qu'aucun n'est disponible.

Renvoie false (0) sinon.

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

Cette fonction peut être utilisée après une tentative échouée de connexion pour décider de la demande d'un utilisateur pour un mot de passe.

`PQconnectionUsedPassword`
Renvoie true (1) si la méthode d'authentification de la connexion a utilisé un mot de passe. Renvoie false (0) sinon.

```
int PQconnectionUsedPassword(const PGconn *conn);
```

Cette fonction peut être utilisée après une connexion, réussie ou en échec, pour détecter si le serveur demande un mot de passe.

`PQgetssl`
Retourne la structure SSL utilisée dans la connexion ou NULL si SSL n'est pas utilisé.

```
SSL *PQgetssl(const PGconn *conn);
```

Cette structure peut être utilisée pour vérifier les niveaux de cryptage, pour vérifier les certificats du serveur, et plus. Référez-vous à la documentation d'OpenSSL™ pour plus d'informations sur cette structure.

Vous pouvez définir USE_SSL pour obtenir le bon prototype de cette fonction. Faire cela inclura automatiquement `ssl.h` à partir d'OpenSSL™.

31.3. Fonctions de commandes d'exécution

Une fois la connexion au serveur de la base de données établie avec succès, les fonctions décrites ici sont utilisées pour exécuter les requêtes SQL et les commandes.

31.3.1. Fonctions principales

`PQexec`
Soumet une commande au serveur et attend le résultat.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Renvoie un pointeur PGresult ou peut-être un pointeur NULL. Un pointeur non NULL sera généralement renvoyé sauf dans des conditions particulières comme un manque de mémoire ou lors d'erreurs sérieuses telles que l'incapacité à envoyer la commande au serveur. Si un pointeur NULL est renvoyé, il devrait être traité comme un résultat PGRES_FATAL_ERROR. Utilisez `PQerrorMessage` pour obtenir plus d'informations sur l'erreur.

Inclure plusieurs commandes SQL (séparées par des points virgules) dans la chaîne de commande est autorisé. Les requêtes multiples envoyées dans un simple appel à `PQexec` sont exécutées dans une seule transaction sauf si des commandes explicites **BEGIN**/**COMMIT** sont incluses dans la chaîne de requête pour la diviser dans de nombreuses transactions. Néanmoins, notez que la structure PGresult renvoyée décrit seulement le résultat de la dernière commande exécutée à partir de la chaîne. Si une des commandes doit échouer, l'exécution de la chaîne s'arrête et le PGresult renvoyé décrit la condition d'erreur.

`PQexecParams`
Soumet une commande au serveur et attend le résultat, avec la possibilité de passer des paramètres séparément du texte de la commande SQL.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` est identique à `PQexec` mais offre des fonctionnalités supplémentaires : des valeurs de paramètres peuvent être spécifiées séparément de la chaîne de commande et les résultats de la requête peuvent être demandés soit au format texte soit au format binaire. `PQexecParams` est supporté seulement dans les connexions avec le protocole 3.0 et ses versions ultérieures ; elle échouera lors de l'utilisation du protocole 2.0.

Voici les arguments de la fonction :

`conn`
L'objet connexion où envoyer la commande.

`command`
La chaîne SQL à exécuter. Si les paramètres sont utilisés, ils sont référencés dans la chaîne avec `$1`, `$2`, etc.

`nParams`
Le nombre de paramètres fournis ; il s'agit de la longueur des tableaux *`paramTypes[]`*, *`paramValues[]`*, *`paramLengths[]`* et *`paramFormats[]`*. (Les pointeurs de tableau peuvent être NULL quand *`nParams`* vaut zéro.)

`paramTypes[]`
Spécifie, par OID, les types de données à affecter aux symboles de paramètres. Si *`paramTypes`* est NULL ou si tout élément spécifique du tableau est zéro, le serveur infère un type de donnée pour le symbole de paramètre de la même façon qu'il le ferait pour une chaîne litérale sans type.

`paramValues[]`
Spécifie les vraies valeurs des paramètres. Un pointeur nul dans ce tableau signifie que le paramètre correspondant est NULL ; sinon, le pointeur pointe vers une chaîne texte terminée par un octet nul (pour le format texte) ou vers des données binaires dans le format attendu par le serveur (pour le format binaire).

`paramLengths[]`
Spécifie les longueurs des données réelles des paramètres du format binaire. Il est ignoré pour les paramètres NULL et les paramètres de format texte. Le pointeur du tableau peut être NULL quand il n'y a pas de paramètres binaires.

`paramFormats[]`
Spécifie si les paramètres sont du texte (placez un zéro dans la ligne du tableau pour le paramètre correspondant) ou binaire (placez un un dans la ligne du tableau pour le paramètre correspondant). Si le pointeur du tableau est nul, alors tous les paramètres sont présumés être des chaînes de texte.

Les valeurs passées dans le format binaire nécessitent de connaître la représentation interne attendue par le moteur. Par exemple, les entiers doivent être passés dans l'ordre réseau pour les octets. Passer des valeurs numeric requiert de connaître le format de stockage du serveur, comme implémenté dans `src/backend/utils/adt/numeric.c::numeric_send()` et `src/backend/utils/adt/numeric.c::numeric_recv()`.

`resultFormat`
Indiquez zéro pour obtenir les résultats dans un format texte et un pour les obtenir dans un format binaire. (Il n'est actuellement pas possible d'obtenir des formats différents pour des colonnes de résultats différentes bien que le protocole le permette.)

Le principal avantage de `PQexecParams` sur `PQexec` est que les valeurs de paramètres pourraient être séparés à partir de la chaîne de commande, évitant ainsi le besoin de guillemets et d'échappements.

Contrairement à `PQexec`, `PQexecParams` autorise au plus une commande SQL dans une chaîne donnée (il peut y avoir des points-virgules mais pas plus d'une commande non vide). C'est une limitation du protocole sous-jacent mais cela a quelque utilité comme défense supplémentaire contre les attaques par injection de SQL.

Astuce

Spécifier les types de paramètres via des OID est difficile, tout particulièrement si vous préférez ne pas coder en dur les valeurs OID particulières dans vos programmes. Néanmoins, vous pouvez éviter de le faire même dans des cas où le serveur lui-même ne peut pas déterminer le type du paramètre ou choisit un type différent de celui que vous voulez. Dans le texte de commande SQL, attachez une conversion explicite au symbole de paramètre pour montrer le type de données que vous enverrez. Par exemple :

```
SELECT * FROM ma_table WHERE x = $1::bigint;
```

Ceci impose le traitement du paramètre `$1` en tant que bigint alors que, par défaut, il se serait vu affecté le même type que `x`. Forcer la décision du type de paramètre, soit de cette façon soit en spécifiant l'OID du type numérique, est fortement recommandé lors de l'envoi des valeurs des paramètres au format binaire car le format binaire a moins de redondance que le format texte et, du coup, il y a moins de chance que le serveur détecte une erreur de correspondance de type pour vous.

`PQprepare`

Soumet une requête pour créer une instruction préparée avec les paramètres donnés et attends la fin de son exécution.

```
PGresult *PQprepare(PGconn *conn,
        const char *stmtName,
        const char *query,
        int nParams,
        const Oid *paramTypes);
```

`PQprepare` crée une instruction préparée pour une exécution ultérieure avec `PQexecPrepared`. Cette fonction autorise les commandes utilisées de façon répété à être analysées et planifiées qu'une seule fois, plutôt qu'à chaque exécution. `PQprepare` est uniquement supporté par les connexions utilisant le protocole 3.0 et ses versions ultérieures ; elle échouera avec le protocole 2.0.

La fonction crée une instruction préparée nommée *`stmtName`* à partir de la chaîne *`query`*, devant contenir une seule commande SQL. *`stmtName`* pourrait être une chaîne vide pour créer une instruction non nommée, auquel cas toute instruction non nommée déjà existante est automatiquement remplacée par cette dernière. Une erreur sera rapportée si le nom de l'instruction est déjà définie dans la session en cours. Si des paramètres sont utilisés, ils sont référencés dans la requête avec `$1`, `$2`, etc. *`nParams`* est le nombre de paramètres pour lesquels des types sont prédéfinis dans le tableau *`paramTypes[]`* (le pointeur du tableau pourrait être NULL quand *`nParams`* vaut zéro). *`paramTypes[]`* spécifie les types de données à affecter aux symboles de paramètres par leur OID. Si *`paramTypes`* est NULL ou si un élément particulier du tableau vaut zéro, le serveur affecte un type de données au symbole du paramètre de la même façon qu'il le ferait pour une chaîne littérale non typée. De plus, la requête pourrait utiliser des symboles de paramètre avec des nombres plus importants que *`nParams`* ; les types de données seront aussi inférés pour ces symboles. (Voir `PQdescribePrepared` comme un moyen de trouver les types de données inférés.)

Comme avec `PQexec`, le résultat est normalement un objet PGresult dont le contenu indique le succès ou l'échec côté serveur. Un résultat NULL indique un manque de mémoire ou une incapacité à envoyer la commande. Utilisez `PQerrorMessage` pour obtenir plus d'informations sur de telles erreurs.

Les instructions préparées avec `PQexecPrepared` peuvent aussi être créées en exécutant les instructions SQL PREPARE(7). De plus, bien qu'il n'y ait aucune fonction libpq pour supprimer une instruction préparée, l'instruction SQL DEALLOCATE(7) peut être utilisée dans ce but.

`PQexecPrepared`

Envoie une requête pour exécuter une instruction séparée avec les paramètres donnés, et attend le résultat.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

`PQexecPrepared` est identique à `PQexecParams` mais la commande à exécuter est spécifiée en nommant l'instruction préparée précédemment au lieu de donner une chaîne de requête. Cette fonctionnalité permet aux commandes utilisées de façon répétée d'être analysées et planifiées seulement une fois plutôt que chaque fois qu'ils sont exécutés. L'instruction doit avoir été préparée précédemment dans la session en cours. `PQexecPrepared` est supporté seulement dans les connexions du protocole 3.0 et ses versions ultérieures ; il échouera lors de l'utilisation du protocole 2.0.

Les paramètres sont identiques à `PQexecParams`, sauf que le nom d'une instruction préparée est donné au lieu d'une chaîne de requête et le paramètre *`paramTypes[]`* n'est pas présente (il n'est pas nécessaire car les types des paramètres de l'instruction préparée ont été déterminés à la création).

`PQdescribePrepared`

Soumet une requête pour obtenir des informations sur l'instruction préparée indiquée et attend le retour de la requête.

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

`PQdescribePrepared` permet à une application d'obtenir des informations si une instruction préparée précédemment. `PQdescribePrepared` est seulement supporté avec des connexions utilisant le protocole 3.0 et ultérieures ; il échouera lors de l'utilisation du protocole 2.0.

`stmtName` peut être `""` ou NULL pour référencer l'instruction non nommée. Sinon, ce doit être le nom d'une instruction pré-

parée existante. En cas de succès, un PGresult est renvoyé avec le code retour PGRES_COMMAND_OK. Les fonctions PQnparams et PQparamtype peuvent utiliser ce PGresult pour obtenir des informations sur les paramètres d'une instruction préparée, et les fonctions PQnfields, PQfname, PQftype, etc fournissent des informations sur les colonnes résultantes (au cas où) de l'instruction.

PQdescribePortal
Soumet une requête pour obtenir des informations sur le portail indiqué et attend le retour de la requête.

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal permet à une application d'obtenir des informations sur un portail précédemment créé. (libpq ne fournit pas d'accès direct aux portails mais vous pouvez utiliser cette fonction pour inspecter les propriétés d'un curseur créé avec la commande SQL **DECLARE CURSOR**.) PQdescribePortal est seulement supporté dans les connexions via le protocole 3.0 et ultérieurs ; il échouera lors de l'utilisation du protocole 2.0.

portalName peut être "" ou NULL pour référencer un portail sans nom. Sinon, il doit correspondre au nom d'un portail existant. En cas de succès, un PGresult est renvoyé avec le code de retour PGRES_COMMAND_OK. Les fonctions PQnfields, PQfname, PQftype, etc peuvent utiliser ce PGresult pour obtenir des informations sur les colonnes résultats (au cas où) du portail.

La structure PGresult encapsule le résultat renvoyé par le serveur. Les développeurs d'applications libpq devraient faire attention au maintien de l'abstraction de PGresult. Utilisez les fonctions d'accès ci-dessous pour obtenir le contenu de PGresult. Évitez la référence aux champs de la structure PGresult car ils sont sujets à des changements dans le futur.

PQresultStatus
Renvoie l'état du résultat d'une commande.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus peut renvoyer une des valeurs suivantes :

PGRES_EMPTY_QUERY
La chaîne envoyée au serveur était vide.

PGRES_COMMAND_OK
Fin avec succès d'une commande ne renvoyant aucune donnée.

PGRES_TUPLES_OK
Fin avec succès d'une commande renvoyant des données (telle que **SELECT** ou **SHOW**).

PGRES_COPY_OUT
Début de l'envoi (à partir du serveur) d'un flux de données.

PGRES_COPY_IN
Début de la réception (sur le serveur) d'un flux de données.

PGRES_BAD_RESPONSE
La réponse du serveur n'a pas été comprise.

PGRES_NONFATAL_ERROR
Une erreur non fatale (une note ou un avertissement) est survenue.

PGRES_FATAL_ERROR
Une erreur fatale est survenue.
Si le statut du résultat est PGRES_TUPLES_OK, alors les fonctions décrites ci-dessous peuvent être utilisées pour récupérer les lignes renvoyées par la requête. Notez qu'une commande **SELECT** qui arrive à récupérer aucune ligne affichera toujours PGRES_TUPLES_OK. PGRES_COMMAND_OK est pour les commandes qui ne peuvent jamais renvoyer de lignes (**INSERT**, **UPDATE**, etc.). Une réponse PGRES_EMPTY_QUERY pourrait indiquer un bogue dans le logiciel client.

Un résultat de statut PGRES_NONFATAL_ERROR ne sera jamais renvoyé directement par PQexec ou d'autres fonctions d'exécution de requêtes ; les résultats de ce type sont passés à l'exécuteur de notifications (voir la Section 31.11, « Traitement des messages »).

PQresStatus
Convertit le type énuméré renvoyé par PQresultStatus en une constante de type chaîne décrivant le code d'état. L'appelant ne devrait pas libérer le résultat.

```
char *PQresStatus(ExecStatusType status);
```

`PQresultErrorMessage`

Renvoie le message d'erreur associé avec la commande ou une chaîne vide s'il n'y a pas eu d'erreurs.

```
char *PQresultErrorMessage(const PGresult *res);
```

S'il y a eu une erreur, la chaîne renvoyée incluera un retour chariot en fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée PGresult associée est passée à `PQclear`.

Suivant immédiatement un appel à `PQexec` ou `PQgetResult`, `PQerrorMessage` (sur la connexion) renverra la même chaîne que `PQresultErrorMessage` (sur le résultat). Néanmoins, un PGresult conservera son message d'erreur jusqu'à destruction alors que le message d'erreur de la connexion changera lorsque des opérations suivantes seront réalisées. Utiliser `PQresultErrorMessage` quand vous voulez connaître le statut associé avec un PGresult particulier ; utilisez `PQerrorMessage` lorsque vous souhaitez connaître le statut à partir de la dernière opération sur la connexion.

`PQresultErrorField`

Renvoie un champ individuel d'un rapport d'erreur.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

`fieldcode` est un identifiant de champ d'erreur ; voir les symboles listés ci-dessous. NULL est renvoyé si PGresult n'est pas un résultat d'erreur ou d'avertissement, ou n'inclut pas le champ spécifié. Les valeurs de champ n'incluront normalement pas un retour chariot en fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée PGresult associée est passée à `PQclear`.

Les codes de champs suivants sont disponibles :

PG_DIAG_SEVERITY
La sévérité ; le contenu du champ peut être `ERROR`, `FATAL` ou `PANIC` dans un message d'erreur, ou `WARNING`, `NOTICE`, `DEBUG`, `INFO` ou `LOG` dans un message de notification, ou une traduction localisée de ceux-ci. Toujours présent.

PG_DIAG_SQLSTATE

Le code SQLSTATE de l'erreur. Ce code identifie le type d'erreur qui est survenu ; il peut être utilisé par des interfaces qui réalisent les opérations spécifiques (telles que la gestion des erreurs) en réponse à une erreur particulière de la base de données. Pour une liste des codes SQLSTATE possibles, voir l'Annexe A, Codes d'erreurs de PostgreSQL™. Ce champ n'est pas localisable et est toujours présent.

PG_DIAG_MESSAGE_PRIMARY
Le principal message d'erreur, compréhensible par un humain (typiquement sur une ligne). Toujours présent.

PG_DIAG_MESSAGE_DETAIL
Détail : un message d'erreur secondaire et optionnel proposant plus d'informations sur le problème. Pourrait être composé de plusieurs lignes.

PG_DIAG_MESSAGE_HINT
Astuce : une suggestion supplémentaire sur ce qu'il faut faire suite à ce problème. Elle a pour but de différer du détail car elle offre un conseil (potentiellement inapproprié) plutôt que des faits établis. Pourrait être composé de plusieurs lignes.

PG_DIAG_STATEMENT_POSITION
Une chaîne contenant un entier décimal indiquant le position du curseur d'erreur comme index dans la chaîne d'instruction originale. Le premier caractère se trouve à l'index 1 et les positions sont mesurées en caractères, et non pas en octets.

PG_DIAG_INTERNAL_POSITION
Ceci est défini de la même façon que le champ PG_DIAG_STATEMENT_POSITION mais c'est utilisé quand la position du curseur fait référence à une commande générée en interne plutôt qu'une soumise par le client. Le champ PG_DIAG_INTERNAL_QUERY apparaîtra toujours quand ce champ apparaît.

PG_DIAG_INTERNAL_QUERY
Le texte d'une commande échouée, générée en interne. Ceci pourrait être, par exemple, une requête SQL lancée par une fonction PL/pgSQL.

PG_DIAG_CONTEXT
Une indication du contexte dans lequel l'erreur est apparue. Actuellement, cela inclut une trace de la pile d'appels des fonctions actives de langages de procédures et de requêtes générées en interne. La trace a une entrée par ligne, la plus récente se trouvant au début.

PG_DIAG_SOURCE_FILE
Le nom du fichier contenant le code source où l'erreur a été rapportée.

PG_DIAG_SOURCE_LINE
Le numéro de ligne dans le code source où l'erreur a été rapportée.

PG_DIAG_SOURCE_FUNCTION
Le nom de la fonction dans le code source où l'erreur a été rapportée.

Le client est responsable du formatage des informations affichées suivant à ses besoins ; en particulier, il doit supprimer les longues lignes si nécessaires. Les caractères de retour chariot apparaissant dans les champs de message d'erreur devraient être traités comme des changements de paragraphes, pas comme des changements de lignes.

Les erreurs générées en interne par libpq auront une sévérité et un message principal mais aucun autre champ. Les erreurs renvoyées par un serveur utilisant un protocole antérieure à la 3.0 inclueront la sévérité, le message principal et, quelques fois, un message détaillé mais aucun autre champ.

Notez que les champs d'erreurs sont seulement disponibles pour les objets PGresult, et non pas pour les objets PGconn ; il n'existe pas de fonction `PQerrorField`.

`PQclear`
Libère le stockage associé avec un PGresult. Chaque résultat de commande devrait être libéré via `PQclear` lorsqu'il n'est plus nécessaire.

```
void PQclear(PGresult *res);
```

Vous pouvez conserver un objet PGresult aussi longtemps que vous en avez besoin ; il ne part pas lorsque vous lancez une nouvelle commande, même pas si vous fermez la connexion. Pour vous en débarrasser, vous devez appeler `PQclear`. En cas d'oubli, ceci résultera en des pertes mémoires pour votre application.

31.3.2. Récupérer l'information provenant des résultats des requêtes

Ces fonctions sont utilisées pour extraire des informations provenant d'un objet PGresult représentant un résultat valide pour une requête (statut `PGRES_TUPLES_OK`). Ils peuvent aussi être utilisés pour extraire des informations à partir d'une opération Describe réussie : le résultat d'un Describe a les mêmes informations de colonnes qu'une exécution réelle de la requête aurait fournie, mais elle ne renvoie pas de lignes. Pour les objets ayant d'autres valeurs de statut, ces fonctions agiront comme si le résultat n'avait aucune ligne et aucune colonne.

`PQntuples`
Renvoie le nombre de lignes (tuples) du résultat de la requête. Comme elle envoie un entier, les gros ensembles de résultat pourraient dépasser la limite des valeurs renvoyées sur les systèmes d'exploitation 32 bits.

```
int PQntuples(const PGresult *res);
```

`PQnfields`
Renvoie le nombre de colonnes (champs) de chaque ligne du résultat de la requête.

```
int PQnfields(const PGresult *res);
```

`PQfname`
Renvoie le nom de la colonne associé avec le numéro de colonne donnée. Les numéros de colonnes commencent à zéro. L'appelant ne devrait pas libérer directement le numéro. Il sera libéré quand la poignée PGresult associée est passée à `PQclear`.

```
char *PQfname(const PGresult *res,
              int column_number);
```

NULL est renvoyé si le numéro de colonne est en dehors de la plage.

`PQfnumber`
Renvoie le numéro de colonne associé au nom de la colonne donné.

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

-1 est renvoyé si le nom donné ne correspond à aucune colonne.

Le nom donné est traité comme un identifiant dans une commande SQL, c'est-à-dire qu'il est mis en minuscule sauf s'il est entre des guillemets doubles. Par exemple, pour le résultat de la requête suivante

```
SELECT 1 AS FOO, 2 AS "BAR";
```

nous devons obtenir les résultats suivants :

```
PQfname(res, 0)              foo
```

```
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

`PQftable`

Renvoie l'OID de la table à partir de laquelle la colonne donnée a été récupérée. Les numéros de colonnes commencent à zéro mais les colonnes des tables ont des numéros différents de zéro.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

`InvalidOid` est renvoyé si le numéro de colonne est en dehors de la plage ou si la colonne spécifiée n'est pas une simple référence à une colonne de table, ou lors de l'utilisation d'un protocole antérieur à la version 3.0. Vous pouvez lancer des requêtes vers la table système `pg_class` pour déterminer exactement quelle table est référencée.

Le type Oid et la constante `InvalidOid` sera définie lorsque vous incluerez le fichier d'en-tête libpq. Ils auront le même type entier.

`PQftablecol`

Renvoie le numéro de colonne (à l'intérieur de la table) de la colonne correspondant à la colonne spécifiée de résultat de la requête. Les numéros de la colonne résultante commencent à 0.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

Zéro est renvoyé si le numéro de colonne est en dehors de la plage, ou si la colonne spécifiée n'est pas une simple référence à une colonne de table, ou lors de l'utilisation d'un protocole antérieur à la version 3.0.

`PQfformat`

Renvoie le code de format indiquant le format de la colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Le code de format zéro indique une représentation textuelle des données alors qu'un code de format un indique une représentation binaire (les autres codes sont réservés pour des définitions futures).

`PQftype`

Renvoie le type de données associé avec le numéro de colonne donné. L'entier renvoyé est le numéro OID interne du type. Les numéros de colonnes commencent à zéro.

```
Oid PQftype(const PGresult *res,
            int column_number);
```

Vous pouvez lancer des requêtes sur la table système `pg_type` pour obtenir les noms et propriétés des différents types de données. Les OID des types de données intégrés sont définis dans le fichier `src/include/catalog/pg_type.h` de la distribution des sources.

`PQfmod`

Renvoie le modificateur de type de la colonne associée avec le numéro de colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfmod(const PGresult *res,
           int column_number);
```

L'interprétation des valeurs du modificateur est spécifique au type ; elles indiquent la précision ou les limites de taille. La valeur -1 est utilisée pour indiquer qu'« aucune information n'est disponible ». La plupart des types de données n'utilisent pas les modificateurs, auquel cas la valeur est toujours -1.

`PQfsize`

Renvoie la taille en octets de la colonne associée au numéro de colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize` renvoie l'espace alloué pour cette colonne dans une ligne de la base de données, en d'autres termes la taille de la représentation interne du serveur du type de données (de façon cohérente, ce n'est pas réellement utile pour les clients). Une va-

leur négative indique que les types de données ont une longueur variable.

`PQbinaryTuples`

Renvoie 1 si PGresult contient des données binaires et 0 s'il contient des données texte.

```
int PQbinaryTuples(const PGresult *res);
```

Cette fonction est obsolète (sauf dans le cas d'une utilisation en relation avec **COPY**) car un seul PGresult peut contenir du texte dans certaines colonnes et des données binaires dans d'autres. `PQfformat` est la fonction préférée. `PQbinaryTuples` renvoie 1 seulement si toutes les colonnes du résultat sont dans un format binaire (format 1).

`PQgetvalue`

Renvoie la valeur d'un seul champ d'une seule ligne d'un PGresult. Les numéros de lignes et de colonnes commencent à zéro. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée PGresult associée est passée à `PQclear`.

```
char* PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Pour les données au format texte, la valeur renvoyée par `PQgetvalue` est une représentation au format chaîne de caractères terminée par un octet nul de la valeur du champ. Pour les données au format binaire, la valeur dans la représentation binaire est déterminée par le type de la donnée, fonctions `typsend` et `typreceive` (la valeur est en fait suivie d'un octet zéro dans ce cas aussi mais ce n'est pas réellement utile car la valeur a des chances de contenir d'autres valeurs NULL embarquées).

Une chaîne vide est renvoyée si la valeur du champ est NULL. Voir `PQgetisnull` pour distinguer les valeurs NULL des valeurs de chaîne vide.

Le pointeur renvoyé par `PQgetvalue` pointe vers le stockage qui fait partie de la structure PGresult. Personne ne devrait modifier les données vers lesquelles il pointe et tout le monde devrait copier explicitement les données dans un autre stockage s'il n'est pas utilisé après la durée de vie de la struture PGresult.

`PQgetisnull`

Teste un champ pour savoir s'il est nul. Les numéros de lignes et de colonnes commencent à zéro.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

Cette fonction renvoie 1 si le champ est nul et 0 s'il contient une valeur non NULL (notez que `PQgetvalue` renverra une chaîne vide, et non pas un pointeur nul, pour un champ nul).

`PQgetlength`

Renvoie la longueur réelle de la valeur d'un champ en octet. Les numéros de lignes et de colonnes commencent à zéro.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

C'est la longueur réelle des données pour la valeur particulière des données, c'est-à-dire la taille de l'objet pointé par `PQgetvalue`. Pour le format textuel, c'est identique à `strlen()`. Pour le format binaire, c'est une information essentielle. Notez que *personne* ne devrait se fier à `PQfsize` pour obtenir la taille réelle des données.

`PQnparams`

Renvoie le nombre de paramètres d'une instruction préparée.

```
int PQnparams(const PGresult *res);
```

Cette fonction est seulement utile pour inspecter le résultat de `PQdescribePrepared`. Pour les autres types de requêtes, il renverra zéro.

`PQparamtype`

Renvoie le type de donnée du paramètre indiqué de l'instruction. Le numérotage des paramètres commence à 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

Cette fonction est seulement utile pour inspecyer le résultat de `PQdescribePrepared`. Pour les autres types de requêtes, il renverra zéro.

`PQprint`

Affiche toutes les lignes et, optionnellement, les noms des colonnes dans le flux de sortie spécifié.

```
void PQprint(FILE* fout,      /* flux de sortie */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct
{
    pqbool  header;      /* affiche les en-têtes des champs et le nombre de
                            lignes */
    pqbool  align;       /* aligne les champs */
    pqbool  standard;    /* vieux format (mort) */
    pqbool  html3;       /* affiche les tables en HTML */
    pqbool  expanded;    /* étend les tables */
    pqbool  pager;       /* utilise le paginateur pour la sortie si nécessaire
                            */
    char    *fieldSep;   /* séparateur de champ */
    char    *tableOpt;   /* attributs des éléments de table HTML */
    char    *caption;    /* titre de la table HTML */
    char    **fieldName; /* Tableau terminé par un NULL des noms de remplacement
                            des champs */
} PQprintOpt;
```

Cette fonction était auparavant utilisée par psql pour afficher les résultats des requêtes mais ce n'est plus le cas. Notez qu'elle assume que les données sont dans un format textuel.

31.3.3. Récupérer d'autres informations de résultats

Ces fonctions sont utilisées pour extraire d'autres informations des objets PGresult.

`PQcmdStatus`

Renvoie l'état de la commande depuis l'instruction SQL qui a généré le PGresult. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand la poignée PGresult associée est passée à `PQclear`.

```
char * PQcmdStatus(PGresult *res);
```

D'habitude, c'est juste le nom de la commande mais elle pourrait inclure des données supplémentaires comme le nombre de lignes traitées.

`PQcmdTuples`

Renvoie le nombre de lignes affectées par la commande SQL.

```
char * PQcmdTuples(PGresult *res);
```

Cette fonction renvoie une chaîne contenant le nombre de lignes affectées par l'instruction SQL qui a généré PGresult. Cette fonction peut seulement être utilisée après l'exécution d'une instruction **SELECT**, **CREATE TABLE AS**, **INSERT**, **UPDATE**, **DELETE**, **MOVE**, **FETCH** ou **COPY**, ou **EXECUTE** avec une instruction préparée contenant une instruction **INSERT**, **UPDATE** ou **DELETE**. Si la commande qui a généré PGresult était autre chose, `PQcmdTuples` renverrait directement une chaîne vide. L'appelant ne devrait pas libérer la valeur de retour directement. Elle sera libérée quand la poignée PGresult associée est passée à `PQclear`.

`PQoidValue`

Renvoie l'OID de la ligne insérée, si la commande SQL était une instruction **INSERT** qui a inséré exactement une ligne dans une table comprenant des OID ou un **EXECUTE** d'une requête préparée contenant une instruction **INSERT** convenable. Sinon, cette fonction renvoie `InvalidOid`. Cette fonction renverra aussi `InvalidOid` si la table touchée par l'instruction **INSERT** ne contient pas d'OID.

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`

Renvoie une chaîne avec l'OID de la ligne insérée si la commande SQL était une instruction **INSERT** qui insère exactement une ligne, ou un **EXECUTE** d'une instruction préparée consistant en une commande **INSERT** (la chaîne sera `0` si l'instruction **INSERT** n'a inséré qu'une seule ligne ou si la table cible n'a pas d'OID). Si la commande n'était pas un **INSERT**, renvoie une chaîne vide.

```
char * PQoidStatus(const PGresult *res);
```

Cette fonction est obsolète et remplacée par `PQoidValue`. Elle n'est pas compatible avec les threads.

31.3.4. Chaîne d'échappement à inclure dans les commandes SQL

`PQescapeLiteral`

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

`PQescapeLiteral` échappe une chaîne pour l'utiliser dans une commande SQL. C'est utile pour insérer des données comme des constantes dans des commandes SQL. Certains caractères, comme les guillemets et les antislashs, doivent être traités avec des caractères d'échappement pour éviter qu'ils soient traités d'après leur signification spéciale par l'analyseur SQL. `PQescapeLiteral` réalise cette opération.

`PQescapeLiteral` renvoie une version échappée du paramètre *`str`* dans une mémoire allouée avec `malloc()`. Cette mémoire devra être libérée en utilisant `PQfreemem()` quand le résultat ne sera plus utile. Un octet zéro de fin n'est pas requis et ne doit pas être compté dans *`length`*. (Si un octet zéro de fin est découvert avant la fin du traitement des *`length`* octets, `PQescapeLiteral` s'arrête au zéro ; ce comportement est identique à celui de `strncpy`.) Les caractères spéciaux de la chaîne en retour ont été remplacés pour qu'ils puissent être traités correctement par l'analyseur de chaînes de PostgreSQL™. Un octet zéro final est aussi ajouté. Les guillemets simples qui doivent entourer les chaînes litérales avec PostgreSQL™ sont inclus dans la chaîne résultante.

En cas d'erreur, `PQescapeLiteral` renvoit NULL et un message convenable est stocké dans l'objet *`conn`*.

Astuce

Il est particulièrement important de faire un échappement propre lors de l'utilisation de chaînes provenant d'une source qui n'est pas forcément de confiance. Sinon, il existe un risque de sécurité : vous vous exposez à une attaque de type « injection SQL » avec des commandes SQL non voulues injectées dans votre base de données.

Notez qu'il n'est pas nécessaire ni correct de faire un échappement quand une valeur est passé en tant que paramètre séparé dans `PQexecParams` ou ce type de routine.

`PQescapeIdentifier`

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

`PQescapeIdentifier` échappe une chaîne pour qu'elle puisse être utilisé en tant qu'identifiant SQL, par exemple pour le nom d'une table, d'une colonne ou d'une fonction. C'est utile quand un identifiant fourni par un utilisateur pourrait contenir des caractères spéciaux qui pourraient autrement ne pas être interprétés comme faisant parti de l'identifiant par l'analyseur SQL ou lorsque l'identifiant pourrait contenir des caractères en majuscule, auquel cas le casse doit être préservée.

`PQescapeIdentifier` renvoit une version du paramètre *`str`* échappée comme doit l'être un identifiant SQL, dans une mémoire allouée avec `malloc()`. Cette mémoire doit être libérée en utilisant `PQfreemem()` quand le résultat n'est plus nécessaire. Un octet zéro de fin n'est pas nécessaire et ne doit pas être comptabilisé dans *`length`*. (Si un octet zéro de fin est trouvé avant le traitement des *`length`* octets, `PQescapeIdentifier` s'arrête au zéro ; ce comportement est identique à celui de `strncpy`.) Les caractères spéciaux de la chaîne en retour ont été remplacés pour que ce dernier soit traité proprement comme un identifiant SQL. Un octet zéro de fin est aussi ajouté. La chaîne de retour sera aussi entourée de guillemets doubles.

En cas d'erreur, `PQescapeIdentifier` renvoit NULL et un message d'erreur convenable est stockée dans l'objet *`conn`*.

Astuce

Comme avec les chaînes litérales, pour empêcher les attaques d'injection SQL, les identifiants SQL doivent être échappés lorsqu'elles proviennent de source non sûre.

`PQescapeStringConn`

```
size_t PQescapeStringConn (PGconn *conn,
```

```
                                        char *to, const char *from, size_t length,
                                        int *error);
```

PQescapeStringConn échappe les chaînes litérales de la même façon que PQescapeLiteral. Contrairement à PQescapeLiteral, l'appelant doit fournir un tampon d'une taille appropriée. De plus, PQescapeStringConn n'ajoute pas de guillemets simples autour des chaînes litérales de PostgreSQL™ ; elles doivent être ajoutées dans la commande SQL où ce résultat sera inséré. Le paramètre *from* pointe vers le premier caractère d'une chaîne à échapper, et le paramètre *length* précise le nombre d'octets contenus dans cette chaîne. Un octet zéro de fin n'est pas nécessaire et ne doit pas être comptabilisé dans *length*. (Si un octet zéro de fin est trouvé avant le traitement des *length* octets, PQescapeStringConn s'arrête au zéro ; ce comportement est identique à celui de strncpy.) *to* doit pointer vers un tampon qui peut contenir au moins un octet de plus que deux fois la valeur de *length*, sinon le comportement de la fonction n'est pas connue. Le comportement est aussi non défini si les chaînes *to* et *from* se surchargent.

Si le paramètre *error* est différent de NULL, alors *error est configuré à zéro en cas de succès et est différent de zéro en cas d'erreur. Actuellement, les seuls conditions permettant une erreur impliquent des encodages multi-octets dans la chaîne source. La chaîne en sortie est toujours générée en cas d'erreur mais il est possible que le serveur la rejettera comme une chaîne malformée. En cas d'erreur, un message convenable est stocké dans l'objet *conn*, que *error* soit NULL ou non.

PQescapeStringConn renvoit le nombre d'octets écrits dans *to*, sans inclure l'octet zéro de fin.

PQescapeString

PQescapeString est une ancienne version de PQescapeStringConn.

```
       size_t PQescapeString (char *to, const char *from, size_t length);
```

La seule différence avec PQescapeStringConn tient dans le fait que PQescapeString n'a pas de paramètres *conn* et *error*. À cause de cela, elle ne peut ajuster son comportement avec les propriétés de la connexion (comme l'encodage des caractères) et du coup, *elle pourrait founir de mauvais résultats*. De plus, elle ne peut pas renvoyer de conditions d'erreur.

PQescapeString peut être utilisé proprement avec des programmes utilisant une seule connexion PostgreSQL™ à la fois (dans ce cas, il peut trouver ce qui l'intéresse « en arrière-plan »). Dans d'autres contextes, c'est un risque en terme de sécurité. Cette fonction devrait être évitée et remplacée autant que possible par la fonction PQescapeStringConn.

PQescapeByteaConn

Échappe des données binaires à utiliser à l'intérieur d'une commande SQL avec le type bytea. Comme avec PQescapeStringConn, c'est seulement utilisé pour insérer des données directement dans une chaîne de commande SQL.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

Certaines valeurs d'octets *doivent* être échappées lorsqu'elles font partie d'un littéral bytea dans une instruction SQL. PQescapeByteaConn échappe les octets en utilisant soit un codage hexadécimal soit un échappement avec des antislashs. Voir Section 8.4, « Types de données binaires » pour plus d'informations.

Le paramètre *from* pointe sur le premier octet de la chaîne à échapper et le paramètre *from_length* donne le nombre d'octets de cette chaîne binaire (un octet zéro de terminaison n'est ni nécessaire ni compté). Le paramètre *to_length* pointe vers une variable qui contiendra la longueur de la chaîne échappée résultante. Cette longueur inclut l'octet zéro de terminaison.

PQescapeByteaConn renvoie une version échappée du paramètre *from* dans la mémoire allouée avec malloc(). Cette mémoire doit être libérée avec PQfreemem lorsque le résultat n'est plus nécessaire. Tous les caractères spéciaux de la chaîne de retour sont remplacés de façon à ce qu'ils puissent être traités proprement par l'analyseur de chaînes littérales de PostgreSQL™ et par l'entrée bytea de la fonction. Un octet zéro de terminaison est aussi ajouté. Les guillemets simples qui englobent les chaînes littérales de PostgreSQL™ ne font pas partie de la chaîne résultante.

En cas d'erreur, un pointeur NULL est renvoyé et un message d'erreur adéquat est stocké dans l'objet *conn*. Actuellement, la seule erreur possible est une mémoire insuffisante pour stocker la chaîne résultante.

PQescapeBytea

PQescapeBytea est une version obsolète de PQescapeByteaConn.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
```

```
                                    size_t *to_length);
```

La seule différence avec PQescapeByteaConn est que PQescapeBytea ne prend pas de paramètre PGconn. De ce fait, PQescapeBytea peut seulement être utilisé correctement dans des programmes qui n'utilisent qu'une seule connexion PostgreSQL™ à la fois (dans ce cas, il peut trouver ce dont il a besoin « en arrière-plan »). Il *pourrait donner de mauvais résultats* s'il était utilisé dans des programmes qui utilisent plusieurs connexions de bases de données (dans ce cas, utilisez plutôt PQescapeByteaConn).

PQunescapeBytea

Convertit une représentation de la chaîne en donnés binaires -- l'inverse de PQescapeBytea. Ceci est nécessaire lors de la récupération de données bytea en format texte, mais pas lors de sa récupération au format binaire.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

Le paramètre *from* pointe vers une chaîne de telle façon qu'elle pourrait provenir de PQgetvalue lorsque la colonne est de type bytea. PQunescapeBytea convertit cette représentation de la chaîne en sa représentation binaire. Elle renvoie un pointeur vers le tampon alloué avec malloc(), ou NULL en cas d'erreur, et place la taille du tampon dans *to_length*. Le résultat doit être libéré en utilisant PQfreemem lorsque celui-ci n'est plus nécessaire.

Cette conversion n'est pas l'inverse exacte de PQescapeBytea car la chaîne n'est pas échappée avec PQgetvalue. Cela signifie en particulier qu'il n'y a pas besoin de réfléchir à la mise entre guillemets de la chaîne, et donc pas besoin d'un paramètre PGconn.

31.4. Traitement des commandes asynchrones

La fonction PQexec est adéquate pour soumettre des commandes aux applications standards, synchrones. Néanmoins, il a quelques déficiences pouvant être d'importance à certains utilisateurs :

- PQexec attend que la commande se termine. L'application pourrait avoir d'autres travaux à réaliser (comme le rafraichissement de l'interface utilisateur), auquel cas il ne voudra pas être bloqué en attente de la réponse.
- Comme l'exécution de l'application cliente est suspendue en attendant le résultat, il est difficile pour l'application de décider qu'elle voudrait annuler la commande en cours (c'est possible avec un gestionnaire de signaux mais pas autrement).
- PQexec ne peut renvoyer qu'une structure PGresult. Si la chaîne de commande soumise contient plusieurs commandes SQL, toutes les structures PGresult sont annulées par PQexec, sauf la dernière.

Les applications qui n'apprécient pas ces limitations peuvent utiliser à la place les fonctions sous-jacentes à partir desquelles PQexec est construit : PQsendQuery et PQgetResult. Il existe aussi PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared et PQsendDescribePortal, pouvant être utilisées avec PQgetResult pour dupliquer les fonctionnalités de respectivement PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared et PQdescribePortal.

PQsendQuery

Soumet une commande au serveur sans attendre le(s) résultat(s). 1 est renvoyé si la commande a été correctement envoyée et 0 dans le cas contraire (auquel cas, utilisez la fonction PQerrorMessage pour obtenir plus d'informations sur l'échec).

```
int PQsendQuery(PGconn *conn, const char *command);
```

Après un appel réussi à PQsendQuery, appelez PQgetResult une ou plusieurs fois pour obtenir les résultats. PQsendQuery ne peut pas être appelé de nouveau (sur la même connexion) tant que PQgetResult ne renvoie pas de pointeur nul, indiquant que la commande a terminé.

PQsendQueryParams

Soumet une commande et des paramètres séparés au serveur sans attendre le(s) résultat(s).

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

Ceci est équivalent à PQsendQuery sauf que les paramètres de requêtes peuvent être spécifiés à partir de la chaîne de requête. Les paramètres de la fonction sont gérés de façon identique à PQexecParams. Comme PQexecParams, cela ne

fonctionnera pas pour les connexions utilisant le protocole 2.0 et cela ne permettra qu'une seule commande dans la chaîne de requête.

`PQsendPrepare`

Envoie une requête pour créer une instruction préparée avec les paramètres donnés et redonne la main sans attendre la fin de son exécution.

```
int PQsendPrepare(PGconn *conn,
const char *stmtName,
const char *query,
int nParams,
const Oid *paramTypes);
```

Ceci est la version asynchrone de `PQprepare` : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 sinon. Après un appel terminé avec succès, appelez `PQgetResult` pour déterminer si le serveur a créé avec succès l'instruction préparée. Les paramètres de la fonction sont gérés de façon identique à `PQprepare`. Comme `PQprepare`, cela ne fonctionnera pas sur les connexions utilisant le protocole 2.0.

`PQsendQueryPrepared`

Envoie une requête pour exécuter une instruction préparée avec des paramètres donnés sans attendre le(s) résultat(s).

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

Ceci est similaire à `PQsendQueryParams` mais la commande à exécuter est spécifiée en nommant une instruction précédemment préparée au lieu de donner une chaîne contenant la requête. Les paramètres de la fonction sont gérés de façon identique à `PQexecPrepared`. Comme `PQexecPrepared`, cela ne fonctionnera pas pour les connexions utilisant le protocole 2.0.

`PQsendDescribePrepared`

Soumet une requête pour obtenir des informations sur l'instruction préparée indiquée sans attendre sa fin.

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

Ceci est la version asynchrone de `PQdescribePrepared` : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 dans le cas contraire. Après un appel réussi, appelez `PQgetResult` pour obtenir les résultats. Les paramètres de la fonction sont gérés de façon identique à `PQdescribePrepared`. Comme `PQdescribePrepared`, cela ne fontionnera pas avec les connexions utilisant le protocole 2.0.

`PQsendDescribePortal`

Soumet une requête pour obtenir des informations sur le portail indiqué sans attendre la fin de la commande.

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

Ceci est la version asynchrone de `PQdescribePortal` : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 dans le cas contraire. Après un appel réussi, appelez `PQgetResult` pour obtenir les résultats. Les paramètres de la fonction sont gérés de façon identique à `PQdescribePortal`. Comme `PQdescribePortal`, cela ne fontionnera pas avec les connexions utilisant le protocole 2.0.

`PQgetResult`

Attend le prochain résultat d'un appel précédant à `PQsendQuery`, `PQsendQueryParams`, `PQsendPrepare` ou `PQsendQueryPrepared`, et le renvoie. Un pointeur nul est renvoyé quand la commande est terminée et qu'il n'y aura plus de résultats.

```
PGresult *PQgetResult(PGconn *conn);
```

`PQgetResult` doit être appelé de façon répété jusqu'à ce qu'il retourne un pointeur nul indiquant que la commande s'est terminée (si appelé à un moment où aucune commande n'est active, `PQgetResult` renverra seulement un pointeur nul à la fois). Chaque résultat non nul provenant de `PQgetResult` devrait être traité en utilisant les mêmes fonctions d'accès à PGresult que celles précédemment décrites. N'oubliez pas de libérer chaque objet résultat avec `PQclear` une fois que vous en avez terminé. Notez que `PQgetResult` bloquera seulement si la commande est active et que les données nécessaires en réponse n'ont pas encore été lues par `PQconsumeInput`.

Utiliser PQsendQuery et PQgetResult résout un des problèmes de PQexec : si une chaîne de commande contient plusieurs commandes SQL, les résultats de ces commandes peuvent être obtenus individuellement (ceci permet une simple forme de traitement en parallèle : le client peut gérer les résultats d'une commande alors que le serveur travaille sur d'autres requêtes de la même chaîne de commandes). Néanmoins, appeler PQgetResult causera toujours un blocage du client jusqu'à la fin de la prochaine commande SQL. Ceci est évitable en utilisant proprement deux fonctions supplémentaires :

PQconsumeInput

Si l'entrée est disponible à partir du serveur, consommez-la.

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput renvoie normalement 1 indiquant « aucune erreur », mais renvoie zéro s'il y a eu une erreur (auquel cas PQerrorMessage peut être consulté). Notez que le résultat ne dit pas si des données ont été récupérées en entrée. Après avoir appelé PQconsumeInput, l'application devrait vérifier PQisBusy et/ou PQnotifies pour voir si leur état a changé.

PQconsumeInput pourrait être appelé même si l'application n'est pas encore préparé à gérer un résultat ou une notification. La fonction lira les données disponibles et les sauvegardera dans un tampon indiquant ainsi qu'une lecture d'un select() est possible. L'application peut donc utiliser PQconsumeInput pour effacer la condition select() immédiatement, puis pour examiner les résultats autant que possible.

PQisBusy

Renvoie 1 si une commande est occupée, c'est-à-dire que PQgetResult bloquerait en attendant une entrée. Un zéro indiquerait que PQgetResult peut être appelé avec l'assurance de ne pas être bloqué.

```
int PQisBusy(PGconn *conn);
```

PQisBusy ne tentera pas lui-même de lire les données à partir du serveur ; du coup, PQconsumeInput doit être appelé d'abord ou l'état occupé ne s'arrêtera jamais.

Une application typique de l'utilisation de ces fonctions aura une boucle principale utilisant select() ou poll() pour attendre toutes les conditions auxquelles il doit répondre. Une des conditions sera la disponibilité des données à partir du serveur, ce qui signifie des données lisibles pour select() sur le descripteur de fichier identifié par PQsocket. Lorsque la boucle principale détecte la disponibilité de données, il devrait appeler PQconsumeInput pour lire l'en-tête. Il peut ensuite appeler PQisBusy suivi par PQgetResult si PQisBusy renvoie false (0). Il peut aussi appeler PQnotifies pour détecter les messages **NOTIFY** (voir la Section 31.7, « Notification asynchrone »).

Un client qui utilise PQsendQuery/PQgetResult peut aussi tenter d'annuler une commande en cours de traitement par le serveur ; voir la Section 31.5, « Annuler des requêtes en cours d'exécution ». Mais quelque soit la valeur renvoyée par PQcancel, l'application doit continuer avec la séquence normale de lecture du résultat en utilisant PQgetResult. Une annulation réussie causera simplement une fin plus rapide de la commande.

En utilisant les fonctions décrites ci-dessus, il est possible d'éviter le blocage pendant l'attente de données du serveur. Néanmoins, il est toujours possible que l'application se bloque en attendant l'envoi vers le serveur. C'est relativement peu fréquent mais cela peut arriver si de très longues commandes SQL ou données sont envoyées (c'est bien plus probable si l'application envoie des données via **COPY IN**). Pour empêcher cette possibilité et réussir des opérations de bases de données totalement non bloquantes, les fonctions supplémentaires suivantes pourraient être utilisées.

PQsetnonblocking

Initialise le statut non bloquant de la connexion.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Initialise l'état de la connexion à non bloquant si *arg* vaut 1 et à bloquant si *arg* vaut 0. Renvoie 0 si OK, -1 en cas d'erreur.

Dans l'état non bloquant, les appels à PQsendQuery, PQputline, PQputnbytes et PQendcopy ne bloqueront pas mais renverront à la place une erreur s'ils ont besoin d'être de nouveau appelés.

Notez que PQexec n'honore pas le mode non bloquant ; s'il est appelé, il agira d'une façon bloquante malgré tout.

PQisnonblocking

Renvoie le statut bloquant de la connexion à la base de données.

```
int PQisnonblocking(const PGconn *conn);
```

Renvoie 1 si la connexion est en mode non bloquant, 1 dans le cas contraire.

PQflush

Tente de vider les données des queues de sortie du serveur. Renvoie 0 en cas de succès (ou si la queue d'envoi est vide), -1 en cas d'échec quelque soit la raison ou 1 s'il a été incapable d'envoyer encore toutes les données dans la queue d'envoi (ce cas arrive seulement si la connexion est non bloquante).

```
int PQflush(PGconn *conn);
```

Après avoir envoyé une commande ou des données dans une connexion non bloquante, appelez `PQflush`. S'il renvoie 1, attendez que la socket soit disponible en écriture et appelez-la de nouveau ; répétez cela jusqu'à ce qu'il renvoie 0. Une fois que `PQflush` renvoie 0, attendez que la socket soit disponible en lecture puis lisez la réponse comme décrit ci-dessus.

31.5. Annuler des requêtes en cours d'exécution

Une application client peut demander l'annulation d'une commande qui est toujours en cours d'exécution par le serveur en utilisant les fonctions décrites dans cette section.

`PQgetCancel`

Crée une structure de données contenant les informations nécessaires à l'annulation d'une commande lancée sur une connexion particulière à la base de données.

```
PGcancel *PQgetCancel(PGconn *conn);
```

`PQgetCancel` crée un objet fonction PGcancel avec un objet connexion PGconn. Il renverra NULL si le paramètre *`conn`* donné est NULL ou est une connexion invalide. L'objet PGcancel est une structure opaque qui n'a pas pour but d'être accédé directement par l'application ; elle peut seulement être passée à `PQcancel` ou `PQfreeCancel`.

`PQfreeCancel`

Libère une structure de données créée par `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` libère un objet donné par `PQgetCancel`.

`PQcancel`

Demande que le serveur abandonne l'exécution de la commande en cours.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

La valeur renvoyée est 1 si la demande d'annulation a été correctement envoyée et 0 sinon. Si non, *`errbuf`* contient un message d'erreur expliquant pourquoi. *`errbuf`* doit être un tableau de caractères d'une taille de *`errbufsize`* octets (la taille recommandée est de 256 octets).

Un envoi réussi ne garantit pas que la demande aura un quelconque effet. Si l'annulation est réelle, la commande en cours terminera plus tôt et renverra une erreur. Si l'annulation échoue (disons, parce que le serveur a déjà exécuté la commande), alors il n'y aura aucun résultat visible.

`PQcancel` peut être invoqué de façon sûr par le gestionnaire de signaux si *`errbuf`* est une variable locale dans le gestionnaire de signaux. L'objet PGcancel est en lecture seule pour ce qui concerne `PQcancel`, pour qu'il puisse aussi être appelé à partir d'un thread séparé de celui manipulant l'objet PGconn.

`PQrequestCancel`

Demande que le serveur abandonne le traitement de la commande en cours.

```
int PQrequestCancel(PGconn *conn);
```

`PQrequestCancel` est une variante obsolète de `PQcancel`. Elle opère directement sur l'objet PGconn et, en cas d'échec, stocke le message d'erreur dans l'objet PGconn (d'où il peut être récupéré avec `PQerrorMessage`). Bien qu'il s'agisse de la même fonctionnalité, cette approche est hasardeuse en cas de programmes compatibles avec les threads ainsi que pour les gestionnaires de signaux car il est possible que la surcharge du message d'erreur de PGconn gênera l'opération en cours sur la connexion.

31.6. Interface à chemin rapide

PostgreSQL™ fournit une interface rapide pour envoyer des appels de fonctions simples au serveur.

Astuce

Cette interface est quelque peu obsolète car vous pourriez réaliser les mêmes choses avec des performances similaires et plus de fonctionnalités en initialisant une instruction préparée pour définir l'appel de fonction. Puis, exécuter l'instruction avec une transmission binaire des paramètres et des substitutions de résultats pour un appel de fonction à chemin rapide.

La fonction `PQfn` demande l'exécution d'une fonction du serveur via l'interface de chemin rapide :

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

L'argument *`fnid`* est l'OID de la fonction à exécuter. *`args`* et *`nargs`* définissent les paramètres à passer à la fonction ; ils doivent correspondre à la liste d'arguments déclarés de la fonction. Quand le champ *`isint`* d'une structure est vrai, la valeur de *`u.integer`* est envoyée au serveur en tant qu'entier de la longueur indiquée (qui doit être 1, 2 ou 4 octets) ; les bons échanges d'octets se passent. Quand *`isint`* est faux, le nombre d'octets indiqué sur *`*u.ptr`* est envoyé au traitement ; les données doivent être dans le format attendu par le serveur pour la transmission binaire du type de données de l'argument de la fonction. *`result_buf`* est le tampon dans lequel placer le code de retour. L'appelant doit avoir alloué suffisamment d'espace pour stocker le code de retour (il n'y a pas de vérification !). La longueur actuelle du résultat sera renvoyé dans l'entier pointé par *`result_len`*. Si un résultat sur un entier de 1, 2 ou 4 octets est attendu, initialisez *`result_is_int`* à 1, sinon initialisez-le à 0. Initialiser *`result_is_int`* à 1 fait que libpq échange les octets de la valeur si nécessaire, de façon à ce que la bonne valeur int soit délivrée pour la machine cliente. Quand *`result_is_int`* vaut 0, la chaîne d'octets au format binaire envoyée par le serveur est renvoyée non modifiée.

`PQfn` renvoie toujours un pointeur PGresult valide. L'état du résultat devrait être vérifié avant que le résultat ne soit utilisé. Le demandeur est responsable de la libération de la structure PGresult avec `PQclear` lorsque celle-ci n'est plus nécessaire.

Notez qu'il n'est pas possible de gérer les arguments nuls, les résultats nuls et les résultats d'ensembles nuls en utilisant cette interface.

31.7. Notification asynchrone

PostgreSQL™ propose des notifications asynchrone via les commandes **LISTEN** et **NOTIFY**. Une session cliente enregistre son intérêt dans un canal particulier avec la commande **LISTEN** (et peut arrêter son écoute avec la commande **UNLISTEN**). Toutes les sessions écoutant un canal particulier seront notifiées de façon asynchrone lorsqu'une commande **NOTIFY** avec ce nom de canal sera exécutée par une session. Une chaîne de « charge » peut être renseigné pour fournir des données supplémentaires aux processus en écoute.

Les applications libpq soumettent les commandes **LISTEN**, **UNLISTEN** et **NOTIFY** comme des commandes SQL ordinaires. L'arrivée des messages **NOTIFY** peut être détectée ensuite en appelant `PQnotifies`.

La fonction `PQnotifies` renvoie la prochaine notification à partir d'une liste de messages de notification non gérés reçus à partir du serveur. Il renvoie un pointeur nul s'il n'existe pas de notifications en attente. Une fois qu'une notification est renvoyée à partir de `PQnotifies`, elle est considérée comme étant gérée et sera supprimée de la liste des notifications.

```
PGnotify* PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;              /* nom du canal de la notification */
    int  be_pid;                /* ID du processus serveur notifiant */
```

```
    char *extra;                /* chaîne de charge pour la notification */
} PGnotify;
```

Après avoir traité un objet PGnotify renvoyé par `PQnotifies`, assurez-vous de libérer le pointeur `PQfreemem`. Il est suffisant de libérer le pointeur PGnotify ; les champs *`relname`* et *`extra`* ne représentent pas des allocations séparées (le nom de ces champs est historique ; en particulier, les noms des canaux n'ont pas besoin d'être liés aux noms des relations.)

Exemple 31.2, « Deuxième exemple de programme pour libpq » donne un programme d'exemple illustrant l'utilisation d'une notification asynchrone.

`PQnotifies` ne lit pas réellement les données à partir du serveur ; il renvoie simplement les messages précédemment absorbés par une autre fonction de libpq. Dans les précédentes versions de libpq, la seule façon de s'assurer une réception à temps des messages **NOTIFY** consistait à soumettre constamment des commandes de soumission, même vides, puis de vérifier `PQnotifies` après chaque `PQexec`. Bien que ceci fonctionnait, cela a été abandonné à cause de la perte de puissance.

Une meilleure façon de vérifier les messages **NOTIFY** lorsque vous n'avez pas de commandes utiles à exécuter est d'appeler `PQconsumeInput` puis de vérifier `PQnotifies`. Vous pouvez utiliser `select()` pour attendre l'arrivée des données à partir du serveur, donc en utilisant aucune puissance du CPU sauf lorsqu'il y a quelque chose à faire (voir `PQsocket` pour obtenir le numéro du descripteur de fichiers à utiliser avec `select()`). Notez que ceci fonctionnera bien que vous soumettez les commandes avec `PQsendQuery`/`PQgetResult` ou que vous utilisez simplement `PQexec`. Néanmoins, vous devriez vous rappeler de vérifier `PQnotifies` après chaque `PQgetResult` ou `PQexec` pour savoir si des notifications sont arrivées lors du traitement de la commande.

31.8. Fonctions associées avec la commande COPY

Dans PostgreSQL™, la commande **COPY** a des options pour lire ou écrire à partir de la connexion réseau utilisée par libpq. Les fonctions décrites dans cette section autorisent les applications à prendre avantage de cette capacité en apportant ou en consommant les données copiées.

Le traitement complet est le suivant. L'application lance tout d'abord la commande SQL **COPY** via `PQexec` ou une des fonctions équivalents. La réponse à ceci (s'il n'y a pas d'erreur dans la commande) sera un objet PGresult avec un code de retour `PGRES_COPY_OUT` ou `PGRES_COPY_IN` (suivant la direction spécifiée pour la copie). L'application devrait alors utiliser les fonctions de cette section pour recevoir ou transmettre des lignes de données. Quand le transfert de données est terminé, un autre objet PGresult est renvoyé pour indiquer le succès ou l'échec du transfert. Son statut sera `PGRES_COMMAND_OK` en cas de succès et `PGRES_FATAL_ERROR` si un problème a été rencontré. À ce point, toute autre commande SQL pourrait être exécutée via `PQexec` (il n'est pas possible d'exécuter d'autres commandes SQL en utilisant la même connexion tant que l'opération **COPY** est en cours).

Si une commande **COPY** est lancée via `PQexec` dans une chaîne qui pourrait contenir d'autres commandes supplémentaires, l'application doit continuer à récupérer les résultats via `PQgetResult` après avoir terminé la séquence **COPY**. C'est seulement quand `PQgetResult` renvoie NULL que vous pouvez être certain que la chaîne de commandes `PQexec` est terminée et qu'il est possible de lancer d'autres commandes.

Les fonctions de cette section devraient seulement être exécutées pour obtenir un statut de résultat `PGRES_COPY_OUT` ou `PGRES_COPY_IN` à partir de `PQexec` ou `PQgetResult`.

Un objet PGresult gérant un de ces statuts comporte quelques données supplémentaires sur l'opération **COPY** qui commence. La données supplémentaire est disponible en utilisant les fonctions qui sont aussi utilisées en relation avec les résultats de requêtes :

`PQnfields`
: Renvoie le nombre de colonnes (champs) à copier.

`PQbinaryTuples`
: 0 indique que le format de copie complet est textuel (lignes séparées par des retours chariots, colonnes séparées par des caractères de séparation, etc). 1 indique que le format de copie complet est binaire. Voir COPY(7) pour plus d'informations.

`PQfformat`
: Renvoie le code de format (0 pour le texte, 1 pour le binaire) associé avec chaque colonne de l'opération de copie. Les codes de format par colonne seront toujours zéro si le format de copie complet est textuel mais le format binaire supporte à la fois des colonnes textuelles et des colonnes binaires (néanmoins, avec l'implémentation actuelle de **COPY**, seules les colonnes binaires apparaissent dans une copie binaire donc les formats par colonnes correspondent toujours au format complet actuellement).

Note

Ces valeurs de données supplémentaires sont seulement disponibles en utilisant le protocole 3.0. Lors de

l'utilisation du protocole 2.0, toutes ces fonctions renvoient 0.

31.8.1. Fonctions d'envoi de données pour COPY

Ces fonctions sont utilisées pour envoyer des données lors d'un `COPY FROM STDIN`. Elles échoueront si elles sont appelées alors que la connexion ne se trouve pas dans l'état `COPY_IN`.

`PQputCopyData`

Envoie des données au serveur pendant un état `COPY_IN`.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Transmet les données de **COPY** dans le tampon spécifié (*`buffer`*), sur *`nbytes`* octets, au serveur. Le résultat vaut 1 si les données ont été envoyées, zéro si elles n'ont pas été envoyées car la tentative pourrait bloquer (ce cas n'est possible que lors d'une connexion en mode non bloquant) ou -1 si une erreur s'est produite (utilisez `PQerrorMessage` pour récupérer des détails si la valeur de retour vaut -1. Si la valeur vaut zéro, attendez qu'il soit prêt en écriture et ré-essayez).

L'application pourrait diviser le flux de données de **COPY** dans des chargements de tampon de taille convenable. Les limites n'ont pas de signification sémantique lors de l'envoi. Le contenu du flux de données doit correspondre au format de données attendu par la commande **COPY** ; voir COPY(7) pour des détails.

`PQputCopyEnd`

Envoie une indication de fin de transfert au serveur lors de l'état `COPY_IN`.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Termine l'opération `COPY_IN` avec succès si *`errormsg`* est NULL. Si *`errormsg`* n'est pas NULL alors **COPY** échoue, la chaîne pointée par *`errormsg`* étant utilisée comme message d'erreur (néanmoins, vous ne devriez pas supposer que ce message d'erreur précis reviendra du serveur car le serveur pourrait avoir déjà échouée sur la commande **COPY** pour des raisons qui lui sont propres). Notez aussi que l'option forçant l'échec ne fonctionnera pas lors de l'utilisation de connexions avec un protocole pre-3.0.

Le résultat est 1 si la donnée de fin a été envoyée, zéro si elle ne l'a pas été car cette tentative serait bloquante (ce cas est uniquement possible si la connexion est dans un mode non bloquant) ou -1 si une erreur est survenue (utilisez `PQerrorMessage` pour récupérer les détails si le code de retour est -1. Si la valeur vaut zéro, attendez que le serveur soit prêt en écriture et ré-essayez de nouveau).

Après un appel réussi à `PQputCopyEnd`, appelez `PQgetResult` pour obtenir le statut de résultat final de la commande **COPY**. Vous pouvez attendre que le résultat soit disponible de la même façon. Puis, retournez aux opérations normales.

31.8.2. Fonctions de réception des données de COPY

Ces fonctions sont utilisées pour recevoir des données lors d'un `COPY TO STDOUT`. Elles échoueront si elles sont appelées alors que la connexion n'est pas dans l'état `COPY_OUT`

`PQgetCopyData`

Reçoit des données à partir du serveur lors d'un état `COPY_OUT`.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Tente d'obtenir une autre ligne de données du serveur lors d'une opération **COPY**. Les données ne sont renvoyées qu'une ligne à la fois ; si seulement une ligne partielle est disponible, elle n'est pas renvoyée. Le retour d'une ligne avec succès implique l'allocation d'une portion de mémoire pour contenir les données. Le paramètre *`buffer`* ne doit pas être NULL. *`*buffer`* est initialisé pour pointer vers la mémoire allouée ou vers NULL au cas où aucun tampon n'est renvoyé. Un tampon résultat non NULL devra être libéré en utilisant `PQfreemem` lorsqu'il ne sera plus utile.

Lorsqu'une ligne est renvoyée avec succès, le code de retour est le nombre d'octets de la donnée dans la ligne (et sera donc supérieur à zéro). La chaîne renvoyée est toujours terminée par un octet nul bien que ce ne soit utile que pour les **COPY** textuels. Un résultat zéro indique que la commande **COPY** est toujours en cours mais qu'aucune ligne n'est encore disponible (ceci est seulement possible lorsque *`async`* est vrai). Un résultat -1 indique que **COPY** a terminé. Un résultat -2 indique

qu'une erreur est survenue (consultez `PQerrorMessage` pour en connaître la raison).

Lorsque *`async`* est vraie (différent de zéro), `PQgetCopyData` ne bloquera pas en attente d'entrée ; il renverra zéro si **COPY** est toujours en cours mais qu'aucune ligne n'est encore disponible (dans ce cas, attendez qu'il soit prêt en lecture puis appelez `PQconsumeInput` avant d'appeler `PQgetCopyData` de nouveau). Quand *`async`* est faux (zéro), `PQgetCopyData` bloquera tant que les données ne seront pas disponibles ou tant que l'opération n'aura pas terminée.

Après que `PQgetCopyData` ait renvoyé -1, appelez `PQgetResult` pour obtenir le statut de résultat final de la commande **COPY**. Vous pourriez attendre la disponibilité de ce résultat comme d'habitude. Puis, retournez aux opérations habituelles.

31.8.3. Fonctions obsolètes pour COPY

Ces fonctions représentent d'anciennes méthodes de gestion de **COPY**. Bien qu'elles fonctionnent toujours, elles sont obsolètes à cause de leur pauvre gestion des erreurs, des méthodes non convenables de détection d'une fin de transmission, et du manque de support des transferts binaires et des transferts non bloquants.

`PQgetline`

Lit une ligne de caractères terminée par un retour chariot (transmis par le serveur) dans un tampon de taille *`length`*.

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

Cette fonction copie jusqu'à *`length`*-1 caractères dans le tampon et convertit le retour chariot en un octet nul. `PQgetline` renvoie EOF à la fin de l'entrée, 0 si la ligne entière a été lu et 1 si le tampon est complet mais que le retour chariot à la fin n'a pas encore été lu.

Notez que l'application doit vérifier si un retour chariot est constitué de deux caractères `\.`, ce qui indique que le serveur a terminé l'envoi des résultats de la commande **COPY**. Si l'application peut recevoir des lignes de plus de *`length`*-1 caractères, une attention toute particulière est nécessaire pour s'assurer qu'elle reconnaisse la ligne `\.` correctement (et ne la confond pas, par exemple, avec la fin d'une longue ligne de données).

`PQgetlineAsync`

Lit une ligne de données **COPY** (transmise par le serveur) dans un tampon sans blocage.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

Cette fonction est similaire à `PQgetline` mais elle peut être utilisée par des applications qui doivent lire les données de **COPY** de façon asynchrone, c'est-à-dire sans blocage. Après avoir lancé la commande **COPY** et obtenu une réponse `PGRES_COPY_OUT`, l'application devrait appeler `PQconsumeInput` et `PQgetlineAsync` jusqu'à ce que le signal de fin des données ne soit détecté.

Contrairement à `PQgetline`, cette fonction prend la responsabilité de détecter la fin de données.

À chaque appel, `PQgetlineAsync` renverra des données si une ligne de données complète est disponible dans le tampon d'entrée de libpq. Sinon, aucune ligne n'est renvoyée jusqu'à l'arrivée du reste de la ligne. La fonction renvoie -1 si le marqueur de fin de copie des données a été reconnu, 0 si aucune donnée n'est disponible ou un nombre positif indiquant le nombre d'octets renvoyés. Si -1 est renvoyé, l'appelant doit ensuite appeler `PQendcopy` puis retourner aux traitements habituels.

Les données renvoyées ne seront pas étendues au delà de la limite de la ligne. Si possible, une ligne complète sera retournée en une fois. Mais si le tampon offert par l'appelant est trop petit pour contenir une ligne envoyée par le serveur, alors une ligne de données partielle sera renvoyée. Avec des données textuelles, ceci peut être détecté en testant si le dernier octet renvoyé est `\n` ou non (dans un **COPY** binaire, l'analyse réelle du format de données **COPY** sera nécessaire pour faire la détermination équivalente). La chaîne renvoyée n'est pas terminée par un octet nul (si vous voulez ajouter un octet nul de terminaison, assurez-vous de passer un *`bufsize`* inférieur de 1 par rapport à l'espace réellement disponible).

`PQputline`

Envoie une chaîne terminée par un octet nul au serveur. Renvoie 0 si tout va bien et EOF s'il est incapable d'envoyer la chaîne.

```
int PQputline(PGconn *conn,
              const char *string);
```

Le flux de données de **COPY** envoyé par une série d'appels à `PQputline` a le même format que celui renvoyé par `PQgetlineAsync`, sauf que les applications ne sont pas obligées d'envoyer exactement une ligne de données par appel à `PQput-`

`line` ; il est correct d'envoyer une ligne partielle ou plusieurs lignes par appel.

Note

Avant le protocole 3.0 de PostgreSQL™, il était nécessaire pour l'application d'envoyer explicitement les deux caractères `\.` comme ligne finale pour indiquer qu'il a terminé l'envoi des données du **COPY** data. Bien que ceci fonctionne toujours, cette méthode est abandonnée et la signification spéciale de `\.` pourrait être supprimée dans une prochaine version. Il est suffisant d'appeler `PQendcopy` après avoir envoyé les vraies données.

`PQputnbytes`

Envoie une chaîne non terminée par un octet nul au serveur. Renvoie 0 si tout va bien et EOF s'il n'a pas été capable d'envoyer la chaîne.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

C'est exactement comme `PQputline` sauf que le tampon de donnée n'a pas besoin d'être terminé avec un octet nul car le nombre d'octets envoyés est spécifié directement. Utilisez cette procédure pour envoyer des données binaires.

`PQendcopy`

Se synchronise avec le serveur.

```
int PQendcopy(PGconn *conn);
```

Cette fonction attend que le serveur ait terminé la copie. Il devrait soit indiquer quand la dernière chaîne a été envoyée au serveur en utilisant `PQputline` soit le moment où la dernière chaîne a été reçue du serveur en utilisant `PGgetline`. Si ce n'est pas fait, le serveur renverra un « out of sync » (perte de synchronisation) au client. Suivant le retour de cette fonction, le serveur est prêt à recevoir la prochaine commande SQL. Le code de retour 0 indique un succès complet et est différent de zéro dans le cas contraire (utilisez `PQerrorMessage` pour récupérer des détails sur l'échec).

Lors de l'utilisation de `PQgetResult`, l'application devrait répondre à un résultat `PGRES_COPY_OUT` en exécutant `PQgetline` de façon répétée, suivi par un `PQendcopy` une fois la ligne de terminaison aperçue. Il devrait ensuite retourner à la boucle `PQgetResult` jusqu'à ce que `PQgetResult` renvoie un pointeur nul. De façon similaire, un résultat `PGRES_COPY_IN` est traité par une série d'appels à `PQputline` suivis par un `PQendcopy`, ensuite retour à la boucle `PQgetResult`. Cet arrangement vous assurera qu'une commande **COPY** intégrée dans une série de commandes SQL sera exécutée correctement.

Les anciennes applications soumettent un **COPY** via `PQexec` et assument que la transaction est faite après un `PQendcopy`. Ceci fonctionnera correctement seulement si **COPY** est la seule commande SQL dans la chaîne de commandes.

31.9. Fonctions de contrôle

Ces fonctions contrôlent divers détails du comportement de libpq.

`PQclientEncoding`

Renvoie l'encodage client.

```
int PQclientEncoding(const PGconn *conn);
```

Notez qu'il renvoie l'identifiant d'encodage, pas une chaîne symbolique telle que `EUC_JP`. Pour convertir un identifiant d'encodage en nom, vous pouvez utiliser :

```
char *pg_encoding_to_char(int encoding_id);
```

`PQsetClientEncoding`

Configure l'encodage client.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

`conn` est la connexion au serveur, et *`encoding`* est l'encodage que vous voulez utiliser. Si la fonction initialise l'encodage avec succès, elle renvoie 0, sinon -1. L'encodage actuel de cette connexion peut être déterminé en utilisant `PQclientEncoding`.

`PQsetErrorVerbosity`

Détermine la verbosité des messages renvoyés par `PQerrorMessage` et `PQresultErrorMessage`.

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

`PQsetErrorVerbosity` initialise le mode de verbosité, renvoyant le paramétrage précédant de cette connexion. Dans le mode *terse*, les messages renvoyés incluent seulement la sévérité, le texte principal et la position ; ceci tiendra normalement sur une seule ligne. Le mode par défaut produit des messages qui inclut ces champs ainsi que les champs détail, astuce ou contexte (ils pourraient être sur plusieurs lignes). Le mode *VERBOSE* inclut tous les champs disponibles. Modifier la verbosité n'affecte pas les messages disponibles à partir d'objets PGresult déjà existants, seulement ceux créés après.

`PQtrace`

Active les traces de communication entre client et serveur dans un flux fichier de débogage.

```
void PQtrace(PGconn *conn, FILE *stream);
```

Note

Sur Windows, si la bibliothèque libpq et une application sont compilées avec des options différentes, cet appel de fonction arrêtera brutalement l'application car la représentation interne des pointeurs `FILE` diffère. Spécifiquement, les options multi-threaded/single-threaded release/debug et static/dynamic devraient être identiques pour la bibliothèque et les applications qui l'utilisent.

`PQuntrace`

Désactive les traces commencées avec `PQtrace`.

```
void PQuntrace(PGconn *conn);
```

31.10. Fonctions diverses

Comme toujours, certains fonctions ne sont pas catégorisables.

`PQfreemem`

Libère la mémoire allouée par libpq.

```
void PQfreemem(void *ptr);
```

Libère la mémoire allouée par libpq, particulièrement `PQescapeByteaConn`, `PQescapeBytea`, `PQunescapeBytea`, et `PQnotifies`. Il est particulièrement important que cette fonction, plutôt que `free()`, soit utilisée sur Microsoft Windows. Ceci est dû à l'allocation de la mémoire dans une DLL et la relâcher dans l'application fonctionne seulement si les drapeaux multi-thread/mon-thread, release/debug et static/dynamic sont les mêmes pour la DLL et l'application. Sur les plateformes autres que Microsoft Windows, cette fonction est identique à la fonction `free()` de la bibliothèque standard.

`PQconninfoFree`

Libère les structures de données allouées par `PQconndefaults` ou `PQconninfoParse`.

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

Un simple appel à `PQfreemem` ne suffira pas car le tableau contient des références à des chaînes supplémentaires.

`PQencryptPassword`

Prépare la forme chiffrée du mot de passe PostgreSQL™.

```
char * PQencryptPassword(const char *passwd, const char *user);
```

Cette fonction est utilisée par les applications clientes qui souhaitent envoyées des commandes comme `ALTER USER joe PASSWORD 'passe'`. Une bonne pratique est de ne pas envoyer le mot de passe en clair dans une telle commande car le mot de passe serait exposé dans les journaux, les affichages d'activité, et ainsi de suite. À la place, utilisez cette fonction pour convertir le mot de passe en clair en une forme chiffrée avant de l'envoyer. Les arguments sont le mot de passe en clair et le nom SQL de l'utilisateur. La valeur renvoyée est une chaîne allouée par `malloc` ou NULL s'il ne reste plus de mémoire. L'appelant assume que la chaîne ne contient aucun caractère spécial qui nécessiterait un échappement. Utilisez `PQfreemem` pour libérer le résultat une fois terminé.

`PQmakeEmptyPGresult`

Construit un objet PGresult vide avec la statut indiqué.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

C'est une fonction interne de la libpq pour allouer et initialiser un objet PGresult vide. Cette fonction renvoit NULL si la mémoire n'a pas pu être allouée. Elle est exportée car certaines applications trouveront utiles de générer eux-mêmes des objets de résultat (tout particulièrement ceux avec des statuts d'erreur). Si *`conn`* n'est pas NULL et que *`status`* indique une erreur, le message d'erreur actuel de la connexion indiquée est copié dans PGresult. De plus, si *`conn`* n'est pas NULL, toute procédure d'événement enregistrée dans la connexion est copiée dans le PGresult. (Elles n'obtiennent pas d'appels `PGEVT_RESULTCREATE`, mais jetez un œil à `PQfireResultCreateEvents`.) Notez que `PQclear` devra être appelé sur l'objet, comme pour un PGresult renvoyé par libpq lui-même.

`PQfireResultCreateEvents`

Déclenche un événement `PGEVT_RESULTCREATE` (voir Section 31.12, « Système d'événements ») pour chaque procédure d'événement enregistré dans l'objet PGresult. Renvoit autre chose que zéro en cas de succès, zéro si la procédure d'événement échoue.

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

L'argument `conn` est passé aux procédures d'événement mais n'est pas utilisé directement. Il peut être `NULL` si les procédures de l'événement ne l'utiliseront pas.

Les procédures d'événements qui ont déjà reçu un événement `PGEVT_RESULTCREATE` ou `PGEVT_RESULTCOPY` pour cet objet ne sont pas déclenchées de nouveau.

La raison principale pour séparer cette fonction de `PQmakeEmptyPGResult` est qu'il est souvent approprié de créer un PGresult et de le remplir avec des données avant d'appeler les procédures d'événement.

`PQcopyResult`

Fait une copie de l'objet PGresult. La copie n'est liée en aucune façon au résultat source et `PQclear` doit être appelée dans que la copie n'est plus nécessaire. Si la fonction échoue, NULL est renvoyé.

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

Cela n'a pas pour but de faire une copie exacte. Le résultat renvoyé a toujours le statut `PGRES_TUPLES_OK`, et ne copie aucun message d'erreur dans la source. (Néanmoins, il copie la chaîne de statut de commande.) L'argument *`flags`* détermine le reste à copier. C'est un OR bit à bit de plusieurs drapeaux. `PG_COPYRES_ATTRS` indique la copie des attributs du résultat source (définition des colonnes). `PG_COPYRES_TUPLES` indique la copie des lignes du résultat source. (Cela implique de copier aussi les attributs.) `PG_COPYRES_NOTICEHOOKS` indique la copie des gestionnaires de notification du résultat source. `PG_COPYRES_EVENTS` indique la copie des événements du résultat source. (Mais toute instance de données associée avec la source n'est pas copiée.)

`PQsetResultAttrs`

Initialise les attributs d'un objet PGresult.

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc
*attDescs);
```

Les *`attDescs`* fournis sont copiés dans le résultat. Si le pointeur *`attDescs`* est NULL ou si *`numAttributes`* est inférieur à 1, la requête est ignorée et la fonction réussit. Si *`res`* contient déjà les attributs, la fonction échouera. Si la fonction

échoue, la valeur de retour est zéro. Si la fonction réussit, la valeur de retour est différente de zéro.

`PQsetvalue`
Initialise la valeur d'un champ d'une ligne d'un objet PGresult.

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int
len);
```

La fonction fera automatiquement grossir le tableau de lignes internes des résultats, si nécessaire. Néanmoins, l'argument *tup_num* doit être inférieur ou égal à `PQntuples`, ceci signifiant que la fonction peut seulement faire grossir le tableau des lignes une ligne à la fois. Mais tout champ d'une ligne existante peut être modifié dans n'importe quel ordre. Si une valeur à *field_num* existe déjà, elle sera écrasée. Si *len* vaut `-1` ou si *value* est `NULL`, la valeur du champ sera configurée à la valeur SQL `NULL`. *value* est copié dans le stockage privé du résultat, donc n'est plus nécessaire au retour de la fonction. Si la fonction échoue, la valeur de retour est zéro. Dans le cas contraire, elle a une valeur différente de zéro.

`PQresultAlloc`
Alloue un stockage supplémentaire pour un objet PGresult.

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

Toute mémoire allouée avec cette fonction est libérée quand *res* est effacée. Si la fonction échoue, la valeur de retour vaut `NULL`. Le résultat est garanti d'être correctement aligné pour tout type de données, comme pour un `malloc`.

31.11. Traitement des messages

Les messages de note et d'avertissement générés par le serveur ne sont pas renvoyés par les fonctions d'exécution des requêtes car elles n'impliquent pas d'échec dans la requête. À la place, elles sont passées à la fonction de gestion des messages et l'exécution continue normalement après le retour du gestionnaire. La fonction par défaut de gestion des messages affiche le message sur `stderr` mais l'application peut surcharger ce comportement en proposant sa propre fonction de gestion.

Pour des raisons historiques, il existe deux niveaux de gestion de messages, appelés la réception des messages et le traitement. Pour la réception, le comportement par défaut est de formater le message et de passer une chaîne au traitement pour affichage. Néanmoins, une application qui choisit de fournir son propre receveur de messages ignorera typiquement la couche d'envoi de messages et effectuera tout le travail au niveau du receveur.

La fonction `PQsetNoticeReceiver` initialise ou examine le receveur actuel de messages pour un objet de connexion. De la même façon, `PQsetNoticeProcessor` initialise ou examine l'émetteur actuel de messages.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Chacune de ces fonctions reçoit le pointeur de fonction du précédent receveur ou émetteur de messages et configure la nouvelle valeur. Si vous fournissez un pointeur de fonction nul, aucune action n'est réalisée mais le pointeur actuel est renvoyé.

Quand un message de note ou d'avertissement est reçu du serveur ou généré de façon interne par libpq, la fonction de réception du message est appelée. Le message lui est passé sous la forme d'un PGresult PGRES_NONFATAL_ERROR (ceci permet au receveur d'extraire les champs individuels en utilisant `PQresultErrorField` ou le message complet préformaté en utilisant `PQresultErrorMessage`). Le même pointeur void passé à `PQsetNoticeReceiver` est aussi renvoyé (ce pointeur peut être utilisé pour accéder à un état spécifique de l'application si nécessaire).

Le receveur de messages par défaut extrait simplement le message (en utilisant `PQresultErrorMessage`) et le passe au système de traitement du message.

Ce dernier est responsable de la gestion du message de note ou d'avertissement donné au format texte. La chaîne texte du message

est passée avec un retour chariot supplémentaire, plus un pointeur sur void identique à celui passé à `PQsetNoticeProcessor` (ce pointeur est utilisé pour accéder à un état spécifique de l'application si nécessaire).

Le traitement des messages par défaut est simplement

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

Une fois que vous avez initialisé un receveur ou une fonction de traitement des messages, vous devez vous attendre à ce que la fonction soit appelée aussi longtemps que l'objet PGconn ou qu'un objet PGresult réalisé à partir de celle-ci existent. À la création d'un PGresult, les pointeurs de gestion actuels de PGconn sont copiés dans PGresult pour une utilisation possible par des fonctions comme `PQgetvalue`.

31.12. Système d'événements

Le système d'événements de libpq est conçu pour notifier les gestionnaires d'événements enregistrés de l'arrivée d'événements intéressants de la libpq, comme par exemple la création ou la destruction d'objets PGconn et PGresult. Un cas d'utilisation principal est de permettre aux applications d'associer leur propres données avec un PGconn ou un PGresult et de s'assurer que les données soient libérées au bon moment.

Chaque gestionnaire d'événement enregistré est associé avec deux types de données, connus par libpq comme des pointeurs opaques, c'est-à-dire `void *`. Il existe un pointeur *passthrough* fournie par l'application quand le gestionnaire d'événements est enregistré avec un PGconn. Le pointeur passthrough ne change jamais pendant toute la durée du PGconn et des PGresult générés grâce à lui ; donc s'il est utilisé, il doit pointer vers des données vivantes. De plus, il existe une pointeur de *données instanciées*, qui commence à NULL dans chaque objet PGconn et PGresult. Ce pointeur peut être manipulé en utilisant les fonctions `PQinstanceData`, `PQsetInstanceData`, `PQresultInstanceData` et `PQsetResultInstanceData`. Notez que, contrairement au pointeur passthrough, les PGresult n'héritent pas automatiquement des données instanciées d'un PGconn. libpq ne sait pas vers quoi pointent les pointeurs passthrough et de données instanciées, et n'essaiera hamais de les libérer -- cela tient de la responsabilité du gestionnaire d'événements.

31.12.1. Types d'événements

La variable `PGEventId` de type enum précise tous les types d'événements gérés par le système d'événements. Toutes ces valeurs ont des noms commençant avec `PGEVT`. Pour chaque type d'événement, il existe une structure d'informations sur l'événement, précisant les paramètres passés aux gestionnaires d'événement. Les types d'événements sont :

`PGEVT_REGISTER`

L'événement d'enregistrement survient quand `PQregisterEventProc` est appelé.; C'est le moment idéal pour initialiser toute structure `instanceData` qu'une procédure d'événement pourrait avoir besoin. Seul un événement d'enregistrement sera déclenché par gestionnaire d'évévenement sur une connexion. Si la procédure échoue, l'enregistrement est annulé.

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

Quand un événement `PGEVT_REGISTER` est reçu, le pointeur *`evtInfo`* doit être converti en un PGEventRegister *. Cette structure contient un PGconn qui doit être dans le statut `CONNECTION_OK` ; garanti si `PQregisterEventProc` est appelé juste après avoir obtenu un bon PGconn. Lorsqu'elle renvoit un code d'erreur, le nettoyage doit être réalisé car aucun événement `PGEVT_CONNDESTROY` ne sera envoyé.

`PGEVT_CONNRESET`

L'événement de réinitialisation de connexion est déclenché après un `PQreset` ou un `PQresetPoll`. Dans les deux cas, l'événement est seulement déclenché si la ré-initialisation est réussie. Si la procédure échoue, la réinitialisation de connexion échouera ; la structure PGconn est placée dans le statut `CONNECTION_BAD` et `PQresetPoll` renverra `PGRES_POLLING_FAILED`.

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

Quand un événement PGEVT_CONNRESET est reçu, le pointeur *evtInfo* doit être converti en un PGEventConnReset *. Bien que le PGconn a été réinitialisé, toutes les données de l'événement restent inchangées. Cet événement doit être utilisé pour ré-initialiser/recharger/re-requêter tout instanceData associé. Notez que même si la procédure d'événement échoue à traiter PGEVT_CONNRESET, elle recevra toujours un événement PGEVT_CONNDESTROY à la fermeture de la connexion.

PGEVT_CONNDESTROY

L'événement de destruction de la connexion est déclenchée en réponse à PQfinish. Il est de la responsabilité de la procédure de l'événement de nettoyer proprement ses données car libpq n'a pas les moyens de gérer cette mémoire. Un échec du nettoyage amènera des pertes mémoire.

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

Quand un événement PGEVT_CONNDESTROY est reçu, le pointeur *evtInfo* doit être converti en un PGEventConnDestroy *. Cet événement est déclenché avant que PQfinish ne réalise d'autres nettoyages. La valeur de retour de la procédure est ignorée car il n'y a aucun moyen d'indiquer un échec de PQfinish. De plus, un échec de la procédure ne doit pas annuler le nettoyage de la mémoire non désirée.

PGEVT_RESULTCREATE

L'événement de création de résultat est déclenché en réponse à l'utilisation d'une fonction d'exécution d'une requête, par exemple PQgetResult. Cet événement sera déclenché seulement après la création réussie du résultat.

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

Quand un événement PGEVT_RESULTCREATE est reçu, le pointeur *evtInfo* doit être converti en un PGEventResultCreate *. Le paramètre *conn* est la connexion utilisée pour générer le résultat. C'est le moment idéal pour initialiser tout instanceData qui doit être associé avec le résultat. Si la procédure échoue, le résultat sera effacé et l'échec sera propagé. Le procédure d'événement ne doit pas tenter un PQclear sur l'objet résultat lui-même. Lors du renvoi d'un code d'échec, tout le nettoyage doit être fait car aucun événement PGEVT_RESULTDESTROY ne sera envoyé.

PGEVT_RESULTCOPY

L'événement de copie du résultat est déclenché en réponse à un PQcopyResult. Cet événement se déclenchera seulement une fois la copie terminée. Seules les procédures qui ont gérées avec succès l'événement PGEVT_RESULTCREATE ou PGEVT_RESULTCOPY pour le résultat source recevront les événements PGEVT_RESULTCOPY.

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

Quand un événement PGEVT_RESULTCOPY est reçu, le pointeur *evtInfo* doit être converti en un PGEventResultCopy *. Le résultat résultat *src* correspond à ce qui a été copié alors que le résultat *dest* correspond à la destination. Cet événement peut être utilisé pour fournir une copie complète de instanceData, ce que PQcopyResult ne peut pas faire. Si la procédure échoue, l'opération complète de copie échouera et le résultat *dest* sera effacé. Au renvoi d'un code d'échec, tout le nettoyage doit être réalisé car aucun événement PGEVT_RESULTDESTROY ne sera envoyé pour le résultat de destination.

PGEVT_RESULTDESTROY

L'événement de destruction de résultat est déclenché en réponse à la fonction PQclear. C'est de la responsabilité de l'événement de nettoyer proprement les données de l'événement car libpq n'a pas cette capacité en matière de gestion de mémoire. Si le nettoyage échoue, cela sera la cause de pertes mémoire.

```
typedef struct
{
    PGresult *result;
```

```
} PGEventResultDestroy;
```

Quand un événement PGEVT_RESULTDESTROY est reçu, le pointeur *evtInfo* doit être converti en un PGEventResultDestroy *. Cet événement est déclenché avant que PQclear ne puisse faire de nettoyage. La valeur de retour de la procédure est ignorée car il n'existe aucun moyen d'indiquer un échec à partir de PQclear. De plus, un échec de la procédure ne doit pas annuler le nettoyage de la mémoire non désirée.

31.12.2. Procédure de rappel de l'événement

PGEventProc

PGEventProc est une définition de type pour un pointeur vers une procédure d'événement, c'est-à-dire la fonction utilisateur appelée pour les événements de la libpq. La signature d'une telle fonction doit être :

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

Le paramètre *evtId* indique l'événement PGEVT qui est survenu. Le pointeur *evtInfo* doit être converti vers le type de structure approprié pour obtenir plus d'informations sur l'événement. Le paramètre *passThrough* est le pointeur fourni à PQregisterEventProc quand la procédure de l'événement a été enregistrée. La fonction doit renvoyer une valeur différente de zéro en cas de succès et zéro en cas d'échec.

Une procédure d'événement particulière peut être enregistrée une fois seulement pour un PGconn. Ceci est dû au fait que l'adresse de la procédure est utilisée comme clé de recherche pour identifier les données instanciées associées.

Attention

Sur Windows, les fonctions peuvent avoir deux adresses différentes : une visible de l'extérieur de la DLL et une visible de l'intérieur. Il faut faire attention que seule une de ces adresses est utilisée avec les fonctions d'événement de la libpq, sinon une confusion en résultera. La règle la plus simple pour écrire du code qui fonctionnera est de s'assurer que les procédures d'événements sont déclarées static. Si l'adresse de la procédure doit être disponible en dehors de son propre fichier source, il faut exposer une fonction séparée pour renvoyer l'adresse.

31.12.3. Fonctions de support des événements

PQregisterEventProc

Enregistre une procédure de rappel pour les événements avec libpq.

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

Une procédure d'évenement doit être enregistré une fois pour chaque PGconn pour lequel vous souhaitez recevoir des événements. Il n'existe pas de limites, autre que la mémoire, sur le nombre de procédures d'événements qui peuvent être enregistrées avec une connexion. La fonction renvoie une valeur différente de zéro en cas de succès, et zéro en cas d'échec.

L'argument *proc* sera appelé quand se déclenchera un événement libpq. Son adresse mémoire est aussi utilisée pour rechercher instanceData. L'argument *name* est utilisé pour faire référence à la procédure d'évenement dans les messages d'erreur. Cette valeur ne peut pas être NULL ou une chaîne de longueur nulle. La chaîne du nom est copiée dans PGconn, donc ce qui est passé n'a pas besoin de durer longtemps. Le pointeur *passThrough* est passé à *proc* à chaque arrivée d'un événement. Cet argument peut être NULL.

PQsetInstanceData

Initialise instanceData de la connexion pour la procédure *proc* avec *data*. Cette fonction renvoit zéro en cas d'échec et autre chose en cas de réussite. (L'échec est seulement possible si *proc* n'a pas été correctement enregistré dans le résultat.)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

PQinstanceData

Renvoie le `instanceData` de la connexion associée avec *`connproc`* ou NULL s'il n'y en a pas.

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

`PQresultSetInstanceData`

Initialise le `instanceData` du résultat pour la procédure *`proc`* avec *`data`*. Cette fonction renvoit zéro en cas d'échec et autre chose en cas de réussite. (L'échec est seulement possible si proc n'a pas été correctement enregistré dans le résultat.)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

`PQresultInstanceData`

Renvoie le `instanceData` du résultat associé avec *`proc`* ou NULL s'il n'y en a pas.

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

31.12.4. Exemple d'un événement

Voici un exemple d'une gestion de données privées associée aux connexions et aux résultats de la libpq.

```
/* en-tête nécssaire pour les événements de la libpq (note : inclut libpq-fe.h) */
#include <libpq-events.h>

/* la donnée instanciée : instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn = PQconnectdb("dbname = postgres");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* appelée une fois pour toute connexion qui doit recevoir des événements.
     * Envoit un PGEVT_REGISTER à myEventProc.
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* la connexion instanceData est disponible */
    data = PQinstanceData(conn, myEventProc);

    /* Envoit un PGEVT_RESULTCREATE à myEventProc */
```

```
    res = PQexec(conn, "SELECT 1 + 1");

    /* le résultat instanceData est disponible */
    data = PQresultInstanceData(res, myEventProc);

    /* Si PG_COPYRES_EVENTS est utilisé, envoit un PGEVT_RESULTCOPY à myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

    /* le résultat instanceData est disponible si PG_COPYRES_EVENTS a été
     * utilisé lors de l'appel à PQcopyResult.
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    /* Les deux fonctions de nettoyage envoient PGEVT_RESULTDESTROY à myEventProc */
    PQclear(res);
    PQclear(res_copy);

    /* Envoit un PGEVT_CONNDESTROY à myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* associe des données spécifiques de l'application avec la connexion */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* libère les données instanciées car la connexion est en cours de
destruction */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* associe des données spécifiques à l'application avec les résultats
(copié de la connexion) */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
```

```
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src, myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            /* associe des données spécifiques à l'application avec les résultats
(copié d'un résultat) */
            PQsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);

            /* libère les données instanciées car le résultat est en cours de
destruction */
            if (data)
              free_mydata(data);
            break;
        }

        /* unknown event id, just return TRUE. */
        default:
            break;
    }

    return TRUE; /* event processing succeeded */
}
```

31.13. Variables d'environnement

Les variables d'environnement suivantes peuvent être utilisées pour sélectionner des valeurs par défaut pour les paramètres de connexion, valeurs qui seront utilisées par `PQconnectdb`, `PQsetdbLogin` et `PQsetdb` si aucune valeur n'est directement précisée par le code d'appel. Elles sont utiles pour éviter de coder en dur les informations de connexion à la base de données dans les applications clients, par exemple.

- `PGHOST` se comporte de la même façon que le paramètre de configuration host.
- `PGHOSTADDR` se comporte de la même façon que le paramètre de configuration hostaddr. Elle peut être initialisée avec `PGHOST` pour éviter la surcharge des recherches DNS.
- `PGPORT` se comporte de la même façon que le paramètre de configuration port.
- `PGDATABASE` se comporte de la même façon que le paramètre de configuration dbname.
- `PGUSER` se comporte de la même façon que le paramètre de configuration user.
- `PGPASSWORD` se comporte de la même façon que le paramètre de configuration password. L'utilisation de cette variable d'environnement n'est pas recommandée pour des raisons de sécurité (certains systèmes d'exploitation autorisent les utilisateurs autres que root à voir les variables d'environnement du processus via ps) ; à la place, considérez l'utilisation du fichier `~/.pgpass` (voir la Section 31.14, « Fichier de mots de passe »).
- `PGPASSFILE` spécifie le nom du fichier de mot de passe à utiliser pour les recherches. Sa valeur par défaut est `~/.pgpass` (voir la Section 31.14, « Fichier de mots de passe »).
- `PGSERVICE` se comporte de la même façon que le paramètre de configuration service.
- `PGSERVICEFILE` indique le nom du fichier service de connexion par utilisateur. S'il n'est pas configuré, sa valeur par défaut est `~/.pg_service.conf` (voir Section 31.15, « Fichier des connexions de service »).
- `PGREALM` initialise le domaine Kerberos à utiliser avec PostgreSQL™ s'il est différent du domaine local. Si `PGREALM` est initialisé, les applications libpq tenteront une authentification avec les serveurs de ce domaine et utiliseront les fichiers tickets sé-

parés pour éviter les conflits avec les fichiers tickets locaux. Cette variable d'environnement est seulement utilisée si l'authentification Kerberos est sélectionnée par le serveur.

- `PGOPTIONS` se comporte de la même façon que le paramètre de configuration options.
- `PGAPPNAME` se comporte de la même façon que le paramètre de connexion application_name.
- `PGSSLMODE` se comporte de la même façon que le paramètre de configuration sslmode.
- `PGREQUIRESSL` se comporte de la même façon que le paramètre de configuration requiressl.
- `PGSSLKEY` spécifie le jeton matériel qui stocke la clé secrète pour le certificat client. La valeur de cette variable doit consister d'un nom de moteur séparé par une virgule (les moteurs sont les modules chargeables d'OpenSSL™) et un identifiant de clé spécifique au moteur. Si elle n'est pas configurée, la clé secrète doit être conservée dans un fichier.
- `PGSSLCERT` se comporte de la même façon que le paramètre de configuration sslcert.
- `PGSSLKEY` se comporte de la même façon que le paramètre de configuration sslkey.
- `PGSSLROOTCERT` se comporte de la même façon que le paramètre de configuration sslrootcert.
- `PGSSLCRL` se comporte de la même façon que le paramètre de configuration sslcrl.
- `PGKRBSRVNAME` se comporte de la même façon que le paramètre de configuration krbsrvname.
- `PGGSSLIB` se comporte de la même façon que le paramètre de configuration gsslib.
- `PGCONNECT_TIMEOUT` se comporte de la même façon que le paramètre de configuration connect_timeout.

Les variables d'environnement par défaut peuvent être utilisées pour spécifier le comportement par défaut de chaque session PostgreSQL™ (voir aussi les commandes ALTER USER(7) et ALTER DATABASE(7) pour des moyens d'initialiser le comportement par défaut sur des bases par utilisateur ou par bases de données).

- `PGDATESTYLE` initialise le style par défaut de la représentation de la date et de l'heure (équivalent à `SET datestyle TO ...`).
- `PGTZ` initialise le fuseau horaire par défaut (équivalent à `SET timezone TO ...`).
- `PGCLIENTENCODING` initialise le codage de l'ensemble des caractères par défaut (équivalent à `SET client_encoding TO ...`).
- `PGGEQO` initialise le mode par défaut pour l'optimiseur générique de requêtes (équivalent à `SET geqo TO ...`).

Référez-vous à la commande SQL SET(7) pour plus d'informations sur des valeurs correctes pour ces variables d'environnement.

Les variables d'environnement suivantes déterminent le comportement interne de libpq ; elles surchargent les valeurs internes par défaut.

- `PGSYSCONFDIR` configure le répertoire contenant le fichier `pg_service.conf` et dans une future version d'autres fichiers de configuration globaux au système.
- `PGLOCALEDIR` configure le répertoire contenant les fichiers `locale` pour l'internationalisation des messages.

31.14. Fichier de mots de passe

Le fichier `.pgpass`, situé dans le répertoire personnel de l'utilisateur, ou le fichier référencé par `PGPASSFILE` est un fichier contenant les mots de passe à utiliser si la connexion requiert un mot de passe (et si aucun mot de passe n'a été spécifié). Sur Microsoft Windows, le fichier est nommé `%APPDATA%\postgresql\pgpass.conf` (où `%APPDATA%` fait référence au sous-répertoire Application Data du profile de l'utilisateur).

Ce fichier devra être composé de lignes au format suivant (une ligne par connexion) :

```
nom_hote:port:database:nomutilisateur:motdepasse
```

Chacun des quatre premiers champs pourraient être une valeur littérale ou `*` (qui correspond à tout). La première ligne réalisant une correspondance pour les paramètres de connexion sera utilisée (du coup, placez les entrées plus spécifiques en premier lorsque vous utilisez des jokers). Si une entrée a besoin de contenir `:` ou `\`, échappez ce caractère avec `\`. Un nom d'hôte `localhost` correspond à la fois à une connexion TCP (nom d'hôte `localhost`) et à une connexion par socket de domaine Unix (`pghost` vide ou le répertoire par défaut du socket) provenant de la machine locale. Dans un serveur en standby, le nom de la base de données `replication` correspond aux connexions réalisées par le serveur maître pour la réplication en flux.

Sur les systèmes Unix, les droits sur `.pgpass` doivent interdire l'accès au groupe et au reste du monde ; faites-le par cette commande : **chmod 0600 ~/.pgpass**. Si les droits sont moins stricts que cela, le fichier sera ignoré. Sur Microsoft Windows, il est supposé que le fichier est stocké dans un répertoire qui est sécurisé, donc aucune vérification des droits n'est effectuée.

31.15. Fichier des connexions de service

Le fichier des connexions de service autorise l'association des paramètres de connexions avec un seul nom de service. Ce nom de service peut ensuite être spécifié par une connexion libpq et les paramétrages associés seront utilisés. Ceci permet de modifier les paramètres de connexion sans avoir à recompiler l'application libpq. Le nom de service peut aussi être spécifié en utilisant la variable d'environnement `PGSERVICE`.

Le fichier de service pour la connexion peut être un fichier par utilisateur sur `~/.pg_service.conf` ou à l'emplacement indiqué par la variable d'environnement `PGSERVICEFILE`. Il peut aussi être un fichier global au système dans le répertoire `etc/pg_service.conf` ou dans le répertoire indiqué par la variable d'environnement `PGSYSCONFDIR`. Si les définitions de service de même nom existent dans le fichier utilisateur et système, le fichier utilisateur est utilisé.

Le fichier utiliser le format des « fichiers INI » où le nom de la section et les paramètres sont des paramètres de connexion ; voir Section 31.1, « Fonctions de contrôle de connexion à la base de données » pour une liste. Par exemple :

```
# comment
[mabase]
host=unhote
port=5433
user=admin
```

Un fichier exemple est fourni sur `share/pg_service.conf.sample`.

31.16. Recherches LDAP des paramètres de connexion

Si libpq a été compilé avec le support de LDAP (option `--with-ldap` du script **configure**), il est possible de récupérer les options de connexion comme `host` ou `dbname` via LDAP à partir d'un serveur central. L'avantage en est que, si les paramètres de connexion d'une base évolue, l'information de connexion n'a pas à être modifiée sur toutes les machines clientes.

La recherche LDAP des paramètres de connexion utilise le fichier service `pg_service.conf` (voir Section 31.15, « Fichier des connexions de service »). Une ligne dans `pg_service.conf` commençant par `ldap://` sera reconnue comme une URL LDAP et une requête LDAP sera exécutée. Le résultat doit être une liste de paires `motclé = valeur` qui sera utilisée pour configurer les options de connexion. L'URL doit être conforme à la RFC 1959 et être de la forme :

```
ldap://[hôte[:port]]/base_recherche?attribut?étendue_recherche?filtre
```

où *`hôte`* vaut par défaut `localhost` et *`port`* vaut par défaut 389.

Le traitement de `pg_service.conf` se termine après une recherche réussie dans LDAP, mais continu si le serveur LDAP ne peut pas être contacté. Cela fournit un moyen de préciser d'autres URL LDAP pointant vers d'autres serveurs LDAP, des paires classiques `motclé = valeur` ou les options de connexion par défaut. Si vous obtenez à la place un message d'erreur, ajoutez une ligne syntaxiquement incorrecte après l'URL LDAP.

Un exemple d'une entrée LDAP qui a été créée à partir d'un fichier LDIF

```
version: 1
dn: cn=mabase,dc=masociété,dc=com
changetype: add
objectclass: top
objectclass: groupOfUniqueNames
cn: mabase
uniqueMember: host=monserveur.masociété.com
uniqueMember: port=5439
uniqueMember: dbname=mabase
uniqueMember: user=monutilisateur_base
uniqueMember: sslmode=require
```

amènera l'exécution de l'URL LDAP suivante :

```
ldap://ldap.masociété.com/dc=masociété,dc=com?uniqueMember?one?(cn=mabase)
```

Vous pouvez mélanger des entrées d'un fichier de service standard avec des recherches par LDAP. Voici un exemple complet dans

pg_service.conf :

```
# seuls l'hôte et le port sont stockés dans LDAP,
# spécifiez explicitement le nom de la base et celui de l'utilisateur
[customerdb]
dbname=clients
user=utilisateurappl
ldap://ldap.acme.com/cn=serveur,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

31.17. Support de SSL

PostgreSQL™ dispose d'un support natif des connexions SSL pour crypter les connexions client/serveur et améliorer ainsi la sécurité. Voir la Section 17.8, « Connexions tcp/ip sécurisées avec ssl » pour des détails sur la fonctionnalité SSL côté serveur.

libpq lit le fichier de configuration système d'OpenSSL™. Par défaut, ce fichier est nommé `openssl.cnf` et est placé dans le répertoire indiqué par `openssl version -d`. Cette valeur par défaut peut être surchargé en configurant la variable d'environnement `OPENSSL_CONF` avec le nom du fichier de configuration souhaité.

31.17.1. Vérification du certificat

Par défaut, PostgreSQL™ ne vérifie pas le certificat du serveur. Cela signifie qu'il est possible de se faire passer pour le serveur final (par exemple en modifiant un enregistrement DNS ou en prenant l'adresse IP du serveur) sans que le client ne le sache. Pour empêcher ceci, la vérification du certificat SSL doit être activée.

Si le paramètre `sslmode` est configuré à `verify-ca`, libpq vérifiera que le serveur est de confiance en vérifiant que le certificat a bien été généré par une autorité de certificats (CA) de confiance. Si `sslmode` est configuré à `verify-full`, libpq vérifiera *aussi* que le nom du serveur correspond à son certificat. La connexion SSL échouera si le certificat du serveur n'établit pas ces correspondances. La connexion SSL échouera si le certificat du serveur ne peut pas être vérifié. `verify-full` est recommandé pour les environnements les plus sensibles à la sécurité.

Dans le mode `verify-full`, l'attribut `cn` (Common Name) du certificat est testé par rapport au nom du serveur . Si l'attribut `cn` commence avec un astérisque(`*`), il sera traité comme un joker, et correspondra à tous les caractères *sauf* un point (`.`). Cela signifie que le certificat ne pourra pas être utilisé pour des sous-domaines complets. Si la connexion se fait en utilisant une adresse IP au lieu d'un nom d'hôte, l'adresse IP sera vérifiée (sans faire de recherche DNS).

Pour permettre la vérification du certificat du serveur, le certificat d'un ou plusieurs CA de confiance doit être placé dans le fichier `~/.postgresql/root.crt` dans le répertoire personnel de l'utilisateur. Su Microsoft Windows, le fichier est nommé `%APPDATA%\postgresql\root.crt`.

Les entrées de la liste de révocation des certificats (CRL) sont aussi vérifiées si le fichier `~/.postgresql/root.crl` existe (`%APPDATA%\postgresql\root.crl` sur Microsoft Windows).

L'emplacement du certificat racine et du CRL peuvent être changés avec les paramètres de connexion `sslrootcert` et `sslcrl`, ou les variables d'environnement `PGSSLROOTCERT` et `PGSSLCRL`.

> **Note**
>
> Pour une compatibilité ascendantes avec les anciennes versions de PostgreSQL, si un certificat racine d'autorité existe, le comportement de `sslmode=require` sera identique à celui de `verify-ca`. Cela signifie que le certificat du serveur est validé par l'autorité de certificat. Il ne faut pas se baser sur ce comportement. Les applications qui ont besoin d'une validation du certificat doivent toujours utiliser `verify-ca` ou `verify-full`.

31.17.2. Certificats des clients

Si le serveur réclame un certificat de confiance du client, libpq enverra le certificat stocké dans le fichier `~/.postgresql/postgresql.crt` du répertoire personnel de l'utilisateur. Le certificat doit être signé par une des autorités (CA) de confiance du serveur. Un fichier de clé privé correspondant `~/.postgresql/postgresql.key` doit aussi être présent. Le fichier de clé privée ne doit pas permettre son accès pour le groupe ou pour le reste du monde ; cela se fait avec la commande **chmod 0600 ~/.postgresql/postgresql.key**. Sur Microsoft Windows, ces fichiers sont nommés `%APPDATA%\postgresql\postgresql.crt` et `%APPDATA%\postgresql\postgresql.key`, et il n'existe pas de vérification de droits car ce répertoire est présumé sécurisé. L'emplacement des fichiers certificat et clé peut être surchargé par les paramètres de connexion `sslcert` et `sslkey`, ou les variables d'environnement `PGSSLCERT` et `PGSSLKEY`.

Dans certains cas, le certificat du client peut être signé par une autorité de certificat « intermédiaire », plutôt que par un qui est di-

rectement accepté par le serveur. Pour utiliser un tel certificat, ajoutez le certificat de l'autorité signataire du fichier `postgresql.crt`, alors son certificat de l'autorité parente, et ainsi de suite jusqu'à arriver à l'autorité « racine » qui est accepté par le serveur. Le certificat racine doit être inclus dans chaque cas où `postgresql.crt` contient plus d'un certificat.

Notez que `root.crt` liste les autorités de certificat de haut-niveau qui sont considérées de confiance pour les certificats serveur signataires. En principe, il n'a pas besoin de lister l'autorité de certificats qui a signé le certificat du client, bien que dans la plupart des cas, l'autorité du certificat sera aussi de confiance pour les certificats serveur.

31.17.3. Protection fournie dans les différents modes

Les différentes valeurs du paramètre `sslmode` fournissent différents niveaux de protection. SSL peut fournir une protection contre trois types d'attaques différentes :

Tableau 31.2. Attaques SSL

Type	Description
L'écoute	Si une troisième partie peut examiner le trafic réseau entre le client et le serveur, il peut lire à la fois les informations de connexion (ceci incluant le nom de l'utilisateur et son mot de passe) ainsi que les données qui y passent. SSL utilise le chiffrement pour empêcher cela.
Man in the middle (MITM)	Si une troisième partie peut modifier les données passant entre le client et le serveur, il peut prétendre être le serveur et, du coup, voir et modifier les données *y compris si elles sont chiffrées*. La troisième partie peut ensuite renvoyer les informations de connexion et les données au serveur d'origine, rendant à ce dernier impossible la détection de cette attaque. Les vecteurs communs pour parvenir à ce type d'attaque sont l'empoisonnement des DNS et la récupération des adresses IP où le client est dirigé vers un autre serveur que celui attendu. Il existe aussi plusieurs autres méthodes d'attaque pour accomplir ceci. SSL utilise la vérification des certificats pour empêcher ceci, en authentifiant le serveur auprès du client.
Impersonnification	Si une troisième partie peut prétendre être un client autorisé, il peut tout simplement accéder aux données auquel il n'a pas droit. Typiquement, cela peut arrier avec une gestion incorrecte des mots de passe. SSL utilise les certificats clients pour empêcher ceci, en s'assurant que seuls les propriétaires de certificats valides peuvent accéder au serveur.

Pour qu'une connexion soit sûre, l'utilisation de SSL doit être configurée *sur le client et sur le serveur* avant que la connexion ne soit effective. Si c'est seulement configuré sur le serveur, le client pourrait envoyer des informations sensibles (comme les mots de passe) avant qu'il ne sache que le serveur réclame une sécurité importante. Dans libpq, les connexions sécurisées peuvent être garanties en configurant le paramètre `sslmode` à `verify-full` ou `verify-ca`, et en fournissant au système un certificat racine à vérifier. Ceci est analogue à l'utilisation des URL `https` pour la navigation web chiffrée.

Une fois que le serveur est authentifié, le client peut envoyer des données sensibles. Cela signifie que jusqu'à ce point, le client n'a pas besoin de savoir si les certificats seront utilisés pour l'authentification, rendant particulièrement sûr de ne spécifier que ceci dans la configuration du serveur.

Toutes les options SSL ont une surcharge du type chiffrement et échange de clés. Il y a donc une balance entre performance et sécurité. La table suivante illustre les risques que les différentes valeurs de `sslmode` cherchent à protéger, et ce que cela apporte en sécurité et fait perdre en performances :

Tableau 31.3. Description des modes SSL

`sslmode`	Protection contre l'écoute	Protection contre l'attaque MITM	Remarques
`disable`	Non	Non	Peu m'importe la sécurité, je ne veux pas la surcharge apportée par le chiffrement.
`allow`	Peut-être	Non	Peu m'importe la sécurité, mais

sslmode	Protection contre l'écoute	Protection contre l'attaque MITM	Remarques
			je vais accepter la surcharge du chiffrement si le serveur insiste là-dessus.
prefer	Peut-être	Non	Peu m'importe la sécurité, mais j'accepte la surcharge du chiffrement si le serveur le supporte.
require	Oui	Non	Je veux chiffrer mes données, et j'accepte la surcharge. Je fais confiance au résreau pour me connecter toujours au serveur que je veux.
verify-ca	Oui	Depends on CA-policy	Je veux chiffrer mes données, et j'accepte la surcharge. Je veux aussi être sûr que je me connecte à un serveur en qui j'ai confiance.
verify-full	Oui	Oui	Je veux chiffrer mes données, et j'accepte la surcharge. Je veux être sûr que je me connecte à un serveur en qui j'ai confiance et que c'est bien celui que j'indique.

La différence entre `verify-ca` et `verify-full` dépend de la politique du CA racine. Si un CA publique est utilisé, `verify-ca` permet les connexions à un serveur que *quelqu'un d'autre* a pu enregistrer avec un CA accepté. Dans ce cas, `verify-full` devrait toujours être utilisé. Si un CA local est utilisé, voire même un certificat signé soi-même, utiliser `verify-ca` fournit souvent suffisamment de protection.

La valeur par défaut pour `sslmode` est `prefer`. Comme l'indique la table ci-dessus, cela n'a pas de sens d'un point de vue de la sécurité, et cela ne promet qu'une surcharge en terme de performance si possible. C'est uniquement fourni comme valeur par défaut pour la compatibilité ascendante, et n'est pas recommandé pour les déploiements de serveurs nécessitant de la sécurité.

31.17.4. Utilisation des fichiers SSL

Tableau 31.4. Utilisation des fichiers SSL libpq/client

Fichier	Contenu	Effet
~/.postgresql/postgresql.crt	certificat client	requis par le serveur
~/.postgresql/postgresql.key	clé privée du client	prouve le certificat client envoyé par l'utilisateur ; n'indique pas que le propriétaire du certificat est de confiance
~/.postgresql/root.crt	autorité de confiance du certificat	vérifie que le certificat du serveur est signé par une autorité de confiance
~/.postgresql/root.crl	certificats révoqués par les autorités	le certificat du serveur ne doit pas être sur cette liste

31.17.5. Initialisation de la bibliothèque SSL

Si votre application initialise les bibliothèques `libssl` et/ou `libcrypto` et que libpq est construit avec le support de SSL, vous devez appeler la fonction `PQinitOpenSSL` pour indiquer à libpq que les bibliothèques `libssl` et/ou `libcrypto` ont été initialisées par votre application, de façon à ce que libpq n'initialise pas elle-aussi ces bibliothèques. Voir *http://h71000.www7.hp.com/doc/83final/BA554_90007/ch04.html* pour plus de détails sur l'API SSL.

`PQinitOpenSSL`

Permet aux applications de sélectionner les bibliothèques de sécurité à initialiser.

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

Quand *do_ssl* est différent de zéro, libpq initialisera la bibliothèque OpenSSL avant d'ouvrir une connexion à la base de données. Quand *do_crypto* est différent de zéro, la bibliothèque `libcrypto` sera initialisée. Par défaut (si `PQinitOpenSSL` n'est pas appelé), les deux bibliothèques sont initialisées. Quand le support de SSL n'est pas intégré, cette fonction est présente mais ne fait rien.

Si votre application utilise et initialise soit OpenSSL soit `libcrypto`, vous *devez* appeler cette fonction avec des zéros pour les paramètres appropriés avant d'ouvrir la première connexion à la base de données. De plus, assurez-vous que vous avez fait cette initialisation avant d'ouvrir une connexion à la base de données.

`PQinitSSL`

Permet aux applications de sélectionner les bibliothèques de sécurité à initialiser.

```
void PQinitSSL(int do_ssl);
```

Cette fonction est équivalent à `PQinitOpenSSL(do_ssl, do_ssl)`. C'est suffisant pour les applications qui initialisent à la fois OpenSSL et `libcrypto` ou aucune des deux.

`PQinitSSL` est présente depuis PostgreSQL™ 8.0, alors que `PQinitOpenSSL` a été ajoutée dans PostgreSQL™ 8.4, donc `PQinitSSL` peut être préférée pour les applications qui ont besoin de fonctionner avec les anciennes versions de libpq.

31.18. Comportement des programmes threadés

libpq est réentrante et sûre avec les threads par défaut. Vous pourriez avoir besoin d'utiliser des options de compilation supplémentaires en ligne lorsque vous compiler le code de votre application. Référez-vous aux documentations de votre système pour savoir comment construire des applications actives au niveau thread ou recherchez `PTHREAD_CFLAGS` et `PTHREAD_LIBS` dans `src/Makefile.global`. Cette fonction permet d'exécuter des requêtes sur le statut de libpq concernant les threads :

`PQisthreadsafe`

Renvoie le statut de sûreté des threads pour libpq library.

```
int PQisthreadsafe();
```

Renvoie 1 si libpq supporte les threads, 0 dans le cas contraire.

Une restriction : il ne doit pas y avoir deux tentatives de threads manipulant le même objet PGconn à la fois. En particulier, vous ne pouvez pas lancer des commandes concurrentes à partir de threads différents à travers le même objet de connexion (si vous avez besoin de lancer des commandes concurrentes, utilisez plusieurs connexions).

Les objets PGresult sont en lecture seule après leur création et, du coup, ils peuvent être passés librement entre les threads. Néanmoins, si vous utilisez une des fonctions de modification d'un PGresult décrites dans Section 31.10, « Fonctions diverses » ou Section 31.12, « Système d'événements », vous devez aussi éviter toute opération concurrente sur le même PGresult.

Les fonctions obsolètes `PQrequestCancel` et `PQoidStatus` ne gèrent pas les threads et ne devraient pas être utilisées dans des programmes multithread. `PQrequestCancel` peut être remplacé par `PQcancel`. `PQoidStatus` peut être remplacé par `PQoidValue`.

Si vous utilisez Kerberos avec votre application (ainsi que dans libpq), vous aurez besoin de verrouiller les appels Kerberos car les fonctions Kerberos ne sont pas sûres lorsqu'elles sont utilisées avec des threads. Voir la fonction `PQregisterThreadLock` dans le code source de libpq pour récupérer un moyen de faire un verrouillage coopératif entre libpq et votre application.

Si vous expérimentez des problèmes avec les applications utilisant des threads, lancez le programme dans `src/tools/thread` pour voir si votre plateforme à des fonctions non compatibles avec les threads. Ce programme est lancé par `configure` mais, dans le cas des distributions binaires, votre bibliothèque pourrait ne pas correspondre à la bibliothèque utilisée pour construire les binaires.

31.19. Construire des applications avec libpq

Pour construire (c'est-à-dire compiler et lier) un programme utilisant libpq, vous avez besoin de faire tout ce qui suit :

- Incluez le fichier d'en-tête `libpq-fe.h` :

  ```
  #include <libpq-fe.h>
  ```

  Si vous ne le faites pas, alors vous obtiendrez normalement les messages d'erreurs similaires à ceci

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- Pointez votre compilateur sur le répertoire où les fichiers d'en-tête de PostgreSQL™ ont été installés en fournissant l'option `-Irépertoire` à votre compilateur (dans certains cas, le compilateur cherchera dans le répertoire en question par défaut, donc vous pouvez omettre cette option). Par exemple, votre ligne de commande de compilation devrait ressembler à ceci :

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  Si vous utilisez des makefiles, alors ajoutez cette option à la variable `CPPFLAGS` :

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  S'il existe une chance pour que votre programme soit compilé par d'autres utilisateurs, alors vous ne devriez pas coder en dur l'emplacement du répertoire. À la place, vous pouvez exécuter l'outil **pg_config** pour trouver où sont placés les fichiers d'en-tête sur le système local :

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  Un échec sur la spécification de la bonne option au compilateur résultera en un message d'erreur tel que

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- Lors de l'édition des liens du programme final, spécifiez l'option `-lpq` de façon à ce que les bibliothèques libpq soient intégrées, ainsi que l'option `-Lrépertoire` pour pointer le compilateur vers le répertoire où les bibliothèques libpq résident (de nouveau, le compilateur cherchera certains répertoires par défaut). Pour une portabilité maximale, placez l'option `-L` avant l'option `-lpq`. Par exemple :

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  Vous pouvez aussi récupérer le répertoire des bibliothèques en utilisant **pg_config** :

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  Les messages d'erreurs, pointant vers des problèmes de ce style, pourraient ressembler à ce qui suit.

  ```
  testlibpq.o: In function `main':
  testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to `PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
  ```

  Ceci signifie que vous avez oublié `-lpq`.

  ```
  /usr/bin/ld: cannot find -lpq
  ```

  Ceci signifie que vous avez oublié l'option `-L` ou que vous n'avez pas indiqué le bon répertoire.

31.20. Exemples de programmes

Ces exemples (et d'autres) sont disponibles dans le répertoire `src/test/examples` de la distribution des sources.

Exemple 31.1. Premier exemple de programme pour libpq

```
/*
 * testlibpq.c
 *
 *      Test the C version of libpq, the PostgreSQL frontend library.
```

```
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * Our test case here involves using a cursor, for which we must be inside
     * a transaction block.  We could do the whole thing with a single
     * PQexec() of "select * from pg_database", but that's too trivial to make
     * a good example.
     */

    /* Start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * Should PQclear PGresult whenever it is no longer needed to avoid memory
     * leaks
     */
    PQclear(res);

    /*
     * Fetch rows from pg_database, the system catalog of databases
     */
    res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
```

```
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in myportal");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* close the portal ... we don't bother to check for errors ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Exemple 31.2. Deuxième exemple de programme pour libpq

```
/*
 * testlibpq2.c
 *      Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
```

```
 * and do this four times:
 *
 *   INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * Issue LISTEN command to enable notifications from the rule's NOTIFY.
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid memory
     * leaks
     */
    PQclear(res);

    /* Quit after four notifies are received. */
    nnotifies = 0;
    while (nnotifies < 4)
    {
```

```
        /*
         * Sleep until something happens on the connection.  We use select(2)
         * to wait for input, but you could also use poll() or similar
         * facilities.
         */
        int         sock;
        fd_set      input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
            break;              /* shouldn't happen */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* Now check for input */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend pid %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
        }
    }

    fprintf(stderr, "Done.\n");

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Exemple 31.3. Troisième exemple de programme pour libpq

```
/*
 * testlibpq3.c
 *      Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE TABLE test1 (i int4, t text, b bytea);
 *
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
```

```
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <libpq-fe.h>

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * This function prints a query result that is a binary-format fetch from
 * a table defined as in the comment above.  We split it out because the
 * main() function uses it twice.
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* Use PQfnumber to avoid assumptions about field order in result */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* Get the field values (we ignore possibility they are null!) */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * The binary representation of INT4 is in network byte order, which
         * we'd better coerce to the local byte order.
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * The binary representation of TEXT is, well, text, and since libpq
         * was nice enough to append a zero byte to it, it'll work just fine
         * as a C string.
         *
         * The binary representation of BYTEA is a bunch of bytes, which could
         * include embedded nulls so we have to pay attention to field length.
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
```

```
                PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
                PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * The point of this program is to illustrate use of PQexecParams() with
     * out-of-line parameters, as well as binary transmission of data.
     *
     * This first example transmits the parameters as text, but receives the
     * results in binary format.  By using out-of-line parameters we can
     * avoid a lot of tedious mucking about with quoting and escaping, even
     * though the data is text.  Notice how we don't have to do anything
     * special with the quote mark in the parameter value.
     */

    /* Here is our out-of-line parameter value */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
                       1,       /* one param */
                       NULL,    /* let the backend deduce param type */
                       paramValues,
                       NULL,    /* don't need param lengths since text */
                       NULL,    /* default to all text params */
                       1);      /* ask for binary results */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
```

```
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /*
     * In this second example we transmit an integer parameter in binary
     * form, and again retrieve the results in binary form.
     *
     * Although we tell PQexecParams we are letting the backend deduce
     * parameter type, we really force the decision by casting the parameter
     * symbol in the query text.  This is a good safety measure when sending
     * binary parameters.
     */

    /* Convert integer value "2" to network byte order */
    binaryIntVal = htonl((uint32_t) 2);

    /* Set up parameter arrays for PQexecParams */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* binary */

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE i = $1::int4",
                       1,       /* one param */
                       NULL,    /* let the backend deduce param type */
                       paramValues,
                       paramLengths,
                       paramFormats,
                       1);      /* ask for binary results */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Chapitre 32. Objets larges

PostgreSQL™ a des fonctionnalités concernant les *objets larges*, fournissant un accès style flux aux données utilisateurs stockées dans une structure spéciale. L'accès en flux est utile pour travailler avec des valeurs de données trop larges pour être manipuler convenablement en entier.

Ce chapitre décrit l'implémentation, la programmation et les interfaces du langage de requêtes pour les données de type objet large dans PostgreSQL™. Nous utilisons la bibliothèque C libpq pour les exemples de ce chapitre mais la plupart des interfaces natives de programmation de PostgreSQL™ supportent des fonctionnalités équivalentes. D'autres interfaces pourraient utiliser l'interface des objets larges en interne pour fournir un support générique des valeurs larges. Ceci n'est pas décrit ici.

32.1. Introduction

Tous les objets larges sont placés dans une seule table système appelée `pg_largeobject`. PostgreSQL™ supporte aussi un système de stockage appelé « TOAST » qui stocke automatiquement les valeurs ne tenant pas sur une page de la base de données dans une aire de stockage secondaire par table. Ceci rend partiellement obsolète la fonctionnalité des objets larges. Un avantage restant des objets larges est qu'il autorise les valeurs de plus de 2 Go en taille alors que les champs TOAST peuvent être d'au plus 1 Go. Néanmoins, les objets larges peuvent être modifiés au hasard en utilisant une API de lecture/écriture qui est plus efficace que la réalisation de telles opérations utilisant TOAST.

32.2. Fonctionnalités d'implémentation

L'implémentation des objets larges, les coupe en « morceaux » (*chunks*) stockés dans les lignes de la base de données. Un index B-tree garantit des recherches rapides sur le numéro du morceau lors d'accès aléatoires en lecture et écriture.

À partir de PostgreSQL™ 9.0, les « Large Objects » ont un propriétaire et un ensemble de droits d'accès pouvant être gérés en utilisant les commandes GRANT(7) et REVOKE(7). Pour une compatibilité avec des versions précédentes, voir lo_compat_privileges. Les droits `SELECT` sont requis pour lire un « Large Object », et les droits `UPDATE` sont requis pour écrire ou tronquer. Seul le propriétaire du « Large Object » ou le propriétaire de la base de données peut supprimer, ajouter un commentaire ou modifier le propriétaire d'un « Large Object ».

32.3. Interfaces client

Cette section décrit les possibilités que les bibliothèques d'interfaces clientes de PostgreSQL™ fournissent pour accéder aux objets larges. Toutes les manipulations d'objets larges utilisant ces fonctions *doivent* prendre place dans un bloc de transaction SQL. L'interface des objets larges de PostgreSQL™ prend comme modèle l'interface des systèmes de fichiers Unix avec des fonctions analogues pour `open`, `read`, `write`, `lseek`, etc.

Les applications clientes utilisant l'interface des objets larges dans libpq doivent inclure le fichier d'en-tête `libpq/libpq-fs.h` et établir un lien avec la bibliothèque libpq.

32.3.1. Créer un objet large

La fonction

```
Oid lo_creat(PGconn *conn, int mode);
```

crée un nouvel objet large. La valeur de retour est un OID assigné au nouvel objet large ou InvalidOid (zéro) en cas d'erreur. *`mode`* est inutilisée et ignorée sur PostgreSQL™ 8.1 ; néanmoins, pour la compatibilité avec les anciennes versions, il est préférable de l'initialiser à INV_READ, INV_WRITE, ou INV_READ | INV_WRITE (ces constantes symboliques sont définies dans le fichier d'en-tête `libpq/libpq-fs.h`).

Un exemple :

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

La fonction

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

crée aussi un nouvel objet large. L'OID à affecter peut être spécifié par *`lobjId`* ; dans ce cas, un échec survient si l'OID est déjà utilisé pour un autre objet large. Si *`lobjId`* vaut InvalidOid (zero), alors `lo_create` affecte un OID inutilisé (ceci est le même comportement que `lo_creat`). La valeur de retour est l'OID qui a été affecté au nouvel objet large ou InvalidOid (zero) en cas d'échec.

`lo_create` est nouveau depuis PostgreSQL™ 8.1 ; si cette fonction est utilisée à partir d'un serveur d'une version plus an-

cienne, elle échouera et renverra InvalidOid.

Un exemple :

```
inv_oid = lo_create(conn, desired_oid);
```

32.3.2. Importer un objet large

Pour importer un fichier du système d'exploitation en tant qu'objet large, appelez

```
Oid lo_import(PGconn *conn, const char *filename);
```

filename spécifie le nom du fichier à importer comme objet large. Le code de retour est l'OID assigné au nouvel objet large ou InvalidOid (zero) en cas d'échec. Notez que le fichier est lu par la bibliothèque d'interface du client, pas par le serveur. Donc il doit exister dans le système de fichier du client et lisible par l'application du client.

La fonction

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

importe aussi un nouvel « large object ». L'OID à affecter peut être indiqué par *lobjId* ; dans ce cas, un échec survient si l'OID est déjà utilisé pour un autre « large object ». Si *lobjId* vaut InvalidOid (zéro) alors `lo_import_with_oid` affecte un OID inutilisé (et donc obtient ainsi le même comportement que `lo_import`). La valeur de retour est l'OID qui a été affecté au nouveau « large object » ou InvalidOid (zéro) en cas d'échec.

`lo_import_with_oid` est une nouveauté de PostgreSQL™ 8.4, et utilise en interne `lo_create` qui était une nouveauté de la 8.1 ; si cette fonction est exécutée sur un serveur en 8.0 voire une version précédente, elle échouera et renverra InvalidOid.

32.3.3. Exporter un objet large

Pour exporter un objet large en tant que fichier du système d'exploitation, appelez

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

L'argument *lobjId* spécifie l'OID de l'objet large à exporter et l'argument *filename* spécifie le nom du fichier. Notez que le fichier est écrit par la bibliothèque d'interface du client, pas par le serveur. Renvoie 1 en cas de succès, -1 en cas d'échec.

32.3.4. Ouvrir un objet large existant

Pour ouvrir un objet large existant pour lire ou écrire, appelez

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

L'argument *lobjId* spécifie l'OID de l'objet large à ouvrir. Les bits *mode* contrôlent si l'objet est ouvert en lecture (INV_READ), écriture (INV_WRITE) ou les deux (ces constantes symboliques sont définies dans le fichier d'en-tête `libpq/libpq-fs.h`). Un objet large ne peut pas être ouvert avant d'avoir été créé. `lo_open` renvoie un descripteur (positif) d'objet large pour une utilisation future avec `lo_read`, `lo_write`, `lo_lseek`, `lo_tell` et `lo_close`. Le descripteur est uniquement valide pour la durée de la transaction en cours. En cas d'échec, -1 est renvoyé.

Actuellement, le serveur ne fait pas de distinction entre les modes INV_WRITE et INV_READ `|` INV_WRITE : vous êtes autorisé à lire à partir du descripteur dans les deux cas. Néanmoins, il existe une différence significative entre ces modes et INV_READ seul : avec INV_READ, vous ne pouvez pas écrire sur le descripteur et la donnée lue à partir de ce dernier, reflètera le contenu de l'objet large au moment où `lo_open` a été exécuté dans la transaction active, quelques soient les possibles écritures par cette transaction ou par d'autres. Lire à partir d'un descripteur ouvert avec INV_WRITE renvoie des données reflétant toutes les écritures des autres transactions validées ainsi que les écritures de la transaction en cours. Ceci est similaire à la différence de comportement entre les modes de transaction `SERIALIZABLE` et `READ COMMITTED` pour les requêtes SQL **SELECT**.

Un exemple :

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

32.3.5. Écrire des données dans un objet large

La fonction

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

écrit *len* octets de *buf* dans le descripteur *fd* de l'objet large. L'argument *fd* doit avoir été renvoyé par un appel précédent à `lo_open`. Le nombre d'octets réellement écrits est renvoyé. Dans le cas d'une erreur, une valeur négative est renvoyée.

32.3.6. Lire des données à partir d'un objet large

La fonction

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

lit *len* octets du descripteur de l'objet large *fd* et les place dans *buf*. L'argument *fd* doit avoir été renvoyé par un appel précédent à `lo_open`. Le nombre d'octets réellement lus est renvoyé. Dans le cas d'une erreur, une valeur négative est renvoyée.

32.3.7. Recherche dans un objet large

Pour modifier l'emplacement courant de lecture ou écriture associé au descripteur d'un objet large, on utilise

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

Cette fonction déplace le pointeur d'emplacement courant pour le descripteur de l'objet large identifié par *fd* au nouvel emplacement spécifié avec le décalage (*offset*). Les valeurs valides pour *whence* sont SEEK_SET (rechercher depuis le début de l'objet), SEEK_CUR (rechercher depuis la position courante) et SEEK_END (rechercher depuis la fin de l'objet). Le code de retour est le nouvel emplacement du pointeur ou -1 en cas d'erreur.

32.3.8. Obtenir la position de recherche d'un objet large

Pour obtenir la position actuelle de lecture ou écriture d'un descripteur d'objet large, appelez

```
int lo_tell(PGconn *conn, int fd);
```

En cas d'erreur, le code de retour est négatif.

32.3.9. Tronquer un Objet Large

Pour tronquer un objet large avec une longueur donnée, on utilise

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

tronque l'objet large décrit par *fd* avec la longueur *len*. l'argument *fd* doit avoir été renvoyé par un appel précédent à `lo_open`. Si le paramètre *len* est plus grand que la taille de l'objet courant, l'objet sera completé avec des octets de valeur null ('\0').

Le décalage reste inchangé.

En cas de succès `lo_truncate` retourne zero. En cas d'erreur, la valeur de retour est négative.

`lo_truncate` est une nouveauté de PostgreSQL™ 8.3; si cette fonction est également exécuté sur un version plus ancienne du serveur, elle échouera et retournera une valeur négative.

32.3.10. Fermer un descripteur d'objet large

Un descripteur d'objet large peut être fermé en appelant

```
int lo_close(PGconn *conn, int fd);
```

où *fd* est un descripteur d'objet large renvoyé par `lo_open`. En cas de succès, `lo_close` renvoie zéro. Une valeur negative en cas d'échec.

Tous les descripteurs d'objets larges restant ouverts à la fin d'une transaction seront automatiquement fermés.

32.3.11. Supprimer un objet large

Pour supprimer un objet large de la base de données, on utilise

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

L'argument *lobjId* spécifie l'OID de l'objet large à supprimer. En cas d'erreur, le code de retour est négatif.

32.4. Fonctions du côté serveur

Ce sont des fonctions côté serveur appelables à partir de SQL et correspondant à chaque fonction côté client décrite ci-dessus ; en fait, pour leur grande part, les fonctions côté client sont simplement des interfaces vers les fonctions équivalentes côté serveur. Celles qui sont réellement utiles à appeler via des commandes SQL sont `lo_creat`, `lo_create`, `lo_unlink`, `lo_import` et `lo_export`. Voici des exemples de leur utilisation :

```
CREATE TABLE image (
```

```
    nom             text,
    donnees         oid
);

SELECT lo_creat(-1);       -- renvoie l'OID du nouvel objet large

SELECT lo_create(43213);   -- tente de créer l'objet large d'OID 43213

SELECT lo_unlink(173454);  -- supprime l'objet large d'OID 173454

INSERT INTO image (nom, donnees)
    VALUES ('superbe image', lo_import('/etc/motd'));

INSERT INTO image (nom, donnees)  -- identique à ci-dessus, mais précise l'OID à
utiliser
    VALUES ('superbe image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.donnees, '/tmp/motd') FROM image
    WHERE nom = 'superbe image';
```

Les fonctions `lo_import` et `lo_export` côté serveur se comportent considérablement différemment de leurs analogues côté client. Ces deux fonctions lisent et écrivent des fichiers dans le système de fichiers du serveur en utilisant les droits du propriétaire du serveur de base de données. Du coup, leur utilisation est restreinte aux superutilisateurs PostgreSQL. Au contraire des fonctions côté serveur, les fonctions d'import et d'export côté client lisent et écrivent des fichiers dans le système de fichiers du client en utilisant les droits du programme client. Les fonctions côté client ne nécessitent pas le droit superutilisateur.

Les fonctionnalités de `lo_read` et `lo_write` sont aussi disponibles via les appels côté serveur mais les noms des fonctions côté serveur diffèrent des interfaces côté client (elles n'ont pas les tirets bas). Vous devez appelez ces fonctions sous le nom de `loread` et de `lowrite`.

32.5. Programme d'exemple

L'Exemple 32.1, « Exemple de programme sur les objets larges avec libpq » est un programme d'exemple qui montre une utilisation de l'interface des objets larges avec libpq. Des parties de ce programme disposent de commentaires au bénéfice de l'utilisateur. Ce programme est aussi disponible dans la distribution des sources (`src/test/examples/testlo.c`).

Exemple 32.1. Exemple de programme sur les objets larges avec libpq

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test utilisant des objets larges avec libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    importe le fichier "in_filename" dans la base de données
 *    en tant qu'objet "lobjOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
```

```
    int         fd;

    /*
     * ouvre le fichier à lire
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * crée l'objet large
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "can't create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * lit le fichier Unix écrit dans le fichier inversion
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s";, buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
```

```
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile
 *    exporte l'objet large "lobjOid" dans le fichier
 * "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * ouvre l' « objet » large
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    /*
     * ouvre le fichier à écrire
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * lit à partir du fichier inversion et écrit dans le fichier Unix
```

```
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * initialise la connexion
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);
```

```
    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapitre 33. ECPG - SQL embarqué dans du C

Ce chapitre décrit le paquetage SQL embarqué pour PostgreSQL™. Il a été écrit par Linus Tolke (<linus@epact.se>) et Michael Meskes (<meskes@postgresql.org>). Originellement, il a été écrit pour fonctionner avec le langage C. Il fonctionne aussi avec le C++, mais il ne reconnaît pas encore toutes les constructions C++.

Cette documentation est assez incomplète. Mais du fait de la standardisation de cette interface, des informations complémentaires sont disponibles au travers de nombreuses ressources traitant du SQL.

33.1. Concept

Un programme SQL embarqué consiste en du code écrit dans un langage de programmation ordinaire, le C dans le cas présent, mélangé à des commandes SQL incluses dans des sections spécialement marquées. Pour construire le programme, le code source est d'abord passé au préprocesseur SQL embarqué qui le convertit en un programme C ordinaire. Il peut alors être traité par un compilateur C.

Le SQL embarqué a des avantages par rapport aux autres méthodes de gestion de commandes SQL dans du code C. Premièrement, il gère le passage laborieux des informations de et vers les variables du programme C. Deuxièmement, le code SQL du programme est vérifié syntaxiquement au moment de la construction. Troisièmement, le SQL embarqué en C est spécifié dans le standard SQL et supporté par de nombreux systèmes de bases de données SQL. L'implantation PostgreSQL™ est conçue pour correspondre au mieux à ce standard. Il est de ce fait assez facile de porter les programmes SQL embarqués écrits pour d'autres bases de données SQL vers PostgreSQL™.

Comme indiqué précédemment, les programmes écrits pour l'interface SQL embarqué sont des programmes C normaux contenant un code spécial inséré pour réaliser les actions en relation avec la base de données. Ce code spécial a toujours la forme :

```
EXEC SQL ...;
```

Ces instructions prennent syntaxiquement la place d'une instruction C. Suivant l'instruction particulière, elles peuvent apparaître dans le contexte global ou à l'intérieur d'une fonction. Les instructions SQL embarquées suivent les règles de sensibilité à la casse d'un code SQL normal, et non pas ceux du C.

Les sections suivantes expliquent toutes les instructions SQL embarquées.

33.2. Se connecter au serveur de bases de données

La connexion à une base de données se fait à l'aide de l'instruction suivante :

```
EXEC SQL CONNECT TO cible [AS nom-connexion] [USER nom-utilisateur];
```

La *cible* peut être indiquée d'une des façons suivantes :

- *nom_base*[@*nomhôte*][:*port*]
- tcp:postgresql:*//nomhôte* [:*port*] [*/nom_base*][? *options*]
- unix:postgresql:*//nomhôte*[: *port*][*/nom_base*][? *options*]
- une chaîne SQL contenant une des formes précédentes
- une référence à une variable contenant une des formes précédentes (voir les exemples)
- DEFAULT

Si la cible de connexion est indiquée littéralement (c'est-à-dire non pas via une variable de référence) et que la valeur n'est pas mise entre guillemets, les règles d'insensibilité à la casse du SQL standard sont appliquées. Dans ce cas, il est possible, si cela s'avère nécessaire, d'encadrer séparément les paramètres individuels de guillemets doubles. En pratique, l'utilisation d'une chaîne littérale (entre guillemets simples) ou d'une variable de référence engendre moins d'erreurs. La cible de connexion DEFAULT initie une connexion à la base de données standard avec l'utilisateur standard. Aucun nom d'utilisateur ou de connexion ne peut être indiqué isolément dans ce cas.

Il existe également différentes façons de préciser le nom de l'utilisateur :

- *nomutilisateur*
- *nomutilisateur/ motdepasse*
- *nomutilisateur* IDENTIFIED BY *motdepasse*

- *nomutilisateur* USING *motdepasse*

Comme indiqué ci-dessus, les paramètres *nomutilisateur* et *motdepasse* peuvent être un identificateur SQL, une chaîne SQL ou une référence à une variable de type caractère.

nom-connexion est utilisé pour gérer plusieurs connexions dans un même programme. Il peut être omis si le programme n'utilise qu'une seule connexion. La connexion la plus récemment ouverte devient la connexion courante, utilisée par défaut lorsqu'une instruction SQL est à exécuter (voir plus loin dans ce chapitre).

Quelques exemples d'instructions **CONNECT** :

```
EXEC SQL CONNECT TO ma_base@sql.mondomaine.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mondomaine.com/ma_base AS maconnexion USER
john;

EXEC SQL BEGIN DECLARE SECTION;
const char *cible = "ma_base@sql.mondomaine.com";
const char *utilisateur = "john";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :cible USER :utilisateur;
```

La dernière forme utilise la variante de référence de variable caractère, citée plus haut. L'utilisation de variables C dans les instructions SQL, en les préfixant d'un caractère deux-points, est expliqué dans les prochaines sections.

Le format de la cible de connexion n'est pas spécifié dans le standard SQL. Ainsi, lorsque l'on souhaite développer des applications portables, il est préférable d'utiliser une syntaxe fondée sur le dernier exemple ci-dessus pour encapsuler la chaîne de la cible de connexion.

33.3. Fermer une connexion

Pour fermer une connexion, l'instruction suivante est utilisée :

```
EXEC SQL DISCONNECT [connexion];
```

connexion peut être indiquée de différentes façons :

- *nom-connexion*
- DEFAULT
- CURRENT
- ALL

Si aucun nom de connexion n'est indiqué, la connexion en cours est fermée.

Il est toujours préférable qu'une application ferme explicitement chaque connexion qu'elle a ouverte.

33.4. Exécuter des commandes SQL

Toute commande SQL peut être exécutée à l'intérieur d'une application SQL embarquée. Ci-dessous se trouvent quelques exemples de procédures.

Création d'une table :

```
EXEC SQL CREATE TABLE foo (nombre integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(nombre);
EXEC SQL COMMIT;
```

Insertion de lignes :

```
EXEC SQL INSERT INTO foo (nombre, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Suppression de lignes :

```
EXEC SQL DELETE FROM foo WHERE nombre = 9999;
EXEC SQL COMMIT;
```

Sélection d'une ligne :

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Sélection à l'aide de curseurs :

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT nombre, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Actualisations :

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE nombre = 9999;
EXEC SQL COMMIT;
```

Les lexèmes de la forme « `:quelquechose` » sont des *variables hôtes*, c'est-à-dire qu'elles font référence à des variables du programme C. Elles sont expliquées dans la Section 33.6, « Utiliser des variables hôtes ».

Dans le mode par défaut, les instructions ne sont validées que lorsque **EXEC SQL COMMIT** est exécuté. L'interface SQL embarqué supporte aussi la validation automatique des transactions (similaire au comportement de libpq) via l'option `-t` en ligne de commande pour **ecpg** (voir ci-dessous) ou via l'instruction `EXEC SQL SET AUTOCOMMIT TO ON`. En mode de validation automatique, chaque commande est automatiquement validée sauf si elle est à l'intérieur d'un bloc de transaction explicite. Ce mode peut être explicitement désactivé en utilisant `EXEC SQL SET AUTOCOMMIT TO OFF`.

33.5. Choisir une connexion

Les instructions SQL affichées dans la section précédente sont exécutées à partir de la connexion courante, c'est-à-dire la dernière à avoir été ouverte. Il y a deux façons de gérer l'utilisation de plusieurs connexions dans une application.

La première option est de choisir explicitement une connexion pour chaque instruction SQL, par exemple :

```
EXEC SQL AT nom-connexion SELECT ...;
```

Cette option est particulièrement adaptée si l'application a besoin d'utiliser alternativement plusieurs connexions.

Si l'application utilise plusieurs threads (fil) d'exécution, ils ne peuvent pas concurrement partager une connexion. Il faut, soit contrôler explicitement l'accès à la connexion (en utilisant des mutex), soit utiliser une connexion pour chaque thread. Si chaque thread utilise sa propre connexion, il est nécessaire d'utiliser la clause AT pour préciser la connexion utilisée par le thread.

La seconde option consiste à exécuter une instruction pour basculer la connexion courante. L'instruction est :

```
EXEC SQL SET CONNECTION nom-connexion;
```

Cette option est particulièrement intéressante si un grand nombre d'instructions doivent être exécutées à partir de la même connexion. Elle ne gère pas les threads.

33.6. Utiliser des variables hôtes

La Section 33.4, « Exécuter des commandes SQL » présente l'exécution d'instructions SQL à partir d'un programme SQL embarqué. Certaines de ces instructions n'utilisent que des valeurs fixes. Elles n'offrent pas la possibilité d'insérer des valeurs fournies par l'utilisateur dans les instructions. Elles ne permettent pas non plus au programme de traiter les valeurs renvoyées par la requête. Ces types d'instructions ne sont pas vraiment utiles dans les applications réelles. Cette section explique en détails l'échange de données entre le programme C et les instructions SQL embarquées à l'aide d'un mécanisme simple appelé *variables hôtes*. Dans un programme SQL embarqué, les instructions SQL sont considérées comme *invitées* dans le code du programme C qui est le *langage hôte*. Du coup, les variables du programme C sont appelées *variables hôtes*.

33.6.1. Aperçu

L'échange de données entre le programme C et les instructions SQL est particulièrement simple en SQL embarqué. Plutôt que de laisser le programme copier les données dans l'instruction, ce qui implique un certain nombre de complications, dont la bonne mise entre guillemets de la valeur, il est plus simple d'écrire le nom de la variable C dans l'instruction SQL en la préfixant par un

caractère deux-points. Par exemple :

```
EXEC SQL INSERT INTO unetable VALUES (:v1, 'foo', :v2);
```

Cette instruction fait référence à deux variables C nommées `v1` et `v2`, et utilise également une chaîne SQL pour illustrer l'absence de restriction à l'utilisation d'un type de données ou d'un autre.

Ce style d'insertion de variables C dans des instructions SQL fonctionne dans tous les cas où l'on attend une expression de valeur dans une instruction SQL.

33.6.2. Sections de déclaration

Pour passer des données du programme à la base de données, comme paramètres d'une requête par exemple, ou pour passer des données de la base au programme, les variables C supposées contenir ces données doivent être déclarées dans des sections spécialement marquées pour que le préprocesseur du SQL embarqué soit averti de leur présence.

Cette section commence avec :

```
EXEC SQL BEGIN DECLARE SECTION;
```

et se termine avec :

```
EXEC SQL END DECLARE SECTION;
```

Entre ces lignes, on trouve des déclarations normales de variables C, comme :

```
int   x = 4;
char  foo[16], bar[16];
```

Une valeur initiale optionnelle peut être affectée à la variable. La portée de la variable est déterminée par son emplacement dans la section de déclaration du programme. Les variables peuvent aussi être déclarées avec la syntaxe suivante qui crée implicitement une section de déclaration :

```
EXEC SQL int i = 4;
```

Le nombre de sections de déclarations dans un programme n'est pas limité.

Les déclarations sont aussi placées dans le fichier de sortie comme des variables C normales. Du coup, il n'est plus nécessaire de les déclarer à nouveau. Les variables qui n'ont pas pour but d'être utilisées dans des commandes SQL peuvent être normalement déclarées en dehors des sections spéciales.

La définition d'une structure ou union doit aussi être saisie dans une section `DECLARE`. Sinon, le préprocesseur, ne connaissant pas leur définition, ne peut pas gérer ces types.

33.6.3. Les différents types de variables hôtes

Des tableaux (*array*), définitions de type (*typedef*), structures (*struct*) et pointeurs (*pointer*) peuvent aussi être utilisés comme variables hôtes. Il existe également des types spéciaux de variables hôtes qui n'existent qu'en ECPG.

Quelques exemples sur les variables hôtes :

Tableaux

Une des utilisations les plus communes d'une déclaration de tableaux est certainement l'allocation d'un tableau de caractères :

```
EXEC SQL BEGIN DECLARE SECTION;
    char chaine[50];
EXEC SQL END DECLARE SECTION;
```

C'est à l'utilisateur de gérer la longueur du tableau. Si une telle variable hôte est utilisée comme variable cible d'une requête qui renvoie une chaîne de plus de 49 caractères, un dépassement de tampon survient.

Typedefs

Le mot clé `typedef` est utilisé pour faire correspondre de nouveaux types à des types déjà existants.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char montypecaractere[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

On peut aussi utiliser :

```
EXEC SQL TYPE serial_t IS long;
```

Cette déclaration n'a pas besoin de faire partie d'une section de déclaration.

Pointeurs

Des pointeurs peuvent être déclarés vers les types les plus communs. Néanmoins, Les pointeurs en tant que variables cibles de requêtes ne peuvent être utilisés sans allocation automatique. Voir Section 33.9, « Utiliser les aires de descripteur » pour plus d'informations sur l'allocation automatique.

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

Types spéciaux de variables

ECPG contient certains types spéciaux qui facilitent l'interaction avec les données du serveur SQL. Par exemple, le support des types varchar, numeric, date, timestamp et interval a été implanté. Section 33.8, « Bibliothèque pgtypes » contient des fonctions basiques de gestion de ces types. Il n'est ainsi pas nécessaire d'envoyer une requête au serveur SQL simplement pour ajouter un interval à une variable de type timestamp, par exemple.

Le type spécial VARCHAR est converti en une struct nommée pour chaque variable. La déclaration :

```
VARCHAR var[180];
```

est convertie en :

```
struct varchar_var { int len; char arr[180]; } var;
```

Cette structure convient pour interfacer des données SQL de type varchar.

33.6.4. SELECT INTO et FETCH INTO

Les sections précédentes expliquent le passage de données entre l'application et une commande SQL. Pour récupérer le résultat d'une requête, le SQL embarqué fournit des variantes spéciales des commandes habituelles **SELECT** et **FETCH**. Ces commandes ont une clause `INTO` particulière qui indique les variables hôtes de stockage des valeurs récupérées.

Par exemple :

```
/*
 * Soit la table :
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

La clause `INTO` apparaît donc entre la liste de sélection et la clause `FROM`. Le nombre d'éléments dans la liste du **select** et dans la liste qui suit `INTO` (aussi appelée liste cible) doivent être identiques.

Exemple utilisant la commande **FETCH** :

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do {
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Ici, la clause `INTO` apparaît après toutes les autres clauses.

33.6.5. Indicateurs

Les exemples ci-dessus ne gèrent pas les valeurs NULL. En fait, ces exemples de récupération affichent une erreur s'ils récupèrent une valeur NULL à partir de la base de données. Pour être capable de passer des valeurs NULL à la base de données ou de récupérer des valeurs NULL de la base de données, il est nécessaire d'ajouter une deuxième spécification de variable hôte pour chaque variable hôte contenant des données. Cette seconde variable est appelée l'*indicateur* et contient un drapeau indiquant si la valeur est NULL, auquel cas la valeur de la variable hôte réelle est ignorée. Exemple qui gère correctement la récupération de valeurs NULL :

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

La variable indicateur `val_ind` vaut zéro si la valeur n'est pas nulle, elle est négative dans le cas contraire.

L'indicateur a une autre fonction : si la valeur de l'indicateur est positive, cela signifie que la valeur n'est pas nulle mais qu'elle a été tronquée lors de son stockage dans la variable hôte.

33.7. SQL dynamique

Dans de nombreux cas, les instructions SQL particulières qu'une application doit exécuter sont connues au moment de l'écriture de l'application. Néanmoins, dans certains cas, les instructions SQL sont composées à l'exécution ou fournies par une source externe. Dans ce cas, il n'est pas possible d'embarquer directement les instructions SQL dans le code source C, mais il existe une fonctionnalité permettant d'appeler des instructions SQL arbitraires fournies par l'intermédiaire d'une variable de type chaîne.

La façon la plus simple d'exécuter une instruction SQL arbitraire est d'utiliser la commande **EXECUTE IMMEDIATE**. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

Les instructions de ce type ne peuvent pas être utilisées pour récupérer des données (c'est-à-dire un **SELECT**).

Une façon plus puissante d'exécuter des instructions SQL arbitraires est de les préparer une seule fois et de les exécuter ensuite aussi souvent que nécessaire. Il est également possible de préparer une version généralisée d'une instruction, puis d'exécuter les versions spécifiques en substituant les paramètres. Lors de la préparation de l'instruction, il suffit d'écrire des points d'interrogation aux endroits où des paramètres seront substitués par la suite. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

Si l'instruction exécutée retourne des valeurs, il est nécessaire d'ajouter une clause `INTO` :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO v1, v2, v3 USING 37;
```

Une commande **EXECUTE** peut avoir une clause `INTO`, une clause `USING`, les deux ou aucune.

Lorsqu'une instruction préparée n'est plus utile, il est préférable de la désallouer :

```
EXEC SQL DEALLOCATE PREPARE name;
```

33.8. Bibliothèque pgtypes

La bibliothèque pgtypes établit une correspondance entre les types PostgreSQL™ et les équivalents en C. Elle fournit aussi des fonctions permettant des calculs simples sur ces types en C, c'est-à-dire sans l'aide du serveur PostgreSQL™. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
free(out);
```

33.8.1. Le type numeric

Le type numeric permet des calculs de précision arbitraire. Voir Section 8.1, « Types numériques » pour le type équivalent dans le serveur PostgreSQL™. Du fait de la précision arbitraire, une variable de ce type doit pouvoir étendre et diminuer sa taille dynamiquement. C'est pourquoi il est obligatoire d'utiliser les fonctions `PGTYPESnumeric_new` et `PGTYPESnumeric_free` pour créer des variables numériques, et uniquement en mémoire *heap*. Le type decimal, similaire mais limité en précision, peut être créé sur la pile comme sur le « heap ».

Les fonctions suivantes peuvent être utilisées avec le type numeric :

`PGTYPESnumeric_new`
: réclame un pointeur vers une variable de type numeric nouvellement allouée.

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`
: libère un type numeric en vidant toute sa mémoire.

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`
: analyse un type numeric à partir de sa notation en chaîne.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Les formats valides sont, par exemple : `-2`, `.794`, `+3.44`, `592.49E07` ou `-32.84e-4`. Si la valeur a pu être analysée correctement, un pointeur valide est renvoyé. Dans le cas contraire, il s'agit d'un pointeur NULL. Actuellement, ECPG analyse toujours la chaîne complète et, du coup, ne supporte pas le stockage de l'adresse du premier caractère invalide dans `*endptr`. `endptr` peut être initialisé à NULL en toute sécurité.

`PGTYPESnumeric_to_asc`
: renvoie un pointeur vers une chaîne allouée via `malloc` qui contient la représentation en chaîne du type numeric `num`.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

La valeur numerique est affichée avec `dscale` chiffres décimaux, arrondie si nécessaire.

`PGTYPESnumeric_add`
: additionne deux variables de type numeric dans une troisième.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

La fonction additionne les variables `var1` et `var2` dans la variable résultat `result`. Elle renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_sub`
: soustrait deux variables de type numeric et renvoie le résultat dans une troisième.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

La fonction soustrait la variable `var2` à la variable `var1`. Le résultat de cette opération est stocké dans la variable `result`.

La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_mul`
multiplie deux variables de type numeric et renvoie le résultat dans une troisième.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

La fonction multiplie les variables `var1` et `var2`. Le résultat de cette opération est stocké dans la variable `result`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_div`
divise deux variables de type numeric et renvoie le résultat dans une troisième.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

La fonction divise la variable `var1` par `var2`. Le résultat de cette opération est stocké dans la variable `result`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_cmp`
compare deux variables de type numeric.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

La fonction compare deux variables de type numeric. En cas d'erreur, `INT_MAX` est renvoyé. En cas de succès, la fonction renvoie un des trois résultats possibles :

- 1, si `var1` est plus grand que `var2`
- -1, si `var1` est plus petit que `var2`
- 0, si `var1` est égal à `var2`

`PGTYPESnumeric_from_int`
convertit une variable de type int en une variable de type numeric.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

La fonction accepte une variable de type signed int (entier signé) et la stocke dans la variable `var` de type numeric. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_from_long`
convertit une variable de type long int en une variable de type numeric.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

La fonction accepte une variable de type signed long int (entier long signé) et la stocke dans la variable `var` de type numeric. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_copy`
copie une variable de type numeric dans une autre.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

la fonction copie la valeur de la variable sur laquelle pointe `src` dans la variable sur laquelle pointe `dst`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_from_double`
convertit une variable de type double en numeric.

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

La fonction accepte une variable de type double et la stocke dans la variable sur laquelle pointe `dst`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_to_double`
convertit une variable de type numeric en double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

La fonction convertit la valeur numérique de la variable pointée par `nv` dans la variable de type double pointée par `dp`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur, dépassement inclus. En cas de dépassement, la variable globale `errno` est positionnée à `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_int`
convertit une variable de type numeric en int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

La fonction convertit la valeur numérique de la variable pointée par `nv` en une variable de type integer pointée par `ip`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur, dépassement inclus. En cas de dépassement, la variable globale `errno` est initialisée à `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_long`
convertit une variable de type numeric en long int.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

La fonction convertit la valeur numérique de la variable pointée par `nv` en une variable de type long integer pointée par `lp`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur, dépassement inclus. En cas de dépassement, la variable globale `errno` est initialisée à `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_decimal`
convertit une variable de type numeric en decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

La fonction convertit la valeur numérique de la variable pointée par `nv` en une variable de type decimal pointée par `dst`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur, dépassement inclus. En cas de débordement, la variable globale `errno` est initialisée à `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_from_decimal`
convertit une variable de type decimal en numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

La fonction convertit la valeur décimale de la variable pointée par `nv` en une variable de type numeric pointée par `dst`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur. Comme le type decimal est une version limitée du type numeric, aucun dépassement ne peut survenir avec cette conversion.

33.8.2. Le type date

Le type date en C permet aux programmes de gérer les données de type date SQL. Voir Section 8.5, « Types date/heure » pour le type équivalent dans le serveur PostgreSQL™.

Les fonctions qui suivent peuvent être utilisées pour travailler avec le type date :

`PGTYPESdate_from_timestamp`
extrait la date d'une variable de type timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

La fonction reçoit une variable de type timestamp comme seul argument et en retourne la partie date.

`PGTYPESdate_from_asc`
analyse une date à partir de sa représentation textuelle.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

La fonction reçoit une chaîne C (char*) `str` et un pointeur vers une autre chaîne C, `endptr`. À ce jour, ECPG analyse toujours la chaîne complète et ne peut donc pas stocker l'adresse du premier caractère invalide dans `*endptr`. `endptr` peut être initialisé à NULL en toute sécurité.

La fonction estime systématiquement recevoir des dates au format MDY. Il n'y a actuellement aucune variable qui permette

de modifier cela dans ECPG.

Tableau 33.1, « Formats de saisie valides pour `PGTYPESdate_from_asc` » affiche les format de saisie autorisés.

Tableau 33.1. Formats de saisie valides pour `PGTYPESdate_from_asc`

Saisie	Résultat
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

`PGTYPESdate_to_asc`
renvoie la représentation textuelle d'une variable date.

```
char *PGTYPESdate_to_asc(date dDate);
```

La fonction reçoit la date `dDate` comme seul paramètre. Elle affiche la date sous la forme `1999-01-18`, c'est-à-dire dans le format `YYYY-MM-DD`.

`PGTYPESdate_julmdy`
extrait les valeurs du jour, du mois et de l'année à partir d'une variable de type date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

La fonction reçoit la date `d` et un pointeur sur un tableau de trois valeurs entières `mdy`. Le nom de la variable indique l'ordre de séquence : `mdy[0]` contient le numéro du mois, `mdy[1]` le jour et `mdy[2]` l'année.

`PGTYPESdate_mdyjul`
crée une valeur date à partir d'un tableau de trois entiers qui précisent le jour, le mois et l'année de la date.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

La fonction prend le tableau de trois entiers (`mdy`) comme premier argument et un pointeur sur une variable de type date comme second argument. Ce dernier reçoit le résultat de l'opération.

`PGTYPESdate_dayofweek`
renvoie un nombre représentant le jour de la semaine pour une valeur de type date.

```
int PGTYPESdate_dayofweek(date d);
```

La fonction reçoit la variable `d` de type date comme seul argument et renvoie un entier indiquant le jour de la semaine associé à cette date.

- 0 - Dimanche
- 1 - Lundi
- 2 - Mardi
- 3 - Mercredi
- 4 - Jeudi
- 5 - Vendredi
- 6 - Samedi

`PGTYPESdate_today`
récupère la date courante.

```
void PGTYPESdate_today(date *d);
```

La fonction reçoit un pointeur vers une variable date (`d`) et l'initialise à la date couratne.

`PGTYPESdate_fmt_asc`
convertit une variable de type date en sa représentation textuelle à l'aide d'un masque de format.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

La fonction reçoit la date à convertir (`dDate`), le masque de format (`fmtstring`) et la chaîne de récupération de la représentation textuelle de la date (`outbuf`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

Les chaînes suivantes indiquent les champs à utiliser :

- `dd` - le numéro du jour dans le mois ;
- `mm` - le numéro du mois dans l'année ;
- `yy` - le numéro de l'année sur deux chiffres ;
- `yyyy` - le numéro de l'année sur quatre chiffres ;
- `ddd` - l'abréviation du nom du jour ;
- `mmm` - l'abréviation du nom du mois.

Tous les autres caractères sont copiés un à un dans la chaîne de sortie.

Tableau 33.2, « Formats valides en entrée de `PGTYPESdate_fmt_asc` » indique quelques formats possibles. Ceci vous donnera une idée de la façon d'utiliser cette fonction. Toutes les lignes en sortie sont basées sur la même date : 23 novembre 1959.

Tableau 33.2. Formats valides en entrée de `PGTYPESdate_fmt_asc`

Format	Résultat
mmddyy	112359
ddmmyy	231159
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959

Format	Résultat
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

`PGTYPESdate_defmt_asc`
utilise un masque de format pour convertir une chaîne C (char*) en une valeur de type date.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

Cette fonction prend en argumant un pointeur sur la valeur de type date de stockage du résultat de l'opération (`d`), le masque du format à utiliser pour analyser la date (`fmt`) et la chaîne C qui contient la représentation textuelle de la date (`str`). La représentation textuelle doit correspondre au masque de format. Il n'est toutefois pas nécessaire d'avoir une correspondance exacte. La fonction analyse simplement l'ordre séquentiel et cherche les libellés `yy` ou `yyyy` qui indiquent la position de l'année, `mm` pour celle du mois et `dd` pour celle du jour.

Tableau 33.3, « Formats valides en entrée de `rdefmtdate` » indique les quelques formats possibles. Ceci vous donnera une idée de la façon d'utiliser ces fonctions.

Tableau 33.3. Formats valides en entrée de `rdefmtdate`

Format	Chaîne
ddmmyy	21-2-54
ddmmyy	2-12-54
ddmmyy	20111954
ddmmyy	130464
mmm.dd.yyyy	MAR-12-1967
yy/mm/dd	1954, February 3rd
mmm.dd.yyyy	041269
yy/mm/dd	En l'an 2525, au mois de juillet, on trouvera bien quelqu'un de vivant le 28ème jour (In the year 2525, in the month of July, mankind will be alive on the 28th day)
dd-mm-yy	J'ai dit le 28 juillet de l'an 2525 (I said on the 28th of July in the year 2525)
mmm.dd.yyyy	9/14/58
yy/mm/dd	47/03/29
mmm.dd.yyyy	oct 28 1975
mmddyy	Nov 14th, 1985

33.8.3. Le type timestamp

Le type timestamp en C permet aux programmes de gérer les données de type timestamp SQL. Voir Section 8.5, « Types date/heure » pour le type équivalent dans le serveur PostgreSQL™.

Le fonctions qui suivent sont utilisées pour travailler avec le type timestamp :

`PGTYPEStimestamp_from_asc`
analyse une variable de type timestamp à partir de sa représentation textuelle et la stocke dans une variable de type timestamp.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

La fonction prend en arguments la chaîne à analyser (`str`) et un pointeur sur une chaîne C de type char* (`endptr`). Actuellement, ECPG analyse toujours la chaîne complète et ne permet pas de stocker l'adresse du premier caractère invalide dans `*endptr`. `endptr` peut être initialisé à NULL en toute sécurité.

La fonction renvoie la variable de type timestamp analysée en cas de succès. En cas d'erreur, `PGTYPESInvalidTimes-`

`tamp` est renvoyée et `errno` vaut `PGTYPES_TS_BAD_TIMESTAMP`. Voir `PGTYPESInvalidTimestamp` pour d'importantes informations sur cette valeur.

En général, la chaîne saisie peut contenir toute combinaison de spécification autorisée de date, d'espace et de spécification autorisée d'heure. Les fuseaux horaires ne sont pas supportés par ECPG. Il peut les analyser mais n'applique aucun calcul au contraire du serveur PostgreSQL™ par exemple. Les fuseaux horaires sont ignorés silencieusement.

Tableau 33.4, « Formats de saisie valides pour `PGTYPEStimestamp_from_asc` » contient quelques exemples pour les chaînes en entrée.

Tableau 33.4. Formats de saisie valides pour `PGTYPEStimestamp_from_asc`

Saisie	Résultat
`1999-01-08 04:05:06`	`1999-01-08 04:05:06`
`January 8 04:05:06 1999 PST`	`1999-01-08 04:05:06`
`1999-Jan-08 04:05:06.789-8`	`1999-01-08 04:05:06.789 (fuseau horaire ignoré)`
`J2451187 04:05-08:00`	`1999-01-08 04:05:00 (fuseau horaire ignoré)`

`PGTYPEStimestamp_to_asc`
convertit une date en chaîne C char*.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

La fonction prend comme seul argument la variable `tstamp` de type timestamp et renvoie une chaîne allouée qui contient la représentation textuelle du timestamp.

`PGTYPEStimestamp_current`
récupère le timestamp courant.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

La fonction récupère le timestamp courant et le sauvegarde dans la variable de type timestamp pointée par `ts`.

`PGTYPEStimestamp_fmt_asc`
convertit une variable de type timestamp en une chaîne C de type char* à l'aide d'un masque de formatage.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char
*fmtstr);
```

La fonction prend en arguments un pointeur vers la variable de type timestamp (`ts`), un pointeur vers le tampon de sortie (`output`), la longueur maximale allouée pour le tampon de sortie (`str_len`) et le masque de formatage à utiliser pour la conversion (`fmtstr`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

Les caractères de formatage qui suivent peuvent être utilisés comme masque de formatage. Ces caractères sont identiques à ceux utilisés dans la fonction `strftime` de libc™. Tout caractère n'en faisant pas partie est copié tel quel dans le tampon de sortie.

- `%A` - remplacé par la représentation nationale du nom complet du jour de la semaine ;
- `%a` - remplacé par la représentation nationale de l'abréviation du nom de la semaine ;
- `%B` - remplacé par la représentation nationale du nom complet du mois ;
- `%b` - remplacé par la représentation nationale de l'abréviation du nom du mois ;
- `%C` - remplacé par (année / 100) en nombre décimal, les chiffres seuls sont précédés d'un zéro ;
- `%c` - remplacé par la représentation nationale de l'heure et de la date ;
- `%D` - est équivalent à `%m/%d/%y`.
- `%d` - est remplacé par le jour du mois en nombre décimal (01-31) ;

- `%E*` `%O*` - extensions POSIX de la locale. Les séquences `%Ec` `%EC` `%Ex` `%EX` `%Ey` `%EY` `%Od` `%Oe` `%OH` `%OI` `%Om` `%OM` `%OS` `%Ou` `%OU` `%OV` `%Ow` `%OW` `%Oy` sont supposées fournir d'autres représentations.

  De plus, `%OB` permet de représenter les autres noms des mois (utilisés en autonome, sans mention du jour) ;
- `%e` - est remplacé par le jour du mois en nombre décimal (1-31) ; les chiffres seuls sont précédés d'un zéro ;
- `%F` - est équivalent à `%Y-%m-%d` ;
- `%G` - est remplacé par une année comme nombre décimal avec le siècle. Il s'agit de l'année qui contient la plus grande partie de la semaine (lundi étant le premier jour de la semaine) ;
- `%g` - est remplacé par la même année que `%G` mais en décimal sans le siècle (00-99) ;
- `%H` - est remplacé par l'heure (sur 24 heures) en nombre décimal (00-23) ;
- `%h` - idem à `%b` ;
- `%I` - est remplacé par l'heure (sur 12 heures) en nombre décimal (01-12) ;
- `%j` - est remplacé par le jour de l'année en nombre décimal (001-366) ;
- `%k` - est remplacé par l'heure (sur 24 heures) en nombre décimal (0-23), les chiffres seuls sont précédés d'un zéro ;
- `%l` - est remplacé par l'heure (sur 12 heures) en nombre décimal (1-12), les chiffres seuls sont précédés d'un zéro ;
- `%M` - est remplacé par la minute en nombre décimal (00-59) ;
- `%m` - est remplacé par le mois en nombre décimal (01-12) ;
- `%n` - est remplacé par un retour chariot ;
- `%O*` - identique à `%E*` ;
- `%p` - remplacé par la représentation nationale de « ante meridiem » ou « post meridiem » suivant le cas ;
- `%R` - est équivalent à `%H:%M` ;
- `%r` - est équivalent à `%I:%M:%S %p` ;
- `%S` - est remplacé par la seconde en nombre décimal (00-60) ;
- `%s` - est remplacé par le nombre de secondes depuis Epoch, UTC ;
- `%T` - est équivalent à `%H:%M:%S` ;
- `%t` - est remplacé par une tabulation ;
- `%U` - est remplacé par le numéro de la semaine dans l'année (dimanche étant le premier jour de l'année) en nombre décimal (00-53) ;
- `%u` - est remplacé par le jour de la semaine (lundi étant le premier jour de la semaine) en nombre décimal (1-7) ;
- `%V` - est remplacé par le numéro de la semaine dans l'année (lundi étant le premier jour de la semaine) en nombre décimal (01-53). Si la semaine contenant le 1er janvier a quatre jours ou plus dans la nouvelle année, alors c'est la semaine 1 ; sinon c'est la dernière semaine de l'année précédente et la semaine suivante est la semaine 1 ;
- `%v` - est équivalent à `%e-%b-%Y` ;
- `%W` - est remplacé par le numéro de semaine de l'année (lundi étant le premier jour de la semaine) en nombre décimal (00-53) ;
- `%w` - est remplacé par le jour de la semaine (dimanche étant le premier jour de la semaine) en nombre décimal (0-6) ;
- `%X` - est remplacé par la représentation nationale de l'heure ;
- `%x` - est remplacé par la représentation nationale de la date ;
- `%Y` - est remplacé par l'année avec le siècle en nombre décimal ;
- `%y` - est remplacé par l'année sans le siècle en nombre décimal (00-99) ;
- `%Z` - est remplacé par le nom du fuseau horaire ;
- `%z` - est remplacé par le décalage du fuseau horaire à partir d'UTC ; un signe plus au début signifie à l'est d'UTC, un signe moins signifie à l'ouest d'UTC ; les heures et les minutes suivent avec deux chiffres chacun et aucun délimiteur entre eux (format habituel pour les en-têtes de date au standard RFC 822) ;

- `%+` - est remplacé par la représentation nationale de la date et l'heure ;
- `%-*` - extension GNU libc. N'ajoute aucune espace de remplissage lors du traitement des sorties numériques ;
- `$_*` - extension GNU libc. Indique explicitement l'espace comme caractère de remplissage ;
- `%0*` - extension GNU libc. Indique explicitement zéro comme caractère de remplissage ;
- `%%` - est remplacé par `%`.

`PGTYPESTimestamp_sub`
: soustrait une valeur de type timestamp à une autre valeur et sauvegarde le résultat dans une variable de type interval.

```
int PGTYPESTimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

La fonction soustrait la variable timestamp pointée par `ts2` de la variable timestamp pointée par `ts1` et stocke le résultat dans la variable de type interval pointée par `iv`.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`PGTYPESTimestamp_defmt_asc`
: analyse une valeur de type timestamp à partir de sa représentation textuelle à l'aide d'un masque de formatage.

```
int PGTYPESTimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

La fonction reçoit la représentation textuelle d'une valeur de type timestamp dans la variable `str` ainsi que le masque de formatage à utiliser dans la variable `fmt`. Le résultat est stocké dans la variable pointée par `d`.

Si le masque de formatage `fmt` est NULL, la fonction utilise la masque de formatage par défaut qui est `%Y-%m-%d %H:%M:%S`.

C'est la fonction inverse de `PGTYPESTimestamp_fmt_asc`. Voir la documentation pour découvrir les saisies possibles pour le masque de formatage.

`PGTYPESTimestamp_add_interval`
: ajoute une variable de type interval à une variable de type timestamp.

```
int PGTYPESTimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

La fonction recçoit un pointeur vers une variable de type timestamp, `tin`, et un pointeur vers une variable de type interval `span`. Elle ajoute l'interval au timestamp et sauvegarde le résultat, de type timestamp, dans la variable pointée par `tout`.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`PGTYPESTimestamp_sub_interval`
: soustrait une variable de type interval à une variable de type timestamp.

```
int PGTYPESTimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

La fonction soustrait la variable de type interval pointée par `span` à la variable timestamp pointée par `tin` et sauvegarde le résultat dans une variable pointée par `tout`.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

33.8.4. Type interval

Le type interval en C permet aux programmes de gérer des données du type SQL interval. Voir Section 8.5, « Types date/heure » pour le type équivalent dans le serveur PostgreSQL™.

Les fonctions qui suivent peuvent être utilisées pour travailler avec le type interval :

`PGTYPESinterval_new`
: renvoie un pointeur vers une variable interval nouvellement allouée.

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`
: libère la mémoire d'une variable interval précédemment allouée.

```
void PGTYPESinterval_new(interval *intvl);
```

`PGTYPESinterval_from_asc`
analyse une variable de type interval à partir de sa représentation textuelle.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

La fonction analyse la chaîne `str` en entrée et renvoie un pointeur vers une variable allouée de type interval. Actuellement, ECPG analyse toujours la chaîne complète et ne supporte donc pas le stockage de l'adresse du premier caractère invalide dans `*endptr`. `endptr` peut ainsi être initialisé à NULL en toute sécurité.

`PGTYPESinterval_to_asc`
convertit une variable de type interval en sa représentation textuelle.

```
char *PGTYPESinterval_to_asc(interval *span);
```

La fonction convertit la variable de type interval pointée par `span` en une chaîne C de type char*. Son contenu ressemble à : `@ 1 day 12 hours 59 mins 10 secs`.

`PGTYPESinterval_copy`
copie une variable de type interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

La fonction copie la variable de type interval pointée par `intvlsrc` dans la variable pointée par `intvldest`. La mémoire de la variable de destination doit être allouée avant d'appeler la fonction.

33.8.5. Type decimal

Le type decimal est similaire au type numeric. Néanmoins, il est limité à une précision maximale de 30 chiffres significatifs. Contrairement au type numeric qui ne peut être créé que sur la partie *heap* de la mémoire, le type decimal peut être créé soit sur la pile soit sur le *heap* (au moyen des fonctions `PGTYPESdecimal_new()` et `PGTYPESdecimal_free()`). Il existe d'autres fonctions qui gèrent le type decimal dans le mode de compatibilité Informix™ décrit dans Section 33.10, « Mode de compatibilité Informix™ ».

Les fonctions qui suivent sont utilisées pour travailler avec le type decimal et ne sont pas uniquement contenues dans la bibliothèque `libcompat`.

`PGTYPESdecimal_new`
réclame un pointeur vers une variable de type decimal nouvellement allouée.

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`
libère toute la mémoire d'un type decimal.

```
void PGTYPESdecimal_free(decimal *var);
```

33.8.6. Valeurs errno de pgtypeslib

`PGTYPES_NUM_BAD_NUMERIC`
Un argument doit contenir une variable de type numeric (ou pointer sur une variable numeric) mais, en fait, sa représentation en mémoire est invalide.

`PGTYPES_NUM_UNDERFLOW`
Un dépassement (*overflow*) est survenu. Comme le type numeric peut gérer une précision quasiment arbitraire, transtyper une variable de type numeric peut causer un dépassement.

`PGTYPES_NUM_OVERFLOW`
Un soupassement (*underflow*, erreur de traitement survenant quand la valeur absolue d'une quantité calculée est plus petite que les limites de précision) est survenu. Comme le type numeric peut gérer une précision quasiment arbitraire, transtyper une

variable de type numeric peut causer un soupassement.

`PGTYPES_NUM_DIVIDE_ZERO`
Tentative de division par zéro.

`PGTYPES_DATE_BAD_DATE`

`PGTYPES_DATE_ERR_EARGS`

`PGTYPES_DATE_ERR_ENOSHORTDATE`

`PGTYPES_INTVL_BAD_INTERVAL`

`PGTYPES_DATE_ERR_ENOTDMY`

`PGTYPES_DATE_BAD_DAY`

`PGTYPES_DATE_BAD_MONTH`

`PGTYPES_TS_BAD_TIMESTAMP`

33.8.7. Constantes spéciales de pgtypeslib

`PGTYPESInvalidTimestamp`
Une valeur de type timestamp qui représente un timestamp invalide. Ce code est renvoyé par la fonction `PGTYPEStimestamp_from_asc` en cas d'erreur lors de l'analyse. Du fait de la représentation interne du type de données timestamp, `PGTYPESInvalidTimestamp` est aussi en même temps un timestamp valide. Il est configuré à `1899-12-31 23:59:59`. Pour détecter les erreurs, il est impératif de s'assurer que l'application ne teste pas seulement `PGTYPESInvalidTimestamp` mais aussi `errno != 0` après chaque appel à `PGTYPEStimestamp_from_asc`.

33.9. Utiliser les aires de descripteur

Une aire de descripteur SQL est une méthode plus sophistiquée pour traiter les résultats d'une instruction **SELECT**, **FETCH** ou **DESCRIBE**. Une aire de descripteur SQL groupe les données d'une ligne avec les éléments des méta-données en une seule structure de données. Les méta-données sont particulièrement utiles lors de l'exécution de requêtes SQL dynamiques, où la nature des colonnes résultantes pourrait ne pas être connue à l'avance. PostgreSQL fournit deux moyens pour utiliser les aires de descripteur : les aires nommées de descripteur SQL et la structure C SQLDA.

33.9.1. Aires nommées de descripteur SQL

Une aire nommée de descripteur SQL consiste en un en-tête contenant des informations concernant le descripteur entier et un ou plusieurs éléments aires de descripteur, chacun décrivant une colonne de la ligne de résultat.

Avant de pouvoir utiliser une aire de descripteur SQL, vous devez en allouer une :

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

L'identifiant sert en tant que « nom de variable » de l'aire de descripteur. Quand vous n'avez plus besoin du descripteur, vous pouvez le libérer :

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

Pour utiliser une aire de descripteur, spécifiez-la en tant que cible de stockage dans une clause `INTO` au lieu de liter les variables hôtes :

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

Si l'ensemble de résultats est vide, l'aire de descripeteur contiendra malgré tout les méta-données de la requête, autrement dit les noms des champs.

Pour des requêtes préparées pas encore exécutées, l'instruction **DESCRIBE** peut être utilisée pour obtenir les méta-données sur l'ensemble de résultats :

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
```

```
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

Avant PostgreSQL 9.0, le mot-clé `SQL` était optionnel, donc l'utilisation de `DESCRIPTOR` et `SQL DESCRIPTOR` produisaient des aires nommées de descripteur SQL. Maintenant, le mot-clé est obligatoire. Omettre ce mot-clé produit des aires de descripteur SQLDA, voir Section 33.9.2, « Aires de descripteur SQLDA ».

Dans les instructions **DESCRIBE** et **FETCH**, les mot-clés `INTO` et `USING` peuvent être utilisés de façon similaire : l'ensemble de résultat et les méta-données sont placés dans une aire de descripteur.

Maintenant, comment sortir les données de l'aire de descripteur ? l'aire des descripteur est en quelque sorte une structure avec des champs nommés. Pour récupérer la valeur d'un champ à partir d'un en-tête et pour stocker cette valeur dans une variable hôte, utilisez la commande suivante :

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

Actuellement, il existe un seul champ d'en-tête défini : *`COUNT`*, qui indique le nombre d'éléments existant dans l'aire de descripteur (autrement dit, le nombre de colonnes contenues dans le résultat). La variable hôte doit être de type entier. Pour obtenir un champ à partir de cet élément aire de descripteur, utilisez la commande suivante :

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

`num` peut être un entier constant ou une variable hôte contenant un entier. Les champs possibles sont :

`CARDINALITY` (integer)
: nombre de lignes dans l'ensemble de résultat

`DATA`
: élément de données actuel (du coup, le type de données de ce champ dépend de la requête)

`DATETIME_INTERVAL_CODE` (integer)
: ?

`DATETIME_INTERVAL_PRECISION` (integer)
: non implémenté

`INDICATOR` (integer)
: un indicateur précisant une valeur NULL ou un tronquage de la valeur

`KEY_MEMBER` (integer)
: non implémenté

`LENGTH` (integer)
: longueur du champ en caractères

`NAME` (string)
: nom de la colonne

`NULLABLE` (integer)
: non implémenté

`OCTET_LENGTH` (integer)
: longueur de la représentation en caractères du datum en octets

`PRECISION` (integer)
: précision (pour le type numeric)

`RETURNED_LENGTH` (integer)
: longueur du datum en caractères

`RETURNED_OCTET_LENGTH` (integer)
: longueur de la représentation en caractères du datum en octets

`SCALE` (integer)
: échelle (pour le type numeric)

`TYPE` (integer)
: code numérique du type de données de la colonne

Dans les instructions **EXECUTE**, **DECLARE** et **OPEN**, l'effet des mot-clés `INTO` et `USING` est différent. Une aire de descripteur peut aussi être construite manuellement pour fournir les paramètres en entrée pour une requête ou un curseur. `USING SQL DESCRIPTOR` *`name`* est la façon de passer les paramètres en entrée pour une requête avec paramètres. L'instruction pour construire une aire nommée de descripteur SQL se trouve ci-dessous :

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQL accepte de récupérer plus d'un enregistrement avec une instruction **FETCH**. Stocker les données dans des variables hôtes dans ce cas suppose que la variable est un tableau. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

33.9.2. Aires de descripteur SQLDA

Une aire de descripteur SQLDA est une structure C qui peut aussi être utilisée pour obtenir l'ensemble de résultats et les méta-données d'une requête. Une structure conserve un enregistrement de l'ensemble de résultat.

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

Notez que le mot-clé `SQL` est omis. Les paragraphes sur les cas d'utilisation des mot-clés `INTO` et `USING` dans Section 33.9.1, « Aires nommées de descripteur SQL » s'appliquent aussi ici avec un ajout. Dans une instruction **DESCRIBE**, le mot-clé `DESCRIPTOR` peut être complètement omis si le mot-clé `INTO` est utilisé :

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

La structure de SQLDA est :

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};

struct sqlvar_struct
{
        short           sqltype;
        short           sqllen;
        char       *sqldata;
        short      *sqlind;
        struct sqlname sqlname;
};

struct sqlda_struct
{
        char            sqldaid[8];
        long            sqldabc;
        short           sqln;
        short           sqld;
        struct sqlda_struct *desc_next;
        struct sqlvar_struct    sqlvar[1];
};

typedef struct sqlvar_struct    sqlvar_t;
```

```
typedef struct sqlda_struct     sqlda_t;
```

Les données allouées pour une structure SQLDA sont variables car elles dépendent du nombre de champs dans l'ensemble du résultat et dépendent aussi de la longueur des valeurs des données de type chaîne de caractères dans un enregistrement. Les champs individuels de la structure `SQLDA` sont :

`sqldaid`
: Il contient la chaîne constante "`SQLDA  `".

`sqldabc`
: Il contient la taille de l'espace alloué en octets.

`sqln`
: Il contient le nombre de paramètres en entrée pour une requête avec paramètres lorsqu'elle est passée via **OPEN**, **DECLARE** ou **EXECUTE** en utilisant le mot-clé `USING`. Dans le cas où elle est passée en sortie des instructions **SELECT**, **EXECUTE** ou **FETCH**, sa valeur est identique à celle de l'instruction `sqld`.

`sqld`
: Il contient le nombre de champs dans l'ensemble de résultats.

`desc_next`
: Si la requête renvoie plus d'un enregistrement, plusieurs structures SQLDA liées sont renvoyées et `desc_next` contient un pointeur vers la prochaine entrée dans la liste.

`sqlvar`
: C'est le tableau de champs de l'ensemble des résultats. Les champs disponibles sont les suivants :

  `sqltype`
  Il contient l'identifiant du type pour le champ. Pour les valeurs, voir `enum ECPGttype` dans `ecpgtype.h`.

  `sqllen`
  Il contient la longueur binaire du champ. Par exemple, quatre octets pour ECPGt_int.

  `sqldata`
  `(char *)sqldata` pointe vers les données.

  `sqlind`
  `(char *)sqlind` pointe vers l'indicateur NULL pour les données. 0 signifie NOT NULL, -1 signifie NULL.

  `sqlname`
  `struct sqlname sqlname` contient le nom d'un champ dans une structure :

  ```
  struct sqlname
  {
          short           length;
          char            data[NAMEDATALEN];
  };
  ```

  `length`
  `sqlname.length` contient la longueur du nom du champ.

  `data`
  `sqlname.data` contient le nom réel du champ.

33.10. Mode de compatibilité Informix™

ecpg peut fonctionner en *mode compatible avec Informix*. Dans ce mode, ecpg tente de se comporter comme le précompilateur Informix™ pour Informix™ E/SQL. Dans les grandes lignes, cela permet d'utiliser le signe dollar à la place de la primitive `EXEC SQL` pour introduire des commandes SQL embarquées :

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

Il existe deux modes de compatibilité : `INFORMIX` et `INFORMIX_SE`.

Lors de l'édition de liens pour des programmes qui utilisent ce mode de compatibilité, il est nécessaire de lier les programmes avec `libcompat`, fournie avec ECPG.

En dehors de la syntaxe expliquée précédemment, le mode de compatibilité Informix™ ajoute à ECPG quelques fonctions pour la saisie, la sortie et la transformation de données ainsi que des instructions SQL embarquées connues d'E/SQL.

Le mode de compatibilité Informix™ est très fortement connecté à la bibliothèque pgtypeslib d'ECPG. pgtypeslib fait correspondre les types de données SQL aux types de données internes du programme C hôte ; la plupart des fonctions supplémentaires du mode de compatibilité Informix™ permettent d'opérer sur ces types. L'étendue de la compatibilité est toutefois limitée. L'idée n'est pas de mimer le comportement d'Informix™ ; cela permet de réaliser plus ou moins les mêmes opérations et fournit des fonctions de même nom et de même comportement de base, mais il n'est pas question de replacer au pied levé une utilisation d'Informix™. De plus, certains types de données sont différents. Ainsi, les types datetime et interval de PostgreSQL™ ne gèrent pas les échelles telles que `YEAR TO MINUTE`, par exemple ; ECPG ne les supporte donc pas non plus.

33.10.1. Types supplémentaires

Le pseudo-type « string » d'Informix pour stocker des données de type chaîne de caractères coupée à droite est maintenant supporté dans le mode Informix sans utiliser `typedef`. En fait, dans le mode Informix, ECPG refuse de traiter les fichiers sources contenant `typedef sometype string;`

```
    EXEC SQL BEGIN DECLARE SECTION;
    string userid; /* cette variable contiendra des données coupées */
    EXEC SQL END DECLARE SECTION;

    EXEC SQL FETCH MYCUR INTO :userid;
```

33.10.2. Instructions supplémentaires/manquantes du SQL embarqué

`CLOSE DATABASE`
: ferme la connexion en cours. En fait, il s'agit d'un synonyme de la commande ECPG `DISCONNECT CURRENT` :

```
    $CLOSE DATABASE;                /* ferme la connexion actuelle */
    EXEC SQL CLOSE DATABASE;
```

`FREE cursor_name`
: Dû aux différences de fonctionnement entre ECPG et ESQL/C d'Informix (i.e. which steps are purely grammar transformations and which steps rely on the underlying runtime library) there is no `FREE cursor_name` statement in ECPG. This is because in ECPG, `DECLARE CURSOR` doesn't translate to a function call into the runtime library that uses to the cursor name. This means that there's no runtime bookkeeping of SQL cursors in the ECPG runtime library, only in the PostgreSQL server.

`FREE statement_name`
: `FREE statement_name` est un synonyme pour `DEALLOCATE PREPARE statement_name`.

33.10.3. Aires de descripteurs SQLDA compatibles Informix

Le mode compatible Informix accepte une structure différente de celle décrite dans Section 33.9.2, « Aires de descripteur SQLDA ». Voir ci-dessous :

```
struct sqlvar_compat
{
        short   sqltype;
        int     sqllen;
        char       *sqldata;
        short      *sqlind;
        char       *sqlname;
        char       *sqlformat;
        short   sqlitype;
        short   sqlilen;
        char       *sqlidata;
        int     sqlxid;
        char       *sqltypename;
        short   sqltypelen;
```

```
        short   sqlownerlen;
        short   sqlsourcetype;
        char       *sqlownername;
        int     sqlsourceid;

        char       *sqlilongdata;
        int     sqlflags;
        void       *sqlreserved;
};

struct sqlda_compat
{
        short           sqld;
        struct sqlvar_compat *sqlvar;
        char            desc_name[19];
        short           desc_occ;
        struct sqlda_compat *desc_next;
        void               *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

Les propriétés globales sont :

`sqld`
: Le nombre de champs compris dans le descripteur `SQLDA`.

`sqlvar`
: Un pointeur vers les propriétés par champ.

`desc_name`
: Non utilisé, rempli d'octets zéro.

`desc_occ`
: Taille de la structure allouée.

`desc_next`
: Pointeur vers la prochaine structure SQLDA si l'ensemble de résultats contient plus d'un enregistrement.

`reserved`
: Pointeur non utilisé, contient NULL. Conservé pour la compatibilité Informix.

Les propriétés par champ sont ci-dessous, stockées dans le tableau `sqlvar` :

`sqltype`
: Type du champ. Les constantes sont dans `sqltypes.h`.

`sqllen`
: Longueur du champ de données.

`sqldata`
: Pointeur vers le champ de données. Le pointeur est de type `char *`, les données pointées sont au format binaire. Par exemple :

```
    int intval;

    switch (sqldata->sqlvar[i].sqltype)
    {
      case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
      ...
    }
```

`sqlind`
: Pointeur vers l'indicateur NULL. Si renvoyé par DESCRIBE ou FETCH, alors il s'agit toujours d'un pointeur valide. Si utilisé en entrée pour `EXECUTE ... USING sqlda;` alors la valeur du pointeur NULL signifie que la valeur pour ce champ est

non NULL. Sinon, c'est un pointeur valide et `sqlitype` doit être correctement initialisé. Par exemple :

```
    if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
      printf("value is NULL\n");
```

`sqlname`
: Nom du champ. Chaîne de caractères terminée par un zéro.

`sqlformat`
: Réservé dans Informix, valeur de `PQfformat()` pour le champ.

`sqlitype`
: Type des données de l'indicateur NULL. C'est toujours SQLSMINT lorsque des données sont renvoyées du serveur. Quand `SQLDA` est utilisé pour une requête avec paramètres, les données sont traitées suivant le tuype de l'ensemble.

`sqlilen`
: Longueur des données de l'indicateur NULL.

`sqlxid`
: Type étendu du champ, résultat de `PQftype()`.

`sqltypename`, `sqltypelen`, `sqlownerlen`, `sqlsourcetype`, `sqlownername`, `sqlsourceid`, `sqlflags`, `sqlreserved`
: Non utilisé.

`sqlilongdata`
: Il est identique à `sqldata` si `sqllen` a une taille supérieure à 32 Ko.

Exemple :

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* ceci n'a pas besoin d'être intégré à un DECLARE SECTION
*/

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
    EXEC SQL WHENEVER NOT FOUND GOTO out;

    while (1)
    {
      EXEC SQL FETCH mycursor USING sqlda;
    }

    EXEC SQL CLOSE mycursor;

    free(sqlda); /* La structure principale doit être libérée avec free(),
                  * sqlda et sqlda->sqlvar sont dans une aire allouée */
```

Pour plus d'informations, voir l'en-tête `sqlda.h` et les tests de régressions `src/interfaces/ecpg/test/compat_informix/sqlda.pgc`.

33.10.4. Fonctions supplémentaires

`decadd`
additionne deux valeurs de type decimal.

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

La fonction reçoit un pointeur vers le premier opérande de type decimal (`arg1`), un pointeur vers le second opérande de type decimal (`arg2`) et un pointeur vers une valeur de type decimal destinée à contenir la somme (`sum`). En cas de succès, la fonction renvoie 0. ECPG_INFORMIX_NUM_OVERFLOW est renvoyé dans le cas d'un dépassement de tampon, et ECPG_INFORMIX_NUM_UNDERFLOW dans le cas d'un soupassement. -1 est renvoyé s'il s'agit d'une autre erreur et `errno` est configuré avec le numéro `errno` respectif dans pgtypeslib.

`deccmp`
compare deux variables de type decimal.

```
int deccmp(decimal *arg1, decimal *arg2);
```

La fonction reçoit un pointeur vers la première valeur de type decimal (`arg1`), un pointeur vers la seconde (`arg2`) et renvoie un entier qui indique la plus grande valeur.

- 1, si la valeur pointée par `arg1` est plus grande que la valeur pointée par `var2` ;
- -1, si la valeur pointée par `arg1` est plus petite que la valeur pointée par `arg2`
- 0, si les valeurs pointées par `arg1` et `arg2` sont identiques.

`deccopy`
copie une valeur de type decimal.

```
void deccopy(decimal *src, decimal *target);
```

La fonction prend en premier argument un pointeur vers la valeur de type decimal à copier (`src`) et un pointeur vers la structure cible de type decimal (`target`) comme second argument.

`deccvasc`
convertit une valeur de sa représentation ASCII en valeur de type decimal.

```
int deccvasc(char *cp, int len, decimal *np);
```

La fonction reçoit un pointeur vers la chaîne contenant le représentation textuelle du nombre à convertir (`cp`) ainsi que sa longueur `len`. `np` est un pointeur vers la valeur de type decimal qui sauvegarde le résultat de l'opération.

Quelques formats valides : `-2`, `.794`, `+3.44`, `592.49E07` ou `-32.84e-4`.

La fonction renvoie 0 en cas de succès. Si un dépassement ou un soupassement survient, `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW`, respectivement, est renvoyé. Si la représentation ASCII ne peut être analysée, `ECPG_INFORMIX_BAD_NUMERIC` est renvoyé (ou `ECPG_INFORMIX_BAD_EXPONENT` si le problème survient lors de l'analyse de l'exposant).

`deccvdbl`
convertit une valeur de type double en une valeur de type decimal.

```
int deccvdbl(double dbl, decimal *np);
```

La fonction prend la variable de type double à convertir en premier argument (`dbl`) et comme second argument (`np`) un pointeur vers la variable décimale amenée à contenir le résultat de l'opération.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`deccvint`
convertit une valeur de type int en une valeur de type decimal.

```
int deccvint(int in, decimal *np);
```

La fonction prend la variable de type int à convertir comme premier argument (`in`) et comme second argument (`np`) un pointeur vers la variable de type decimal aménée à contenir le résultat de l'opération.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`deccvlong`
convertit une valeur de type long en une valeur de type decimal.

```
int deccvlong(long lng, decimal *np);
```

La fonction prend la variable de type long à convertir comme premier argument (`lng`) et comme second argument (`np`) un pointeur vers la variable de type decimal amenée à contenir le résultat de l'opération.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`decdiv`
effectue la division entre deux variables de type decimal.

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

La fonction reçoit des pointeurs vers les variables représentant le premier opérande (`n1`) et le second (`n2`) et calcule `n1`/`n2`. Le résultat (`result`) est un pointeur vers la variable amenée à contenir le résultat de l'opération.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur. Si un dépassement ou un soupassement survient, `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW`, respectivement, est renvoyé. En cas de tentative de division par zéro, la fonction renvoie `ECPG_INFORMIX_DIVIDE_ZERO`.

`decmul`
effectue la multiplication de deux valeurs décimales.

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

La fonction reçoit des pointeurs vers les variables représentant le premier opérande (`n1`) et le second (`n2`) et calcule `n1`/`n2`. Le résultat (`result`) est un pointeur vers la variable amenée à contenir le résultat de l'opération.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur. Si un dépassement ou un soupassement survient, `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW`, respectivement, est renvoyé.

`decsub`
soustrait une valeur de type decimal à une autre.

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

La fonction reçoit des pointeurs vers les variables représentant le premier opérande (`n1`) et le second (`n2`) et calcule `n1-n2`. Le résultat (`result`) est un pointeur vers la variable amenée à contenir le résultat de l'opération.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur. Si un dépassement ou soupassement survient, `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW`, respectivment, est renvoyé.

`dectoasc`
convertit une variable de type decimal en sa représentation ASCII dans une chaîne C char*.

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

La fonction reçoit un pointeur vers une variable de type decimal (`np`) qu'elle convertit dans sa représentation textuelle. `cp` est le tampon de stockage du résultat de l'opération. Le paramètre `right` indique le nombre de chiffres à inclure à droite du point décimal dans le résultat. Le résultat est arrondi à ce nombre de chiffres décimaux. Configurer `right` à -1 indique que tous les chiffres décimaux disponibles doivent être inclus dans la sortie. Si la longueur du tampon de sortie, indiquée par `len`, n'est pas suffisante pour contenir la représentation textuelle, seul le caractère `*` est stocké dans le résultat et -1 est renvoyé.

Cette fonction renvoie -1 si le tampon `cp` est trop petit ou `ECPG_INFORMIX_OUT_OF_MEMORY` si la mémoire disponible est épuisée.

`dectodbl`
convertit une variable de type decimal en double.

```
int dectodbl(decimal *np, double *dblp);
```

La fonction reçoit un pointeur vers la valeur du type decimal à convertir (`np`) et un pointeur vers la variable de type double de stockage du résultat de l'opération (`dblp`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`dectoint`
convertit une variable de type decimal en integer.

```
int dectoint(decimal *np, int *ip);
```

La fonction reçoit un pointeur vers la valeur du type decimal à convertir (`np`) et un pointeur vers la variable de type integer de stockage du résultat de l'opération (`ip`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur. Si un dépassement survient, `ECPG_INFORMIX_NUM_OVERFLOW` est renvoyé.

L'implantation d'ECPG diffère de celle d'Informix™. Informix™ limite un entier à la plage -32767 .. 32767 alors que les limites de l'implantation d'ECPG dépendent de l'architecture (`-INT_MAX .. INT_MAX`).

`dectolong`
convertit une variable de type decimal en long integer.

```
int dectolong(decimal *np, long *lngp);
```

Cette fonction reçoit un pointeur vers la valeur du type decimal à convertir (`np`) et un pointeur vers la variable de type long de stockage du résultat de l'opération (`lngp`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur. Si un dépassement survient, `ECPG_INFORMIX_NUM_OVERFLOW` est renvoyé.

L'implantation d'ECPG diffère de celle d'Informix™. Informix™ limite un entier long à la plage -2.147.483.647 .. 2.147.483.647 alors que les limites de l'implantation d'ECPG dépendent de l'architecture (`-LONG_MAX .. LONG_MAX`).

`rdatestr`
convertit une date en une chaîne C de type char*.

```
int rdatestr(date d, char *str);
```

La fonction prend deux arguments. Le premier est la date à convertir (`d`) et le second est un pointeur vers la chaîne cible. Le format de sortie est toujours `yyyy-mm-dd`. Il est donc nécessaire d'allouer au minimum 11 octets (ce qui inclut le terminateur NUL) pour la chaîne.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

L'implantation d'ECPG diffère de celle d'Informix™. Dans Informix™, le format peut être modifié en configurant les variables d'environnement. En revanche, avec ECPG, le format de sortie ne peut pas être modifié.

`rstrdate`
analyse la représentation textuelle d'une date.

```
int rstrdate(char *str, date *d);
```

La fonction reçoit la représentation textuelle de la date à convertir (`str`) et un pointeur vers une variable de type date (`d`). Cette fonction ne permet pas de préciser un masque de formatage. Elle utilise le masque de formatage d'Informix™, `mm/dd/yyyy`. En interne, cette fonction est implantée via `rdefmtdate`. Du coup, `rstrdate` n'est pas plus rapide et si c'est possible, il est préférable d'opter pour `rdefmtdate` qui permet d'expliciter le masque de formatage.

Cette fonction renvoie les mêmes valeurs que `rdefmtdate`.

`rtoday`
récupère la date courante.

```
void rtoday(date *d);
```

La fonction reçoit un pointeur vers une variable de type date (`d`) qu'elle positionne à la date courante.

En interne, cette fonction utilise la fonction `PGTYPESdate_today`.

`rjulmdy`
extrait les valeurs du jour, du mois et de l'année à partir d'une variable de type date.

```
int rjulmdy(date d, short mdy[3]);
```

La fonction reçoit la date `d` et un pointeur vers un tableau de trois valeurs de type short integer, `mdy`. Le nom de la variable indique l'ordre séquentiel : `mdy[0]` contient le numéro du mois, `mdy[1]` contient le numéro du jour et `mdy[2]` contient l'année.

Cette fonction renvoie toujours 0 actuellement.

En interne, cette fonction utilise la fonction `PGTYPESdate_julmdy`.

`rdefmtdate`

utilise un masque de formatage pour convertir une chaîne de caractères en une valeur de type date.

```
int rdefmtdate(date *d, char *fmt, char *str);
```

La fonction reçoit un pointeur vers la valeur de type date de stockage du résultat de l'opération (`d`), le masque de formatage à utiliser pour analyser la date (`fmt`) et la chaîne C char* contenant la représentation textuelle de la date (`str`). La représentation textuelle doit correspondre au masque de formatage. Il n'est toutefois pas nécessaire d'avoir une correspondance caractère par caractère entre la chaîne et le masque de formatage. La fonction n'analyse que l'ordre séquentiel et recherche les constantes `yy` ou `yyyy` pour la position de l'année, `mm` pour la position du mois et `dd` pour la position du jour.

La fonction renvoie les valeurs suivantes :

- 0 - la fonction a terminé avec succès ;
- `ECPG_INFORMIX_ENOSHORTDATE` - la date ne contient pas de délimiteur entre le jour, le mois et l'année. Dans ce cas, la taille de la chaîne en entrée doit être de six ou huit octets exactement mais elle ne l'est pas ;
- `ECPG_INFORMIX_ENOTDMY` - la chaîne de formatage n'indique pas correctement l'ordre séquentiel de l'année, du mois et du jour ;
- `ECPG_INFORMIX_BAD_DAY` - la chaîne en entrée ne contient pas de jour valide ;
- `ECPG_INFORMIX_BAD_MONTH` - la chaîne en entrée ne contient pas de mois valide ;
- `ECPG_INFORMIX_BAD_YEAR` - la chaîne en entrée ne contient pas d'année valide.

En interne, cette fonction est codée pour utiliser la fonction `PGTYPESdate_defmt_asc`. La référence présente un tableau d'exemples de saisie.

`rfmtdate`

convertit une variable de type date en sa représentation textuelle en utilisant un masque de formatage.

```
int rfmtdate(date d, char *fmt, char *str);
```

La fonction reçoit la date à convertir (`d`), le masque de formatage (`fmt`) et la chaîne de stockage de la représentation textuelle de la date (`str`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

En interne, cette fonction utilise la fonction `PGTYPESdate_fmt_asc`. La référence présente divers exemples.

`rmdyjul`

crée une valeur de type date à partir d'un tableau de trois entiers indiquant le jour, le mois et l'année de la date.

```
int rmdyjul(short mdy[3], date *d);
```

La fonction reçoit le tableau de trois entiers courts (`mdy`) et un pointeur vers une variable de type date pour stocker le résultat de l'opération.

Actuellement, la fonction renvoie toujours 0.

En interne, cette fonction utilise la fonction `PGTYPESdate_mdyjul`.

`rdayofweek`

renvoie un numéro représentant le jour de la semaine pour une valeur de type date.

```
int rdayofweek(date d);
```

La fonction prend comme seul argument la variable de type date `d` et renvoie un entier qui indique le jour de la semaine pour

cette date.

- 0 - Dimanche
- 1 - Lundi
- 2 - Mardi
- 3 - Mercredi
- 4 - Jeudi
- 5 - Vendredi
- 6 - Samedi

En interne, cette fonction utilise la fonction `PGTYPESdate_dayofweek`.

`dtcurrent`
récupère le timestamp courant.

```
void dtcurrent(timestamp *ts);
```

La fonction récupère la valeur du timestamp courant et la sauvegarde dans la variable de type timestamp pointée par `ts`.

`dtcvasc`
analyse une variable de type timestamp à partir d'une représentation textuelle pour en faire une variable de type timestamp.

```
int dtcvasc(char *str, timestamp *ts);
```

La fonction reçoit la chaîne à analyser (`str`) et un pointeur vers la variable de type timestamp pour stocker le résultat de l'opération (`ts`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

En interne, cette fonction utilise la fonction `PGTYPEStimestamp_from_asc`. Cette référence présente un tableau d'exemples de saisie.

`dtcvfmtasc`
analyse une variable de type timestamp à partir de sa représentation textuelle en utilisant un masque de formatage pour en faire une variable de type timestamp.

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

La fonction reçoit la chaîne à analyser (`inbuf`), le masque de formatage à utiliser (`fmtstr`) et un pointeur vers une variable de type timestamp pour stocker le résultat de l'opération (`dtvalue`).

Cette fonction utilise la fonction `PGTYPEStimestamp_defmt_asc`. Voir la documentation pour une liste des spécificateurs de format utilisables.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`dtsub`
soustrait une variable de type timestamp à une autre et renvoie une variable de type interval.

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

La fonction soustrait la variable timestamp pointée par `ts2` à la variable timestamp pointée par `ts1` et stocke le résultat dans la variable interval pointée par `iv`.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`dttoasc`
convertit une variable de type timestamp en une chaîne C de type char*.

```
int dttoasc(timestamp *ts, char *output);
```

La fonction reçoit un pointeur vers la variable de type timestamp à convertir (`ts`) et la chaîne de stockage du résultat de l'opération, `output`. Elle convertit `ts` en sa représentation textuelle dans le standard SQL, définie comme `YYYY-MM-DD HH:MM:SS`.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`dttofmtasc`
convertit une variable de type timestamp en une chaîne C de type char* en utilisant un masque de formatage.

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

La fonction reçoit un pointeur vers la variable de type timestamp à convertir (`ts`) et la chaîne de stockage du résultat de l'opération, `output`, la longueur maximale allouée au tampon de sortie (`str_len`) et le masque de formatage à utiliser pour la conversion (`fmtstr`).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

En interne, cette fonction utilise la fonction `PGTYPEStimestamp_fmt_asc`. Cette référence fournit les informations concernant les spécificateurs de format de masque utilisables.

`intoasc`
convertit une variable de type timestamp en une chaîne C de type char*.

```
int intoasc(interval *i, char *str);
```

La fonction reçoit un pointeur vers la variable de type timestamp à convertir (`i`) et la chaîne de stockage du résultat de l'opération, `str`. Elle convertit `i` en sa représentation textuelle dans le standard SQL, définie comme `YYYY-MM-DD HH:MM:SS`.

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

`rfmtlong`
convertit un entier long en sa représentation textuelle en utilisant un masque de formatage.

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

La fonction reçoit la valeur `lng_val` de type long, le masque de formatage `fmt` et un pointeur vers un tampon de sortie `outbuf`. Elle convertit la valeur de type long en sa représentation textuelle selon le masque de formatage.

Le masque de formatage est composé de caractères de spécification de formats à prendre parmi :

- `*` (astérisque) - remplace une position autrement vide par une astérisque ;
- `&` (esperluette, perluette ou « et commercial ») - remplace une position autrement vide par un zéro ;
- `#` - remplace les zéros de début de chaîne par des espaces ;
- `<` - justifie le nombre à gauche dans la chaîne ;
- `,` (virgule) - groupe les nombres de plus de quatre chiffres en groupes de trois chiffres séparés par des virgules ;
- `.` (point) - sépare la partie décimale de la partie entière ;
- `-` (moins) - apparaît si le nombre est négatif ;
- `+` (plus) - apparaît si le nombre est positif ;
- `(` - remplace le signe moins devant le nombre négatif. Le signe moins n'apparaît pas ;
- `)` - remplace le signe moins et est affiché derrière la valeur négative ;
- `$` - représente le symbole monétaire.

`rupshift`
convertit une chaîne en majuscules.

```
void rupshift(char *str);
```

La fonction reçoit un pointeur vers la chaîne et transforme chaque caractère minuscule en majuscule.

`byleng`
renvoie le nombre de caractères dans une chaîne sans compter les espaces en fin de chaîne.

```
int byleng(char *str, int len);
```

La fonction attend une chaîne de longueur fixe comme premier argument (`str`) et sa longueur comme deuxième argument (`len`). Elle renvoie le nombre de caractères significatifs, c'est-à-dire la longueur de la chaîne sans les espaces de fin de chaîne.

`ldchar`

copie une chaîne de longueur fixe dans une chaîne terminée par NUL.

```
void ldchar(char *src, int len, char *dest);
```

La fonction reçoit une chaîne de longueur fixe à copier (`src`), sa longueur (`len`) et un pointeur vers la chaîne de destination (`dest`). Il est nécessaire de réserver au moins `len+1` octets pour la chaîne vers laquelle `dest` pointe. La fonction copie au maximum `len` octets vers le nouvel emplacement (moins si la chaîne source contient des espaces en fin de chaîne) et ajoute le terminateur de chaîne NUL.

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

Cette fonction existe mais elle n'est pas codée actuellement !

`rtypalign`

```
int rtypalign(int offset, int type);
```

Cette fonction existe mais elle n'est pas codée actuellement !

`rtypmsize`

```
int rtypmsize(int type, int len);
```

Cette fonction existe mais elle n'est pas codée actuellement !

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

Cette fonction existe mais elle n'est pas codée actuellement !

`rsetnull`

initialise une variable à NULL.

```
int rsetnull(int t, char *ptr);
```

La fonction reçoit un entier qui indique le type de la variable et un pointeur vers la variable elle-même qui est convertie en un pointeur de chaîne C, type char*.

Les types suivants existent :

- `CCHARTYPE` - pour une variable de type char ou char* ;
- `CSHORTTYPE` - pour une variable de type short int ;
- `CINTTYPE` - pour une variable de type int ;
- `CBOOLTYPE` - pour une variable de type boolean ;
- `CFLOATTYPE` - pour une variable de type float ;
- `CLONGTYPE` - pour une variable de type long ;
- `CDOUBLETYPE` - pour une variable de type double ;
- `CDECIMALTYPE` - pour une variable de type decimal ;
- `CDATETYPE` - pour une variable de type date ;
- `CDTIMETYPE` - pour une variable de type timestamp.

Exemple d'appel de la fonction :

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`
teste si une variable est NULL.

```
int risnull(int t, char *ptr);
```

La fonction reçoit le type de la variable à tester (`t`) ainsi qu'un pointeur vers cette variable (`ptr`). Ce dernier a besoin d'être converti en char*. Voir la fonction `rsetnull` pour une liste des types de variable possibles.

Exemple d'utilisation de la fonction :

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

33.10.5. Constantes supplémentaires

Toutes les constantes qui suivent décrivent des erreurs et toutes sont négatives. Dans les descriptions des différentes constantes se trouve la valeur de la constante dans l'implantation actuelle. Il est toutefois préférable de ne pas compter sur ce nombre. Cela dit on peut se reposer sur le fait que toutes les valeurs sont négatives.

`ECPG_INFORMIX_NUM_OVERFLOW`
Les fonctions renvoient cette valeur si un dépassement survient lors d'un calcul. En interne, elle vaut -1200 (définition Informix™).

`ECPG_INFORMIX_NUM_UNDERFLOW`
Les fonctions renvoient cette valeur si un soupassement survient lors d'un calcul. En interne, elle vaut -1201 (définition Informix™).

`ECPG_INFORMIX_DIVIDE_ZERO`
Les fonctions renvoient cette valeur si une tentative de division par zéro est observée. En interne, elle vaut -1202 (définition Informix™).

`ECPG_INFORMIX_BAD_YEAR`
Les fonctions renvoient cette valeur si une mauvaise valeur pour l'année est trouvée lors de l'analyse d'une date. En interne, elle vaut -1204 (définition Informix™).

`ECPG_INFORMIX_BAD_MONTH`
Les fonctions renvoient cette valeur si une mauvaise valeur pour le mois est trouvée lors de l'analyse d'une date. En interne, elle vaut -1205 (définition Informix™).

`ECPG_INFORMIX_BAD_DAY`
Les fonctions renvoient cette valeur si une mauvaise valeur pour le jour est trouvée lors de l'analyse d'une date. En interne, elle vaut -1206 (définition Informix™).

`ECPG_INFORMIX_ENOSHORTDATE`
Les fonctions renvoient cette valeur si une routine d'analyse nécessite une représentation courte de date mais qu'elle n'obtient pas une chaîne de la bonne longueur. En interne, elle vaut -1209 (définition Informix™).

`ECPG_INFORMIX_DATE_CONVERT`
Les fonctions renvoient cette valeur en cas de problème lors de la conversion de la date. En interne, elle vaut -1210 (définition Informix™).

`ECPG_INFORMIX_OUT_OF_MEMORY`
Les fonctions renvoient cette valeur en cas de manque de mémoire. En interne, elle vaut -1211 (définition Informix™).

`ECPG_INFORMIX_ENOTDMY`

Les fonctions renvoient cette valeur si une routine d'analyse attend un masque de formatage (comme `mmddyy`) mais que certains champs n'ont pas été indiqués correctement. En interne, elle vaut -1212 (définition Informix™).

`ECPG_INFORMIX_BAD_NUMERIC`
Les fonctions renvoient cette valeur si une routine d'analyse ne peut pas créer la représentation textuelle d'une valeur numérique, parce qu'elle contient des erreurs, ou si une routine ne peut terminer un calcul impliquant des variables numériques parce que l'une d'elle, au moins, est invalide. En interne, elle vaut -1213 (définition Informix™).

`ECPG_INFORMIX_BAD_EXPONENT`
Les fonctions renvoient cette valeur en cas de mauvais exposant. En interne, elle vaut -1216 (définition Informix™).

`ECPG_INFORMIX_BAD_DATE`
Les fonctions renvoient cette valeur en cas de mauvaise valeur. En interne, elle vaut -1218 (définition Informix™).

`ECPG_INFORMIX_EXTRA_CHARS`
Les fonctions renvoient cette valeur en cas de caractères superflus. En interne, elle vaut -1264 (définition Informix™).

33.11. Gestion des erreurs

Cette section décrit la gestion des conditions exceptionnelles et des avertissements dans un programme SQL embarqué. Il existe plusieurs fonctions non exclusives pour cela.

33.11.1. Configurer des rappels

Une méthode simple de récupération des erreurs et des avertissements consiste à configurer une action spécifique à exécuter à chaque fois qu'une condition particulière survient. En général :

```
EXEC SQL WHENEVER condition action;
```

`condition` peut prendre une des valeurs suivantes :

`SQLERROR`
l'action indiquée est appelée lorsqu'une erreur survient pendant l'exécution d'une instruction SQL ;

`SQLWARNING`
l'action indiquée est appelée lorsqu'un avertissement survient pendant l'exécution d'une instruction SQL ;

`NOT FOUND`
l'action indiquée est appelée lorsqu'une instruction ne récupère ou n'affecte aucune ligne (cette condition n'est pas une erreur mais il peut être intéressant de la gérer de façon particulière).

`action` peut prendre une des valeurs suivantes :

`CONTINUE`
la condition est ignorée. C'est le comportement par défaut ;

`GOTO` *`label`*, `GO TO` *`label`*
saute au label indiqué (à l'iade d'une instruction C `goto`) ;

`SQLPRINT`
affiche un message sur la sortie standard. Cela est utile pour les programmes simples ou lors du prototypage. Les détails du message ne peuvent pas être configurés ;

`STOP`
appelle `exit(1)`, ce qui termine le programme ;

`DO BREAK`
exécute l'instruction C `break`. Cela ne doit être utilisé que dans les boucles et les instructions `switch` ;

`CALL` *`nom`* `(`*`args`*`)`, `DO` *`nom`* `(`*`args`*`)`
appelle les fonctions C indiquées avec les arguments indiqués.

Le standard SQL ne définit que les actions `CONTINUE` et `GOTO` (et `GO TO`).

L'exemple suivant est utilisable dans un programme simple. Il affiche un message lorsqu'un avertissement survient et termine le programme quand une erreur se produit :

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

L'instruction `EXEC SQL WHENEVER` est une directive du préprocesseur SQL, pas une instruction C. Les actions sur erreur ou avertissement qu'elle définit s'appliquent à toutes les instructions SQL embarquées qui apparaissent avant l'endroit où le gestionnaire est défini, à moins qu'une action différente n'ait été définie pour la même condition entre le premier `EXEC SQL WHENEVER` et l'instruction SQL qui engendre la condition, quelque soit le flux de contrôle du programme C. De ce fait, aucun des deux extraits de programme C qui suivent n'a le comportement désiré :

```
/*
 * FAUX
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*
 * FAUX
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

33.11.2. sqlca

Pour une gestion plus puissante des erreurs, l'interface du SQL embarqué fournit une variable globale de nom `sqlca` qui a la structure suivante :

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmcSQLERRMC_LEN;
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(Dans un programme multithreadé, chaque thread obtient automatiquement sa propre copie de `sqlca`. Ceci fonctionne de façon similaire à la gestion de la variable globale C standard `errno`.)

`sqlca` couvre à la fois les avertissements et les erreurs. Si plusieurs avertissements ou erreurs surviennent lors de l'exécution d'une instruction, alors `sqlca` ne contient que les informations relatives à la dernière.

Si aucune erreur ne survient dans la dernière instruction SQL, `sqlca.sqlcode` vaut 0 et `sqlca.sqlstate` vaut `"00000"`. Si un avertissement ou une erreur survient, alors `sqlca.sqlcode` est négatif et `sqlca.sqlstate` est différent de `"00000"`. Une valeur positive de `sqlca.sqlcode` indique une condition sans dommage, telle que « la dernière requête n'a retourné aucune ligne ». `sqlcode` et `sqlstate` sont deux schémas de code d'erreur différents ; les détails apparaissent ci-dessous.

Si la dernière instruction SQL a réussi, alors `sqlca.sqlerrd[1]` contient l'OID de la ligne traitée, si applicable, et `sqlca.sqlerrd[2]` contient le nombre de lignes traitées ou renvoyées, si applicable à la commande.

En cas d'erreur ou d'avertissement, `sqlca.sqlerrm.sqlerrmc` contient une chaîne décrivant l'erreur. Le champ `sqlca.sqlerrm.sqlerrml` contient la longueur du message d'erreur stocké dans `sqlca.sqlerrm.sqlerrmc` (le résultat de `strlen()`, sans réel intérêt pour un programmeur C). Certains messages sont trop longs pour entrer dans le tableau `sqlerrmc` de taille fixe ; ils sont alors tronqués.

En cas d'avertissement, `sqlca.sqlwarn[2]` est positionné à `W`. (Dans tous les autres cas, il est positionné à quelque chose de différent de `W`.) Si `sqlca.sqlwarn[1]` est positionné à `W`, alors une valeur a été tronquée lors de son stockage dans une variable hôte. `sqlca.sqlwarn[0]` est positionné à `W` si tout autre élément est positionné pour indiquer un avertissement.

Les champs *`sqlcaid`*, *`sqlcabc`*, *`sqlerrp`* et les éléments restant de *`sqlerrd`* et *`sqlwarn`* ne contiennent actuellement aucune information utile.

La structure `sqlca` n'est pas définie dans le standard SQL mais elle est implantée dans plusieurs autres systèmes de bases de données SQL. Leurs définitions sont similaires dans l'esprit, mais l'écriture d'applications portables nécessite une étude attentive des autres implantations.

33.11.3. SQLSTATE vs SQLCODE

Les champs `sqlca.sqlstate` et `sqlca.sqlcode` sont deux schémas différents fournissant des codes d'erreur. Les deux sont dérivés du standard SQL mais `SQLCODE` est indiqué comme obsolète dans l'édition SQL-92 du standard et a été supprimé dans les éditions ultérieures. C'est pourquoi il est fortement recommandé que les nouvelles applications utilisent `SQLSTATE` dans les nouvelles applications.

`SQLSTATE` est un tableau de cinq caractères. Ces cinq caractères contiennent des chiffres ou des lettres en majuscules représentant les codes de différentes conditions d'erreur ou d'avertissement. `SQLSTATE` dispose d'un schéma hiérarchique : les deux premiers caractères indiquent la classe générale de la condition, les trois derniers caractères indiquent une sous-classe de la condition générale. Un succès est indiqué par le code `00000`. Les codes `SQLSTATE` sont pour la plupart définis dans le standard SQL. Le serveur PostgreSQL™ supporte nativement les codes d'erreurs `SQLSTATE` ; de ce fait, un haut degré de cohérence peut être atteint en utilisant ce schéma de code d'erreur dans toutes les applications. Pour plus d'informations, voir l'Annexe A, Codes d'erreurs de PostgreSQL™.

`SQLCODE`, schéma obsolète de codes d'erreur, est un simple entier. Une valeur 0 indique un succès, une valeur positive indique un succès avec des informations supplémentaires, une valeur négative indique une erreur. Le standard SQL ne définit que la valeur positive +100, qui indique que la dernière commande n'a renvoyé ou modifié aucune ligne, et aucune valeur négative spécifique. De ce fait, ce schéma n'est que faiblement portable et ne propose pas d'affectation de code hiérarchique. Historiquement, le processeur de SQL embarqué pour PostgreSQL™ a affecté quelques valeurs à `SQLCODE` spécifiques pour sa propre utilisation. ces valeurs sont listées ci-dessous avec leurs valeurs numériques et leurs noms symboliques. Elles ne sont pas portables vers d'autres implantations SQL. Pour simplifier le portage d'applications vers la schéma `SQLSTATE`, le code `SQLSTATE` correspondant est également affiché. Il n'y a toutefois pas de correspondance une-à-une ou une-à-plusieurs entre les deux schémas (en fait, c'est plutôt plusieurs-à-plusieurs). Il est préférable de consulter le schéma `SQLSTATE` global dans l'Annexe A, Codes d'erreurs de PostgreSQL™ pour chaque cas.

Valeurs affectées à `SQLCODE` :

-12 (ECPG_OUT_OF_MEMORY)
: la mémoire virtuelle est épuisée, (SQLSTATE YE001) ;

-200 (ECPG_UNSUPPORTED)
: le préprocesseur a engendré quelque chose que la bibliothèque ne connaît pas. Il peut s'agir de l'exécution de versions incompatibles du préprocesseur et de la bibliothèque, (SQLSTATE YE002). ;

-201 (ECPG_TOO_MANY_ARGUMENTS)
: la commande indique plus de variables hôtes que n'en attend la commande (SQLSTATE 07001 ou 07002) ;

-202 (ECPG_TOO_FEW_ARGUMENTS)
: la commande indique moins de variables hôtes que n'en attend la commande (SQLSTATE 07001 ou 07002) ;

-203 (ECPG_TOO_MANY_MATCHES)
: une requête retourne plusieurs lignes alors que l'instruction est préparée à ne stocker qu'une seule ligne de résultat (par exemple, parce que les variables indiquées ne sont pas des tableaux) (SQLSTATE 21000) ;

-204 (ECPG_INT_FORMAT)
: la variable hôte est de type int alors que la donnée dans la base est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un int. La bibliothèque utilise `strtol()` pour cette conversion (SQLSTATE 42804) ;

-205 (ECPG_UINT_FORMAT)
la variable hôte est de type unsigned int alors que la donnée de la base est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un unsigned int. La bibliothèque utilise `strtoul()` pour cette conversion (SQLSTATE 42804) ;

-206 (ECPG_FLOAT_FORMAT)
la variable hôte est de type float alors que la donnée de la base est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un float. La bibliothèque utilise `strtod()` pour cette conversion (SQLSTATE 42804) ;

-211 (ECPG_CONVERT_BOOL)
la variable hôte est de type bool alors que la donnée de la base n'est ni `'t'` ni `'f'` (SQLSTATE 42804) ;

-212 (ECPG_EMPTY)
l'instruction envoyée au serveur PostgreSQL™ est vide (ceci ne peut normalement pas survenir dans un programme SQL embarqué et peut donc indiquer une erreur interne) (SQLSTATE YE002) ;

-213 (ECPG_MISSING_INDICATOR)
une valeur NULL est retournée alors qu'aucune variable indicateur de nullité n'est fournie (SQLSTATE 22002) ;

-214 (ECPG_NO_ARRAY)
une variable ordinaire est utilisée à un endroit qui requiert un tableau (SQLSTATE 42804) ;

-215 (ECPG_DATA_NOT_ARRAY)
la base de données a retourné une variable ordinaire à un endroit qui requiert une valeur de tableau (SQLSTATE 42804) ;

-220 (ECPG_NO_CONN)
le programme tente d'accéder à une connexion qui n'existe pas (SQLSTATE 08003) ;

-221 (ECPG_NOT_CONN)
le programme tente d'accéder à une connexion qui existe mais n'est pas ouverte (ceci est une erreur interne) (SQLSTATE YE002) ;

-230 (ECPG_INVALID_STMT)
l'instruction utilisée n'a pas été préparée (SQLSTATE 26000) ;

-240 (ECPG_UNKNOWN_DESCRIPTOR)
le descripteur indiqué n'a pas été trouvé. L'instruction utilisée n'a pas été préparée (SQLSTATE 33000) ;

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)
l'index du descripteur indiqué est hors échelle (SQLSTATE 07009) ;

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)
un élément invalide du descripteur est demandé (erreur interne) (SQLSTATE YE002) ;

-243 (ECPG_VAR_NOT_NUMERIC)
lors de l'exécution d'une instruction dynamique, la base de données a retourné une valeur numérique alors que la variable hôte n'est pas numérique (SQLSTATE 07006) ;

-244 (ECPG_VAR_NOT_CHAR)
lors de l'exécution d'une instruction dynamique, la base de données a retourné une valeur non numérique alors que la variable hôte est numérique (SQLSTATE 07006) ;

-400 (ECPG_PGSQL)
le serveur PostgreSQL™ a engendré une erreur. Le message contient le message d'erreur du serveur PostgreSQL™ ;

-401 (ECPG_TRANS)
le serveur PostgreSQL™ signale que la transaction ne peut être commencée, validée ou annulée (SQLSTATE 08007) ;

-402 (ECPG_CONNECT)
la tentative de connexion à la base de données a échoué (SQLSTATE 08001) ;

100 (ECPG_NOT_FOUND)
il s'agit d'une condition sans gravité qui indique que la dernière commande n'a récupéré ni traité aucune ligne, ou que la fin du curseur est atteinte (SQLSTATE 02000).

33.12. Directives du préprocesseur

33.12.1. Inclusion de fichiers

Pour inclure un fichier externe dans un programme SQL embarqué, il suffit d'utiliser :

```
EXEC SQL INCLUDE nomfichier;
```

Le préprocesseur du SQL embarqué cherche un fichier nommé `nomfichier.h`, le prétraite et l'inclut dans la sortie C résultante. De ce fait, les instructions C embarquées du fichier inclus sont correctement gérées.

Ce *n'est pas* la même chose que :

```
#include <nomfichier.h>
```

parce que ce fichier n'est alors pas sujet au prétraitement des commandes SQL. Naturellement, la directive C `#include` peut toujours être utilisée pour inclure d'autres fichiers d'en-tête.

> **Note**
>
> Le nom du fichier inclus est sensible à la casse, même si le reste de la commande `EXEC SQL INCLUDE` suit les règles habituelles de sensibilité à la casse du SQL.

33.12.2. Directives #define et #undef

Le SQL embarqué utilise un concept similaire à la directive `#define` utilisée en C :

```
EXEC SQL DEFINE nom;
EXEC SQL DEFINE nom valeur;
```

Un nom peut être défini :

```
EXEC SQL DEFINE HAVE_FEATURE;
```

Des constantes peuvent aussi être définies :

```
EXEC SQL DEFINE MON_NOMBRE 12;
EXEC SQL DEFINE MA_CHAINE 'abc';
```

`undef` est utilisé pour supprimer une définition :

```
EXEC SQL UNDEF MON_NOMBRE;
```

Les versions C de `#define` et `#undef` peuvent, évidemment, continuer à être utilisées dans le programme qui embarque le SQL. La différence se situe à l'endroit où les valeurs définies sont évaluées. Si `EXEC SQL DEFINE` est utilisé, alors le préprocesseur **ecpg** évalue les définitions et effectue les substitutions. Par exemple, si on écrit :

```
EXEC SQL DEFINE MON_NOMBRE 12;
...
EXEC SQL UPDATE Tbl SET col = MON_NOMBRE;
```

alors **ecpg** effectue la substitution et le compilateur C ne voit aucun nom ou identifiant `MON_NOMBRE`. `#define` ne peut pas être utilisé pour une constante utilisée dans une requête SQL embarquée car, dans ce cas, le précompilateur ecpg n'est pas capable de voir cette déclaration.

33.12.3. Directives ifdef, ifndef, else, elif et endif

Les directives suivantes peuvent être utilisées pour compiler des sections de code de façon conditionnelle :

`EXEC SQL ifdef nom;`
: vérifie un *nom* et traite les lignes qui suivent si *nom* a été créé avec `EXEC SQL define nom` ;

`EXEC SQL ifndef nom;`
: vérifie un *nom* et traite les lignes qui suivent si *nom n'a pas* été créé avec `EXEC SQL define nom` ;

`EXEC SQL else;`
: commence le traitement d'une section alternative à une section introduite avec `EXEC SQL ifdef nom` ou `EXEC SQL ifndef nom` ;

`EXEC SQL elif nom;`
: vérifie *nom* et commence le traitement d'une section alternative si *nom* a été créé avec `EXEC SQL define nom` ;

`EXEC SQL endif;`
: termine une section alternative.

Exemple :

```
EXEC SQL ifndef TZVAR;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

33.13. Traiter les programmes en SQL embarqué

Après avoir vu comment former des programmes C incluant du SQL embarqué, il est tout aussi intéressant de savoir comment les compiler.

Avant d'être compilé, le fichier est passé au travers du préprocesseur C pour le SQL embarqué. Ce préprocesseur, appelé aussi pré-compilateur, convertit les instructions SQL utilisées en appels de fonctions spéciales. Après compilation, le programme doit être lié avec une bibliothèque spéciale contenant les fonctions nécessaires. Ces fonctions traitent l'information issues des arguments, exécutent la commande SQL via l'interface libpq et placent le résultat dans les arguments indiqués pour la sortie.

Le préprocesseur, appelé `ecpg`, est inclus dans toute installation standard de PostgreSQL™. Les programmes en SQL embarqué sont nommés typiquement avec une extension `.pgc`. Un fichier programme nommé `prog1.pgc` peut être passé au préprocesseur par la simple commande :

```
ecpg prog1.pgc
```

Ceci crée un fichier nommé `prog1.c`. Si les fichiers en entrée ne suivent pas le modèle de nommage suggéré, il est possible de nommer explicitement le fichier de sortie en utilisant l'option `-o`.

Le fichier traité par le préprocesseur peut être compilé normalement. Par exemple :

```
cc -c prog1.c
```

Les fichiers sources en C engendrés incluent les fichiers d'en-tête provenant de l'installation de PostgreSQL™. De ce fait, si PostgreSQL™ a été installé à un emplacement qui n'est pas parcouru par défaut, il faut ajouter une option comme `-I/usr/local/pgsql/include` sur la ligne de commande de la compilation.

Pour lier un programme SQL embarqué, il faut inclure la bibliothèque `libecpg` de cette façon :

```
cc -o monprog prog1.o prog2.o ... -lecpg
```

Là encore, il peut être nécessaire d'ajouter une option comme `-L/usr/local/pgsql/lib` sur la ligne de commande.

Si le processus de construction d'un grand projet est géré avec make, il peut être plus pratique d'inclure la règle implicite suivante dans les fichiers makefile :

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

La syntaxe complète de la commande **ecpg** est détaillée dans ecpg(1).

La bibliothèque ecpg est compatible avec les threads par défaut. Néanmoins, vous pourriez avoir besoin d'utiliser des options en ligne de commande pour les threads, histoire de pouvoir compiler votre code.

33.14. Fonctions de bibliothèque

La bibliothèque `libecpg` contient principalement des fonctions « cachées » utilisées pour coder les fonctionnalités exprimées par les commandes SQL embarquées. Mais il existe quelques fonctions qu'il peut être utile d'appeler directement. Cela a toutefois des conséquences sur la portabilité du code.

- `ECPGdebug(int` *`on`*`, FILE *`*`flux`*`)` active le débogage si elle est appelée avec une valeur différente de zéro pour le premier argument. Les traces de débogage sont envoyées sur le *`flux`*. Les traces contiennent toutes les instructions SQL avec toutes les variables en entrée et les résultats du serveur PostgreSQL™. Cela peut être très utile pour rechercher des erreurs dans les instructions SQL.

Note

Sur Windows, si les bibliothèques ecpg et une application sont compilées avec des options différentes, cet appel

de fonction arrête brutalement l'application car la représentation interne des pointeurs `FILE` diffère. En particulier, les options multi-threaded/single-threaded, release/debug et static/dynamic doivent être identiques pour la bibliothèque et pour toutes les applications qui utilisent cette bibliothèque.

- `ECPGget_PGconn(const char *connection_name)` renvoie un lien vers la connexion à la base de données d'après le nom de la connexion. Si *`connection_name`* vaut `NULL`, le lien courant est renvoyé. Si aucun lien ne peut être identifié, la fonction renvoie `NULL`. Le lien de la connexion renvoyé peut être utilisé pour appeler toute autre fonction de la bibliothèque libpq, si nécessaire.

  Note

  C'est généralement une mauvaise idée que de manipuler les liens de connexions à la base de données à partir d'ecpg directement avec des routines de la bibliothèque libpq.

- `ECPGtransactionStatus(const char *connection_name)` renvoie l'état de la transactions en cours sur la connexion identifiée par son nom via l'argument *`connection_name`*. Voir Section 31.2, « Fonctions de statut de connexion » et la fonction `PQtransactionStatus()` de la bibliothèque libpq pour des détails sur les codes de statut renvoyés.
- `ECPGstatus(int no_ligne, const char* nom_connexion)` retourne vrai si une connexion à une base de données est active, faux sinon. *`nom_connexion`* peut être `NULL` si une seule connexion est utilisée.

33.15. Fonctionnement interne

Cette section explique le fonctionnement interne d'ECPG. Cette information est parfois utile pour aider les utilisateurs à comprendre comment utiliser ECPG.

Les quatre premières lignes écrites par **ecpg** sur la sortie sont des lignes figées. Deux sont des commentaires et deux sont des lignes d'inclusion de fichiers d'en-tête nécessaires pour l'interfaçage avec la bibliothèque. Ensuite, le préprocesseur lit le fichier et écrit la sortie. Normalement, il ne fait que tout répéter sur la sortie.

Lorsqu'**ecpg** lit une instruction **EXEC SQL**, il intervient et la modifie. La commande commence avec **EXEC SQL** et se termine avec **;**. Tout ce qui se trouve entre est traité comme une instruction SQL et analysé pour substitution de variable.

La substitution de variable intervient quand un symbole débute par un caractère deux-points (`:`). La variable possédant ce nom est recherchée parmi toutes les variables précédemment déclarées dans une section `EXEC SQL DECLARE`.

La fonction la plus importante de la bibliothèque est `ECPGdo`, qui prend en charge l'exécution de la plupart des commandes. Elle prend un nombre variable d'arguments. Ceci permet d'aller jusqu'à environ 50 arguments, ce qui, il faut l'espérer, ne pose de problème sur aucune plateforme.

Les arguments sont :

Un numéro de ligne
: C'est le numéro de ligne de la ligne originale ; utilisé seulement dans les messages d'erreur.

Une chaîne
: C'est la commande SQL à exécuter. Elle est modifiée par les variables en entrée, c'est-à-dire les variables qui n'étaient pas connues au moment de la compilation mais qui doivent être précisées dans la commande. À l'emplacement des variables, la chaîne contient le signe `?`.

Variables en entrée
: Chaque variable en entrée entraîne la création de dix arguments (voir ci-dessous).

`ECPGt_EOIT`
: Un enum indiquant qu'il n'y a plus de variables en entrée.

Variables en sortie
: Chaque variable en sortie entraîne la création de dix arguments (voir ci-dessous). La valeur des variables est fournie par la fonction.

`ECPGt_EORT`
: Un enum indiquant qu'il n'y a plus de variables.

Pour chaque variable de la commande SQL, la fonction récupère dix arguments :

1. le type comme symbole spécial ;
2. un pointeur sur la valeur ou un pointeur sur le pointeur ;
3. la taille de la variable dans le cas d'un type char ou varchar ;
4. le nombre d'éléments du tableau (pour les parcours de tableaux) ;
5. le décalage pour obtenir le prochain élément dans le tableau (pour les parcours de tableaux) ;
6. le type de variable indicateur comme un symbole spécial ;
7. un pointeur vers la variable indicateur ;
8. 0 ;
9. le nombre d'éléments dans le tableau indicateur (pour les parcours de tableaux) ;

10 le décalage pour obtenir le prochain élément dans le tableau indicateur (pour les parcours de tableaux).
.
Toutes les commandes SQL ne sont pas traitées ainsi. Par exemple, une instruction d'ouverture de curseur comme :

```
EXEC SQL OPEN curseur;
```

n'est pas copiée sur la sortie. À la place, la commande **DECLARE** de déclaration du curseur est utilisée à la position de la commande **OPEN** car, en fait, elle ouvre le curseur.

Exemple complet de sortie du préprocesseur d'un fichier `foo.pgc` (les détails peuvent varier pour chaque version particulière du préprocesseur) :

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int resultat;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :resultat FROM matable WHERE index = :index;
```

est traduit en :

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int resultat;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM matable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(resultat),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(L'indentation a été ajoutée ici pour des raisons de lisibilité et n'est pas réalisée par le préprocesseur)

Chapitre 34. Schéma d'information

Le schéma d'information consiste en un ensemble de vues contenant des informations sur les objets définis dans la base de données courante. Le schéma d'information est défini dans le standard SQL et, donc supposé portable et stable -- contrairement aux catalogues système qui sont spécifiques à PostgreSQL™ et modelés suivant l'implantation. Néanmoins, les vues du schéma d'information ne contiennent pas d'informations sur les fonctionnalités spécifiques à PostgreSQL™ ; pour cela, on utilise catalogues système et autres vues spécifiques à PostgreSQL™.

34.1. Le schéma

Le schéma d'information est lui-même un schéma nommé `information_schema`. Ce schéma existe automatiquement dans toutes les bases de données. Le propriétaire de ce schéma est l'utilisateur initial du cluster. Il a naturellement tous les droits sur ce schéma, dont la possibilité de le supprimer (mais l'espace gagné ainsi sera minuscule).

Par défaut, le schéma d'information n'est pas dans le chemin de recherche des schémas. Il est donc nécessaire d'accéder à tous les objets qu'il contient via des noms qualifiés. Comme les noms de certains objets du schéma d'information sont des noms génériques pouvant survenir dans les applications utilisateur, il convient d'être prudent avant de placer le schéma d'information dans le chemin.

34.2. Types de données

Les colonnes des vues du schéma d'information utilisent des types de données spéciaux, définis dans le schéma d'information. Ils sont définis comme des domaines simples sur des types internes. Vous normal, il est préférable de ne pas utiliser ces types en dehors du schéma d'information, mais les applications doivent pouvoir les utiliser si des sélections sont faites dans le schéma d'information.

Ces types sont :

cardinal_number
: Un entier non négatif.

character_data
: Une chaîne de caractères (sans longueur maximale indiquée).

sql_identifier
: Une chaîne de caractères. Elle est utilisée pour les identifiants SQL, le type de données character_data est utilisé pour tout autre type de données texte.

time_stamp
: Un domaine au-dessus du type timestamp with time zone

yes_or_no
: Un domaine dont le type correspond à une chaîne de caractères, qui contient soit `YES` soit `NO`. C'est utilisé pour représenter des données booléennes (true/false) dans le schéma d'informations. (Le schéma d'informations était inventé avant l'ajout du type boolean dans le standard SQL, donc cette convention est nécessaire pour conserver la compatibilité avec le schéma d'informations.)

Chaque colonne du schéma d'information est de l'un des ces cinq types.

34.3. `information_schema_catalog_name`

`information_schema_catalog_name` est une table qui contient en permanence une ligne et une colonne contenant le nom de la base de données courante (catalogue courant dans la terminologie SQL).

Tableau 34.1. Colonnes de `information_schema_catalog_name`

Nom	Type de données	Description
`catalog_name`	sql_identifier	Nom de la base de données contenant ce schéma d'informations

34.4. `administrable_role_authorizations`

La vue `administrable_role_authorizations` identifie tous les rôles pour lesquelles l'utilisateur courant possède l'option ADMIN.

Tableau 34.2. Colonnes de `administrable_role_authorizations`

Nom	Type de données	Description
`grantee`	sql_identifier	Nom du rôle pour lequel cette appartenance de rôle a été donnée (peut être l'utilisateur courant ou un rôle différent dans le cas d'appartenances de rôles imbriquées).
`role_name`	sql_identifier	Nom d'un rôle
`is_grantable`	yes_or_no	Toujours `YES`

34.5. `applicable_roles`

La vue `applicable_roles` identifie tous les rôles dont l'utilisateur courant peut utiliser les droits. Cela signifie qu'il y a certaines chaînes de donnation des droits de l'utilisateur courant au rôle en question. L'utilisateur lui-même est un rôle applicable. L'ensemble de rôles applicables est habituellement utilisé pour la vérification des droits.

Tableau 34.3. Colonnes de `applicable_roles`

Nom	Type de données	Description
`grantee`	sql_identifier	Nom du rôle à qui cette appartenance a été donnée (peut être l'utilisateur courant ou un rôle différent dans le cas d'appartenances de rôles imbriquées)
`role_name`	sql_identifier	Nom d'un rôle
`is_grantable`	yes_or_no	`YES` si le bénéficiaire a l'option ADMIN sur le rôle, `NO` dans le cas contraire

34.6. `attributes`

La vue `attributes` contient des informations sur les attributs des types de données composites définis dans la base. (La vue ne donne pas d'informations sur les colonnes de table, qui sont quelque fois appelées attributs dans le contexte de PostgreSQL.)

Tableau 34.4. Colonnes de `attributes`

Nom	Type de données	Description
`udt_catalog`	sql_identifier	Nom de la base contenant le type de données (toujours la base courante)
`udt_schema`	sql_identifier	Nom du schéma contenant le type de données
`udt_name`	sql_identifier	Nom du type de données
`attribute_name`	sql_identifier	Nom de l'attribut
`ordinal_position`	cardinal_number	Position de l'attribut dans le type de données (le décompte commence à 1)
`attribute_default`	character_data	Expression par défaut de l'attribut
`is_nullable`	yes_or_no	`YES` si l'attribut peut être NULL, `NO` dans le cas contraire.
`data_type`	character_data	Type de données de l'attribut s'il s'agit d'un type interne ou `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), sinon `USER-DEFINED` (dans ce cas, le type est identifié dans

Nom	Type de données	Description
		attribute_udt_name et les colonnes associées).
character_maximum_length	cardinal_number	Si data_type identifie un caractère ou une chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximale n'a été déclarée.
character_octet_length	cardinal_number	Si data_type identifie un type caractère, la longueur maximale en octets (bytes) d'un datum ; NULL pour tous les autres types de données. La longueur maximale en octets dépend de la longueur maximum déclarée en caractères (voir ci-dessus) et l'encodage du serveur.
numeric_precision	cardinal_number	Si data_type identifie un type numérique, cette colonne contient la précision (déclarée ou implicite) du type pour cet attribut. La précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2) comme le précise la colonne numeric_precision_radix. Pour tous les autres types de données, cette colonne vaut NULL.
numeric_precision_radix	cardinal_number	Si data_type identifie un type numérique, cette colonne indique la base d'expression des colonnes numeric_precision et numeric_scale. La valeur est soit 2 soit 10. Pour tous les autres types de données, cette colonne est NULL.
numeric_scale	cardinal_number	Si data_type identifie un type numérique exact, cette colonne contient l'échelle (déclarée ou implicite) du type pour cet attribut. L'échelle indique le nombre de chiffres significatifs à droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2) comme le précise la colonne numeric_precision_radix. Pour tous les autres types de données, cette colonne est NULL.
datetime_precision	cardinal_number	Si data_type identifie une date, une heure, un horodatage ou un interval, cette colonne contient la précision en secondes (déclarée ou implicite) pour cet attribut, c'est-à-dire le nombre de chiffres décimaux suivant le point décimal de la valeur en secondes. Pour tous les autres types de données, cette colonne est NULL.
interval_type	character_data	Pas encore implanté
interval_precision	character_data	Pas encore implanté
attribute_udt_catalog	sql_identifier	Nom de la base dans laquelle le type de données de l'attribut est défini (toujours la base courante)
attribute_udt_schema	sql_identifier	Nom du schéma dans lequel le type de données de l'attribut est défini

Nom	Type de données	Description
attribute_udt_name	sql_identifier	Nom du type de données de l'attribut
scope_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
scope_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
scope_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
maximum_cardinality	cardinal_number	Toujours NULL car les tableaux ont toujours une cardinalité maximale dans PostgreSQL™
dtd_identifier	sql_identifier	Un identifiant du descripteur du type de données de la colonne, unique parmi les descripteurs de types de données de la table. Ceci est principalement utile pour des jointures avec d'autres instances de tels identifiants. (Le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il reste identique dans les versions futures.)
is_derived_reference_attribute	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™

Voir aussi dans Section 34.12, « columns », une vue structurée de façon similaire, pour plus d'informations sur certaines colonnes.

34.7. check_constraint_routine_usage

La vue check_constraint_routine_usage identifie les routines (fonctions et procédures) utilisées par une contrainte de vérification. Seules sont présentées les routines qui appartiennent à un rôle couramment actif.

Tableau 34.5. Colonnes de check_constraint_routine_usage

Nom	Type de données	Description
constraint_catalog	sql_identifier	Nom de la base contenant la contrainte (toujours la base courante)
constraint_schema	sql_identifier	Nom du schéma contenant la contrainte
constraint_name	sql_identifier	Nom de la contrainte
specific_catalog	sql_identifier	Nom de la base contenant la fonction (toujours la base courante)
specific_schema	sql_identifier	Nom du schéma contenant la fonction
specific_name	sql_identifier	Le « nom spécifique » de la fonction. Voir Section 34.33, « routines » pour plus d'informations.

34.8. check_constraints

La vue check_constraints contient toutes les contraintes de vérification définies sur une table ou un domaine, possédées par un rôle couramment actif (le propriétaire d'une table ou d'un domaine est le propriétaire de la contrainte).

Tableau 34.6. Colonnes de check_constraints

Nom	Type de données	Description
constraint_catalog	sql_identifier	Nom de la base de données contenant la contrainte (toujours la base de données courante)

Nom	Type de données	Description
constraint_schema	sql_identifier	Nom du schéma contenant la contrainte
constraint_name	sql_identifier	Nom de la contrainte
check_clause	character_data	L'expression de vérification de la contrainte

34.9. column_domain_usage

La vue column_domain_usage identifie toutes les colonnes (d'une table ou d'une vue) utilisant un domaine défini dans la base de données courante et possédé par un rôle couramment actif.

Tableau 34.7. Colonnes de column_domain_usage

Nom	Type de données	Description
domain_catalog	sql_identifier	Nom de la base de données contenant le domaine (toujours la base de données courante)
domain_schema	sql_identifier	Nom du schéma contenant le domaine
domain_name	sql_identifier	Nom du domaine
table_catalog	sql_identifier	Nom de la base de données contenant la table (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma contenant la table
table_name	sql_identifier	Nom de la table
column_name	sql_identifier	Nom de la colonne

34.10. column_privileges

La vue column_privileges identifie tous les droits octroyés sur les colonnes à un rôle couramment actif ou par un rôle couramment actif. Il existe une ligne pour chaque combinaison colonne, donneur (*grantor*) et receveur (*grantee*).

Si un droit a été donné sur une table entière, il s'affichera dans cette vue comme un droit sur chaque colonne, mais seulement pour les types de droits où la granularité par colonne est possible : SELECT, INSERT, UPDATE, REFERENCES.

Tableau 34.8. Colonnes de column_privileges

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle ayant accordé le privilège
grantee	sql_identifier	Nom du rôle receveur
table_catalog	sql_identifier	Nom de la base de données qui contient la table qui contient la colonne (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table qui contient la colonne
table_name	sql_identifier	Nom de la table qui contient la colonne
column_name	sql_identifier	Nom de la colonne
privilege_type	character_data	Type de privilège : SELECT, INSERT, UPDATE ou REFERENCES
is_grantable	yes_or_no	YES si le droit peut être accordé, NO sinon

34.11. column_udt_usage

La vue column_udt_usage identifie toutes les colonnes qui utilisent les types de données possédés par un rôle actif. Avec PostgreSQL™, les types de données internes se comportent comme des types utilisateur, ils apparaissent aussi ici. Voir aussi la Section 34.12, « columns » pour plus de détails.

Tableau 34.9. Colonnes de column_udt_usage

Nom	Type de données	Description
udt_catalog	sql_identifier	Nom de la base de données dans laquelle le type de donnée de la colonne (le type sous-jacent du domaine, si applicable) est défini (toujours la base de données courante).
udt_schema	sql_identifier	Nom du schéma dans lequel le type de donnée de la colonne (le type sous-jacent du domaine, si applicable) est défini.
udt_name	sql_identifier	Nom du type de données de la colonne (le type sous-jacent du domaine, si applicable).
table_catalog	sql_identifier	Nom de la base de données contenant la table (toujours la base de données courante).
table_schema	sql_identifier	Nom du schéma contenant la table.
table_name	sql_identifier	Nom de la table.
column_name	sql_identifier	Nom de la colonne.

34.12. columns

La vue columns contient des informations sur toutes les colonnes de table (et colonnes de vue) de la base. Les colonnes système (oid, etc.) ne sont pas incluses. Seules les colonnes auxquelles l'utilisateur a accès (par propriété ou par privilèges) sont affichées.

Tableau 34.10. Colonnes de columns

Nom	Type de données	Description
table_catalog	sql_identifier	Nom de la base de données contenant la table (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma contenant la table
table_name	sql_identifier	Nom de la table
column_name	sql_identifier	Nom de la colonne
ordinal_position	cardinal_number	Position de la colonne dans la table (la numérotation commençant à 1)
column_default	character_data	Expression par défaut de la colonne
is_nullable	yes_or_no	YES si la colonne est *NULLable* (elle admet une absence de valeur), NO dans le cas contraire. La contrainte NOT NULL n'est pas la seule façon de définir qu'une colonne n'est pas *NULLable*.
data_type	character_data	Le type de données de la colonne, s'il s'agit d'un type interne ou ARRAY s'il s'agit d'un tableau (dans ce cas, voir la vue element_types), USER-DEFINED dans les autres cas (le type est alors identifié dans udt_name et colonnes associées). Si la colonne est fondée sur un domaine, cette colonne est une référence au type sous-jacent du domaine (et le domaine est identifié dans domain_name et colonnes associées).
character_maximum_length	cardinal_number	Si data_type identifie un type chaîne de caractères ou chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximale n'a été déclarée.
character_octet_length	cardinal_number	Si data_type identifie un type caractère, la longueur maximale en octets (bytes) d'un datum ; NULL pour tous les autres types de données. La longueur maximale en octets dépend de la longueur maximum déclarée en caractères (voir ci-dessus) et l'encodage du serveur.
numeric_precision	cardinal_number	Si data_type identifie un type numérique, cette colonne contient la précision (déclarée ou implicite) du type pour ce domaine. La précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en

Nom	Type de données	Description
		binaire (base 2) comme indiqué dans la colonne `numeric_precision_radix`. Pour tous les autres types de données, la colonne est NULL.
`numeric_precision_radix`	cardinal_number	Si `data_type` identifie un type numérique, cette colonne indique dans quelle base les valeurs des colonnes `numeric_precision` et `numeric_scale` sont exprimées. La valeur est 2 ou 10. Pour tous les autres types de données, cette colonne est NULL.
`numeric_scale`	cardinal_number	Si `data_type` identifie un type numeric exact, cette colonne contient l'échelle (déclarée ou implicite) du type pour ce domaine. L'échelle indique le nombre de chiffres significatifs à la droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme indiqué dans la colonne `numeric_precision_radix`. Pour tous les autres types de données, cette colonne est NULL.
`datetime_precision`	cardinal_number	Si `data_type` identifie une date, une heure, un horodatage ou un interval, cette colonne contient la précision en secondes (déclarée ou implicite) pour cet attribut, c'est-à-dire le nombre de chiffres décimaux suivant le point décimal de la valeur en secondes. Pour tous les autres types de données, cette colonne est NULL.
`interval_type`	character_data	Pas encore implanté
`interval_precision`	character_data	Pas encore implanté
`character_set_catalog`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`character_set_schema`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`character_set_name`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`collation_catalog`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`collation_schema`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`collation_name`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`domain_catalog`	sql_identifier	Si la colonne a un type domaine, le nom de la base de données où le type est défini (toujours la base de données courante), sinon NULL.
`domain_schema`	sql_identifier	Si la colonne a un type domaine, le nom du schéma où le domaine est défini, sinon NULL.
`domain_name`	sql_identifier	Si la colonne a un type de domaine, le nom du domaine, sinon NULL.
`udt_catalog`	sql_identifier	Nom de la base de données où le type de données de la colonne (le type sous-jacent du domaine, si applicable) est défini (toujours la base de données courante).
`udt_schema`	sql_identifier	Nom du schéma où le type de données de la colonne (le type sous-jacent du domaine, si applicable) est défini.
`udt_name`	sql_identifier	Nom du type de données de la colonne (le type sous-jacent du domaine, si applicable).
`scope_catalog`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`scope_schema`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`scope_name`	sql_identifier	S'applique à une fonctionnalité non disponible dans Post-

Nom	Type de données	Description
		greSQL™.
maximum_cardinality	cardinal_number	Toujours NULL car les tableaux ont toujours une cardinalité maximale illimitée avec PostgreSQL™.
dtd_identifier	sql_identifier	Un identifiant du descripteur du type de données de la colonne, unique parmi les descripteurs de type de données contenus dans la table. Ceci est principalement utile pour joindre d'autres instances de ces identifiants. (Le format spécifique de l'identifiant n'est pas défini et rien ne permet d'assurer qu'il restera inchangé dans les versions futures.)
is_self_referencing	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
is_identity	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
identity_generation	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
identity_start	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
identity_increment	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
identity_maximum	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
identity_minimum	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
identity_cycle	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
is_generated	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
generation_expression	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
is_updatable	yes_or_no	YES si la colonne est actualisable, NO dans le cas contraire (les colonnes des tables sont toujours modifiables, les colonnes des vues ne le sont pas nécessairement).

Puisqu'en SQL les possibilités de définir les types de données sont nombreuses, et que PostgreSQL™ offre des possibilités supplémentaires, leur représentation dans le schéma d'information peut s'avérer complexe.

La colonne data_type est supposée identifier le type de données interne sous-jacent de la colonne. Avec PostgreSQL™, cela signifie que le type est défini dans le schéma du catalogue système pg_catalog. Cette colonne est utile si l'application sait gérer les types internes (par exemple, formater les types numériques différemment ou utiliser les données dans les colonnes de précision). Les colonnes udt_name, udt_schema et udt_catalog identifient toujours le type de données sous-jacent de la colonne même si la colonne est basée sur un domaine.

Puisque PostgreSQL™ traite les types internes comme des types utilisateur, les types internes apparaissent aussi ici. Il s'agit d'une extension du standard SQL.

Toute application conçue pour traiter les données en fonction du type peut utiliser ces colonnes, car, dans ce cas, il importe peu de savoir si la colonne est effectivement fondée sur un domaine. Si la colonne est fondée sur un domaine, l'identité du domaine est stockée dans les colonnes domain_name, domain_schema et domain_catalog. Pour assembler les colonnes avec leurs types de données associés et traiter les domaines comme des types séparés, on peut écrire coalesce(domain_name, udt_name), etc.

34.13. constraint_column_usage

La vue constraint_column_usage identifie toutes les colonnes de la base de données courante utilisées par des contraintes. Seules sont affichées les colonnes contenues dans une table possédée par un rôle connecté. Pour une contrainte de vérification, cette vue identifie les colonnes utilisées dans l'expression de la vérification. Pour une contrainte de clé étrangère, cette vue identifie les colonnes que la clé étrangère référence. Pour une contrainte d'unicité ou de clé primaire, cette vue identifie les colonnes

contraintes.

Tableau 34.11. Colonnes de `constraint_column_usage`

Nom	Type de données	Description
`table_catalog`	sql_identifier	Nom de la base de données contenant la table contenant la colonne utilisée par certaines contraintes (toujours la base de données courante)
`table_schema`	sql_identifier	Nom du schéma contenant la table contenant la colonne utilisée par certaines contraintes
`table_name`	sql_identifier	Nom de la table contenant la colonne utilisée par certaines contraintes
`column_name`	sql_identifier	Nom de la colonne utilisée par certaines contraintes
`constraint_catalog`	sql_identifier	Nom de la base de données contenant la contrainte (toujours la base de données courante)
`constraint_schema`	sql_identifier	Nom du schéma contenant la contrainte
`constraint_name`	sql_identifier	Nom de la contrainte

34.14. `constraint_table_usage`

La vue `constraint_table_usage` identifie toutes les tables de la base de données courante utilisées par des contraintes et possédées par un rôle actuellement activé. (Cela diffère de la vue `table_constraints` qui identifie toutes les contraintes et la table où elles sont définies.) Pour une contrainte de clé étrangère, cette vue identifie la table que la clé étrangère référence. Pour une contrainte d'unicité ou de clé primaire, cette vue identifie simplement la table à laquelle appartient la contrainte. Les contraintes de vérification et les contraintes de non nullité (NOT NULL) ne sont pas incluses dans cette vue.

Tableau 34.12. Colonnes de `constraint_table_usage`

Nom	Type de données	Description
`table_catalog`	sql_identifier	Nom de la base de données contenant la table utilisée par quelques contraintes (toujours la base de données courante)
`table_schema`	sql_identifier	Nom du schéma contenant la table utilisée par quelque contrainte
`table_name`	sql_identifier	Nom de la table utilisée par quelque contrainte
`constraint_catalog`	sql_identifier	Nom de la base de données contenant la contrainte (toujours la base de données courante)
`constraint_schema`	sql_identifier	Nom du schéma contenant la contrainte
`constraint_name`	sql_identifier	Nom de la contrainte

34.15. `data_type_privileges`

La vue `data_type_privileges` identifie tous les descripteurs de type de données auxquels l'utilisateur a accès, parce qu'il en est le propriétaire ou parce qu'il dispose de quelque droit sur le descripteur. Un descripteur de type de données est créé lorsqu'un type de données est utilisé dans la définition d'une colonne de table, d'un domaine ou d'une fonction (en tant que paramètre ou code de retour). Il stocke alors quelques informations sur l'utilisation du type de données (par exemple la longueur maximale déclarée, si applicable). Chaque descripteur de type de données se voit affecter un identifiant unique parmi les descripteurs de type de données affectés à un objet (table, domaine, fonction). Cette vue n'est probablement pas utile pour les applications, mais elle est utilisée pour définir d'autres vues dans le schéma d'information.

Tableau 34.13. Colonnes de `data_type_privileges`

Nom	Type de données	Description
`object_catalog`	sql_identifier	Nom de la base de données contenant l'objet décrit (toujours la base de données courante)

Nom	Type de données	Description
object_schema	sql_identifier	Nom du schéma contenant l'objet décrit
object_name	sql_identifier	Nom de l'objet décrit
object_type	character_data	Le type d'objet décrit : fait partie de TABLE (le descripteur de type de données concerne une colonne de cette table), DOMAIN (le descripteur concerne ce domaine), ROUTINE (le descripteur est lié à un type de paramètre ou de code de retour de cette fonction).
dtd_identifier	sql_identifier	L'identifiant du descripteur de type de données, unique parmi les descripteurs de type de données pour le même objet.

34.16. domain_constraints

La vue domain_constraints contient toutes les contraintes appartenant aux domaines définis dans la base courante.

Tableau 34.14. Colonnes de domain_constraints

Nom	Type de données	Description
constraint_catalog	sql_identifier	Nom de la base de données contenant la contrainte (toujours la base de données courante)
constraint_schema	sql_identifier	Nom du schéma contenant la contrainte
constraint_name	sql_identifier	Nom de la contrainte
domain_catalog	sql_identifier	Nom de la base de données contenant le domaine (toujours la base de données courante)
domain_schema	sql_identifier	Nom du schéma contenant le domaine
domain_name	sql_identifier	Nom du domaine
is_deferrable	yes_or_no	YES si la vérification de la contrainte peut être différée, NO sinon
initially_deferred	yes_or_no	YES si la vérification de la contrainte, qui peut être différée, est initialement différée, NO sinon

34.17. domain_udt_usage

La vue domain_udt_usage identifie tous les domaines utilisant les types de données possédés par un rôle actif. Sous PostgreSQL™, les types de données internes se comportent comme des types utilisateur. Ils sont donc inclus ici.

Tableau 34.15. Colonnes de domain_udt_usage

Nom	Type de données	Description
udt_catalog	sql_identifier	Nom de la base de données de définition du type de données domaine (toujours la base de données courante)
udt_schema	sql_identifier	Nom du schéma de définition du type de données domaine
udt_name	sql_identifier	Nom du type de données domaine
domain_catalog	sql_identifier	Nom de la base de données contenant le domaine (toujours la base de données courante)
domain_schema	sql_identifier	Nom du schéma contenant le domaine
domain_name	sql_identifier	Nom du domaine

34.18. domains

La vue domains contient tous les domaines définis dans la base de données courante.

Tableau 34.16. Colonnes de `domains`

Nom	Type de données	Description
`domain_catalog`	sql_identifier	Nom de la base de données contenant le domaine (toujours la base de données courante)
`domain_schema`	sql_identifier	Nom du schéma contenant le domaine
`domain_name`	sql_identifier	Nom du domaine
`data_type`	character_data	Type de données du domaine s'il s'agit d'un type interne, ou `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), sinon `USER-DEFINED` (dans ce cas, le type est identifié dans `udt_name` et comprend des colonnes associées).
`character_maximum_length`	cardinal_number	Si le domaine a un type caractère ou chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximale n'a été déclarée.
`character_octet_length`	cardinal_number	Si le domaine a un type caractère, la longueur maximale en octets (bytes) d'un datum ; NULL pour tous les autres types de données. La longueur maximale en octets dépend de la longueur maximum déclarée en caractères (voir ci-dessus) et l'encodage du serveur.
`character_set_catalog`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`character_set_schema`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`character_set_name`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`collation_catalog`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`collation_schema`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`collation_name`	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`numeric_precision`	cardinal_number	Si le domaine a un type numérique, cette colonne contient la précision (déclarée ou implicite) du type de cette colonne. Cette précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme indiqué dans la colonne `numeric_precision_radix`. Pour les autres types de données, cette colonne est NULL.
`numeric_precision_radix`	cardinal_number	Si le domaine a un type numérique, cette colonne indique la base des valeurs des colonnes `numeric_precision` et `numeric_scale`. La valeur est soit 2 soit 10. Pour tous les autres types de données, cette colonne est NULL.
`numeric_scale`	cardinal_number	Si le domaine contient un type numeric, cette colonne contient l'échelle (déclarée ou implicite) du type pour cette colonne. L'échelle indique le nombre de chiffres significatifs à droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme indiqué dans la colonne `numeric_precision_radix`. Pour tous les autres types de données, cette colonne est NULL.
`datetime_precision`	cardinal_number	Si le domaine contient un type date, heure ou intervalle, la précision déclarée ; NULL pour les autres types de données ou si la précision n'a pas été déclarée.
`interval_type`	character_data	Pas encore implanté

Nom	Type de données	Description
interval_precision	character_data	Pas encore implanté
domain_default	character_data	Expression par défaut du domaine
udt_catalog	sql_identifier	Nom de la base de données dans laquelle est défini le type de données domaine (toujours la base de données courante)
udt_schema	sql_identifier	Nom du schéma où le type de données domaine est défini
udt_name	sql_identifier	Nom du type de données domaine
scope_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
maximum_cardinality	cardinal_number	Toujours NULL car les tableaux n'ont pas de limite maximale de cardinalité dans PostgreSQL™
dtd_identifier	sql_identifier	Un identifiant du descripteur de type de données du domaine, unique parmi les descripteurs de type de données restant dans le domaine (ce qui est trivial car un domaine contient seulement un descripteur de type de données). Ceci est principalement utile pour joindre d'autres instances de tels identifiants (le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il restera identique dans les versions futures).

34.19. element_types

La vue element_types contient les descripteurs de type de données des éléments de tableaux. Lorsqu'une colonne de table, domaine, paramètre de fonction ou code de retour de fonction est définie comme un type tableau, la vue respective du schéma d'information contient seulement ARRAY dans la colonne data_type. Pour obtenir des informations sur le type d'élément du tableau, il est possible de joindre la vue respective avec cette vue. Par exemple, pour afficher les colonnes d'une table avec les types de données et les types d'élément de tableau, si applicable, on peut écrire :

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

Cette vue n'inclut que les objets auxquels l'utilisateur courant a accès, parce que propriétaire ou disposant de quelque privilège.

Tableau 34.17. Colonnes de element_types

Nom	Type de données	Description
object_catalog	sql_identifier	Nom de la base de données contenant l'objet qui utilise le tableau décrit (toujours la base de données courante)
object_schema	sql_identifier	Nom du schéma contenant l'objet utilisant le tableau décrit
object_name	sql_identifier	Nom de l'objet utilisant le tableau décrit
object_type	character_data	Le type de l'objet utilisant le tableau décrit : il fait partie de TABLE (le tableau est utilisé par une colonne de cette table), DOMAIN (le tableau est utilisé par ce domaine), ROUTINE (le tableau est utilisé par un paramètre ou le type du code de retour de cette fonction).
collection_type_identifier	sql_identifier	L'identifiant du descripteur de type de données du tableau en cours de description. Utilisez cette colonne en la joignant avec les colonnes dtd_identifier des autres vues du

Nom	Type de données	Description
		schéma d'informations.
data_type	character_data	Le type de données des éléments du tableau s'il s'agit d'un type interne, sinon USER-DEFINED (dans ce cas, le type est identifié comme udt_name et dispose de colonnes associées).
character_maximum_length	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
character_octet_length	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
character_set_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
character_set_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
character_set_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
numeric_precision	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
numeric_precision_radix	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
numeric_scale	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
datetime_precision	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
interval_type	character_data	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
interval_precision	character_data	Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL™
domain_default	character_data	Pas encore implanté
udt_catalog	sql_identifier	Nom de la base de données pour lequel le type de données est défini (toujours la base de données courante)
udt_schema	sql_identifier	Nom du schéma dans lequel est défini le type de données des éléments
udt_name	sql_identifier	Nom du type de données des éléments
scope_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.

Nom	Type de données	Description
maximum_cardinality	cardinal_number	Toujours NULL car les tableaux n'ont pas de limite maximale de cardinalité dans PostgreSQL™
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the element. This is currently not useful.

34.20. enabled_roles

La vue enabled_roles identifie les « rôles actuellement actifs ». Les rôles actifs sont définis récursivement comme l'utilisateur courant avec tous les rôles qui ont été donnés aux rôles activés avec l'héritage automatique. En d'autres termes, ce sont les rôles dont l'utilisateur courant est automatiquement membre, par héritage direct ou indirect.

Pour la vérification des permissions, l'ensemble des « rôles applicables » est appliqué, ce qui peut être plus large que l'ensemble des rôles actifs. Il est, de ce fait, généralement préférable d'utiliser la vue applicable_roles à la place de celle-ci ; voir aussi là.

Tableau 34.18. Colonnes de enabled_roles

Nom	Type de données	Description
role_name	sql_identifier	Nom d'un rôle

34.21. foreign_data_wrapper_options

La vue foreign_data_wrapper_options contient toutes les options définies par les wrappers de données distantes dans la base de données en cours. Seuls les wrappers accessibles par l'utilisateur connecté sont affichés (qu'il soit propriétaire ou qu'il ait des droits dessus).

Tableau 34.19. Colonnes de foreign_data_wrapper_options

Nom	Type de données	Description
foreign_data_wrapper_catalog	sql_identifier	Nom de la base de données dans laquelle est défini le wrapper de données distantes (toujours la base de connexion)
foreign_data_wrapper_name	sql_identifier	Nom du wrapper
option_name	sql_identifier	Nom d'une option
option_value	character_data	Valeur de l'option

34.22. foreign_data_wrappers

La vue foreign_data_wrappers contient tous les wrappers de données distantes définis dans le base de données en cours. Seuls sont affichés les wrappers pour lesquels l'utilisateur connecté a accès (qu'il soit propriétaire ou qu'il ait des droits dessus).

Tableau 34.20. Colonnes de foreign_data_wrappers

Nom	Type de données	Description
foreign_data_wrapper_catalog	sql_identifier	Nom de la base de données contenant le wrapper de données distantes (toujours la base de données en cours)
foreign_data_wrapper_name	sql_identifier	Nom du wrapper
authorization_identifier	sql_identifier	Nom du propriétaire du serveur distant
library_name	character_data	Nom du fichier de la bibliothèque implémentant ce wrapper
foreign_data_wrapper_language	character_data	Langage utilisé pour implémenter ce wrapper

34.23. foreign_server_options

La vue foreign_server_options contient toutes les options définies pour les serveurs distants de la base de données en cours. Ne sont affichés que les serveurs distants pour lesquels l'utilisateur connecté a des droits (qu'il soit propriétaire ou qu'il ait quelques droits dessus).

Tableau 34.21. Colonnes de foreign_server_options

Nom	Type de données	Description
foreign_server_catalog	sql_identifier	Nom de la base de données contenant le serveur distant (toujours la base de données en cours)
foreign_server_name	sql_identifier	Nom du serveur distant
option_name	sql_identifier	Nom d'une option
option_value	character_data	Valeur de l'option

34.24. foreign_servers

La vue foreign_servers contient tous les serveurs distants définis dans la base en cours. Ne sont affichés que les serveurs distants pour lesquels l'utilisateur connecté a des droits (qu'il soit propriétaire ou qu'il ait quelques droits dessus).

Tableau 34.22. Colonnes de foreign_servers

Nom	Type de données	Description
foreign_server_catalog	sql_identifier	Nom de la base de données dans laquelle ce serveur distant est défini (toujours la base de données en cours)
foreign_server_name	sql_identifier	Nom du serveur distant
foreign_data_wrapper_catalog	sql_identifier	Nom de la base de données qui contient le wrapper de données distantes utilisé par le serveur distant (toujours la base de données en cours)
foreign_data_wrapper_name	sql_identifier	Nom du wrapper de données distantes utilisé par le serveur distant
foreign_server_type	character_data	Information sur le type de serveur distant, si indiqué lors de la création
foreign_server_version	character_data	Information sur la version de serveur distant, si indiqué lors de la création
authorization_identifier	sql_identifier	Nom du propriétaire du serveur distant

34.25. key_column_usage

La vue key_column_usage identifie toutes les colonnes de la base de données courante restreintes par une contrainte unique, clé primaire ou clé étrangère. Les contraintes de vérification ne sont pas incluses dans cette vue. Seules sont affichées les colonnes auxquelles l'utilisateur a accès, parce qu'il est le propriétaire de la table ou qu'il dispose de quelque privilège.

Tableau 34.23. Colonnes de key_column_usage

Nom	Type de données	Description
constraint_catalog	sql_identifier	Nom de la base de données contenant la contrainte (toujours la base de données courante)
constraint_schema	sql_identifier	Nom du schéma contenant la contrainte
constraint_name	sql_identifier	Nom de la contrainte
table_catalog	sql_identifier	Nom de la base de données contenant la table contenant la

Nom	Type de données	Description
		colonne contrainte (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma contenant la table contenant la colonne contrainte
table_name	sql_identifier	Nom de la table contenant la colonne contrainte
column_name	sql_identifier	Nom de la colonne contrainte
ordinal_position	cardinal_number	Position ordinale de la colonne dans la clé de contrainte (la numérotation commence à 1)
position_in_unique_constraint	cardinal_number	Pour une contrainte de type clé étrangère, la position ordinale de la colonne référencée dans sa contrainte d'unicité (la numérotation commence à 1) ; sinon null

34.26. parameters

La vue parameters contient des informations sur les paramètres (arguments) de toutes les fonctions de la base de données courante. Seules sont affichées les fonctions auxquelles l'utilisateur courant a accès, parce qu'il en est le propriétaire ou qu'il dispose de quelque privilège.

Tableau 34.24. Colonnes de parameters

Nom	Type de données	Description
specific_catalog	sql_identifier	Nom de la base de données contenant la fonction (toujours la base de données courante)
specific_schema	sql_identifier	Nom du schéma contenant la fonction
specific_name	sql_identifier	Le « nom spécifique » de la fonction. Voir la Section 34.33, « routines » pour plus d'informations.
ordinal_position	cardinal_number	Position ordinale du paramètre dans la liste des arguments de la fonction (la numérotation commence à 1)
parameter_mode	character_data	IN pour les paramètres en entrée, OUT pour les paramètres en sortie ou INOUT pour les paramètres en entrée/sortie.
is_result	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
as_locator	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
parameter_name	sql_identifier	Nom du paramètre ou NULL si le paramètre n'a pas de nom
data_type	character_data	Type de données du paramètre s'il s'agit d'un type interne, ou ARRAY s'il s'agit d'un tableau (dans ce cas, voir la vue element_types), sinon USER-DEFINED (dans ce cas, le type est identifié dans udt_name et dispose de colonnes associées).
character_maximum_length	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
character_octet_length	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
character_set_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
character_set_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
character_set_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans Post-

Nom	Type de données	Description
		greSQL™.
collation_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
numeric_precision	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
numeric_precision_radix	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
numeric_scale	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
datetime_precision	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
interval_type	character_data	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
interval_precision	character_data	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
udt_catalog	sql_identifier	Nom de la base de données sur laquelle est défini le paramètre (toujours la base de données courante)
udt_schema	sql_identifier	Nom du schéma dans lequel est défini le type de données du paramètre
udt_name	sql_identifier	Nom du type de données du paramètre
scope_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
maximum_cardinality	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL™
dtd_identifier	sql_identifier	Un identifiant du descripteur de type de données du paramètre, unique parmi les descripteurs de type de données restant dans la fonction. Ceci est principalement utile pour réaliser une jointure avec les autres instances de tels identifiants (le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il reste identique dans les prochaines versions).

34.27. referential_constraints

La vue referential_constraints contient toutes les contraintes référentielles (clés étrangères) au sein de la base de données courante. Seuls sont affichés les contraintes pour lesquelles l'utilisateur connecté a accès en écriture sur la table référençante (parce qu'il est le propriétaire ou qu'il a d'autres droits que SELECT).

Tableau 34.25. Colonnes de referential_constraints

Nom	Type de données	Description
constraint_catalog	sql_identifier	Nom de la base de données contenant la contrainte (toujours la base de données courante)
constraint_schema	sql_identifier	Nom du schéma contenant la contrainte
constraint_name	sql_identifier	Nom de la contrainte
unique_constraint_catalog	sql_identifier	Nom de la base de données contenant la contrainte d'unicité ou de clé primaire que la contrainte de clé étrangère référence (toujours la base de données courante)

Nom	Type de données	Description
unique_constraint_schema	sql_identifier	Nom du schéma contenant la contrainte d'unicité ou de clé primaire que la contrainte de clé étrangère référence
unique_constraint_name	sql_identifier	Nom de la contrainte d'unicité ou de clé primaire que la contrainte de clé étrangère référence
match_option	character_data	Correspondances de la contrainte de clé étrangère : FULL, PARTIAL ou NONE.
update_rule	character_data	Règle de mise à jour associée à la contrainte de clé étrangère : CASCADE, SET NULL, SET DEFAULT, RESTRICT ou NO ACTION.
delete_rule	character_data	Règle de suppression associée à la contrainte de clé étrangère : CASCADE, SET NULL, SET DEFAULT, RESTRICT ou NO ACTION.

34.28. role_column_grants

La vue role_column_grants identifie tous les privilèges de colonne octroyés pour lesquels le donneur ou le bénéficiaire est un rôle actuellement actif. Plus d'informations sous column_privileges. La seule différence réelle entre cette vue et column_privileges est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour PUBLIC.

Tableau 34.26. Colonnes de role_column_grants

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a octroyé le privilège
grantee	sql_identifier	Nom du rôle bénéficiaire
table_catalog	sql_identifier	Nom de la base de données qui contient la table qui contient la colonne (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table qui contient la colonne
table_name	sql_identifier	Nom de la table qui contient la colonne
column_name	sql_identifier	Nom de la colonne
privilege_type	character_data	Type de privilège : SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES ou TRIGGER
is_grantable	yes_or_no	YES si le droit peut être transmis, NO sinon

34.29. role_routine_grants

La vue role_routine_grants identifie tous les privilèges de routine octriyés lorsque le donneur ou le bénéficiaire est un rôle actif. Plus d'informations sous routine_privileges. La seule différence réelle entre cette vue et routine_privileges est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour PUBLIC.

Tableau 34.27. Colonnes de role_routine_grants

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a octroyé le privilège
grantee	sql_identifier	Nom du rôle bénéficiaire
specific_catalog	sql_identifier	Nom de la base de données qui contient la fonction (toujours la base de données courante)
specific_schema	sql_identifier	Nom du schéma qui contient la fonction
specific_name	sql_identifier	Le « nom spécifique » de la fonction. Voir la Section 34.33, « routines » pour plus d'informations.
routine_catalog	sql_identifier	Nom de la base de données qui contient la fonction

Nom	Type de données	Description
		(toujours la base de données courante)
routine_schema	sql_identifier	Nom du schéma qui contient la fonction
routine_name	sql_identifier	Nom de la fonction (peut être dupliqué en cas de surcharge)
privilege_type	character_data	Toujours EXECUTE (seul type de privilège sur les fonctions)
is_grantable	yes_or_no	YES si le droit peut être transmis, NO sinon

34.30. role_table_grants

La vue role_table_grants identifie tous les privilèges de tables octroyés lorsque le donneur ou le bénéficiaire est un rôle actif. Plus d'informations sous table_privileges. La seule différence réelle entre cette vue et table_privileges est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour PUBLIC.

Tableau 34.28. Colonnes de role_table_grants

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a octroyé le privilège
grantee	sql_identifier	Nom du rôle bénéficiaire
table_catalog	sql_identifier	Nom de la base de données qui contient la table (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table
table_name	sql_identifier	Nom de la table
privilege_type	character_data	Type du privilège : SELECT, DELETE, INSERT, UPDATE, REFERENCES ou TRIGGER
is_grantable	yes_or_no	YES si le droit peut être transmis, NO sinon
with_hierarchy	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.

34.31. role_usage_grants

La vue role_usage_grants identifie les privilèges d'USAGE sur différents types d'objets où le donneur ou le receveur des droits est un rôle actuellement activé. Plus d'informations sous usage_privileges. Dans le futur, cette vue pourrait contenir des informations plus utiles. La seule différence réelle entre cette vue et usage_privileges est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour PUBLIC.

Tableau 34.29. Colonnes de role_usage_grants

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a octroyé le privilège
grantee	sql_identifier	Nom du rôle bénéficiaire
object_catalog	sql_identifier	Nom de la base de données qui contient l'objet (toujours la base de données courante)
object_schema	sql_identifier	Nom du schéma qui contient l'objet, if applicable, sinon une chaîne vide
object_name	sql_identifier	Nom de l'objet
object_type	character_data	DOMAIN, FOREIGN DATA WRAPPER ou FOREIGN SERVER
privilege_type	character_data	Toujours USAGE
is_grantable	yes_or_no	YES si le droit peut être transmis, NO sinon

34.32. routine_privileges

La vue routine_privileges identifie tous les droits sur les fontions à un rôle actuellement activé ou par un rôle actuellement activé. Il existe une ligne pour chaque combinaison fonction, donneur, bénéficiaire.

Tableau 34.30. Colonnes de routine_privileges

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a accordé le privilège
grantee	sql_identifier	Nom du rôle bénéficiaire
specific_catalog	sql_identifier	Nom de la base de données qui contient la fonction (toujours la base de données courante)
specific_schema	sql_identifier	Nom du schéma qui contient la fonction
specific_name	sql_identifier	Le « nom spécifique » de la fonction. Voir la Section 34.33, « routines » pour plus d'informations.
routine_catalog	sql_identifier	Nom de la base de données qui contient la fonction (toujours la base de données courante)
routine_schema	sql_identifier	Nom du schéma qui contient la fonction
routine_name	sql_identifier	Nom de la fonction (peut être dupliqué en cas de surcharge)
privilege_type	character_data	Toujours EXECUTE (seul priilège de fonctions)
is_grantable	yes_or_no	YES si le droit peut être transmis, NO sinon

34.33. routines

La vue routines contient toutes les fonctions de la base de données courante. Seules sont affichées les fonctions auxquelles l'utilisateur courant a accès (qu'il en soit le propriétaire ou dispose de de privilèges).

Tableau 34.31. Colonnes de routines

Nom	Type de données	Description
specific_catalog	sql_identifier	Nom de la base de données qui contient la fonction (toujours la base de données courante)
specific_schema	sql_identifier	Nom du schéma qui contient la fonction
specific_name	sql_identifier	Le « nom spécifique » de la fonction. Ce nom identifie de façon unique la fonction dans le schéma, même si le nom réel de la fonction est surchargé. Le format du nom spécifique n'est pas défini, il ne devrait être utilisé que dans un but de comparaison avec d'autres instances de noms spécifiques de routines.
routine_catalog	sql_identifier	Nom de la base de données qui contient la fonction (toujours la base de données courante)
routine_schema	sql_identifier	Nom du schéma qui contient la fonction
routine_name	sql_identifier	Nom de la fonction (peut être dupliqué en cas de surcharge)
routine_type	character_data	Toujours FUNCTION (dans le futur, il pourrait y avoir d'autres types de routines)
module_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
module_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
module_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
udt_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.

Nom	Type de données	Description
udt_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
udt_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
data_type	character_data	Type de données de retour de la fonction s'il est interne, ARRAY s'il s'agit d'un tableau (dans ce cas, voir la vue element_types), sinon USER-DEFINED (dans ce cas, le type est identifié dans type_udt_name et dispose de colonnes associées).
character_maximum_length	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
character_octet_length	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
character_set_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
character_set_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
character_set_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
collation_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
numeric_precision	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
numeric_precision_radix	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
numeric_scale	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
datetime_precision	cardinal_number	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
interval_type	character_data	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
interval_precision	character_data	Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL™
type_udt_catalog	sql_identifier	Nom de la base de données dans laquelle est défini le type de données de retour de la fonction (toujours la base de données courante)
type_udt_schema	sql_identifier	Nom du schéma dans lequel est défini le type de données de retour de la fonction
type_udt_name	sql_identifier	Nom du type de données de retour de la fonction
scope_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
scope_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
maximum_cardinality	cardinal_number	Toujours NULL car il n'y a pas de limite maximale à la cardinalité des tableaux dans PostgreSQL™
dtd_identifier	sql_identifier	Un identifiant du descripteur de type de données du type de

Nom	Type de données	Description
		données retour, unique parmi les descripteurs de type de données de la fonction. Ceci est principalement utile pour la jointure avec d'autres instances de tels identifiants (le format spécifique de l'identifiant n'est pas défini et il n'est pas certain qu'il restera identique dans les versions futures).
routine_body	character_data	Si la fonction est une fonction SQL, alors SQL, sinon EXTERNAL.
routine_definition	character_data	Le texte source de la fonction (NULL si la fonction n'appartient pas à un rôle actif). (Le standard SQL précise que cette colonne n'est applicable que si routine_body est SQL, mais sous PostgreSQL™ ce champ contient tout texte source précisé à la création de la fonction.)
external_name	character_data	Si la fonction est une fonction C, le nom externe (link symbol) de la fonction ; sinon NULL. (Il s'agit de la même valeur que celle affichée dans routine_definition).
external_language	character_data	Le langage d'écriture de la fonction
parameter_style	character_data	Toujours GENERAL (le standard SQL définit d'autres styles de paramètres qui ne sont pas disponibles avec PostgreSQL™).
is_deterministic	yes_or_no	Si la fonction est déclarée immuable (déterministe dans le standard SQL), alors YES, sinon NO. (Les autres niveaux de volatilité disponibles dans PostgreSQL™ ne peuvent être récupérés via le schéma d'informations).
sql_data_access	character_data	Toujours MODIFIES, ce qui signifie que la fonction peut modifier les données SQL. Cette information n'est pas utile sous PostgreSQL™.
is_null_call	yes_or_no	Si la fonction renvoie automatiquement NULL si un de ces arguments est NULL, alors YES, sinon NO.
sql_path	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
schema_level_routine	yes_or_no	Toujours YES. (L'opposé serait une méthode d'un type utilisateur, fonctionnalité non disponible dans PostgreSQL™).
max_dynamic_result_sets	cardinal_number	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
is_user_defined_cast	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
is_implicitly_invocable	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
security_type	character_data	Si la fonction est exécutée avec les droits de l'utilisateur courant, alors INVOKER. Si la fonction est exécutée avec les droits de l'utilisateur l'ayant définie, alors DEFINER.
to_sql_specific_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
to_sql_specific_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
to_sql_specific_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
as_locator	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
created	time_stamp	S'applique à une fonctionnalité non disponible dans PostgreSQL™
last_altered	time_stamp	S'applique à une fonctionnalité non disponible dans PostgreSQL™

Nom	Type de données	Description
new_savepoint_level	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™
is_udt_dependent	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_from_data_type	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_as_locator	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_char_max_length	cardinal_number	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_char_octet_length	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_char_set_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_char_set_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_char_set_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_collation_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_collation_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_collation_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_numeric_precision	cardinal_number	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_numeric_precision_radix	cardinal_number	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_numeric_scale	cardinal_number	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_datetime_precision	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_interval_type	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_interval_precision	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_type_udt_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_type_udt_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_type_udt_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_scope_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_scope_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_scope_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_maximum_cardinality	cardinal_number	S'applique à une fonctionnalité non disponible dans PostgreSQL™
result_cast_dtd_identifier	sql_identifier	S'applique à une fonctionnalité non disponible dans Post-

Nom	Type de données	Description
		greSQL™

34.34. schemata

La vue schemata contient tous les schémas de la base de données courante dont un rôle actif est propriétaire.

Tableau 34.32. Colonnes de schemata

Nom	Type de données	Description
catalog_name	sql_identifier	Nom de la base de données dans laquelle se trouve le schéma (toujours la base de données courante)
schema_name	sql_identifier	Nom du schéma
schema_owner	sql_identifier	Nom du propriétaire du schéma
default_character_set_catalog	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
default_character_set_schema	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
default_character_set_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
sql_path	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.

34.35. sequences

La vue sequences contient toutes les séquences définies dans la base courante. Seules sont affichées les séquences auxquelles l'utilisateur courant a accès (qu'il en soit le propriétaire ou dispose de privilèges).

Tableau 34.33. Colonnes de sequences

Nom	Type de données	Description
sequence_catalog	sql_identifier	Nom de la base qui contient la séquence (toujours la base en cours)
sequence_schema	sql_identifier	Nom du schéma qui contient la séquence
sequence_name	sql_identifier	Nom de la séquence
data_type	character_data	Type de données de la séquence. Dans PostgreSQL™, c'est toujours bigint.
numeric_precision	cardinal_number	Cette colonne contient la précision (déclarée ou implicite) du type de données de la séquence (voir ci-dessus). La précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), suivant ce qui est indiqué dans la colonne numeric_precision_radix.
numeric_precision_radix	cardinal_number	Cette colonne indique dans quelle base les valeurs de la colonne numeric_precision et numeric_scale sont exprimées, 2 ou 10.
numeric_scale	cardinal_number	Cette colonne contient l'échelle (déclarée ou implicite) du type de données de la séquence (voir ci-dessus). L'échelle indique le nombre de chiffres significatifs à droite du point décimale. Elle peut être exprimée

Nom	Type de données	Description
		en décimal (base 10) ou en binaire (base 2), suivant ce qui est indiqué dans la colonne `numeric_precision_radix`.
`maximum_value`	cardinal_number	Pas encore implanté
`minimum_value`	cardinal_number	Pas encore implanté
`increment`	cardinal_number	Pas encore implanté
`cycle_option`	yes_or_no	Pas encore implanté

34.36. `sql_features`

La table `sql_features` contient des informations sur les fonctionnalités officielles définies dans le standard SQL et supportées par PostgreSQL™. Ce sont les mêmes informations que celles présentées dans l'Annexe D, Conformité SQL. D'autres informations de fond y sont disponibles.

Tableau 34.34. Colonnes de `sql_features`

Nom	Type de données	Description
`feature_id`	character_data	Chaîne identifiant la fonctionnalité
`feature_name`	character_data	Nom descriptif de la fonctionnalité
`sub_feature_id`	character_data	Chaîne identifiant la sous-fonctionnalité ou chaîne de longueur NULL s'il ne s'agit pas d'une sous-fonctionnalité
`sub_feature_name`	character_data	Nom descriptif de la sous-fonctionnalité ou chaîne de longueur NULL s'il ne s'agit pas d'une sous-fonctionnalité
`is_supported`	yes_or_no	`YES` si la fonctionnalité est complètement supportée par la version actuelle de PostgreSQL™, `NO` sinon
`is_verified_by`	character_data	Toujours NULL car le groupe de développement PostgreSQL™ ne réalise pas de tests formels sur la conformité des fonctionnalités
`comments`	character_data	Un commentaire éventuel sur le statut du support de la fonctionnalité

34.37. `sql_implementation_info`

La table `sql_inplementation_info` contient des informations sur différents aspects que le standard SQL laisse à la discrétion de l'implantation. Ces informations n'ont de réel intérêt que dans le contexte de l'interface ODBC ; les utilisateurs des autres interfaces leur trouveront certainement peu d'utilité. Pour cette raison, les éléments décrivant l'implantation ne sont pas décrits ici ; ils se trouvent dans la description de l'interface ODBC.

Tableau 34.35. Colonnes de `sql_implementation_info`

Nom	Type de données	Description
`implementation_info_id`	character_data	Chaîne identifiant l'élément d'information d'implantation
`implementation_info_name`	character_data	Nom descriptif de l'élément d'information d'implantation
`integer_value`	cardinal_number	Valeur de l'élément d'information d'implantation, ou NULL si la valeur est contenue dans la colonne `character_value`
`character_value`	character_data	Valeur de l'élément d'information d'implantation, ou NULL si la valeur est contenue dans la colonne `integer_value`
`comments`	character_data	Un commentaire éventuel de l'élément d'information d'implantation

34.38. sql_languages

La table sql_languages contient une ligne par langage lié au SQL supporté par PostgreSQL™. PostgreSQL™ supporte le SQL direct et le SQL intégré dans le C ; cette table ne contient pas d'autre information.

Tableau 34.36. Colonnes de sql_languages

Nom	Type de données	Description
sql_language_source	character_data	Le nom de la source de définition du langage ; toujours ISO 9075, c'est-à-dire le standard SQL
sql_language_year	character_data	L'année de l'approbation du standard dans sql_language_source ; actuellement 2003
sql_language_conformance	character_data	Le niveau de conformité au standard pour le langage. Pour ISO 9075:2003, c'est toujours CORE.
sql_language_integrity	character_data	Toujours NULL (cette valeur n'a d'intérêt que pour les versions précédentes du standard SQL).
sql_language_implementation	character_data	Toujours NULL
sql_language_binding_style	character_data	Le style de lien du langage, soit DIRECT soit EMBEDDED
sql_language_programming_language	character_data	Le langage de programmation si le style de lien est EMBEDDED, sinon NULL. PostgreSQL™ ne supporte que le langage C.

34.39. sql_packages

La table sql_packages contient des informations sur les paquets de fonctionnalités définis dans le standard SQL supportés par PostgreSQL™. On se référera à l'Annexe D, Conformité SQL pour des informations de base sur les paquets de fonctionnalités.

Tableau 34.37. Colonnes de sql_packages

Nom	Type de données	Description
feature_id	character_data	Chaîne identifiant le paquet
feature_name	character_data	Nom descriptif du paquet
is_supported	yes_or_no	YES si le paquet est complètement supporté par la version actuelle, NO sinon
is_verified_by	character_data	Toujours NULL car le groupe de développement de PostgreSQL™ ne réalise pas de tests formels pour la conformité des fonctionnalités
comments	character_data	Un commentaire éventuel sur l'état de support du paquet

34.40. sql_parts

La table sql_parts contient des informations sur les parties du standard SQL supportées par PostgreSQL™.

Tableau 34.38. Colonnes de sql_parts

Nom	Type de données	Description
feature_id	character_data	Une chaîne d'identification contenant le numéro de la partie
feature_name	character_data	Nom descriptif de la partie

Nom	Type de données	Description
`is_supported`	yes_or_no	`YES` si cette partie est complètement supportée par la version actuelle de PostgreSQL™, `NO` dans le cas contraire
`is_verified_by`	character_data	Toujours NULL, car les développeurs PostgreSQL™ ne testent pas officiellement la conformité des fonctionnalités
`comments`	character_data	Commentaires sur le statut du support de la partie

34.41. sql_sizing

La table `sql_sizing` contient des informations sur les différentes limites de tailles et valeurs maximales dans PostgreSQL™. Ces informations ont pour contexte principal l'interface ODBC ; les utilisateurs des autres interfaces leur trouveront probablement peu d'utilité. Pour cette raison, les éléments de taille individuels ne sont pas décrits ici ; ils se trouvent dans la description de l'interface ODBC.

Tableau 34.39. Colonnes de `sql_sizing`

Nom	Type de données	Description
`sizing_id`	cardinal_number	Identifiant de l'élément de taille
`sizing_name`	character_data	Nom descriptif de l'élément de taille
`supported_value`	cardinal_number	Valeur de l'élément de taille, ou 0 si la taille est illimitée ou ne peut pas être déterminée, ou NULL si les fonctionnalités pour lesquelles l'élément de taille est applicable ne sont pas supportées
`comments`	character_data	Un commentaire éventuel de l'élément de taille

34.42. sql_sizing_profiles

La table `sql_sizing_profiles` contient des informations sur les valeurs `sql_sizing` requises par différents profils du standard SQL. PostgreSQL™ ne garde pas trace des profils SQL, donc la table est vide.

Tableau 34.40. Colonnes de `sql_sizing_profiles`

Nom	Type de données	Description
`sizing_id`	cardinal_number	Identifiant de l'élément de taille
`sizing_name`	character_data	Nom descriptif de l'élément de taille
`profile_id`	character_data	Chaîne identifiant un profil
`required_value`	cardinal_number	La valeur requise par le profil SQL pour l'élément de taille, ou 0 si le profil ne place aucune limite sur l'élément de taille, ou NULL si le profil ne requiert aucune fonctionnalité pour laquelle l'élément de style est applicable
`comments`	character_data	Un commentaire éventuel sur l'élément de taille au sein du profil

34.43. table_constraints

La vue `table_constraints` contient toutes les contraintes appartenant aux tables possédées par l'utilisateur courant ou pour lesquelles l'utilisateur courant dispose de certains droits différents de SELECT.

Tableau 34.41. Colonnes de `table_constraints`

Nom	Type de données	Description
constraint_catalog	sql_identifier	Nom de la base de données qui contient la contrainte (toujours la base de données courante)
constraint_schema	sql_identifier	Nom du schéma qui contient la contrainte
constraint_name	sql_identifier	Nom de la contrainte
table_catalog	sql_identifier	Nom de la base de données qui contient la table (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table
table_name	sql_identifier	Nom de la table
constraint_type	character_data	Type de contrainte : CHECK, FOREIGN KEY, PRIMARY KEY ou UNIQUE
is_deferrable	yes_or_no	YES si la contrainte peut être différée, NO sinon
initially_deferred	yes_or_no	YES si la contrainte, qui peut être différée, est initialement différée, NO sinon

34.44. table_privileges

La vue table_privileges identifie tous les privilèges accordés, à un rôle actif ou par une rôle actif, sur des tables ou vues. Il y a une ligne par combinaison table, donneur, bénéficiaire.

Tableau 34.42. Colonnes de table_privileges

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a accordé le privilège
grantee	sql_identifier	Nom du rôle bénéficiaire
table_catalog	sql_identifier	Nom de la base de données qui contient la table (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table
table_name	sql_identifier	Nom de la table
privilege_type	character_data	Type de privilège : SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES ou TRIGGER
is_grantable	yes_or_no	YES si le droit peut être transmis, NO sinon
with_hierarchy	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™.

34.45. tables

La vue tables contient toutes les tables et vues définies dans la base de données courantes. Seules sont affichées les tables et vues auxquelles l'utilisateur courant a accès (parce qu'il en est le propriétaire ou qu'il possède certains privilèges).

Tableau 34.43. Colonnes de tables

Nom	Type de données	Description
table_catalog	sql_identifier	Nom de la base de données qui contient la table (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table
table_name	sql_identifier	Nom de la table
table_type	character_data	Type de table : BASE TABLE pour une table de base persistante (le type de table normal), VIEW pour une vue ou LOCAL TEMPORARY pour une table temporaire
self_referencing_column_name	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.

Nom	Type de données	Description
reference_generation	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
user_defined_type_catalog	sql_identifier	Si la table est une table typée, le nom de la base de données qui contient le type de données sous-jacent (toujours la base de données actuel), sinon NULL.
user_defined_type_schema	sql_identifier	Si la table est une table typée, le nom du schéma qui contient le type de données sous-jacent, sinon NULL.
user_defined_type_name	sql_identifier	Si la table est une table typée, le nom du type de données sous-jacent, sinon NULL.
is_insertable_into	yes_or_no	YES s'il est possible d'insérer des données dans la table, NO dans le cas contraire. (Il est toujours possible d'insérer des données dans une table de base, pas forcément dans les vues.)
is_typed	yes_or_no	YES si la table est une table typée, NO dans le cas contraire
commit_action	character_data	Si la table est temporaire, alors PRESERVE, sinon NULL. (Le standard SQL définit d'autres actions de validation pour les tables temporaires, actions qui ne sont pas supportées par PostgreSQL™.)

34.46. triggered_update_columns

Pour les triggers de la base de données actuelle qui spécifient une liste de colonnes (comme UPDATE OF colonne1, colonne2), la vue triggered_update_columns identifie ces colonnes. Les triggers qui ne spécifient pas une liste de colonnes ne sont pas inclus dans cette vue. Seules sont affichées les colonnes que l'utilisateur actuel possède ou que l'utilisateur a des droits autre que SELECT.

Tableau 34.44. Colonnes de triggered_update_columns

Nom	Type de données	Description
trigger_catalog	sql_identifier	Nom de la base de données qui contient le déclencheur (toujours la base de données courante)
trigger_schema	sql_identifier	Nom du schéma qui contient le déclencheur
trigger_name	sql_identifier	Nom du déclencheur
event_object_catalog	sql_identifier	Nom de la base de données qui contient la table sur laquelle est défini le déclencheur (toujours la base de données courante)
event_object_schema	sql_identifier	Nom du schéma qui contient la table sur laquelle est défini le déclencheur
event_object_table	sql_identifier	Nom de la table sur laquelle est défini le déclencheur
event_object_column	sql_identifier	Name of the column that the trigger is defined on

34.47. triggers

La vue triggers contient tous les triggers définis dans la base de données actuelles sur les tables que l'utilisateur actuel possède ou sur lesquels il a d'autres droits que le SELECT.

Tableau 34.45. Colonnes de triggers

Nom	Type de données	Description
trigger_catalog	sql_identifier	Nom de la base contenant le trigger (toujours la base de données actuelle)
trigger_schema	sql_identifier	Nom du schéma contenant le trigger
trigger_name	sql_identifier	Nom du trigger

Nom	Type de données	Description
event_manipulation	character_data	Événement qui déclenche le trigger (INSERT, UPDATE ou DELETE)
event_object_catalog	sql_identifier	Nom de la base contenant la table où le trigger est défini (toujours la base de données actuelle)
event_object_schema	sql_identifier	Nom du schéma qui contient la table où le trigger est défini
event_object_table	sql_identifier	Nom de la table où le trigger est défini
action_order	cardinal_number	Pas encore implanté
action_condition	character_data	La condition WHEN du trigger, NULL si aucun (NULL aussi si la table n'appartient pas à un rôle actuellement activé)
action_statement	character_data	Instruction exécutée par le déclencheur (actuellement toujours EXECUTE PROCEDURE *function*(...))
action_orientation	character_data	Indique si le déclencheur est exécuté une fois par ligne traitée ou une fois par instruction (ROW ou STATEMENT)
condition_timing	character_data	Moment de l'exécution du déclencheur (BEFORE ou AFTER)
condition_reference_old_table	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
condition_reference_new_table	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
condition_reference_old_row	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
condition_reference_new_row	sql_identifier	S'applique à une fonctionnalité non disponible dans PostgreSQL™
created	time_stamp	S'applique à une fonctionnalité non disponible dans PostgreSQL™

Les déclencheurs dans PostgreSQL™ ont deux incompatibilités avec le standard SQL qui affectent leur représentation dans le schéma d'information.

Premièrement, les noms des déclencheurs sont locaux à la table sous PostgreSQL™, et ne sont pas des objets du schéma indépendants. De ce fait, il peut exister des déclencheurs de même noms au sein d'un schéma, pour peu qu'ils s'occupent de tables différentes. (trigger_catalog et trigger_schema sont les champs qui décrivent effectivement la table sur laquelle est défini le déclencheur.)

Deuxièmement, les déclencheurs peuvent être définis pour s'exécuter sur plusieurs événements sous PostgreSQL™ (c'est-à-dire ON INSERT OR UPDATE) alors que le standard SQL n'en autorise qu'un. Si un déclencheur est défini pour s'exécuter sur plusieurs événements, il est représenté sur plusieurs lignes dans le schéma d'information, une pour chaque type d'événement.

En conséquence, la clé primaire de la vue triggers est en fait (trigger_catalog, trigger_schema, trigger_name, event_object_table, event_manipulation) et non (trigger_catalog, trigger_schema, trigger_name) comme le spécifie le standard SQL. Néanmoins, si les déclencheurs sont définis de manière conforme au standard SQL (des noms de déclencheurs uniques dans le schéma et un seul type d'événement par déclencheur), il n'y a pas lieu de se préoccuper de ces deux incompatibilités.

34.48. usage_privileges

La vue usage_privileges identifie les privilèges d'USAGE accordés sur différents objets à un rôle actif ou par un rôle actif. Sous PostgreSQL™, cela s'applique aux domaines. Puisqu'il n'y a pas de réels privilèges sur les domaines sous PostgreSQL™, cette vue est affiche les privilèges USAGE implicitement octroyés à PUBLIC pour tous les domaines, wrappers de données distantes et serveurs distants. Il y a une ligne pour chaque combinaison d'objet, de donneur et de receveur.

Comme les domaines n'ont pas de vrais droits dans PostgreSQL™, cette vue affiche des droits USAGE implicites, non donnables à

d'autres, et donnés par le propriétaire à PUBLIC pour tous les domaines. Les autres types d'objets affichent néanmoins de vrais droits.

Tableau 34.46. Colonnes de usage_privileges

Nom	Type de données	Description
grantor	sql_identifier	Nom du rôle qui a donné ce droit
grantee	sql_identifier	Name of the role that the privilege was granted to
object_catalog	sql_identifier	Nom de la base de données qui contient l'objet (toujours la base de données courante)
object_schema	sql_identifier	Nom du schéma qui contient l'objet, if applicable, sinon une chaîne vide
object_name	sql_identifier	Nom de l'objet
object_type	character_data	DOMAIN, FOREIGN DATA WRAPPER ou FOREIGN SERVER
privilege_type	character_data	Toujours USAGE
is_grantable	yes_or_no	YES si le droit peut être donné, NO dans le cas contraire

34.49. user_mapping_options

La vue user_mapping_options contient toutes les options définies pour les correspondances d'utilisateur définies dans la base de données en cours. Seules sont affichées les correspondances pour lesquelles le serveur distant correspondant peut être accédé par l'utilisateur connecté (qu'il en soit le propriétaire ou qu'il ait quelques droits dessus).

Tableau 34.47. Colonnes de user_mapping_options

Nom	Type de données	Description
authorization_identifier	sql_identifier	Nom de l'utilisateur, ou PUBLIC si la correspondance est publique
foreign_server_catalog	sql_identifier	Nom de la base de données dans laquelle est défini le serveur distant correspondant (toujours la base de données en cours)
foreign_server_name	sql_identifier	Nom du serveur distant utilisé par cette correspondance
option_name	sql_identifier	Nom d'une option
option_value	character_data	Valeur de l'option. Cette colonne s'affichera comme NULL sauf si l'utilisateur connecté est l'utilisateur en cours de correspondance ou si la correspondance est pour PUBLIC et que l'utilisateur connecté est le propriétaire de la base de données ou un superutilisateur. Le but est de protéger les informations de mot de passe stockées comme option.

34.50. user_mappings

La vue user_mappings contient toutes les correspondances utilisateurs définies dans la base de données en cours. Seules sont affichées les correspondances pour lesquelles le serveur distant correspondant peut être accédé par l'utilisateur connecté (qu'il en soit le propriétaire ou qu'il ait quelques droits dessus).

Tableau 34.48. Colonnes de user_mappings

Nom	Type de données	Description
authorization_identifier	sql_identifier	Nom de l'utilisateur en cours de corres-

Nom	Type de données	Description
		pondance ou PUBLIC si la correspondance est publique
foreign_server_catalog	sql_identifier	Nom de la base de données dans laquelle est défini le serveur distant correspondant (toujours la base de données en cours)
foreign_server_name	sql_identifier	Nom du serveur distant utilisé par cette correspondance

34.51. view_column_usage

La vue view_column_usage identifie toutes les colonnes utilisées dans l'expression de la requête d'une vue (l'instruction **SELECT** définissant la vue). Une colonne n'est incluse que si la table contenant la colonne appartient à un rôle actif.

> **Note**
>
> Les colonnes des tables système ne sont pas incluses. Cela sera probablement corrigé un jour.

Tableau 34.49. Colonnes de view_column_usage

Nom	Type de données	Description
view_catalog	sql_identifier	Nom de la base de données qui contient la vue (toujours la base de données courante)
view_schema	sql_identifier	Nom du schéma qui contient la vue
view_name	sql_identifier	Nom de la vue
table_catalog	sql_identifier	Nom de la base de données qui contient la table qui contient la colonne utilisée par la vue (toujours la base de données courante)
table_schema	sql_identifier	Nom du schéma qui contient la table qui contient la colonne utilisée par la vue
table_name	sql_identifier	Nom de la table qui contient la colonne utilisée par la vue
column_name	sql_identifier	Nom de la colonne utilisée par la vue

34.52. view_routine_usage

La vue view_routine_usage identifie toutes les routines (fonctions et procédures) utilisées dans la requête d'une vue (l'instruction **SELECT** qui définit la vue). Une routine n'est incluse que si la routine appartient à un rôle actif.

Tableau 34.50. Colonnes de view_routine_usage

Nom	Type de données	Description
table_catalog	sql_identifier	Nom de la base qui contient la vue (toujours la base en cours)
table_schema	sql_identifier	Nom du schéma qui contient la vue
table_name	sql_identifier	Nom de la vue
specific_catalog	sql_identifier	Nom de la base qui contient la fonction (toujours la base en cours)
specific_schema	sql_identifier	Nom du schéma qui contient la fonction
specific_name	sql_identifier	Le « nom spécifique » de la fonction. Voir Section 34.33, « routines » pour plus d'informations.

34.53. view_table_usage

La vue `view_table_usage` identifie toutes les tables utilisées dans l'expression de la requête d'une vue (l'instruction **SELECT** définissant la vue). Une table n'est incluse que son propriétaire est un rôle actif.

> **Note**
>
> Les tables système ne sont pas incluses. Cela sera probablement corrigé un jour.

Tableau 34.51. Colonnes de `view_table_usage`

Nom	Type de données	Description
`view_catalog`	sql_identifier	Nom de la base de données qui contient la vue (toujours la base de données courante)
`view_schema`	sql_identifier	Nom du schéma qui contient la vue
`view_name`	sql_identifier	Nom de la vue
`table_catalog`	sql_identifier	Nom de la base de données qui contient la table utilisée par la vue (toujours la base de données courante)
`table_schema`	sql_identifier	Nom du schéma qui contient la table utilisée par la vue
`table_name`	sql_identifier	Nom de la table utilisée par la vue

34.54. views

La vue `views` contient toutes les vues définies dans la base de données courantes. Seules sont affichées les vues auxquelles l'utilisateur a accès (parce qu'il en est le propriétaire ou qu'il dispose de privilèges).

Tableau 34.52. Colonnes de `views`

Nom	Type de données	Description
`table_catalog`	sql_identifier	Nom de la base de données qui contient la vue (toujours la base de données courante)
`table_schema`	sql_identifier	Nom du schéma qui contient la vue
`table_name`	sql_identifier	Nom de la vue
`view_definition`	character_data	Expression de la requête définissant la vue (NULL si la vue n'appartient pas à un rôle actif)
`check_option`	character_data	S'applique à une fonctionnalité non disponible dans PostgreSQL™.
`is_updatable`	yes_or_no	`YES` si la vue est actualisable (autorise **UPDATE** et **DELETE**), `NO` dans le cas contraire
`is_insertable_into`	yes_or_no	`YES` s'il est possible d'insérer des données dans la vue (autorise **INSERT**), `NO` dans le cas contraire
`is_trigger_updatable`	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™
`is_trigger_deletable`	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™
`is_trigger_insertable_into`	yes_or_no	S'applique à une fonctionnalité non disponible dans PostgreSQL™

Partie V. Programmation serveur

Cette partie traite des possibilités d'extension des fonctionnalités du serveur par l'ajout de fonctions utilisateur, de types de données, de déclencheurs (triggers), etc. Il est préférable de n'aborder ces sujets, avancés, qu'après avoir compris tous les autres.

Les derniers chapitres décrivent les langages de programmation serveur disponibles avec PostgreSQL™ ainsi que les problèmes de ces langages en général. Il est essentiel de lire au minimum les premières sections du Chapitre 35, Étendre SQL (qui traitent des fonctions) avant de se plonger dans les langages de programmation serveur.

Chapitre 35. Étendre SQL

Les sections qui suivent présentent les possibilités d'étendre le langage SQL de requêtage de PostgreSQL™ par l'ajout :

- de fonctions (Section 35.3, « Fonctions utilisateur ») ;
- d'agrégats (Section 35.10, « Agrégats utilisateur ») ;
- de types de données (Section 35.11, « Types utilisateur ») ;
- d'opérateurs (Section 35.12, « Opérateurs définis par l'utilisateur ») ;
- de classes d'opérateurs pour les index (Section 35.14, « Interfacer des extensions d'index »).

35.1. L'extensibilité

PostgreSQL™ est extensible parce qu'il opère grâce à un système de catalogues. Quiconque est familier des systèmes de bases de données relationnelles standard sait que les informations concernant les bases, les tables, les colonnes, etc. y sont stockées dans ce qu'on nomme communément des catalogues systèmes (certains systèmes appellent cela le dictionnaire de données). Pour l'utilisateur, les catalogues ressemblent à des tables ordinaires, mais le SGBD y enregistre ses registres internes. À la différence des autres systèmes, PostgreSQL™ enregistre beaucoup d'informations dans ses catalogues : non seulement l'information concernant les tables et les colonnes, mais aussi l'information concernant les types de données, les fonctions, les méthodes d'accès, etc.

Ces tables peuvent être modifiées par l'utilisateur. Qui plus est, puisque PostgreSQL™ fonde ses opérations sur ces tables, il peut être étendu par les utilisateurs. En comparaison, les systèmes de bases de données conventionnels ne peuvent être étendus qu'en modifiant les procédures dans le code source ou en installant des modules spécifiquement écrits par le vendeur de SGBD.

De plus, le serveur PostgreSQL™ peut incorporer du code utilisateur par chargement dynamique. C'est-à-dire que l'utilisateur peut indiquer un fichier de code objet (par exemple une bibliothèque partagée) qui code un nouveau type ou une nouvelle fonction et PostgreSQL™ le charge au besoin. Il est encore plus facile d'ajouter au serveur du code écrit en SQL. La possibilité de modifier son fonctionnement « à la volée » fait de PostgreSQL™ un outil unique pour le prototypage rapide de nouvelles applications et de structures de stockage.

35.2. Le système des types de PostgreSQL™

Les types de données de PostgreSQL™ sont répartis en types de base, types composites, domaines et pseudo-types.

35.2.1. Les types de base

Les types de base sont ceux qui, comme int4, sont implantés sous le niveau du langage SQL (typiquement dans un langage de bas niveau comme le C). Ils correspondent généralement à ce que l'on appelle les types de données abstraits. PostgreSQL™ ne peut opérer sur de tels types qu'au moyen de fonctions utilisateur et n'en comprend le fonctionnement que dans la limite de la description qu'en a fait l'utilisateur. Les types de base sont divisés en types scalaires et types tableaux. Pour chaque type scalaire, un type tableau est automatiquement créé destiné à contenir des tableaux de taille variable de ce type scalaire.

35.2.2. Les types composites

Les types composites, ou types lignes, sont créés chaque fois qu'un utilisateur crée une table. Il est égalment possible de définir un type composite autonome sans table associée. Un type composite n'est qu'une simple liste de types de base avec des noms de champs associés. Une valeur de type composite est une ligne ou un enregistrement de valeurs de champ. L'utilisateur peut accéder à ces champs à partir de requêtes SQL. La Section 8.15, « Types composites » fournit de plus amples informations sur ces types.

35.2.3. Les domaines

Un domaine est fondé sur un type de base particulier. Il est, dans de nombreux cas, interchangeable avec ce type. Mais un domaine peut également posséder des contraintes qui restreignent ses valeurs à un sous-ensemble des valeurs autorisées pour le type de base.

Les domaines peuvent être créés à l'aide de la commande SQL CREATE DOMAIN(7). Leurs création et utilisation n'est pas l'objet de ce chapitre.

35.2.4. Pseudo-types

Il existe quelques « pseudo-types » pour des besoins particuliers. Les pseudo-types ne peuvent pas apparaître comme champs de table ou comme attributs de types composites, mais ils peuvent être utilisés pour déclarer les types des arguments et des résultats

de fonctions. Dans le système de typage, ils fournissent un mécanisme d'identification des classes spéciales de fonctions. La Tableau 8.24, « Pseudo-Types » donne la liste des pseudo-types qui existent.

35.2.5. Types et fonctions polymorphes

Quatre pseudo-types sont particulièrement intéressants : anyelement, anyarray, anynonarray et anyenum, collectivement appelés *types polymorphes*. Toute fonction déclarée utiliser ces types est dite *fonction polymorphe*. Une fonction polymorphe peut opérer sur de nombreux types de données différents, les types de données spécifiques étant déterminés par les types des données réellement passés lors d'un appel particulier de la fonction.

Les arguments et résultats polymorphes sont liés entre eux et sont résolus dans un type de données spécifique quand une requête faisant appel à une fonction polymorphe est analysée. Chaque occurrence (argument ou valeur de retour) déclarée comme anyelement peut prendre n'importe quel type réel de données mais, lors d'un appel de fonction donné, elles doivent toutes avoir le *même* type réel. Chaque occurrence déclarée comme anyarray peut prendre n'importe quel type de données tableau mais, de la même façon, elles doivent toutes être du *même* type. Si des occurrences sont déclarées comme anyarray et d'autres comme anyelement, le type réel de tableau des occurrences anyarray doit être un tableau dont les éléments sont du même type que ceux apparaissant dans les occurrences de type anyelement. anynonarray est traité de la même façon que anyelement mais ajoute une contrainte supplémentaire. Le type réel ne doit pas être un tableau. anyenum est traité de la même façon que anyelement mais ajoute une contrainte supplémentaire. Le type doit être un type enuméré.

Ainsi, quand plusieurs occurrences d'argument sont déclarées avec un type polymorphe, seules certaines combinaisons de types réels d'argument sont autorisées. Par exemple, une fonction déclarée comme `foo(anyelement, anyelement)` peut prendre comme arguments n'importe quelles valeurs à condition qu'elles soient du même type de données.

Quand la valeur renvoyée par une fonction est déclarée de type polymorphe, il doit exister au moins une occurrence d'argument également polymorphe, et le type réel de donnée passé comme argument détermine le type réel de résultat renvoyé lors de cet appel à la fonction. Par exemple, s'il n'existe pas déjà un mécanisme d'indexation d'éléments de tableau, on peut définir une fonction qui code ce mécanisme : `indice(anyarray, integer) returns anyelement`. La déclaration de fonction contraint le premier argument réel à être de type tableau et permet à l'analyseur d'inférer le type correct de résultat à partir du type réel du premier argument. Une fonction déclarée de cette façon `f(anyarray) returns anyenum` n'accepte que des tableaux contenant des valeurs de type enum.

anynonarray et anyenum ne représentent pas des variables de type séparé ; elles sont du même type que anyelement, mais avec une contrainte supplémentaire. Par exemple, déclarer une fonction `f(anyelement, anyenum)` est équivalent à la déclarer `f(anyenum, anyenum)` : les deux arguments réels doivent être du même type enum.

Une fonction variadic (c'est-à-dire une fonction acceptant un nombre variable d'arguments, comme dans Section 35.4.5, « Fonctions SQL avec un nombre variables d'arguments ») peut être polymorphique : cela se fait en déclarant son dernier paramètre `VARIADIC` anyarray. Pour s'assurer de la correspondance des arguments et déterminer le type de la valeur en retour, ce type de fonction se comporte de la même façon que si vous aviez écrit le nombre approprié de paramètres anynonarray.

35.3. Fonctions utilisateur

PostgreSQL™ propose quatre types de fonctions :

- fonctions en langage de requête (fonctions écrites en SQL, Section 35.4, « Fonctions en langage de requêtes (SQL) »)
- fonctions en langage procédural (fonctions écrites, par exemple, en PL/pgSQL ou PL/Tcl, Section 35.7, « Fonctions en langage de procédures »)
- fonctions internes (Section 35.8, « Fonctions internes »)
- fonctions en langage C (Section 35.9, « Fonctions en langage C »)

Chaque type de fonction peut accepter comme arguments (paramètres) des types de base, des types composites ou une combinaison de ceux-ci. De plus, chaque sorte de fonction peut renvoyer un type de base ou un type composite. Les fonctions pourraient aussi être définies pour renvoyer des ensembles de valeurs de base ou de valeurs composites.

De nombreuses sortes de fonctions peuvent accepter ou renvoyer certains pseudo-types (comme les types polymorphes) mais avec des fonctionnalités variées. Consultez la description de chaque type de fonction pour plus de détails.

Il est plus facile de définir des fonctions SQL aussi allons-nous commencer par celles-ci. La plupart des concepts présentés pour les fonctions SQL seront aussi gérés par les autres types de fonctions.

Lors de la lecture de ce chapitre, il peut être utile de consulter la page de référence de la commande CREATE FUNCTION(7) pour mieux comprendre les exemples. Quelques exemples extraits de ce chapitre peuvent être trouvés dans les fichiers `funcs.sql` et `funcs.c` du répertoire du tutoriel de la distribution source de PostgreSQL™.

35.4. Fonctions en langage de requêtes (SQL)

Les fonctions SQL exécutent une liste arbitraire d'instructions SQL et renvoient le résultat de la dernière requête de cette liste. Dans le cas d'un résultat simple (pas d'ensemble), la première ligne du résultat de la dernière requête sera renvoyée (gardez à l'esprit que « la première ligne » d'un résultat multiligne n'est pas bien définie à moins d'utiliser `ORDER BY`). Si la dernière requête de la liste ne renvoie aucune ligne, la valeur NULL est renvoyée.

Une fonction SQL peut être déclarée de façon à renvoyer un ensemble (set) en spécifiant le type renvoyé par la fonction comme `SETOF` *`un_type`*, ou de façon équivalente en la déclarant comme `RETURNS TABLE(`*`colonnes`*`)`. Dans ce cas, toutes les lignes de la dernière requête sont renvoyées. Des détails supplémentaires sont donnés plus loin dans ce chapitre.

Le corps d'une fonction SQL doit être constitué d'une liste d'une ou de plusieurs instructions SQL séparées par des points-virgule. Un point-virgule après la dernière instruction est optionnel. Sauf si la fonction déclare renvoyer void, la dernière instruction doit être un **SELECT** ou un **INSERT**, **UPDATE** ou un **DELETE** qui a une clause `RETURNING`.

Toute collection de commandes dans le langage SQL peut être assemblée et définie comme une fonction. En plus des requêtes **SELECT**, les commandes peuvent inclure des requêtes de modification des données (**INSERT**, **UPDATE** et **DELETE**) ainsi que d'autres commandes SQL (la seule exception est que vous ne pouvez pas placer de commandes **BEGIN**, **COMMIT**, **ROLLBACK** ou **SAVEPOINT** dans une fonction SQL). Néanmoins, la commande finale doit être un **SELECT** ou doit avoir une clause `RETURNING` qui renvoie ce qui a été spécifié comme type de retour de la fonction. Autrement, si vous voulez définir une fonction SQL qui réalise des actions mais n'a pas de valeur utile à renvoyer, vous pouvez la définir comme renvoyant void. Par exemple, cette fonction supprime les lignes avec des salaires négatifs depuis la table `emp` :

```
CREATE FUNCTION nettoie_emp() RETURNS void AS '
    DELETE FROM emp WHERE salaire < 0;
' LANGUAGE SQL;

SELECT nettoie_emp();

  nettoie_emp
  -----------

  (1 row)
```

La syntaxe de la commande **CREATE FUNCTION** requiert que le corps de la fonction soit écrit comme une constante de type chaîne. Il est habituellement plus agréable d'utiliser les guillemets dollar (voir la Section 4.1.2.4, « Constantes de chaînes avec guillemet dollar ») pour cette constante. Si vous choisissez d'utiliser la syntaxe habituelle avec des guillemets simples, vous devez doubler les marques de guillemet simple (`'`) et les antislashs (`\`), en supposant que vous utilisez la syntaxe d'échappement de chaînes, utilisés dans le corps de la fonction (voir la Section 4.1.2.1, « Constantes de chaînes »).

Les arguments de la fonction SQL sont référencés dans le corps de la fonction en utilisant la syntaxe suivante. `$`*`n`*`:$1` se réfère au premier argument, `$2` au second et ainsi de suite. Si un argument est de type composite, on utilisera la notation par point, par exemple `$1.name`, pour accéder aux attributs de l'argument. Les arguments peuvent seulement être utilisés comme valeurs des données, pas comme des identifieurs. Du coup, par exemple, ceci est correct :

```
INSERT INTO matable VALUES ($1);
```

mais ceci ne fonctionnera pas :

```
INSERT INTO $1 VALUES (42);
```

35.4.1. Fonctions SQL sur les types de base

La fonction SQL la plus simple possible n'a pas d'argument et retourne un type de base tel que integer :

```
CREATE FUNCTION un() RETURNS integer AS $$
    SELECT 1 AS resultat;
$$ LANGUAGE SQL;

-- Autre syntaxe pour les chaînes littérales :
CREATE FUNCTION un() RETURNS integer AS '
    SELECT 1 AS resultat;
' LANGUAGE SQL;

SELECT un();

 un
----
  1
```

Notez que nous avons défini un alias de colonne avec le nom `resultat` dans le corps de la fonction pour se référer au résultat de la fonction mais cet alias n'est pas visible hors de la fonction. En effet, le résultat est nommé `un` au lieu de `resultat`.

Il est presque aussi facile de définir des fonctions SQL acceptant des types de base comme arguments. Dans l'exemple suivant, remarquez comment nous faisons référence aux arguments dans le corps de la fonction avec `$1` et `$2`.

```
CREATE FUNCTION ajoute(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT ajoute(1, 2) AS reponse;

 reponse
---------
       3
```

Voici une fonction plus utile, qui pourrait être utilisée pour débiter un compte bancaire :

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS $$
    UPDATE banque
        SET balance = balance - $2
        WHERE no_compte = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

Un utilisateur pourrait exécuter cette fonction pour débiter le compte 17 de 100 000 euros ainsi :

```
SELECT tf1(17, 100.000);
```

Dans la pratique, on préférera vraisemblablement un résultat plus utile que la constante 1. Une définition plus probable est :

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE banque
        SET balance = balance - $2
        WHERE no_compte = $1;
    SELECT balance FROM banque WHERE no_compte = $1;
$$ LANGUAGE SQL;
```

qui ajuste le solde et renvoie sa nouvelle valeur. La même chose peut se faire en une commande en utilisant la clause `RETURNING` :

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE banque
        SET balance = balance - $2
        WHERE no_compte = $1
    RETURNING balance;
$$ LANGUAGE SQL;
```

35.4.2. Fonctions SQL sur les types composites

Quand nous écrivons une fonction avec des arguments de type composite, nous devons non seulement spécifier l'argument utilisé (comme nous l'avons fait précédemment avec `$1` et `$2`), mais aussi spécifier l'attribut désiré de cet argument (champ). Par exemple, supposons que emp soit le nom d'une table contenant des données sur les employés et donc également le nom du type composite correspondant à chaque ligne de la table. Voici une fonction `double_salaire` qui calcule ce que serait le salaire de quelqu'un s'il était doublé :

```
CREATE TABLE emp (
    nom         text,
    salaire     numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salaire(emp) RETURNS numeric AS $$
    SELECT $1.salaire * 2 AS salaire;
$$ LANGUAGE SQL;
```

```
SELECT nom, double_salaire(emp.*) AS reve
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | reve
------+-------
 Bill |  8400
```

Notez l'utilisation de la syntaxe `$1.salaire` pour sélectionner un champ dans la valeur de la ligne argument. Notez également comment la commande **SELECT** utilise `*` pour sélectionner la ligne courante entière de la table comme une valeur composite (`emp`). La ligne de la table peut aussi être référencée en utilisant seulement le nom de la table ainsi :

```
SELECT nom, double_salaire(emp) AS reve
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

mais cette utilisation est obsolète car elle est facilement obscure.

Quelque fois, il est pratique de construire une valeur d'argument composite en direct. Ceci peut se faire avec la construction `ROW`. Par exemple, nous pouvons ajuster les données passées à la fonction :

```
SELECT nom, double_salaire(ROW(nom, salaire*1.1, age, cubicle)) AS reve
        FROM emp;
```

Il est aussi possible de construire une fonction qui renvoie un type composite. Voici un exemple de fonction renvoyant une seule ligne de type emp :

```
CREATE FUNCTION nouvel_emp() RETURNS emp AS $$
    SELECT text 'Aucun' AS nom,
        1000.0 AS salaire,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

Dans cet exemple, nous avons spécifié chacun des attributs avec une valeur constante, mais un quelconque calcul aurait pu être substitué à ces valeurs.

Notez deux aspects importants à propos de la définition de fonction :

- L'ordre de la liste du SELECT doit être exactement le même que celui dans lequel les colonnes apparaissent dans la table associée au type composite (donner des noms aux colonnes dans le corps de la fonction, comme nous l'avons fait dans l'exemple, n'a aucune interaction avec le système).
- Vous devez transtyper les expressions pour concorder avec la définition du type composite ou bien vous aurez l'erreur suivante :

  ```
  ERROR:  function declared to return emp returns varchar instead of text at column 1
  ```

Une autre façon de définir la même fonction est :

```
CREATE FUNCTION nouveau_emp() RETURNS emp AS $$
    SELECT ROW('Aucun', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Ici, nous écrivons un **SELECT** qui renvoie seulement une colonne du bon type composite. Ceci n'est pas vraiment meilleur dans cette situation mais c'est une alternative pratique dans certains cas -- par exemple, si nous avons besoin de calculer le résultat en appelant une autre fonction qui renvoie la valeur composite désirée.

Nous pourrions appeler cette fonction directement de deux façons :

```
SELECT nouveau_emp();

        nouveau_emp
--------------------------
 (None,1000.0,25,"(2,2)")

SELECT * FROM nouveau_emp();

  nom  | salaire | age | cubicle
-------+---------+-----+---------
```

```
 Aucun |  1000.0 |  25 | (2,2)
```

La deuxième façon est décrite plus complètement dans la Section 35.4.7, « Fonctions SQL comme sources de table ».

Quand vous utilisez une fonction qui renvoie un type composite, vous pourriez vouloir seulement un champ (attribut) depuis ce résultat. Vous pouvez le faire avec cette syntaxe :

```
SELECT (nouveau_emp()).nom;

 nom
------
 None
```

Les parenthèses supplémentaires sont nécessaires pour éviter une erreur de l'analyseur. Si vous essayez de le faire sans, vous obtiendrez quelque chose comme ceci :

```
SELECT nouveau_emp().nom;
ERROR:  syntax error at or near "."
LINE 1: SELECT nouveau_emp().nom;
                             ^
```

Une autre option est d'utiliser la notation fonctionnelle pour extraire un attribut. Une manière simple d'expliquer cela est de dire que nous pouvons échanger les notations `attribut(table)` et `table.attribut`.

```
SELECT nom(nouveau_emp());

 name
------
 None
```

```
-- C'est la même chose que
-- SELECT emp.nom AS leplusjeune FROM emp WHERE emp.age < 30;

SELECT nom(emp) AS leplusjeune FROM emp WHERE age(emp) < 30;

 leplusjeune
-------------
 Sam
 Andy
```

Astuce

L'équivalence entre la notation fonctionnelle et la notation d'attribut rend possible l'utilisation de fonctions sur des types composites pour émuler les « champs calculés ». Par exemple, en utilisant la définition précédente pour `double_salaire(emp)`, nous pouvons écrire

```
SELECT emp.nom, emp.double_salaire FROM emp;
```

Une application utilisant ceci n'aurait pas besoin d'être consciente directement que `double_salaire` n'est pas une vraie colonne de la table (vous pouvez aussi émuler les champs calculés avec des vues).

Une autre façon d'utiliser une fonction renvoyant un type composite est de l'appeler comme une fonction de table, comme décrit dans la Section 35.4.7, « Fonctions SQL comme sources de table ».

35.4.3. Fonctions SQL avec des paramètres nommés

Il est possible d'attacher des noms aux paramètres d'une fonction, par exemple :

```
CREATE FUNCTION tf1 (acct_no integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE no_compte = $1
    RETURNING balance;
$$ LANGUAGE SQL;
```

Dans cet exemple, le premier paramètre a comme nom `acct_no`, et le second paramètre `debit`. En ce qui concerne la fonction SQL, ces noms sont de la décoration. Vous devez toujours faire référence aux paramètres avec la notation `$1`, `$2`, etc dans le

corps de la fonction. (Certains langages de procédure vous autorisent à utiliser les noms des paramètres à la place.) Néanmoins, les noms des paramètres peuvent être utiles dans un but de documentation. Quand une fonction a beaucoup de paramètres, il est aussi utile d'utiliser les noms lors de l'appel à la fonction, comme décrit dans Section 4.3, « Fonctions appelantes ».

35.4.4. Fonctions SQL avec des paramètres en sortie

Une autre façon de décrire les résultats d'une fonction est de la définir avec des *paramètres en sortie* comme dans cet exemple :

```
CREATE FUNCTION ajoute (IN x int, IN y int, OUT sum int)
AS 'SELECT $1 + $2'
LANGUAGE SQL;

SELECT ajoute(3,7);
 ajoute
--------
     10
(1 row)
```

Ceci n'est pas vraiment différent de la version d'`ajoute` montrée dans la Section 35.4.1, « Fonctions SQL sur les types de base ». La vraie valeur des paramètres en sortie est qu'ils fournissent une façon agréable de définir des fonctions qui renvoient plusieurs colonnes. Par exemple,

```
CREATE FUNCTION ajoute_n_produit (x int, y int, OUT sum int, OUT product int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

Ce qui est arrivé ici est que nous avons créé un type composite anonyme pour le résultat de la fonction. L'exemple ci-dessus a le même résultat final que

```
CREATE TYPE produit_ajoute AS (somme int, produit int);

CREATE FUNCTION ajoute_n_produit (int, int) RETURNS produit_ajoute
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

mais ne pas avoir à s'embêter avec la définition séparée du type composite est souvent agréable. Notez que les noms attachés aux paramètres de sortie ne sont pas juste décoratif, mais déterminent le nom des colonnes du type composite anonyme. (Si vous omettez un nom pour un paramètre en sortie, le système choisira un nom lui-même.)

Notez que les paramètres en sortie ne sont pas inclus dans la liste d'arguments lors de l'appel d'une fonction de ce type en SQL. Ceci parce que PostgreSQL™ considère seulement les paramètres en entrée pour définir la signature d'appel de la fonction. Cela signifie aussi que seuls les paramètres en entrée sont importants lors de références de la fonction pour des buts comme sa suppression. Nous pouvons supprimer la fonction ci-dessus avec l'un des deux appels ci-dessous :

```
DROP FUNCTION ajoute_n_produit (x int, y int, OUT somme int, OUT produit int);
DROP FUNCTION ajoute_n_produit (int, int);
```

Les paramètres peuvent être marqués comme `IN` (par défaut), `OUT` ou `INOUT` ou `VARIADIC`. Un paramètre `INOUT` sert à la fois de paramètre en entrée (il fait partie de la liste d'arguments en appel) et comme paramètre de sortie (il fait partie du type d'enregistrement résultat). Les paramètres `VARIADIC` sont des paramètres en entrées, mais sont traités spécifiquement comme indiqué ci-dessous.

35.4.5. Fonctions SQL avec un nombre variables d'arguments

Les fonctions SQL peuvent accepter un nombre variable d'arguments à condition que tous les arguments « optionnels » sont du même type. Les arguments optionnels seront passés à la fonction sous forme d'un tableau. La fonction est déclarée en marquant le dernier paramètre comme `VARIADIC` ; ce paramètre doit être déclaré de type tableau. Par exemple :

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
```

```
 mleast
--------
     -1
(1 row)
```

En fait, tous les arguments à la position ou après la position de l'argument `VARIADIC` sont emballés dans un tableau à une dimension, comme si vous aviez écrit

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- doesn't work
```

Vous ne pouvez pas vraiment écrire cela, ou tout du moins cela ne correspondra pas à la définition de la fonction. Un paramètre marqué `VARIADIC` correspond à une ou plusieurs occurrences de son type d'élément, et non pas de son propre type.

Quelque fois, il est utile de pouvoir passer un tableau déjà construit à une fonction variadic ; ceci ets particulièrement intéressant quand une fonction variadic veut passer son paramètre tableau à une autre fonction. Vous pouvez faire cela en spécifiant `VARIADIC` dans l'appel :

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

Ceci empêche l'expansion du paramètre variadic de la fonction dans le type des éléments, ce qui permet à la valeur tableau de correspondre. `VARIADIC` peut seulement être attaché au dernier argument d'un appel de fonction.

Les paramètres de l'élément tableau générés à partir d'un paramètre variadic sont traités comme n'ayant pas de noms propres. Cela signifie qu'il n'est pas possible d'appeler une fonction variadic en utilisant des arguments nommés (Section 4.3, « Fonctions appelantes »), sauf quand vous spécifiez `VARIADIC`. Par exemple, ceci fonctionnera :

```
SELECT mleast(VARIADIC arr := ARRAY[10, -1, 5, 4.4]);
```

mais pas cela :

```
SELECT mleast(arr := 10);
SELECT mleast(arr := ARRAY[10, -1, 5, 4.4]);
```

35.4.6. Fonctions SQL SQL avec des valeurs par défaut pour les arguments

Les fonctions peuvent être déclarées avec des valeurs par défaut pour certains des paramètres en entrée ou pour tous. Les valeurs par défaut sont insérées quand la fonction est appelée avec moins d'arguments que à priori nécessaires. Comme les arguments peuvent seulement être omis à partir de la fin de la liste des arguments, tous les paramètres après un paramètres disposant d'une valeur par défaut disposeront eux-aussi d'une valeur par défaut. (Bien que l'utilisation de la notation avec des arguments nommés pourrait autoriser une relâche de cette restriction, elle est toujours forcée pour que la notation des arguments de position fonctionne correctement.)

Par exemple :

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
```

```
  15
(1 row)

SELECT foo();  -- échec car il n'y a pas de valeur par défaut pour le premier argument
ERROR:  function foo() does not exist
```

Le signe = peut aussi être utilisé à la place du mot clé DEFAULT,

35.4.7. Fonctions SQL comme sources de table

Toutes les fonctions SQL peuvent être utilisées dans la clause FROM d'une requête mais ceci est particulièrement utile pour les fonctions renvoyant des types composite. Si la fonction est définie pour renvoyer un type de base, la fonction table produit une table d'une seule colonne. Si la fonction est définie pour renvoyer un type composite, la fonction table produit une colonne pour chaque attribut du type composite.

Voici un exemple :

```
CREATE TABLE foo (fooid int, foosousid int, foonom text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION recupfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(foonom) FROM recupfoo(1) AS t1;

 fooid | foosubid | foonom | upper
-------+----------+--------+-------
     1 |        1 | Joe    | JOE
(1 row)
```

Comme le montre cet exemple, nous pouvons travailler avec les colonnes du résultat de la fonction comme s'il s'agissait des colonnes d'une table normale.

Notez que nous n'obtenons qu'une ligne comme résultat de la fonction. Ceci parce que nous n'avons pas utilisé l'instruction SETOF. Cette instruction est décrite dans la prochaine section.

35.4.8. Fonctions SQL renvoyant un ensemble

Quand une fonction SQL est déclarée renvoyer un SETOF *un_type*, la requête finale de la fonction est complètement exécutée et chaque ligne extraite est renvoyée en tant qu'élément de l'ensemble résultat.

Cette caractéristique est normalement utilisée lors de l'appel d'une fonction dans une clause FROM. Dans ce cas, chaque ligne renvoyée par la fonction devient une ligne de la table vue par la requête. Par exemple, supposons que la table foo ait le même contenu que précédemment et écrivons :

```
CREATE FUNCTION recupfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM recupfoo(1) AS t1;
```

Alors nous obtenons :

```
 fooid | foosousid | foonom
-------+-----------+--------
     1 |         1 | Joe
     1 |         2 | Ed
(2 rows)
```

Il est aussi possible de renvoyer plusieurs lignes avec les colonnes définies par des paramètres en sortie, comme ceci :

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
```

```
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

Le point clé ici est que vous devez écrire `RETURNS SETOF record` pour indiquer que la fonction renvoie plusieurs lignes et non pas une seule. S'il n'y a qu'un paramètre en sortie, indiquez le type de paramètre plutôt que record.

Actuellement, les fonctions renvoyant des ensembles peuvent aussi être appelées dans la liste du select d'une requête. Pour chaque ligne générée par la requête, la fonction renvoyant un ensemble est appelée et une ligne est générée pour chaque élément de l'ensemble résultat. Notez cependant que cette fonctionnalité est déconseillée et pourra être supprimée dans une version future. Voici un exemple de fonction renvoyant un ensemble à partir de la liste d'un SELECT :

```
CREATE FUNCTION listeenfant(text) RETURNS SETOF text AS $$
    SELECT nom FROM noeuds WHERE parent = $1
$$ LANGUAGE SQL;

SELECT * FROM noeuds;
   nom        | parent
--------------+--------
 Haut         |
 Enfant1      | Haut
 Enfant2      | Haut
 Enfant3      | Haut
 Sous-Enfant1 | Enfant1
 Sous-Enfant2 | Enfant1
(6 rows)

SELECT listeenfant('Haut');
 listeenfant
--------------
 Enfant1
 Enfant2
 Enfant3
(3 rows)

SELECT nom, listeenfant(nom) FROM noeuds;
  nom    | listeenfant
---------+--------------
 Haut    | Enfant1
 Haut    | Enfant2
 Haut    | Enfant3
 Enfant1 | Sous-Enfant1
 Enfant1 | Sous-Enfant2
(5 rows)
```

Notez, dans le dernier **SELECT**, qu'aucune ligne n'est renvoyée pour `Enfant2`, `Enfant3`, etc. C'est parce que la fonction `listeenfant` renvoie un ensemble vide pour ces arguments et ainsi aucune ligne n'est générée.

Note

Si la dernière commande d'une fonction est **INSERT**, **UPDATE** ou **DELETE** avec une clause `RETURNING`, cette commande sera toujours exécutée jusqu'à sa fin, même si la fonction n'est pas déclarée avec `SETOF` ou que la requête appelante ne renvoie pas toutes les lignes résultats. Toutes les lignes supplémentaires produites par la clause `RETURNING` sont silencieusement abandonnées mais les modifications de table sont pris en compte (et sont toutes terminées avant que la fonction ne se termine).

35.4.9. Fonctions SQL renvoyant `TABLE`

Il existe une autre façon de déclarer une fonction comme renvoyant un ensemble de données. Cela passe par la syntaxe `RETURNS TABLE(`*`colonnes`*`)`. C'est équivalent à utiliser un ou plusieurs paramètres `OUT` et à marquer la fonction comme renvoyant un

`SETOF record` (ou `SETOF` d'un type simple en sortie, comme approprié). Cette notation est indiquée dans les versions récentes du standard SQL et, du coup, devrait être plus portable que `SETOF`.

L'exemple précédent, sum-and-product, peut se faire aussi de la façon suivante :

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Il n'est pas autorisé d'utiliser explicitement des paramètres `OUT` ou `INOUT` avec la notation `RETURNS TABLE` -- vous devez indiquer toutes les colonnes en sortie dans la liste `TABLE`.

35.4.10. Fonctions SQL polymorphes

Les fonctions SQL peuvent être déclarées pour accepter et renvoyer les types « polymorphe » anyelement, anyarray, anynonarray et anyenum. Voir la Section 35.2.5, « Types et fonctions polymorphes » pour une explication plus approfondie. Voici une fonction polymorphe `cree_tableau` qui construit un tableau à partir de deux éléments de type arbitraire :

```
CREATE FUNCTION cree_tableau(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
SELECT cree_tableau(1, 2) AS tableau_entier, cree_tableau('a'::text, 'b') AS
tableau_texte;

 tableau_entier | tableau_texte
----------------+---------------
 {1,2}          | {a,b}
(1 row)
```

Notez l'utilisation du transtypage `'a'::text` pour spécifier le type text de l'argument. Ceci est nécessaire si l'argument est une chaîne de caractères car, autrement, il serait traité comme un type unknown, et un tableau de type unknown n'est pas un type valide. Sans le transtypage, vous obtiendrez ce genre d'erreur :

```
ERROR:  could not determine polymorphic type because input is UNKNOWN
```

Il est permis d'avoir des arguments polymorphes avec un type de renvoi fixe, mais non l'inverse. Par exemple :

```
CREATE FUNCTION est_plus_grand(anyelement, anyelement) RETURNS bool AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT est_plus_grand(1, 2);
 est_plus_grand
----------------
 f
(1 row)

CREATE FUNCTION fonction_invalide() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result datatype
DETAIL:  A function returning a polymorphic type must have at least one
polymorphic argument.
```

Le polymorphisme peut être utilisé avec les fonctions qui ont des arguments en sortie. Par exemple :

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

Le polymorphisme peut aussi être utilisé avec des fonctions variadic. Par exemple :

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
```

```
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat('|', 1, 4, 2);
 concat
--------
 1|4|2
(1 row)
```

35.5. Surcharge des fonctions

Plusieurs fonctions peuvent être définies avec le même nom SQL à condition que les arguments soient différents. En d'autres termes, les noms de fonction peuvent être *surchargés*. Quand une requête est exécutée, le serveur déterminera la fonction à appeler à partir des types de données des arguments et du nombre d'arguments. La surcharge peut aussi être utilisée pour simuler des fonctions avec un nombre variable d'arguments jusqu'à un nombre maximum fini.

Lors de la création d'une famille de fonctions surchargées, vous devriez être attentif à ne pas créer d'ambiguïtés. Par exemple, avec les fonctions :

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

Savoir quelle fonction sera appelée avec une entrée triviale comme `test(1, 1.5)` n'est pas immédiatement clair. Les règles de résolution actuellement implémentées sont décrites dans le Chapitre 10, Conversion de types mais il est déconseillé de concevoir un système qui serait basé subtilement sur ce comportement.

Une fonction qui prend un seul argument d'un type composite devrait généralement ne pas avoir le même nom que tout attribut (champ) de ce type. Rappellez-vous que `attribut(table)` est considéré comme équivalent à `table.attribut`. Dans le cas où il existe une ambiguïté entre une fonction sur un type composite et sur un attribut d'un type composite, l'attribut sera toujours utilisé. Il est possible de contourner ce choix en qualifiant le nom de la fonction avec celui du schéma (c'est-à-dire `schema.fonction(table)`) mais il est préférable d'éviter le problème en ne choisissant aucun nom conflictuel.

Un autre conflit possible se trouve entre les fonctions variadic et les autres. En fait, il est possible de créer à la fois `foo(numeric)` et `foo(VARIADIC numeric[])`. Dans ce cas, il n'est pas simple de savoir lequel sera sélectionné lors d'un appel avec un seul argument numérique, par exemple `foo(10.1)`. La règle est que la fonction apparaissant plsu tôt dans le chemin des schémas est utilisé. De même, si les deux fonctions sont dans le même schéma, la non variadic est préféré.

Lors de la surcharge de fonctions en langage C, il existe une contrainte supplémentaire : le nom C de chaque fonction dans la famille des fonctions surchargées doit être différent des noms C de toutes les autres fonctions, soit internes soit chargées dynamiquement Si cette règle est violée, le comportement n'est pas portable. Vous pourriez obtenir une erreur de l'éditeur de lien ou une des fonctions sera appelée (habituellement l'interne). L'autre forme de clause `AS` pour la commande SQL **CREATE FUNCTION** découple le nom de la fonction SQL à partir du nom de la fonction dans le code source C. Par exemple :

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
LANGUAGE C;
```

Les noms des fonctions C reflètent ici une des nombreuses conventions possibles.

35.6. Catégories de volatilité des fonctions

Chaque fonction a une classification de volatilité (*volatility*) comprenant `VOLATILE`, `STABLE` ou `IMMUTABLE`. `VOLATILE` est la valeur par défaut si la commande CREATE FUNCTION(7) ne spécifie pas de catégorie. La catégorie de volatilité est une promesse à l'optimiseur sur le comportement de la fonction :

- Une fonction `VOLATILE` peut tout faire, y compris modifier la base de données. Elle peut renvoyer différents résultats sur des appels successifs avec les mêmes arguments. L'optimiseur ne fait aucune supposition sur le comportement de telles fonctions. Une requête utilisant une fonction volatile ré-évaluera la fonction à chaque ligne où sa valeur est nécessaire.
- Une fonction `STABLE` ne peut pas modifier la base de données et est garantie de renvoyer les mêmes résultats si elle est appelée avec les mêmes arguments pour toutes les lignes à l'intérieur d'une même instruction. Cette catégorie permet à l'optimiseur d'optimiser plusieurs appels de la fonction dans une seule requête. En particulier, vous pouvez utiliser en toute sécurité une expression contenant une telle fonction dans une condition de parcours d'index (car un parcours d'index évaluera la valeur de la comparaison une seule fois, pas une fois pour chaque ligne, utiliser une fonction `VOLATILE` dans une condition de parcours d'index n'est pas valide).
- Une fonction `IMMUTABLE` ne peut pas modifier la base de données et est garantie de toujours renvoyer les mêmes résultats si elle est appelée avec les mêmes arguments. Cette catégorie permet à l'optimiseur de pré-évaluer la fonction quand une requête l'appelle avec des arguments constants. Par exemple, une requête comme `SELECT ... WHERE x = 2 + 2` peut être simplifiée pour obtenir `SELECT ... WHERE x = 4` car la fonction sous-jacente de l'opérateur d'addition est indiquée `IMMUTABLE`.

Pour une meilleure optimisation des résultats, vous devez mettre un label sur les fonctions avec la catégorie la plus volatile valide pour elles.

Toute fonction avec des effets de bord *doit* être indiquée comme `VOLATILE`, de façon à ce que les appels ne puissent pas être optimisés. Même une fonction sans effets de bord doit être indiquée comme `VOLATILE` si sa valeur peut changer à l'intérieur d'une seule requête ; quelques exemples sont `random()`, `currval()`, `timeofday()`.

Un autre exemple important est que la famille de fonctions `current_timestamp` est qualifiée comme `STABLE` car leurs valeurs ne changent pas à l'intérieur d'une transaction.

Il y a relativement peu de différences entre les catégories `STABLE` et `IMMUTABLE` en considérant les requêtes interactives qui sont planifiées et immédiatement exécutées : il importe peu que la fonction soit exécutée une fois lors de la planification ou une fois au lancement de l'exécution de la requête mais cela fait une grosse différence si le plan est sauvegardé et utilisé plus tard. Placer un label `IMMUTABLE` sur une fonction quand elle ne l'est pas vraiment pourrait avoir comme conséquence de la considérer prématurément comme une constante lors de la planification et résulterait en une valeur erronée lors d'une utilisation ultérieure de ce plan d'exécution. C'est un danger qui arrive lors de l'utilisattion d'instructions préparées ou avec l'utilisation de langages de fonctions mettant les plans d'exécutions en cache (comme PL/pgSQL).

Pour les fonctions écrites en SQL ou dans tout autre langage de procédure standard, la catégorie de volatibilité détermine une deuxième propriété importante, à savoir la visibilité de toute modification de données effectuées par la commande SQL qui a appelé la fonction. Une fonction `VOLATILE` verra les changements, une fonction `STABLE` ou `IMMUTABLE` ne les verra pas. Ce comportement est implantée en utilisant le comportement par images de MVCC (voir Chapitre 13, Contrôle d'accès simultané) : les fonctions `STABLE` et `IMMUTABLE` utilisent une image établie au lancement de la requête appelante alors que les fonctions `VOLATILE` obtiennent une image fraiche au début de chaque requête qu'elles exécutent.

Note

Les fonctions écrites en C peuvent gérer les images de la façon qu'elles le souhaitent, mais il est préférable de coder les fonctions C de la même façon.

À cause du comportement à base d'images, une fonction contenant seulement des commandes **SELECT** peut être indiquée `STABLE` en toute sécurité même s'il sélectionne des données à partir de tables qui pourraient avoir subi des modifications entre temps par des requêtes concurrentes. PostgreSQL™ exécutera toutes les commandes d'une fonction `STABLE` en utilisant l'image établie par la requête appelante et n'aura qu'une vision figée de la base de données au cours de la requête.

Ce même comportement d'images est utilisé pour les commandes **SELECT** à l'intérieur de fonctions `IMMUTABLE`. Il est généralement déconseillé de sélectionner des tables de la base de données à l'intérieur de fonctions `IMMUTABLE` car l'immutabilité sera rompue si le contenu de la table change. Néanmoins, PostgreSQL™ ne vous force pas à ne pas le faire.

Une erreur commune est de placer un label sur une fonction `IMMUTABLE` quand son résultat dépend d'un paramètre de configuration. Par exemple, une fonction qui manipule des types date/heure pourrait bien avoir des résultats dépendant du paramètre timezone. Pour être sécurisées, de telles fonctions devraient avoir le label `STABLE` à la place.

Note

Avant PostgreSQL™ version 8.0, le prérequis que les fonctions `STABLE` et `IMMUTABLE` ne pouvaient pas modifier la base de données n'était pas contraint par le système. Les versions 8.0 et ultérieures le contraignent en réclamant que les fonctions SQL et les fonctions de langages de procédures de ces catégories ne contiennent pas de commandes SQL autre que **SELECT** (ceci n'a pas été complètement testé car de telles fonctions pourraient toujours appeler des fonctions `VOLATILE` qui modifient la base de données. Si vous le faites, vous trouverez que la fonction `STABLE` ou `IMMUTABLE` n'est pas au courant des modifications effectuées sur la base de données par la fonction appelée, car elles sont cachées depuis son image).

35.7. Fonctions en langage de procédures

PostgreSQL™ autorise l'écriture de fonctions définies par l'utilisateur dans d'autres langages que SQL et C. Ces autres langages sont appelés des *langages de procédure* (PL). Les langages de procédures ne sont pas compilés dans le serveur PostgreSQL™ ; ils sont fournis comme des modules chargeables. Voir le Chapitre 38, Langages de procédures et les chapitres suivants pour plus d'informations.

Il y a actuellement quatre langages de procédures disponibles dans la distribution PostgreSQL™ standard : PL/pgSQL, PL/Tcl, PL/Perl et PL/Python. Référez-vous au Chapitre 38, Langages de procédures pour plus d'informations. D'autres langages peuvent être définis par les utilisateurs. Les bases du développement d'un nouveau langage de procédures sont traitées dans le Chapitre 49, Écrire un gestionnaire de langage procédural.

35.8. Fonctions internes

Les fonctions internes sont des fonctions écrites en C qui ont été liées de façon statique dans le serveur PostgreSQL™. Le « corps » de la définition de la fonction spécifie le nom en langage C de la fonction, qui n'est pas obligatoirement le même que le nom déclaré pour l'utilisation en SQL (pour des raisons de rétro compatibilité, un corps vide est accepté pour signifier que le nom de la fonction en langage C est le même que le nom SQL).

Normalement, toutes les fonctions internes présentes dans le serveur sont déclarées pendant l'initialisation du groupe de base de données (voir Section 17.2, « Créer un groupe de base de données ») mais un utilisateur peut utiliser la commande **CREATE FUNCTION** pour créer des noms d'alias supplémentaires pour une fonction interne. Les fonctions internes sont déclarées dans la commande **CREATE FUNCTION** avec le nom de langage `internal`. Par exemple, pour créer un alias de la fonction `sqrt` :

```
CREATE FUNCTION racine_carree(double precision) RETURNS double precision    AS
'dsqrt'
LANGUAGE internal    STRICT;
```

(la plupart des fonctions internes doivent être déclarées « STRICT »)

Note

Toutes les fonctions « prédéfinies » ne sont pas internes (au sens explicité ci-dessus). Quelques fonctions prédéfinies sont écrites en SQL.

35.9. Fonctions en langage C

Les fonctions définies par l'utilisateur peuvent être écrites en C (ou dans un langage pouvant être rendu compatible avec C, comme le C++). Ces fonctions sont compilées en objets dynamiques chargeables (encore appelés bibliothèques partagées) et sont chargées par le serveur à la demande. Cette caractéristique de chargement dynamique est ce qui distingue les fonctions en « langage C » des fonctions « internes » -- les véritables conventions de codage sont essentiellement les mêmes pour les deux (c'est pourquoi la bibliothèque standard de fonctions internes est une source abondante d'exemples de code pour les fonctions C définies par l'utilisateur).

Deux différentes conventions d'appel sont actuellement en usage pour les fonctions C. La plus récente, « version 1 », est indiquée en écrivant une macro d'appel `PG_FUNCTION_INFO_V1()` comme illustré ci-après. L'absence d'une telle macro indique une fonction écrite selon l'ancien style (« version 0 »). Le nom de langage spécifié dans la commande **CREATE FUNCTION** est `C` dans les deux cas. Les fonctions suivant l'ancien style sont maintenant déconseillées en raison de problèmes de portabilité et d'un manque de fonctionnalité mais elles sont encore supportées pour des raisons de compatibilité.

35.9.1. Chargement dynamique

La première fois qu'une fonction définie par l'utilisateur dans un fichier objet particulier chargeable est appelée dans une session,

le chargeur dynamique charge ce fichier objet en mémoire de telle sorte que la fonction peut être appelée. La commande **CREATE FUNCTION** pour une fonction en C définie par l'utilisateur doit par conséquent spécifier deux éléments d'information pour la fonction : le nom du fichier objet chargeable et le nom en C (lien symbolique) de la fonction spécifique à appeler à l'intérieur de ce fichier objet. Si le nom en C n'est pas explicitement spécifié, il est supposé être le même que le nom de la fonction SQL.

L'algorithme suivant, basé sur le nom donné dans la commande **CREATE FUNCTION**, est utilisé pour localiser le fichier objet partagé :

1. Si le nom est un chemin absolu, le fichier est chargé.
2. Si le nom commence par la chaîne `$libdir`, cette chaîne est remplacée par le nom du répertoire de la bibliothèque du paquetage PostgreSQL™, qui est déterminé au moment de la compilation.
3. Si le nom ne contient pas de partie répertoire, le fichier est recherché par le chemin spécifié dans la variable de configuration dynamic_library_path.
4. Dans les autres cas, (nom de fichier non trouvé dans le chemin ou ne contenant pas de partie répertoire non absolu), le chargeur dynamique essaiera d'utiliser le nom donné, ce qui échouera très vraisemblablement (dépendre du répertoire de travail en cours n'est pas fiable).

Si cette séquence ne fonctionne pas, l'extension pour les noms de fichier des bibliothèques partagées spécifique à la plateforme (souvent `.so`) est ajoutée au nom attribué et la séquence est à nouveau tentée. En cas de nouvel échec, le chargement échoue.

Il est recommandé de localiser les bibliothèques partagées soit relativement à `$libdir` ou via le chemin dynamique des bibliothèques. Ceci simplifie les mises à jour de versions si la nouvelle installation est à un emplacement différent. Le répertoire actuel représenté par `$libdir` est trouvable avec la commande `pg_config --pkglibdir`.

L'identifiant utilisateur sous lequel fonctionne le serveur PostgreSQL™ doit pouvoir suivre le chemin jusqu'au fichier que vous essayez de charger. Une erreur fréquente revient à définir le fichier ou un répertoire supérieur comme non lisible et/ou non exécutable par l'utilisateur `postgres`.

Dans tous les cas, le nom de fichier donné dans la commande **CREATE FUNCTION** est enregistré littéralement dans les catalogues systèmes, de sorte que, si le fichier doit être à nouveau chargé, la même procédure sera appliquée.

> **Note**
>
> PostgreSQL™ ne compilera pas une fonction C automatiquement. Le fichier objet doit être compilé avant d'être référencé dans une commande **CREATE FUNCTION**. Voir la Section 35.9.6, « Compiler et lier des fonctions chargées dynamiquement » pour des informations complémentaires.

Pour s'assurer qu'un fichier objet chargeable dynamiquement n'est pas chargé dans un serveur incompatible, PostgreSQL™ vérifie que le fichier contient un « bloc magique » avec un contenu approprié. Ceci permet au serveur de détecter les incompatibilités évidentes comme du code compilet pour une version majeure différente de PostgreSQL™. Un bloc magique est requis à partir de PostgreSQL™ 8.2. Pour inclure un bloc magique, écrivez ceci dans un (et seulement un) des fichiers source du module, après avoir inclus l'en-tête `fmgr.h` :

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

Le test `#ifdef` peut être omis si le code n'a pas besoin d'être compilé avec des versions de PostgreSQL™ antérieures à la 8.2.

Après avoir été utilisé pour la première fois, un fichier objet chargé dynamiquement est conservé en mémoire. Les futurs appels de fonction(s) dans ce fichier pendant la même session provoqueront seulement une légère surcharge due à la consultation d'une table de symboles. Si vous devez forcer le chargement d'un fichier objet, par exemple après une recompilation, commencez une nouvelle session.

De façon optionnelle, un fichier chargé dynamiquement peut contenir des fonctions d'initialisation et de terminaison. Si le fichier inclut une fonction nommée `_PG_init`, cette fonction sera appelée immédiatement après le chargement du fichier. La fonction ne reçoit aucun paramètre et doit renvoyer void. Si le fichier inclut une fonction nommée `_PG_fini`, cette fonction sera appelée tout juste avant le déchargement du fichier. De la même façon, la fonction ne reçoit aucun paramètre et doit renvoyer void. Notez que `_PG_fini` sera seulement appelée lors du déchargement du fichier, pas au moment de la fin du processus. (Actuellement, les déchargements sont désactivés et ne surviendront jamais, bien que cela puisse changer un jour.)

35.9.2. Types de base dans les fonctions en langage C

Pour savoir comment écrire des fonctions en langage C, vous devez savoir comment PostgreSQL™ représente en interne les types

de données de base et comment elles peuvent être passés vers et depuis les fonctions. En interne, PostgreSQL™ considère un type de base comme un « blob de mémoire ». Les fonctions que vous définissez sur un type définissent à leur tour la façon que PostgreSQL™ opère sur lui. C'est-à-dire que PostgreSQL™ ne fera que conserver et retrouver les données sur le disque et utilisera votre fonction pour entrer, traiter et restituer les données.

Les types de base peuvent avoir un des trois formats internes suivants :

- passage par valeur, longueur fixe ;
- passage par référence, longueur fixe ;
- passage par référence, longueur variable.

Les types par valeur peuvent seulement avoir une longueur de 1, 2 ou 4 octets (également 8 octets si `sizeof(Datum)` est de huit octets sur votre machine). Vous devriez être attentif lors de la définition de vos types de sorte à qu'ils aient la même taille sur toutes les architectures. Par exemple, le type `long` est dangereux car il a une taille de quatre octets sur certaines machines et huit octets sur d'autres, alors que le type int est de quatre octets sur la plupart des machines Unix. Une implémentation raisonnable du type int4 sur une machine Unix pourrait être

```
/* entier sur quatre octets, passé par valeur */
typedef int int4;
```

D'autre part, les types à longueur fixe d'une taille quelconque peuvent être passés par référence. Par exemple, voici l'implémentation d'un type PostgreSQL™ :

```
/* structure de 16 octets, passée par référence */
typedef struct
{
    double  x, y;
} Point;
```

Seuls des pointeurs vers de tels types peuvent être utilisés en les passant dans et hors des fonctions PostgreSQL™. Pour renvoyer une valeur d'un tel type, allouez la quantité appropriée de mémoire avec `palloc`, remplissez la mémoire allouée et renvoyez un pointeur vers elle (de plus, si vous souhaitez seulement renvoyer la même valeur qu'un de vos arguments en entrée qui se trouve du même type, vous pouvez passer le `palloc` supplémentaire et simplement renvoyer le pointeur vers la valeur en entrée).

Enfin, tous les types à longueur variable doivent aussi être passés par référence. Tous les types à longueur variable doivent commencer avec un champ d'une longueur d'exactement quatre octets et toutes les données devant être stockées dans ce type doivent être localisées dans la mémoire à la suite immédiate de ce champ longueur. Le champ longueur contient la longueur totale de la structure, c'est-à-dire incluant la longueur du champ longueur lui-même.

Another important point is to avoid leaving any uninitialized bits within data type values; for example, take care to zero out any alignment padding bytes that might be present in structs. Without this, logically-equivalent constants of your data type might be seen as unequal by the planner, leading to inefficient (though not incorrect) plans.

Avertissement

Ne *jamais* modifier le contenu d'une valeur en entrée passée par référence. Si vous le faites, il y a de forts risques pour que vous réussissiez à corrompre les données sur disque car le pointeur que vous avez reçu pourrait bien pointer directement vers un tampon disque. La seule exception à cette règle est expliquée dans la Section 35.10, « Agrégats utilisateur ».

Comme exemple, nous pouvons définir le type text comme ceci :

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Il est évident que le champ déclaré ici n'est pas assez long pour contenir toutes les chaînes possibles. Comme il est impossible de déclarer une structure de taille variable en C, nous nous appuyons sur le fait que le compilateur C ne vérifie pas la plage des indices de tableau. Nous allouons juste la quantité d'espace nécessaire et ensuite nous accédons au tableau comme s'il avait été déclaré avec la bonne longueur (c'est une astuce courante que vous pouvez trouver dans beaucoup de manuels de C).

En manipulant les types à longueur variable, nous devons être attentifs à allouer la quantité correcte de mémoire et à fixer correctement le champ longueur. Par exemple, si nous voulons stocker 40 octets dans une structure text, nous devrions utiliser un fragment de code comme celui-ci :

```
#include "postgres.h"
```

```
...
char buffer[40]; /* notre donnée source */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` est équivalent à `sizeof(int4)` mais est considéré comme une meilleure tournure de référence à la taille de l'overhead pour un type à longueur variable. De plus, le champ length *doit* être configuré en utilisant la macro `SET_VARSIZE`, et non plus par simple affectation.

Le Tableau 35.1, « Équivalence des types C et des types SQL intégrés » spécifie la correspondance entre les types C et les types SQL quand on écrit une fonction en langage C utilisant les types internes de PostgreSQL™. La colonne « Défini dans » donne le fichier d'en-tête devant être inclus pour accéder à la définition du type (la définition effective peut se trouver dans un fichier différent inclus dans le fichier indiqué. Il est recommandé que les utilisateurs s'en tiennent à l'interface définie). Notez que vous devriez toujours inclure `postgres.h` en premier dans tout fichier source car il déclare un grand nombre d'éléments dont vous aurez besoin de toute façon.

Tableau 35.1. Équivalence des types C et des types SQL intégrés

Type SQL	Type C	Défini dans
abstime	AbsoluteTime	`utils/nabstime.h`
boolean	bool	`postgres.h` (intégration au compilateur)
box	BOX*	`utils/geo_decls.h`
bytea	bytea*	`postgres.h`
"char"	char	(intégré au compilateur)
character	BpChar*	`postgres.h`
cid	CommandId	`postgres.h`
date	DateADT	`utils/date.h`
smallint (int2)	int2 or int16	`postgres.h`
int2vector	int2vector*	`postgres.h`
integer (int4)	int4 or int32	`postgres.h`
real (float4)	float4*	`postgres.h`
double precision (float8)	float8*	`postgres.h`
interval	Interval*	`utils/timestamp.h`
lseg	LSEG*	`utils/geo_decls.h`
name	Name	`postgres.h`
oid	Oid	`postgres.h`
oidvector	oidvector*	`postgres.h`
path	PATH*	`utils/geo_decls.h`
point	POINT*	`utils/geo_decls.h`
regproc	regproc	`postgres.h`
reltime	RelativeTime	`utils/nabstime.h`
text	text*	`postgres.h`
tid	ItemPointer	`storage/itemptr.h`
time	TimeADT	`utils/date.h`
time with time zone	TimeTzADT	`utils/date.h`
timestamp	Timestamp*	`utils/timestamp.h`
tinterval	TimeInterval	`utils/nabstime.h`
varchar	VarChar*	`postgres.h`
xid	TransactionId	`postgres.h`

Maintenant que nous avons passé en revue toutes les structures possibles pour les types de base, nous pouvons donner quelques exemples de vraies fonctions.

35.9.3. Conventions d'appel de la version 0

Nous présentons l'« ancien style » de convention d'appel en premier -- bien que cette approche soit maintenant déconseillée, elle est plus facile à maîtriser au début. Dans la méthode version-0, les arguments et résultats de la fonction C sont simplement déclarés dans le style C normal mais en faisant attention à utiliser la représentation C de chaque type de données SQL comme montré ci-dessus.

Voici quelques exemples :

```
#include "postgres.h"
#include <string.h>
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* par valeur */

int
add_one(int arg)
{
    return arg + 1;
}

/* par référence, taille fixe */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* par référence, taille variable */

text *
copytext(text *t)
{
    /*
     * VARSIZE est la taille totale de la structure en octets.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATA est un pointeur sur la région de données de la structure.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);  /* nombre d'octets */
    return new_t;
}
```

```
text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    return new_text;
}
```

En supposant que le code ci-dessus ait été écrit dans le fichier `funcs.c` et compilé en objet partagé, nous pourrions définir les fonctions pour PostgreSQL™ avec des commandes comme ceci :

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- notez la surcharge du nom de la fonction SQL "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

Ici, *`DIRECTORY`* représente le répertoire contenant le fichier de la bibliothèque partagée (par exemple le répertoire du tutoriel de PostgreSQL™, qui contient le code des exemples utilisés dans cette section). (Un meilleur style aurait été d'écrire seulement `'funcs'` dans la clause `AS`, après avoir ajouté *`DIRECTORY`* au chemin de recherche. Dans tous les cas, nous pouvons omettre l'extension spécifique au système pour les bibliothèques partagées, communément `.so` ou `.sl`.)

Remarquez que nous avons spécifié la fonction comme « STRICT », ce qui signifie que le système devra automatiquement supposer un résultat NULL si n'importe quelle valeur d'entrée est NULL. Ainsi, nous évitons d'avoir à vérifier l'existence d'entrées NULL dans le code de la fonction. Sinon, nous aurions dû contrôler explicitement les valeurs NULL en testant un pointeur NULL pour chaque argument passé par référence (pour les arguments passés par valeur, nous n'aurions même aucun moyen de contrôle !).

Bien que cette convention d'appel soit simple à utiliser, elle n'est pas très portable ; sur certaines architectures, il y a des problèmes pour passer de cette manière des types de données plus petits que int. De plus, il n'y a pas de moyen simple de renvoyer un résultat NULL, ni de traiter des arguments NULL autrement qu'en rendant la fonction strict. La convention version-1, présentée ci-après, permet de surmonter ces objections.

35.9.4. Conventions d'appel de la version 1

La convention d'appel version-1 repose sur des macros pour supprimer la plus grande partie de la complexité du passage d'arguments et de résultats. La déclaration C d'une fonction en version-1 est toujours :

```
Datum nom_fonction(PG_FUNCTION_ARGS)
```

De plus, la macro d'appel :

```
PG_FUNCTION_INFO_V1(nom_fonction);
```

doit apparaître dans le même fichier source (par convention, elle est écrite juste avant la fonction elle-même). Cette macro n'est pas nécessaire pour les fonctions `internal` puisque PostgreSQL™ assume que toutes les fonctions internes utilisent la convention version-1. Elle est toutefois requise pour les fonctions chargées dynamiquement.

Dans une fonction version-1, chaque argument existant est traité par une macro `PG_GETARG_`*`xxx`*`()` correspondant au type de

donnée de l'argument et le résultat est renvoyé par une macro `PG_RETURN_xxx()` correspondant au type renvoyé. `PG_GETARG_xxx()` prend comme argument le nombre d'arguments de la fonction à parcourir, le compteur commençant à 0. `PG_RETURN_xxx()` prend comme argument la valeur effective à renvoyer.

Voici la même fonction que précédemment, codée en style version-1

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* par valeur */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* par référence, longueur fixe */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* La macro pour FLOAT8 cache sa nature de passage par référence. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Ici, la nature de passage par référence de Point n'est pas cachée. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* par référence, longueur variable */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE est la longueur totale de la structure en octets.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
```

```
     * VARDATA est un pointeur vers la région de données de la structure.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);    /* nombre d'octets */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

Les commandes **CREATE FUNCTION** sont les mêmes que pour leurs équivalents dans la version-0.

Au premier coup d'œil, les conventions de codage de la version-1 peuvent sembler inutilement obscures. Pourtant, elles offrent nombre d'améliorations car les macros peuvent cacher les détails superflus. Un exemple est donné par la fonction `add_one_float8` où nous n'avons plus besoin de prêter attention au fait que le type float8 est passé par référence. Un autre exemple de simplification est donné par les macros pour les types à longueur variable `GETARG` qui permettent un traitement plus efficace des valeurs « toasted » (compressées ou hors-ligne).

Une des grandes améliorations dans les fonctions version-1 est le meilleur traitement des entrées et des résultats NULL. La macro `PG_ARGISNULL(n)` permet à une fonction de tester si chaque entrée est NULL (évidemment, ceci n'est nécessaire que pour les fonctions déclarées non « STRICT »). Comme avec les macros `PG_GETARG_xxx()`, les arguments en entrée sont comptés à partir de zéro. Notez qu'on doit se garder d'exécuter `PG_GETARG_xxx()` jusqu'à ce qu'on ait vérifié que l'argument n'est pas NULL. Pour renvoyer un résultat NULL, exécutez la fonction `PG_RETURN_NULL()` ; ceci convient aussi bien dans les fonctions STRICT que non STRICT.

Les autres options proposées dans l'interface de nouveau style sont deux variantes des macros `PG_GETARG_xxx()`. La première d'entre elles, `PG_GETARG_xxx_COPY()`, garantit le renvoi d'une copie de l'argument spécifié où nous pouvons écrire en toute sécurité (les macros normales peuvent parfois renvoyer un pointeur vers une valeur physiquement mise en mémoire dans une table qui ne doit pas être modifiée. En utilisant les macros `PG_GETARG_xxx_COPY()`, on garantit l'écriture du résultat). La seconde variante se compose des macros `PG_GETARG_xxx_SLICE()` qui prennent trois arguments. Le premier est le nombre d'arguments de la fonction (comme ci-dessus). Le second et le troisième sont le décalage et la longueur du segment qui doit être renvoyé. Les décalages sont comptés à partir de zéro et une longueur négative demande le renvoi du reste de la valeur. Ces macros procurent un accès plus efficace à des parties de valeurs à grande dimension dans le cas où elles ont un type de stockage en mémoire « external » (le type de stockage d'une colonne peut être spécifié en utilisant `ALTER TABLE nom_table ALTER COLUMN nom_colonne SET STORAGE typestockage`. `typestockage` est un type parmi `plain`, `external`, `extended` ou `main`).

Enfin, les conventions d'appels de la version-1 rendent possible le renvoi de résultats d'ensemble (Section 35.9.10, « Renvoi d'ensembles »), l'implémentation de fonctions déclencheurs (Chapitre 36, Déclencheurs (triggers)) et d'opérateurs d'appel de langage procédural (Chapitre 49, Écrire un gestionnaire de langage procédural). Le code version-1 est aussi plus portable que celui de version-0 car il ne contrevient pas aux restrictions du protocole d'appel de fonction en C standard. Pour plus de détails, voir `src/backend/utils/fmgr/README` dans les fichiers sources de la distribution.

35.9.5. Écriture du code

Avant de nous intéresser à des sujets plus avancés, nous devons discuter de quelques règles de codage des fonctions en langage C de PostgreSQL™. Bien qu'il soit possible de charger des fonctions écrites dans des langages autre que le C dans PostgreSQL™, c'est habituellement difficile (quand c'est possible) parce que les autres langages comme C++, FORTRAN ou Pascal ne suivent pas fréquemment les mêmes conventions de nommage que le C. C'est-à-dire que les autres langages ne passent pas les arguments et ne renvoient pas les valeurs entre fonctions de la même manière. Pour cette raison, nous supposerons que nos fonctions en langage C sont réellement écrites en C.

Les règles de base pour l'écriture de fonctions C sont les suivantes :

- Utilisez `pg_config --includedir-server` pour découvrir où sont installés les fichiers d'en-tête du serveur PostgreSQL™ sur votre système (ou sur le système de vos utilisateurs).
- Compilez et liez votre code de façon à ce qu'il soit chargé dynamiquement dans PostgreSQL™, ce qui requiert des informations spéciales. Voir Section 35.9.6, « Compiler et lier des fonctions chargées dynamiquement » pour une explication détaillée sur la façon de le faire pour votre système d'exploitation spécifique.
- Rappelez-vous to définir un « bloc magique » pour votre bibliothèque partagée, comme décrit dans Section 35.9.1, « Chargement dynamique ».
- Quand vous allouez de la mémoire, utilisez les fonctions PostgreSQL™ `palloc` et `pfree` au lieu des fonctions correspondantes `malloc` et `free` de la bibliothèque C. La mémoire allouée par `palloc` sera libérée automatiquement à la fin de chaque transaction, empêchant des débordements de mémoire.
- Remettez toujours à zéro les octets de vos structures en utilisant `memset` (ou allouez-les avec `palloc0` dès le départ). Même si vous faites l'affectation de chaque champ de votre structure, il pourrait rester des données indésirables dues à l'alignement des champs (en quelque sorte, des trous dans la structure). Si vous ne le faites pas, il est difficile de supporter les index ou les jointures de découpage car vous devez retenir seulement les bits significatifs de votre structure de donnée pour calculer un découpage. Le planificateur se base parfois sur la comparaison de constantes par une égalité bit à bit, donc vous pouvez obtenir des résultats indésirables si des valeurs, logiquement équivalentes, ne sont pas identiques bit à bit.
- La plupart des types internes PostgreSQL™ sont déclarés dans `postgres.h` alors que les interfaces de gestion des fonctions (PG_FUNCTION_ARGS, etc.) sont dans `fmgr.h`. Du coup, vous aurez besoin d'inclure au moins ces deux fichiers. Pour des raisons de portabilité, il vaut mieux inclure `postgres.h` *en premier* avant tout autre fichier d'en-tête système ou utilisateur. En incluant `postgres.h`, il incluera également `elog.h` et `palloc.h` pour vous.
- Les noms de symboles définis dans les objets ne doivent pas entrer en conflit entre eux ou avec les symboles définis dans les exécutables du serveur PostgreSQL™. Vous aurez à renommer vos fonctions ou variables si vous recevez un message d'erreur à cet effet.

35.9.6. Compiler et lier des fonctions chargées dynamiquement

Avant de pouvoir être utilisées dans PostgreSQL™, les fonctions d'extension écrites en C doivent être compilées et liées d'une certaine façon, ceci afin de produire un fichier dynamiquement chargeable par le serveur. Pour être plus précis, une *bibliothèque partagée* doit être créée.

Pour obtenir plus d'informations que celles contenues dans cette section, il faut se référer à la documentation du système d'exploitation, en particulier les pages traitant du compilateur C, de **cc** et de l'éditeur de lien, **ld**. Par ailleurs, le code source de PostgreSQL™ contient de nombreux exemples fonctionnels dans le répertoire `contrib`. Néanmoins, ces exemples entraînent la création de modules qui dépendent de la disponibilité du code source de PostgreSQL™.

La création de bibliothèques partagées est un processus analogue à celui utilisé pour lier des exécutables : les fichiers sources sont d'abord compilés en fichiers objets puis sont liées ensemble. Les fichiers objets doivent être compilés sous la forme de *code indépendant de sa position* (PIC, acronyme de *position-independent code*) . Conceptuellement, cela signifie qu'ils peuvent être placés dans une position arbitraire de la mémoire lorsqu'ils sont chargés par l'exécutable. (Les fichiers objets destinés aux exécutables ne sont généralement pas compilés de cette manière.) La commande qui permet de lier des bibliothèques partagées nécessite des options spéciales qui la distinguent de celle permettant de lier un exécutable. En théorie, tout du moins. La réalité est, sur certains systèmes, beaucoup plus complexe.

Les exemples suivants considèrent que le code source est un fichier `foo.c` et qu'une bibliothèque partagée `foo.so` doit être créée. Sans précision, le fichier objet intermédiaire est appelé `foo.o`. Une bibliothèque partagée peut contenir plusieurs fichiers objet. Cela dit, un seul est utilisé ici.

`BSD/OS`

L'option du compilateur pour créer des PIC est `-fpic`. L'option de l'éditeur de liens pour créer des bibliothèques partagées est `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

Ceci est applicable à partir de la version 4.0 de `BSD/OS`.

`FreeBSD`

L'option du compilateur pour créer des PIC est `-fpic`. L'option de l'éditeur de liens pour créer des bibliothèques partagées

est `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Ceci est applicable à partir de la version 3.0 de `FreeBSD`.

`HP-UX`

L'option du compilateur du système pour créer des PIC est `+z`. Avec GCC, l'option est `-fpic`. Le commutateur de l'éditeur de liens pour les bibliothèques partagées est `-b`. Ainsi :

```
cc +z -c foo.c
```

ou :

```
gcc -fpic -c foo.c
```

puis :

```
ld -b -o foo.sl foo.o
```

`HP-UX` utilise l'extension `.sl` pour les bibliothèques partagées, à la différence de la plupart des autres systèmes.

`IRIX`

PIC est l'option par défaut. Aucune option de compilation particulière n'est nécessaire. Le commutateur de l'éditeur de liens pour produire des bibliothèques partagées est `-shared` :

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

`Linux`

L'option du compilateur pour créer des PIC est `-fpic`. Sur certaines plateformes et dans certaines situations, `-fPIC` doit être utilisé si `-fpic` ne fonctionne pas. Le manuel de GCC donne plus d'informations. L'option de compilation pour créer des bibliothèques partagées est `-shared`. Un exemple complet ressemble à :

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

`MacOS X`

L'exemple suivant suppose que les outils de développement sont installés.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

`NetBSD`

L'option du compilateur pour créer des PIC est `-fpic`. Pour les systèmes ELF, l'option de compilation pour lier les bibliothèques partagées est `-shared`. Sur les systèmes plus anciens et non-ELF, on utilise `ld -Bshareable`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

`OpenBSD`

L'option du compilateur pour créer des PIC est `-fpic`. Les bibliothèques partagées peuvent être créées avec `ld -Bshareable`.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

`Solaris`

L'option du compilateur pour créer des PIC est `-KPIC` avec le compilateur de Sun et `-fpic` avec GCC. Pour lier les bibliothèques partagées, l'option de compilation est respectivement `-G` ou `-shared`.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

ou

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

`Tru64 UNIX`

Par défaut, il s'agit de PIC. Ainsi, aucune directive particulière n'est à fournir pour la compilation. Pour l'édition de lien, des options spécifiques sont à fournir à **ld** :

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

Une procédure identique doit être employée dans le cas où GCC est utilisé à la place du compilateur du système ; aucune option particulière n'est nécessaire.

`UnixWare`

L'option de compilation pour créer des PIC est `-KPIC` avec le compilateur SCO et `-fpic` avec GCC™. Pour lier des bibliothèques partagées, les options respectives sont `-G` et `-shared`.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

ou

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Astuce

Si cela s'évère trop compliqué, *GNU Libtool*™ peut être utilisé. Cet outil permet de s'affranchir des différences entre les nombreux systèmes au travers d'une interface uniformisée.

La bibliothèque partagée résultante peut être chargée dans PostgreSQL™. Lorsque l'on précise le nom du fichier dans la commande **CREATE FUNCTION**, il faut indiquer le nom de la bibliothèque partagée et non celui du fichier objet intermédiaire. L'extension standard pour les bibliothèques partagées (en général `.so` ou `.sl`) peut être omise dans la commande **CREATE FUNCTION**, et doit l'être pour une meilleure portabilité.

La Section 35.9.1, « Chargement dynamique » indique l'endroit où le serveur s'attend à trouver les fichiers de bibliothèques partagées.

35.9.7. Infrastructure de construction d'extensions

Si vous pensez distribuer vos modules d'extension PostgreSQL™, configurer un système de construction portable peut être assez compliqué. Du coup, l'installation de PostgreSQL™ fournit une infrastructure de construction pour les extensions, appelée PGXS, pour que les modules d'extension simples puissent être construit simplement avec un serveur déjà installé. Notez que cette infrastructure n'a pas pour but d'être un ensemble de travail universel pouvant être utilisé pour construire tous les logiciels s'interfaçant avec PostgreSQL™ ; il automatise simplement les règles de construction communes pour les modules simples d'extension du serveur. Pour des paquetages plus complexes, vous aurez besoin d'écrire votre propre système de construction.

Pour utiliser l'infrastructure de votre extension, vous devez écrire un simple fichier makefile. Dans ce fichier, vous devez configurer quelques variables et inclure enfin le makefile global PGXS. Voici un exemple qui construit un module d'extension nommé `isbn_issn` consistant en une bibliothèque partagée, un script SQL et un fichier texte de documentation :

```
MODULES = isbn_issn
DATA_built = isbn_issn.sql
DOCS = README.isbn_issn

PG_CONFIG := pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Les trois dernières lignes devraient toujours être identiques. Plus tôt dans le fichier, vous affectez des variables ou ajoutez vos propres règles pour make.

Set one of these three variables to specify what is built :

`MODULES`
liste des objets partagés à construire à partir des fichiers source de même base (ne pas inclure le suffixe dans cette liste)

`MODULE_big`
un objet partagé à construire à partir de plusieurs fichiers source (liste les fichiers objets dans `OBJS`)

`PROGRAM`
: un exécutable binaire à construire (liste les fichiers objets dans `OBJS`)

Les variables suivantes peuvent être configurées :

`MODULEDIR`
: sous-répertoire dans lequel les fichiers DATA et DOCS doivent être installés (si non configuré, la variable vaut par défaut `contrib`)

`DATA`
: fichiers spécifiques à installer dans *`prefix`*`/share/$MODULEDIR`

`DATA_built`
: fichiers spécifiques à installer dans *`prefix`*`/share/$MODULEDIR` qui ont besoin d'être construit au début

`DATA_TSEARCH`
: quelques fichiers à installer sous *`prefix`*`/share/tsearch_data`

`DOCS`
: fichiers spécifiques à installer dans *`prefix`*`/doc/$MODULEDIR`

`SCRIPTS`
: fichiers script (pas des binaires) à installer dans *`prefix`*`/bin`

`SCRIPTS_built`
: fichiers script (pas des binaires) à installer dans *`prefix`*`/bin` qui ont besoin d'être construit au début

`REGRESS`
: liste des cas de tests de regression (sans suffixe), voir ci-dessous

`EXTRA_CLEAN`
: fichiers supplémentaires à supprimer dans `make clean`

`PG_CPPFLAGS`
: sera ajouté à `CPPFLAGS`

`PG_LIBS`
: sera ajouté à la ligne de liens `PROGRAM`

`SHLIB_LINK`
: sera ajouté à la ligne de lien `MODULE_big`

`PG_CONFIG`
: chemin vers le programme pg_config indiquant l'installation de PostgreSQL™ qui sert à la construction (généralement `pg_config` pour utiliser le premier programme trouvé dans la variable d'environnement `PATH`)

Nommez ce fichier `Makefile` et placez le dans le répertoire qui contient votre extension. Ensuite, vous pouvez lancer la compilation avec `make`, et plus tard `make install` pour installer votre module. Par défaut, l'extension est compilée et installée pour l'installation de PostgreSQL™ qui correspond au premier programme **pg_config** trouvée dans votre chemin. Vous pouvez utiliser utiliser une autre installation en configurant `PG_CONFIG` pour qu'il pointe vers le bon **pg_config**, soit dans le fichier Makefile soit sur la ligne de commande de `make`.

Attention

Modifier `PG_CONFIG` fonctionne seulement en construisant à partir d'une installation PostgreSQL™ 8.3 ou ultérieure. Les anciennes versions ne l'utilisent pas ; vous devez donc modifier votre variable `PATH` pour sélectionner la bonne installation.

Les scripts listés dans la variable `REGRESS` sont utilisés pour les tests de régression de votre module, tout comme `make installcheck` est utilisé pour votre serveur PostgreSQL™. Pour que ceci fonctionne, vous avez besoin d'avoir un sous-répertoire nommé `sql/` dans le répertoire de votre extension, à l'intérieur duquel vous placez un fichier pour chaque groupe de tests que vous voulez exécuter. Les fichiers doivent avoir l'extension `.sql`, qui ne seront pas inclus dans la liste `REGRESS` dans le Makefile. Pour chaque test, il doit exister un fichier contenant le résultat attendu dans un sous-répertoire nommé `expected/`, avec l'extension `.out`. Les tests sont exécutés via `make installcheck`, et le résultat sera comparé aux fichiers attendus. Les différences seront écrites dans le fichier `regression.diffs` dans un format **diff -c**. Notez qu'essayer d'exécuter un test à qui il manque le fichier sera rapporté comme un « problème », donc assurez-vous que vous avez bien tous les fichiers attendus.

> **Astuce**
>
> La façon la plus simple de créer les fichiers attendus est de créer des fichiers vides, puis de vérifier avec précaution les fichiers résultants après un test d'exécution (à trouver dans un répertoire `results/`), et les copier dans `expected/` s'ils correspondent à votre souhait suite aux tests.

35.9.8. Arguments de type composite

Les types composites n'ont pas une organisation fixe comme les structures en C. Des instances d'un type composite peuvent contenir des champs NULL. De plus, les types composites faisant partie d'une hiérarchie d'héritage peuvent avoir des champs différents des autres membres de la même hiérarchie. En conséquence, PostgreSQL™ propose une interface de fonction pour accéder depuis le C aux champs des types composites.

Supposons que nous voulions écrire une fonction pour répondre à la requête :

```
SELECT nom, c_surpaye(emp, 1500) AS surpaye
    FROM emp
    WHERE nom = 'Bill' OR nom = 'Sam';
```

En utilisant les conventions d'appel de la version 0, nous pouvons définir `c_surpaye` comme :

```
#include "postgres.h"
#include "executor/executor.h"  /* pour GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

bool
c_surpaye(HeapTupleHeader *t, /* la ligne courante d'emp */
           int32 limite)
{
    bool isNULL;
    int32 salaire;

    salaire = DatumGetInt32(GetAttributeByName(t, "salaire", &isNULL));
    if (isNULL)
        return false;
    return salaire > limite;
}
```

Dans le codage version-1, le code ci-dessus devient :

```
#include "postgres.h"
#include "executor/executor.h"  /* pour GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(c_surpaye);

Datum
c_surpaye(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  *t = (HeapTupleHeader *) PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limite = PG_GETARG_INT32(1);
    bool isNULL;
    Datum salaire;

    salaire = GetAttributeByName(t, "salaire", &isNULL);
    if (isNULL)
        PG_RETURN_BOOL(false);
    /* Autrement, nous pourrions préférer de lancer PG_RETURN_NULL() pour un
       salaire NULL.
    */

    PG_RETURN_BOOL(DatumGetInt32(salaire) > limite);
}
```

`GetAttributeByName` est la fonction système PostgreSQL™ qui renvoie les attributs depuis une colonne spécifiée. Elle a trois arguments : l'argument de type HeapTupleHeader passé à la fonction, le nom de l'attribut recherché et un paramètre de retour qui indique si l'attribut est NULL. `GetAttributeByName` renvoie une valeur de type Datum que vous pouvez convertir dans un type voulu en utilisant la macro appropriée `DatumGet`*`XXX`*`()`. Notez que la valeur de retour est insignifiante si le commutateur NULL est positionné ; il faut toujours vérifier le commutateur NULL avant de commencer à faire quelque chose avec le résultat.

Il y a aussi `GetAttributeByNum`, qui sélectionne l'attribut cible par le numéro de colonne au lieu de son nom.

La commande suivante déclare la fonction `c_surpaye` en SQL :

```
CREATE FUNCTION c_surpaye(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_surpaye'
    LANGUAGE C STRICT;
```

Notez que nous avons utilisé `STRICT` pour que nous n'ayons pas à vérifier si les arguments en entrée sont NULL.

35.9.9. Renvoi de lignes (types composites)

Pour renvoyer une ligne ou une valeur de type composite à partir d'une fonction en langage C, vous pouvez utiliser une API spéciale qui fournit les macros et les fonctions dissimulant en grande partie la complexité liée à la construction de types de données composites. Pour utiliser cette API, le fichier source doit inclure :

```
#include "funcapi.h"
```

Il existe deux façons de construire une valeur de données composites (autrement dit un « tuple ») : vous pouvez le construire à partir d'un tableau de valeurs Datum ou à partir d'un tableau de chaînes C qui peuvent passer dans les fonctions de conversion des types de données du tuple. Quelque soit le cas, vous avez d'abord besoin d'obtenir et de construire un descripteur TupleDesc pour la structure du tuple. En travaillant avec des Datums, vous passez le TupleDesc à `BlessTupleDesc`, puis vous appelez `heap_form_tuple` pour chaque ligne. En travaillant avec des chaînes C, vous passez TupleDesc à `TupleDescGetAttInMetadata`, puis vous appelez `BuildTupleFromCStrings` pour chaque ligne. Dans le cas d'une fonction renvoyant un ensemble de tuple, les étapes de configuration peuvent toutes être entreprises une fois lors du premier appel à la fonction.

Plusieurs fonctions d'aide sont disponibles pour configurer le TupleDesc requis. La façon recommandée de le faire dans la plupart des fonctions renvoyant des valeurs composites est d'appeler :

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

en passant la même structure `fcinfo` que celle passée à la fonction appelante (ceci requiert bien sûr que vous utilisez les conventions d'appel version-1). `resultTypeId` peut être spécifié comme `NULL` ou comme l'adresse d'une variable locale pour recevoir l'OID du type de résultat de la fonction. `resultTupleDesc` devrait être l'adresse d'une variable TupleDesc locale. Vérifiez que le résultat est `TYPEFUNC_COMPOSITE` ; dans ce cas, `resultTupleDesc` a été rempli avec le TupleDesc requis (si ce n'est pas le cas, vous pouvez rapporter une erreur pour une « fonction renvoyant un enregistrement appelé dans un contexte qui ne peut pas accepter ce type enregistrement »).

Astuce

`get_call_result_type` peut résoudre le vrai type du résultat d'une fonction polymorphique ; donc, il est utile pour les fonctions qui renvoient des résultats scalaires polymorphiques, pas seulement les fonctions qui renvoient des types composites. Le résultat `resultTypeId` est principalement utile pour les fonctions renvoyant des scalaires polymorphiques.

Note

`get_call_result_type` a une fonction cousine `get_expr_result_type`, qui peut être utilisée pour résoudre le tupe attendu en sortie en un appel de fonction représenté par un arbre d'expressions. Ceci peut être utilisé pour tenter de déterminer le type de résultat sans entrer dans la fonction elle-même. Il existe aussi `get_func_result_type`, qui peut seulement être utilisée quand l'OID de la fonction est disponible. Néanmoins, ces fonctions ne sont pas capables de gérer les fonctions déclarées renvoyer des enregistrements (record). `get_func_result_type` ne peut pas résoudre les types polymorphiques, donc vous devriez utiliser de préférence `get_call_result_type`.

Les fonctions anciennes, et maintenant obsolètes, qui permettent d'obtenir des TupleDesc sont :

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

pour obtenir un TupleDesc pour le type de ligne d'une relation nommée ou :

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

pour obtenir une TupleDesc basée sur l'OID d'un type. Ceci peut être utilisé pour obtenir un TupleDesc soit pour un type de base, soit pour un type composite. Néanmoins, cela ne fonctionnera pas pour une fonction qui renvoie record et cela ne résoudra pas les types polymorphiques.

Une fois que vous avez un TupleDesc, appelez :

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

si vous pensez travailler avec des Datums ou :

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

si vous pensez travailler avec des chaînes C. Si vous écrivez une fonction renvoyant un ensemble, vous pouvez sauvegarder les résultats de ces fonctions dans la structure dans le FuncCallContext -- utilisez le champ *`tuple_desc`* ou *`attinmeta`* respectivement.

Lorsque vous fonctionnez avec des Datums, utilisez :

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

pour construire une donnée utilisateur HeapTuple indiquée dans le format Datum.

Lorsque vous travaillez avec des chaînes C, utilisez :

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

pour construire une donnée utilisateur HeapTuple indiquée dans le format des chaînes C. `values` est un tableau de chaîne C, une pour chaque attribut de la ligne renvoyée. Chaque chaîne C doit être de la forme attendue par la fonction d'entrée du type de donnée de l'attribut. Afin de renvoyer une valeur NULL pour un des attributs, le pointeur correspondant dans le tableau de valeurs (*`values`*) doit être fixé à NULL. Cette fonction demandera à être appelée pour chaque ligne que vous renvoyez.

Une fois que vous avez construit un tuple devant être renvoyé par votre fonction, vous devez le convertir en type Datum. Utilisez :

```
HeapTupleGetDatum(HeapTuple tuple)
```

pour convertir un type HeapTuple en un Datum valide. Ce Datum peut être renvoyé directement si vous envisagez de renvoyer juste une simple ligne ou bien il peut être utilisé pour renvoyer la valeur courante dans une fonction renvoyant un ensemble.

Un exemple figure dans la section suivante.

35.9.10. Renvoi d'ensembles

Il existe aussi une API spéciale procurant le moyen de renvoyer des ensembles (lignes multiples) depuis une fonction en langage C. Une fonction renvoyant un ensemble doit suivre les conventions d'appel de la version-1. Aussi, les fichiers source doivent inclure l'en-tête `funcapi.h`, comme ci-dessus.

Une fonction renvoyant un ensemble (SRF : « set returning function ») est appelée une fois pour chaque élément qu'elle renvoie. La SRF doit donc sauvegarder suffisamment l'état pour se rappeler ce qu'elle était en train de faire et renvoyer le prochain élément à chaque appel. La structure FuncCallContext est offerte pour assister le contrôle de ce processus. À l'intérieur d'une fonction, `fcinfo->flinfo->fn_extra` est utilisée pour conserver un pointeur vers FuncCallContext au cours des appels successifs.

```
typedef struct
{
    /*
     * Number of times we've been called before
     *
     * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(), and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     *
     * max_calls is here for convenience only and setting it is optional.
     * If not set, you must provide alternative means to know when the
     * function is done.
     */
```

```
    uint32 max_calls;

    /*
     * OPTIONAL pointer to result slot
     *
     * This is obsolete and only present for backwards compatibility, viz,
     * user-defined SRFs that use the deprecated TupleDescGetSlot().
     */
    TupleTableSlot *slot;

    /*
     * OPTIONAL pointer to miscellaneous user-provided context information
     *
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your function.
     */
    void *user_fctx;

    /*
     * OPTIONAL pointer to struct containing attribute type input metadata
     *
     * attinmeta is for use when returning tuples (i.e., composite data types)
     * and is not used when returning base data types. It is only needed
     * if you intend to use BuildTupleFromCStrings() to create the return
     * tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for multiple calls
     *
     * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you, and used
     * by SRF_RETURN_DONE() for cleanup. It is the most appropriate memory
     * context for any memory that is to be reused across multiple calls
     * of the SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
    * OPTIONAL pointer to struct containing tuple description
    *
    * tuple_desc is for use when returning tuples (i.e. composite data types)
    * and is only needed if you are going to build the tuples with
    * heap_form_tuple() rather than with BuildTupleFromCStrings().  Note that
    * the TupleDesc pointer stored here should usually have been run through
    * BlessTupleDesc() first.
    */
    TupleDesc tuple_desc;

} FuncCallContext;
```

Une SRF utilise plusieurs fonctions et macros qui manipulent automatiquement la structure FuncCallContext (et s'attendent à la trouver via `fn_extra`). Utilisez :

```
SRF_IS_FIRSTCALL()
```

pour déterminer si votre fonction est appelée pour la première fois. Au premier appel, utilisez :

```
SRF_FIRSTCALL_INIT()
```

pour initialiser la structure FuncCallContext. À chaque appel de fonction, y compris le premier, utilisez :

```
SRF_PERCALL_SETUP()
```

pour une mise à jour correcte en vue de l'utilisation de FuncCallContext et pour nettoyer toutes les données renvoyées précédemment et conservées depuis le dernier passage de la fonction.

Si votre fonction a des données à renvoyer, utilisez :

```
SRF_RETURN_NEXT(funcctx, result)
```

pour les renvoyer à l'appelant. (`result` doit être de type Datum, soit une valeur simple, soit un tuple préparé comme décrit ci-

dessus.) Enfin, quand votre fonction a fini de renvoyer des données, utilisez :

```
SRF_RETURN_DONE(funcctx)
```

pour nettoyer et terminer la SRF.

Lors de l'appel de la SRF, le contexte mémoire courant est un contexte transitoire qui est effacé entre les appels. Cela signifie que vous n'avez pas besoin d'appeler `pfree` sur tout ce que vous avez alloué en utilisant `palloc` ; ce sera supprimé de toute façon. Toutefois, si vous voulez allouer des structures de données devant persister tout au long des appels, vous avez besoin de les conserver quelque part. Le contexte mémoire référencé par *`multi_call_memory_ctx`* est un endroit approprié pour toute donnée devant survivre jusqu'à l'achèvement de la fonction SRF. Dans la plupart des cas, cela signifie que vous devrez basculer vers *`multi_call_memory_ctx`* au moment de la préparation du premier appel.

Voici un exemple complet de pseudo-code :

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Here we are done returning items and just need to clean up: */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

Et voici un exemple complet d'une simple SRF retournant un type composite :

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;
```

```
    /* stuff done only on the first call of the function */
    if (SRF_IS_FIRSTCALL())
    {
       MemoryContext oldcontext;

       /* create a function context for cross-call persistence */
       funcctx = SRF_FIRSTCALL_INIT();

       /* switch to memory context appropriate for multiple function calls */
       oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

       /* total number of tuples to be returned */
       funcctx->max_calls = PG_GETARG_UINT32(0);

       /* Build a tuple descriptor for our result type */
       if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
           ereport(ERROR,
                   (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                    errmsg("function returning record called in context "
                           "that cannot accept type record")));

       /*
        * generate attribute metadata needed later to produce tuples from raw
        * C strings
        */
       attinmeta = TupleDescGetAttInMetadata(tupdesc);
       funcctx->attinmeta = attinmeta;

       MemoryContextSwitchTo(oldcontext);
    }

    /* stuff done on every call of the function */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* do when there is more left to send */
    {
       char       **values;
       HeapTuple    tuple;
       Datum        result;

       /*
       * Prepare a values array for building the returned tuple.
        * This should be an array of C strings which will
        * be processed later by the type input functions.
        */
       values = (char **) palloc(3 * sizeof(char *));
       values[0] = (char *) palloc(16 * sizeof(char));
       values[1] = (char *) palloc(16 * sizeof(char));
       values[2] = (char *) palloc(16 * sizeof(char));

       snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
       snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
       snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

       /* build a tuple */
       tuple = BuildTupleFromCStrings(attinmeta, values);

       /* make the tuple into a datum */
       result = HeapTupleGetDatum(tuple);

       /* clean up (this is not really necessary) */
       pfree(values[0]);
       pfree(values[1]);
       pfree(values[2]);
       pfree(values);
```

```
        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

Voici une façon de déclarer cette fonction en SQL :

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Une façon différente de le faire est d'utiliser des paramètres OUT :

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Notez que dans cette méthode le type en sortie de la fonction est du type record anonyme.

Le répertoire `contrib/tablefunc` situé dans les fichiers source de la distribution contient d'autres exemples de fonctions renvoyant des ensembles.

35.9.11. Arguments polymorphes et types renvoyés

Les fonctions en langage C peuvent être déclarées pour accepter et renvoyer les types « polymorphes » anyelement, anyarray, anynonarray et anyenum. Voir la Section 35.2.5, « Types et fonctions polymorphes » pour une explication plus détaillée des fonctions polymorphes. Si les types des arguments ou du renvoi de la fonction sont définis comme polymorphes, l'auteur de la fonction ne peut pas savoir à l'avance quel type de données sera appelé ou bien quel type doit être renvoyé. Il y a deux routines offertes par `fmgr.h` qui permettent à une fonction en version-1 de découvrir les types de données effectifs de ses arguments et le type qu'elle doit renvoyer. Ces routines s'appellent `get_fn_expr_rettype(FmgrInfo *flinfo)` et `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. Elles renvoient l'OID du type du résultat ou de l'argument ou InvalidOID si l'information n'est pas disponible. L'accès à la structure `flinfo` se fait normalement avec `fcinfo->flinfo`. Le paramètre `argnum` est basé à partir de zéro. `get_call_result_type` peut aussi être utilisé comme alternative à `get_fn_expr_rettype`.

Par exemple, supposons que nous voulions écrire une fonction qui accepte un argument de n'importe quel type et qui renvoie un tableau uni-dimensionnel de ce type :

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element, being careful in case it's NULL */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
```

```
        element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval,
&typalign);

    /* now build the array */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

La commande suivante déclare la fonction `make_array` en SQL :

```
CREATE FUNCTION make_array(anyelement)
    RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE 'C' IMMUTABLE;
```

Notez l'utilisation de STRICT ; ceci est primordial car le code ne se préoccupe pas de tester une entrée NULL.

Il existe une variante du polymorphisme qui est seulement disponible pour les fonctions en langage C : elles peuvent être déclarées prendre des paramètres de type `"any"`. (Notez que ce nom de type doit être placé entre des guillemets doubles car il s'agit d'un mot SQL réservé.) Ceci fonctionne comme anyelement sauf qu'il ne contraint pas les différents arguments `"any"` à être du même type, pas plus qu'ils n'aident à déterminer le type de résultat de la fonction. Une fonction en langage C peut aussi déclarer son paramètre final ainsi : `VARIADIC "any"`. Cela correspondra à un ou plusieurs arguments réels de tout type (pas nécessairement le même type). Ces arguments ne seront *pas* placés dans un tableau comme c'est le cas pour les fonctions variadic normales ; ils seront passés séparément à la fonction. La macro `PG_NARGS()` et les méthodes décrites ci-dessus doivent être utilisées pour déterminer le nombre d'arguments réels et leur type lors de l'utilisation de cette fonctionnalité.

35.9.12. Mémoire partagée et LWLocks

Les modules peuvent réserver des LWLocks et allouer de la mémoire partagée au lancement du serveur. La bibliothèque partagée du module doit être préchargée en l'ajoutant shared_preload_libraries. La mémoire partagée est réservée en appelant :

```
void RequestAddinShmemSpace(int size)
```

à partir de votre fonction `_PG_init`.

Les LWLocks sont réservés en appelant :

```
void RequestAddinLWLocks(int n)
```

à partir de `_PG_init`.

Pour éviter des cas rares possibles, chaque moteur devrait utiliser la fonction `AddinShmemInitLock` lors de la connexion et de l'initialisation de la mémoire partagée, comme indiquée ci-dessous :

```
        static mystruct *ptr = NULL;

        if (!ptr)
        {
                bool    found;

                LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
                ptr = ShmemInitStruct("my struct name", size, &found);
                if (!found)
                {
                        initialize contents of shmem area;
                        acquire any requested LWLocks using:
                        ptr->mylockid = LWLockAssign();
```

```
                }
                LWLockRelease(AddinShmemInitLock);
        }
```

35.10. Agrégats utilisateur

Dans PostgreSQL™, les fonctions d'agrégat sont exprimées comme des *valeurs d'état* et des *fonctions de transition d'état*. C'est-à-dire qu'un agrégat opère en utilisant une valeur d'état qui est mis à jour à chaque ligne traitée. Pour définir une nouvelle fonction d'agrégat, on choisit un type de donnée pour la valeur d'état, une valeur initiale pour l'état et une fonction de transition d'état. La fonction de transition d'état est une fonction ordinaire qui peut être utilisée hors agrégat. Une *fonction finale* peut également être spécifiée pour le cas où le résultat désiré comme agrégat est différent des données conservées comme valeur d'état courant.

Ainsi, en plus des types de données d'argument et de résultat vus par l'utilisateur, il existe un type de données pour la valeur d'état interne qui peut être différent des deux autres.

Un agrégat qui n'utilise pas de fonction finale est un agrégat qui utilise pour chaque ligne une fonction dépendante des valeurs de colonnes. `sum` en est un exemple. `sum` débute à zéro et ajoute la valeur de la ligne courante à son total en cours. Par exemple, pour obtenir un agrégat `sum` qui opère sur un type de données nombres complexes, il suffira décrire la fonction d'addition pour ce type de donnée. La définition de l'agrégat sera :

```
CREATE AGGREGATE somme (complex)
(
    sfunc = ajout_complexe,
    stype = complexe,
    initcond = '(0,0)'
);

SELECT somme(a) FROM test_complexe;

   somme
-----------
 (34,53.9)
```

(Notez que nous nous reposons sur une surcharge de fonction : il existe plus d'un agrégat nommé `sum` mais PostgreSQL™ trouve le type de somme s'appliquant à une colonne de type complex.)

La définition précédente de `sum` retournera zéro (la condition d'état initial) s'il n'y a que des valeurs d'entrée NULL. Dans ce cas, on peut souhaiter qu' elle retourne NULL -- le standard SQL prévoit que la fonction `sum` se comporte ainsi. Cela peut être obtenu par l'omission de l'instruction `initcond`, de sorte que la condition d'état initial soit NULL. Dans ce cas, `sfunc` vérifie l'entrée d'une condition d'état NULL mais, pour `sum` et quelques autres agrégats simples comme `max` et `min`, il suffit d'insérer la première valeur d'entrée non NULL dans la variable d'état et d'appliquer la fonction de transition d'état à partir de la seconde valeur non NULL. PostgreSQL™ fait cela automatiquement si la condition initiale est NULL et si la fonction de transition est marquée « strict » (elle n'est pas appelée pour les entrées NULL).

Par défaut également, pour les fonctions de transition « strict », la valeur d'état précédente reste inchangée pour une entrée NULL. Les valeurs NULL sont ainsi ignorées. Pour obtenir un autre comportement, il suffit de ne pas déclarer la fonction de transition « strict ». À la place, codez-la de façon à ce qu'elle vérifie et traite les entrées NULL.

`avg` (average = moyenne) est un exemple plus complexe d'agrégat. Il demande deux états courants : la somme des entrées et le nombre d'entrées. Le résultat final est obtenu en divisant ces quantités. La moyenne est typiquement implantée en utilisant comme valeur d'état un tableau. Par exemple, l'implémentation intégrée de `avg(float8)` ressemble à :

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

(`float8_accum` nécessite un tableau à trois éléments, et non pas seulement deux, car il accumule la somme des carrés, ainsi que la somme et le nombre des entrées. Cela permet son utilisation pour d'autres agrégats que `avg`.)

Les fonctions d'agrégat peuvent utiliser des fonctions d'état transitionnelles ou des fonctions finales polymorphes. De cette façon, les mêmes fonctions peuvent être utilisées pour de multiples agrégats. Voir la Section 35.2.5, « Types et fonctions polymorphes » pour une explication des fonctions polymorphes. La fonction d'agrégat elle-même peut être spécifiée avec un type de base et des types d'état polymorphes, ce qui permet ainsi à une unique définition de fonction de servir pour de multiples types de données en

entrée. Voici un exemple d'agrégat polymorphe :

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

Dans ce cas, le type d'état effectif pour tout appel d'agrégat est le type tableau avec comme éléments le type effectif d'entrée. Le comportement de l'agrégat est de concaténer toutes les entrées dans un tableau de ce type. (Note : l'agrégat `array_agg` fournit une fonctionnalité similaire, avec de meilleures performances que ne pourrait avoir cette définition.)

Voici le résultat pour deux types de données différents en arguments :

```
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute WHERE attnum > 0
AND attrelid = 'pg_tablespace'::regclass GROUP BY attrelid;
   attrelid    |              array_accum
---------------+--------------------------------------
 pg_tablespace | {spcname,spcowner,spclocation,spcacl}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,text,aclitem[]}
(1 row)
```

Une fonction écrite en C peut détecter si elle est appelée en tant que fonction de transition ou en tant que fonction finale d'un agrégat en appelant `AggCheckCallContext`, par exemple :

```
        if (AggCheckCallContext(fcinfo, NULL))
```

Une raison de surveiller ceci est que, si le retour de cette fonction vaut true pour une fonction de transition, la première valeur doit être une valeur de transition temporaire et peut du coup être modifiée en toute sûreté sans avoir à allouer une nouvelle copie. Voir `int8inc()` pour un exemple. (C'est le *seul* cas où une fonction peut modifier en toute sécurité un argument passé en référence. En particulier, les fonctions finales d'agrégat ne doivent pas modifier leur arguments dans tous les cas car, dans certains cas, elles seront ré-exécutées sur la même valeur de transition finale.)

Pour de plus amples détails, on se réfèrera à la commande CREATE AGGREGATE(7).

35.11. Types utilisateur

Comme cela est décrit dans la Section 35.2, « Le système des types de PostgreSQL™ », PostgreSQL™ peut être étendu pour supporter de nouveaux types de données. Cette section décrit la définition de nouveaux types basiques. Ces types de données sont définis en-dessous du SQL. Créer un nouveau type requiert d'implanter des fonctions dans un langage de bas niveau, généralement le C.

Les exemples de cette section sont disponibles dans `complex.sql` et `complex.c` du répertoire `src/tutorial` de la distribution. Voir le fichier `README` de ce répertoire pour les instructions d'exécution des exemples.

Un type utilisateur doit toujours posséder des fonctions d'entrée et de sortie. Ces fonctions déterminent la présentation du type en chaînes de caractères (pour la saisie par l'utilisateur et le renvoi à l'utilisateur) et son organisation en mémoire. La fonction d'entrée prend comme argument une chaîne de caractères terminée par NULL et retourne la représentation interne (en mémoire) du type. La fonction de sortie prend en argument la représentation interne du type et retourne une chaîne de caractères terminée par NULL.

Il est possible de faire plus que stocker un type, mais il faut pour cela implanter des fonctions supplémentaires gérant les opérations souhaitées.

Soit le cas d'un type complex représentant les nombres complexes. Une façon naturelle de représenter un nombre complexe en mémoire passe par la structure C suivante :

```
typedef struct Complex {
    double      x;
    double      y;
```

```
} Complex;
```

Ce type ne pouvant tenir sur une simple valeur Datum, il sera passé par référence.

La représentation externe du type se fera sous la forme de la chaîne `(x,y)`.

En général, les fonctions d'entrée et de sortie ne sont pas compliquées à écrire, particulièrement la fonction de sortie. Mais lors de la définition de la représentation externe du type par une chaîne de caractères, il faudra peut-être écrire un analyseur complet et robuste, comme fonction d'entrée, pour cette représentation. Par exemple :

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

La fonction de sortie peut s'écrire simplement :

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

Il est particulièrement important de veiller à ce que les fonctions d'entrée et de sortie soient bien inversées l'une par rapport à l'autre. Dans le cas contraire, de grosses difficultés pourraient apparaître lors de la sauvegarde de la base dans un fichier en vue d'une future relecture de ce fichier. Ceci est un problème particulièrement fréquent lorsque des nombres à virgule flottante entrent en jeu.

De manière optionnelle, un type utilisateur peut fournir des routines d'entrée et de sortie binaires. Les entrées/sorties binaires sont normalement plus rapides mais moins portables que les entrées/sorties textuelles. Comme avec les entrées/sorties textuelles, c'est l'utilisateur qui définit précisément la représentation binaire externe. La plupart des types de données intégrés tentent de fournir une représentation binaire indépendante de la machine. Dans le cas du type complex, des convertisseurs d'entrées/sorties binaires pour le type float8 sont utilisés :

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}
```

```
PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Lorsque les fonctions d'entrée/sortie sont écrites et compilées en une bibliothèque partagée, le type complex peut être défini en SQL. Tout d'abord, il est déclaré comme un type shell :

```
CREATE TYPE complex;
```

Ceci sert de paramètre qui permet de mettre en référence le type pendant la définition de ses fonctions E/S. Les fonctions E/S peuvent alors être définies :

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

La définition du type de données peut ensuite être fournie complètement :

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

Quand un nouveau type de base est défini, PostgreSQL™ fournit automatiquement le support pour des tableaux de ce type. Le type tableau a habituellement le nom du type de base préfixé par un caractère souligné (_).

Lorsque le type de données existe, il est possible de déclarer les fonctions supplémentaires de définition des opérations utiles pour ce type. Les opérateurs peuvent alors être définis par dessus ces fonctions et, si nécessaire, des classes d'opérateurs peuvent être créées pour le support de l'indexage du type de données. Ces couches supplémentaires sont discutées dans les sections suivantes.

Si les valeurs du type de données varient en taille (sous la forme interne), le type de données doit être marqué comme TOAST-able (voir Section 54.2, « TOAST »). Vous devez le faire même si les données sont trop petites pour être compressées ou stockées en externe car TOAST peut aussi gagner de la place sur des petites données en réduisant la surcharge de l'en-tête.

Pour cela, la représentation interne doit suivre la disposition standard pour les données de longueur variable : les quatre premiers octets doivent être un champ char[4] dont l'accès n'est jamais direct (appelé *vl_len_*). Vous devez utiliser `SET_VARSIZE()` pour stocker la taille du datum dans ce champ et `VARSIZE()` pour le récupérer. Les fonctions C opérant sur ce type de données doivent toujours faire attention à déballer toutes valeurs TOAST qu'elles récupèrent en utilisant `PG_DETOAST_DATUM` (ce détail est habituellement caché en définissant des macros `GETARG_DATATYPE_P` spécifiques au type). Ensuite, lors de l'exécution de

la commande **CREATE TYPE**, précisez la longueur interne comme `variable` et sélectionnez l'option de stockage approprié.

Si l'alignement n'est pas important (soit seulement pour une fonction spécifique soit parce que le type de données spécifie un alignement par octet), alors il est possible d'éviter `PG_DETOAST_DATUM`. Vous pouvez utiliser `PG_DETOAST_DATUM_PACKED` à la place (habituellement caché par une macro `GETARG_DATATYPE_PP`) et utiliser les macros `VARSIZE_ANY_EXHDR` et `VARDATA_ANY` pour accéder à un datum potentiellement packagé. Encore une fois, les données renvoyées par ces macros ne sont pas alignées même si la définition du type de données indique un alignement. Si l'alignement est important pour vous, vous devez passer par l'interface habituelle, `PG_DETOAST_DATUM`.

> **Note**
>
> Un ancien code déclare fréquemment *`vl_len_`* comme un champ de type int32 au lieu de char[4]. C'est correct tant que la définition de la structure a d'autres champs qui ont au moins un alignement int32. Mais il est dangereux d'utiliser une telle définition de structure en travaillant avec un datum potentiellement mal aligné ; le compilateur peut le prendre comme une indication pour supposer que le datum est en fait aligné, ceci amenant des « core dump » sur des architectures qui sont strictes sur l'alignement.

Pour plus de détails, voir la description de la commande CREATE TYPE(7).

35.12. Opérateurs définis par l'utilisateur

chaque opérateur est un « sucre syntaxique » pour l'appel d'une fonction sous-jacente qui effectue le véritable travail ; aussi devez-vous en premier lieu créer cette fonction avant de pouvoir créer l'opérateur. Toutefois, un opérateur n'est pas *simplement* un « sucre syntaxique » car il apporte des informations supplémentaires qui aident le planificateur de requête à optimiser les requêtes utilisées par l'opérateur. La prochaine section est consacrée à l'explication de ces informations additionnelles.

postgresql™ accepte les opérateurs unaire gauche, unaire droit et binaire. Les opérateurs peuvent être surchargés ; c'est-à-dire que le même nom d'opérateur peut être utilisé pour différents opérateurs à condition qu'ils aient des nombres et des types différents d'opérandes. Quand une requête est exécutée, le système détermine l'opérateur à appeler en fonction du nombre et des types d'opérandes fournis.

Voici un exemple de création d'opérateur pour l'addition de deux nombres complexes. Nous supposons avoir déjà créé la définition du type complex (voir la Section 35.11, « Types utilisateur »). premièrement, nous avons besoin d'une fonction qui fasse le travail, ensuite nous pouvons définir l'opérateur :

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C;

CREATE OPERATOR + ( leftarg = complex, rightarg = complex, procedure =
complex_add, commutator = + );
```

Maintenant nous pouvons exécuter la requête comme ceci :

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

Nous avons montré comment créer un opérateur binaire. Pour créer des opérateurs unaires, il suffit d'omettre un des `leftarg` (pour un opérateur unaire gauche) ou `rightarg` (pour un opérateur unaire droit). La clause `procedure` et les clauses argument sont les seuls éléments requis dans la commande **create operator**. la clause `commutator` montrée dans l'exemple est une indication optionnelle pour l'optimiseur de requête. Des détails supplémentaires sur la clause `commutator` et d'autres compléments d'optimisation sont donnés dans la prochaine section.

35.13. Informations sur l'optimisation d'un opérateur

Une définition d'opérateur PostgreSQL™ peut inclure plusieurs clauses optionnelles qui donnent au système des informations utiles sur le comportement de l'opérateur. Ces clauses devraient être fournies chaque fois que c'est utile car elles peuvent considérablement accélérer l'exécution des requêtes utilisant cet opérateur. Mais si vous le faites, vous devez être sûr de leur justesse ! L'usage incorrect d'une clause d'optimisation peut être la cause de requêtes lentes, des sorties subtilement fausses ou d'autres effets pervers. Vous pouvez toujours abandonner une clause d'optimisation si vous n'êtes pas sûr d'elle ; la seule conséquence est un possible ralentissement des requêtes.

Des clauses additionnelles d'optimisation pourront être ajoutées dans les futures versions de postgresql™. celles décrites ici sont toutes celles que cette version comprend.

35.13.1. COMMUTATOR

Si elle est fournie, la clause `commutator` désigne un opérateur qui est le commutateur de l'opérateur en cours de définition. Nous disons qu'un opérateur A est le commutateur de l'opérateur B si (x A y) est égal à (y B x) pour toute valeur possible de x, y. Notez que B est aussi le commutateur de A. Par exemple, les opérateurs < et > pour un type particulier de données sont habituellement des commutateurs l'un pour l'autre, et l'opérateur + est habituellement commutatif avec lui-même. Mais l'opérateur - n'est habituellement commutatif avec rien.

Le type de l'opérande gauche d'un opérateur commuté est le même que l'opérande droit de son commutateur, et vice versa. Aussi postgresql™ n'a besoin que du nom de l'opérateur commutateur pour consulter le commutateur, et c'est tout ce qui doit être fourni à la clause `commutator` .

Vous avez juste à définir un opérateur auto-commutateur. Mais les choses sont un peu plus compliquées quand vous définissez une paire de commutateurs : comment peut-on définir la référence du premier au second alors que ce dernier n'est pas encore défini ? Il y a deux solutions à ce problème :

- Une façon d'opérer est d'omettre la clause `commutator` dans le premier opérateur que vous définissez et ensuite d'en insérer une dans la définition du second opérateur. Puisque postgresql™ sait que les opérateurs commutatifs vont par paire, quand il voit la seconde définition, il retourne instantanément remplir la clause `commutator` manquante dans la première définition.
- L'autre façon, plus directe, est de simplement inclure les clauses `commutator` dans les deux définitions. quand postgresql™ traite la première définition et réalise que la clause `commutator` se réfère à un opérateur inexistant, le système va créer une entrée provisoire pour cet opérateur dans le catalogue système. Cette entrée sera pourvue seulement de données valides pour le nom de l'opérateur, les types d'opérande droit et gauche et le type du résultat, puisque c'est tout ce que postgresql™ peut déduire à ce point. la première entrée du catalogue pour l'opérateur sera liée à cette entrée provisoire. Plus tard, quand vous définirez le second opérateur, le système mettra à jour l'entrée provisoire avec les informations additionnelles fournies par la seconde définition. Si vous essayez d'utiliser l'opérateur provisoire avant qu'il ne soit complété, vous aurez juste un message d'erreur.

35.13.2. NEGATOR

La clause `negator` dénomme un opérateur qui est l'opérateur de négation de l'opérateur en cours de définition. Nous disons qu'un opérateur A est l'opérateur de négation de l'opérateur B si tous les deux renvoient des résultats booléens et si (x A y) est égal à NOT (x B y) pour toutes les entrées possible x, y. Notez que B est aussi l'opérateur de négation de A. Par exemple, < et >= forment une paire d'opérateurs de négation pour la plupart des types de données. Un opérateur ne peut jamais être validé comme son propre opérateur de négation .

Au contraire des commutateurs, une paire d'opérateurs unaires peut être validée comme une paire d'opérateurs de négation réciproques ; ce qui signifie que (A x) est égal à NOT (B x) pour tout x ou l'équivalent pour les opérateurs unaires à droite.

L'opérateur de négation d'un opérateur doit avoir les mêmes types d'opérandes gauche et/ou droit que l'opérateur à définir comme avec `commutator`. seul le nom de l'opérateur doit être donné dans la clause `negator`.

Définir un opérateur de négation est très utile pour l'optimiseur de requêtes car il permet de simplifier des expressions telles que `not (x = y)` en `x <> y`. ceci arrive souvent parce que les opérations `not` peuvent être insérées à la suite d'autres réarrangements.

Des paires d'opérateurs de négation peuvent être définies en utilisant la même méthode que pour les commutateurs.

35.13.3. RESTRICT

La clause `restrict`, si elle est invoquée, nomme une fonction d'estimation de sélectivité de restriction pour cet opérateur (notez que c'est un nom de fonction, et non pas un nom d'opérateur). Les clauses `restrict` n'ont de sens que pour les opérateurs binaires qui renvoient un type boolean. un estimateur de sélectivité de restriction repose sur l'idée de prévoir quelle fraction des lignes dans une table satisfera une condition de clause `where` de la forme :

```
colonne OP constante
```

pour l'opérateur courant et une valeur constante particulière. Ceci aide l'optimiseur en lui donnant une idée du nombre de lignes qui sera éliminé par les clauses `where` qui ont cette forme (vous pouvez vous demander, qu'arrivera-t-il si la constante est à gauche ? hé bien, c'est une des choses à laquelle sert le `commutator`...).

L'écriture de nouvelles fonctions d'estimation de restriction de sélectivité est éloignée des objectifs de ce chapitre mais, heureusement, vous pouvez habituellement utiliser un des estimateurs standards du système pour beaucoup de vos propres opérateurs. Voi-

ci les estimateurs standards de restriction :

`eqsel` pour =
`neqsel` pour <>
`scalarltsel` pour < ou <=
`scalargtsel` pour > ou >=
Ces catégories peuvent sembler un peu curieuses mais cela prend un sens si vous y réfléchissez. = acceptera typiquement une petite fraction des lignes d'une table ; <> rejettera typiquement seulement une petite fraction des lignes de la table. < acceptera une fraction des lignes en fonction de la situation de la constante donnée dans la gamme de valeurs de la colonne pour cette table (ce qui est justement l'information collectée par la commande **analyze** et rendue disponible pour l'estimateur de sélectivité). <= acceptera une fraction légèrement plus grande que < pour la même constante de comparaison mais elles sont assez proches pour ne pas valoir la peine d'être distinguées puisque nous ne risquons pas de toute façon de faire mieux qu'une grossière estimation. La même remarque s'applique à > et >=.

Vous pouvez fréquemment vous en sortir à bon compte en utilisant soit `eqsel` ou `neqsel` pour des opérateurs qui ont une très grande ou une très faible sélectivité, même s'ils ne sont pas réellement égalité ou inégalité. Par exemple, les opérateurs géométriques d'égalité approchée utilisent `eqsel` en supposant habituellement qu'ils ne correspondent qu'à une petite fraction des entrées dans une table.

Vous pouvez utiliser `scalarltsel` et `scalargtsel` pour des comparaisons de types de données qui possèdent un moyen de conversion en scalaires numériques pour les comparaisons de rang. Si possible, ajoutez le type de données à ceux acceptés par la fonction `convert_to_scalar()` dans `src/backend/utils/adt/selfuncs.c` (finalement, cette fonction devrait être remplacée par des fonctions pour chaque type de données identifié grâce à une colonne du catalogue système `pg_type` ; mais cela n'a pas encore été fait). si vous ne faites pas ceci, les choses fonctionneront mais les estimations de l'optimiseur ne seront pas aussi bonnes qu'elles pourraient l'être.

D'autres fonctions d'estimation de sélectivité conçues pour les opérateurs géométriques sont placées dans `src/backend/utils/adt/geo_selfuncs.c` : `areasel`, `positionsel` et `contsel`. lors de cette rédaction, ce sont seulement des fragments mais vous pouvez vouloir les utiliser (ou mieux les améliorer).

35.13.4. JOIN

La clause `join`, si elle est invoquée, nomme une fonction d'estimation de sélectivité de jointure pour l'opérateur (notez que c'est un nom de fonction, et non pas un nom d'opérateur). Les clauses `join` n'ont de sens que pour les opérateurs binaires qui renvoient un type boolean. un estimateur de sélectivité de jointure repose sur l'idée de prévoir quelle fraction des lignes dans une paire de tables satisfera une condition de clause `where` de la forme :

```
table1.colonne1 OP table2.colonne2
```

pour l'opérateur courant. Comme pour la clause `restrict`, ceci aide considérablement l'optimiseur en lui indiquant parmi plusieurs séquences de jointure possibles laquelle prendra vraisemblablement le moins de travail.

Comme précédemment, ce chapitre n'essaiera pas d'expliquer comment écrire une fonction d'estimation de sélectivité de jointure mais suggérera simplement d'utiliser un des estimateurs standard s'il est applicable :

`eqjoinsel` pour =
`neqjoinsel` pour <>
`scalarltjoinsel` pour < ou <=
`scalargtjoinsel` pour > ou >=
`areajoinsel` pour des comparaisons basées sur une aire 2d
`positionjoinsel` pour des comparaisons basées sur une position 2d
`contjoinsel` pour des comparaisons basées sur un appartenance 2d

35.13.5. HASHES

La clause `hashes` indique au système qu'il est permis d'utiliser la méthode de jointure-découpage pour une jointure basée sur cet opérateur. `hashes` n'a de sens que pour un opérateur binaire qui renvoie un `boolean` et en pratique l'opérateur égalité doit représenter l'égalité pour certains types de données ou paire de type de données.

La jointure-découpage repose sur l'hypothèse que l'opérateur de jointure peut seulement renvoyer la valeur vrai pour des paires de valeurs droite et gauche qui correspondent au même code de découpage. Si deux valeurs sont placées dans deux différents paquets (« buckets »), la jointure ne pourra jamais les comparer avec la supposition implicite que le résultat de l'opérateur de jointure doit être faux. Ainsi, il n'y a aucun sens à spécifier `hashes` pour des opérateurs qui ne représentent pas une certaine forme d'égalité. Dans la plupart des cas, il est seulement pratique de supporter le hachage pour les opérateurs qui prennent le même type de données sur chaque côté. Néanmoins, quelque fois, il est possible de concevoir des fonctions de hachage compatibles pour deux type de données, voire plus ; c'est-à-dire pour les fonctions qui généreront les mêmes codes de hachage pour des valeurs égales même si elles ont des représentations différentes. Par exemple, il est assez simple d'arranger cette propriété lors du hachage d'entiers de

largeurs différentes.

Pour être marqué `hashes`, l'opérateur de jointure doit apparaître dans une famille d'opérateurs d'index de découpage. Ceci n'est pas rendu obligatoire quand vous créez l'opérateur, puisque évidemment la classe référençant l'opérateur peut ne pas encore exister. Mais les tentatives d'utilisation de l'opérateur dans les jointure-découpage échoueront à l'exécution si une telle famille d'opérateur n'existe pas. Le système a besoin de la famille d'opérateur pour définir la fonction de découpage spécifique au type de données d'entrée de l'opérateur. Bien sûr, vous devez également créer des fonctions de découpage appropriées avant de pouvoir créer la famille d'opérateur.

On doit apporter une grande attention à la préparation des fonctions de découpage parce qu'il y a des processus dépendants de la machine qui peuvent ne pas faire les choses correctement. Par exemple, si votre type de données est une structure dans laquelle peuvent se trouver des bits de remplissage sans intérêt, vous ne pouvez pas simplement passer la structure complète à la fonction `hash_any` (à moins d'écrire vos autres opérateurs et fonctions de façon à s'assurer que les bits inutilisés sont toujours zéro, ce qui est la stratégie recommandée). Un autre exemple est fourni sur les machines qui respectent le standard de virgule-flottante ieee, le zéro négatif et le zéro positif sont des valeurs différentes (les motifs de bit sont différents) mais ils sont définis pour être égaux. Si une valeur flottante peut contenir un zéro négatif, alors une étape supplémentaire est nécessaire pour s'assurer qu'elle génère la même valeur de découpage qu'un zéro positif.

Un opérateur joignable par hachage doit avoir un commutateur (lui-même si les types de données des deux opérandes sont identiques, ou un opérateur d'égalité relatif dans le cas contraire) qui apparaît dans la même famille d'opérateur. Si ce n'est pas le cas, des erreurs du planificateur pourraient apparaître quand l'opérateur est utilisé. De plus, une bonne idée (mais pas obligatoire) est qu'une famille d'opérateur de hachage supporte les tupes de données multiples pour fournir des opérateurs d'égalité pour chaque combinaison des types de données ; cela permet une meilleure optimisation.

Note

La fonction sous-jacente à un opérateur de jointure-découpage doit être marquée immuable ou stable. Si elle est volatile, le système n'essaiera jamais d'utiliser l'opérateur pour une jointure hachage.

Note

Si un opérateur de jointure-hachage a une fonction sous-jacente marquée stricte, la fonction doit également être complète : cela signifie qu'elle doit renvoyer TRUE ou FALSE, jamais NULL, pour n'importe quelle double entrée non NULL. Si cette règle n'est pas respectée, l'optimisation de découpage des opérations `in` peut générer des résultats faux (spécifiquement, `in` devrait renvoyer false quand la réponse correcte devrait être NULL ; ou bien il devrait renvoyer une erreur indiquant qu'il ne s'attendait pas à un résultat NULL).

35.13.6. `MERGES`

La clause `merges`, si elle est présente, indique au système qu'il est permis d'utiliser la méthode de jointure-union pour une jointure basée sur cet opérateur. `merges` n'a de sens que pour un opérateur binaire qui renvoie un `boolean` et, en pratique, cet opérateur doit représenter l'égalité pour des types de données ou des paires de types de données.

La jointure-union est fondée sur le principe d'ordonner les tables gauche et droite et ensuite de les comparer en parallèle. Ainsi, les deux types de données doivent être capable d'être pleinement ordonnées, et l'opérateur de jointure doit pouvoir réussir seulement pour des paires de valeurs tombant à la « même place » dans l'ordre de tri. En pratique, cela signifie que l'opérateur de jointure doit se comporter comme l'opérateur égalité. Mais il est possible de faire une jointure-union sur deux types de données distincts tant qu'ils sont logiquement compatibles. Par exemple, l'opérateur d'égalité smallint-contre-integer est susceptible d'opérer une jointure-union. Nous avons seulement besoin d'opérateurs de tri qui organisent les deux types de données en séquences logiquement comparables.

Pour être marqué `MERGES`, l'opérateur de jointure doit apparaître en tant que membre d'égalité d'une famille opérateur d'index `btree`. Ceci n'est pas forcé quand vous créez l'opérateur puisque, bien sûr, la famille d'opérateur référente n'existe pas encore. Mais l'opérateur ne sera pas utilisé pour les jointures de fusion sauf si une famille d'opérateur correspondante est trouvée. L'option `MERGES` agit en fait comme une aide pour le planificateur lui indiquant qu'il est intéressant de chercher une famille d'opérateur correspondant.

Un opérateur joignable par fusion doit avoir un commutateur (lui-même si les types de données des deux opérateurs sont identiques, ou un opérateur d'égalité en relation dans le cas contraire) qui apparaîtdans la même famille d'opérateur. Si ce n'est pas le cas, des erreurs du planificateur pourraient apparaître quand l'opérateur est utilisé. De plus, une bonne idée (mais pas obligatoire) est qu'une famille d'opérateur de hachage supporte les tupes de données multiples pour fournir des opérateurs d'égalité pour chaque combinaison des types de données ; cela permet une meilleure optimisation.

> **Note**
>
> La fonction sous-jacente à un opérateur de jointure-union doit être marquée immuable ou stable. Si elle est volatile, le système n'essaiera jamais d'utiliser l'opérateur pour une jointure union.

35.14. Interfacer des extensions d'index

Les procédures décrites jusqu'à maintenant permettent de définir de nouveaux types, de nouvelles fonctions et de nouveaux opérateurs. Néanmoins, nous ne pouvons pas encore définir un index sur une colonne d'un nouveau type de données. Pour cela, nous devons définir une *classe d'opérateur* pour le nouveau type de données. Plus loin dans cette section, nous illustrerons ce concept avec un exemple : une nouvelle classe d'opérateur pour la méthode d'indexation B-tree qui enregistre et trie des nombres complexes dans l'ordre ascendant des valeurs absolues.

Les classes d'opérateur peuvent être groupées en *familles d'opérateur* pour afficher les relations entre classes compatibles sémantiquement. Quand un seul type de données est impliqué, une classe d'opérateur est suffisant, donc nous allons nous fixer sur ce cas en premier puis retourner aux familles d'opérateur.

35.14.1. Méthodes d'indexation et classes d'opérateurs

La table `pg_am` contient une ligne pour chaque méthode d'indexation (connue en interne comme méthode d'accès). Le support pour l'accès normal aux tables est implémenté dans PostgreSQL™ mais toutes les méthodes d'index sont décrites dans `pg_am`. Il est possible d'ajouter une nouvelle méthode d'indexation en définissant les routines d'interface nécessaires et en créant ensuite une ligne dans la table `pg_am` -- mais ceci est au-delà du sujet de ce chapitre (voir le Chapitre 51, Définition de l'interface des méthodes d'accès aux index).

Les routines pour une méthode d'indexation n'ont pas à connaître directement les types de données sur lesquels opère la méthode d'indexation. Au lieu de cela, une *classe d'opérateur* identifie l'ensemble d'opérations que la méthode d'indexation doit utiliser pour fonctionner avec un type particulier de données. Les classes d'opérateurs sont ainsi dénommées parce qu'une de leur tâche est de spécifier l'ensemble des opérateurs de la clause `WHERE` utilisables avec un index (c'est-à-dire, qui peuvent être requalifiés en balayage d'index). Une classe d'opérateur peut également spécifier des *procédures d'appui*, nécessaires pour les opérations internes de la méthode d'indexation mais sans correspondance directe avec un quelconque opérateur de clause `WHERE` pouvant être utilisé avec l'index.

Il est possible de définir plusieurs classes d'opérateurs pour le même type de données et la même méthode d'indexation. Ainsi, de multiples ensembles de sémantiques d'indexation peuvent être définis pour un seul type de données. Par exemple, un index B-tree exige qu'un tri ordonné soit défini pour chaque type de données auquel il peut s'appliquer. Il peut être utile pour un type de donnée de nombre complexe de disposer d'une classe d'opérateur B-tree qui trie les données selon la valeur absolue complexe, une autre selon la partie réelle, etc. Typiquement, une des classes d'opérateur sera considérée comme plus utile et sera marquée comme l'opérateur par défaut pour ce type de données et cette méthode d'indexation.

Le même nom de classe d'opérateur peut être utilisé pour plusieurs méthodes d'indexation différentes (par exemple, les méthodes d'index B-tree et hash ont toutes les deux des classes d'opérateur nommées `int4_ops`) mais chacune de ces classes est une entité indépendante et doit être définie séparément.

35.14.2. Stratégies des méthode d'indexation

Les opérateurs associés à une classe d'opérateur sont identifiés par des « numéros de stratégie », servant à identifier la sémantique de chaque opérateur dans le contexte de sa classe d'opérateur. Par exemple, les B-trees imposent un classement strict selon les clés, du plus petit au plus grand. Ainsi, des opérateurs comme « plus petit que » et « plus grand que » sont intéressants pour un B-tree. Comme PostgreSQL™ permet à l'utilisateur de définir des opérateurs, PostgreSQL™ ne peut pas rechercher le nom d'un opérateur (par exemple, `<` ou `>=`) et rapporter de quelle comparaison il s'agit. Au lieu de cela, la méthode d'indexation définit un ensemble de « stratégies », qui peuvent être comprises comme des opérateurs généralisés. Chaque classe d'opérateur spécifie l'opérateur effectif correspondant à chaque stratégie pour un type de donnée particulier et pour une interprétation de la sémantique d'index.

La méthode d'indexation B-tree définit cinq stratégies, qui sont exposées dans le Tableau 35.2, « Stratégies B-tree ».

Tableau 35.2. Stratégies B-tree

Opération	Numéro de stratégie
plus petit que	1
plus petit ou égal	2
égal	3

Opération	Numéro de stratégie
plus grand ou égal	4
plus grand que	5

Les index de découpage permettent seulement des comparaisons d'égalité et utilisent ainsi une seule stratégie exposée dans le Tableau 35.3, « Stratégies de découpage ».

Tableau 35.3. Stratégies de découpage

Opération	Numéro de stratégie
égal à	1

Les index GiST sont plus flexibles : ils n'ont pas du tout un ensemble fixe de stratégies. À la place, la routine de support de « cohérence » de chaque classe d'opérateur GiST interprète les numéros de stratégie comme elle l'entend. Comme exemple, plusieurs des classes d'opérateurs GiST indexe les objets géométriques à deux dimensions fournissant les stratégies « R-tree » affichées dans Tableau 35.4, « Stratégies « R-tree » pour GiST à deux dimensions ». Quatre d'entre elles sont des vrais tests à deux dimensions (surcharge, identique, contient, contenu par) ; quatre autres considèrent seulement la direction X ; et les quatre dernières fournissent les mêmes tests dans la direction Y.

Tableau 35.4. Stratégies « R-tree » pour GiST à deux dimensions

Opération	Numéro de stratégie
strictement à gauche de	1
ne s'étend pas à droite de	2
surcharge	3
ne s'étend pas à gauche de	4
strictement à droite de	5
identique	6
contient	7
contenu par	8
ne s'étend pas au dessus	9
strictement en dessous	10
strictement au dessus	11
ne s'étend pas en dessous	12

Les index GIN sont similaires aux index GiST en flexibilité : ils n'ont pas un ensemble fixe de stratégie. À la place, les routines de support de chaque classe d'opérateur interprètent les numéros de stratégie suivant la définition du classe d'opérateur. Comme exemple, les numéros des stratégies utilisés par les classes d'opérateur sur des tableaux sont affichés dans Tableau 35.5, « Stratégies des tableaux GIN ».

Tableau 35.5. Stratégies des tableaux GIN

Opération	Numéro de stratégie
surcharge	1
contient	2
est contenu par	3
identique	4

Notez que tous les opérateurs de stratégie renvoient des valeurs de type booléen. Dans la pratique, tous les opérateurs définis comme stratégies de méthode d'indexation doivent renvoyer un type boolean puisqu'ils doivent apparaître au plus haut niveau d'une clause `WHERE` pour être utilisés avec un index.

35.14.3. Routines d'appui des méthodes d'indexation

Généralement, les stratégies n'apportent pas assez d'informations au système pour indiquer comment utiliser un index. Dans la pratique, les méthodes d'indexation demandent des routines d'appui additionnelles pour fonctionner. Par exemple, les méthodes d'index B-tree doivent être capables de comparer deux clés et de déterminer laquelle est supérieure, égale ou inférieure à l'autre. De la même façon, la méthode d'indexation hash doit être capable de calculer les codes de hachage pour les valeurs de clés. Ces opérations ne correspondent pas à des opérateurs utilisés dans les commandes SQL ; ce sont des routines administratives utilisées en interne par des méthodes d'index.

Comme pour les stratégies, la classe d'opérateur énumère les fonctions spécifiques et le rôle qu'elles doivent jouer pour un type de donnée donné et une interprétation sémantique donnée. La méthode d'indexation définit l'ensemble des fonctions dont elle a besoin et la classe d'opérateur identifie les fonctions exactes à utiliser en les assignant aux « numéros de fonction d'appui » spécifiés par la méthode d'indexage.

Les B-trees demandent une seule fonction d'appui, exposée dans le Tableau 35.6, « Fonctions d'appui de B-tree ».

Tableau 35.6. Fonctions d'appui de B-tree

Fonction	Numéro d'appui
Comparer deux clés et renvoyer un entier inférieur à zéro, zéro ou supérieure à zéro indiquant si la première clé est inférieure, égale ou supérieure à la deuxième.	1

De façon identique, les index de découpage requièrent une fonction d'appui exposée dans le Tableau 35.7, « Fonctions d'appui pour découpage ».

Tableau 35.7. Fonctions d'appui pour découpage

Fonction	Numéro d'appui
Calculer la valeur de découpage pour une clé	1

Les index GiST requièrent sept fonctions d'appui exposées dans le Tableau 35.8, « Fonctions d'appui pour GiST ».

Tableau 35.8. Fonctions d'appui pour GiST

Fonction	Numéro d'appui
consistency - détermine si la clé satisfait le qualifiant de la requête	1
union - calcule l'union d'un ensemble de clés	2
compress - calcule une représentation compressée d'une clé ou d'une valeur à indexer	3
decompress - calcule une représentation décompressée d'une clé compressée	4
penality - calcule la pénalité pour l'insertion d'une nouvelle clé dans un sous-arbre avec la clé du sous-arbre indiqué	5
picksplit - détermine les entrées d'une page qui sont à déplacer vers la nouvelle page et calcule les clés d'union pour les pages résultantes	6
equal - compare deux clés et renvoie true si elles sont identiques	7

Les index GIN requièrent quatre fonctions d'appui exposées dans le Tableau 35.9, « Fonctions d'appui GIN ».

Tableau 35.9. Fonctions d'appui GIN

Fonction	Description
`compare`	Compare deux clés et renvoie un entier plus petit que zéro, zéro ou plus grand que zéro, indiquant si la première clé est plus petit, égal à ou plus grand que la seconde.

Fonction	Description
`extractValue`	Extrait les clés à partir d'une condition de requête
`extractQuery`	Extrait les clés à partir d'une condition de requête
`consistent`	Détermine la valeur correspondant à la condition de requête
`comparePartial`	(méthode optionnelle) compare la clé partielle de la requête et la clé de l'index, et renvoie un entier négatif, nul ou positif, indiquant si GIN doit ignorer cette entrée d'index, traiter l'entrée comme une correspondance ou arrer le parcours d'index

Contrairement aux opérateurs de stratégie, les fonctions d'appui renvoient le type de donnée, quelqu'il soit, que la méthode d'indexation particulière attend, par exemple, dans le cas de la fonction de comparaison des B-trees, un entier signé. Le nombre et le type des arguments pour chaque fonction de support peuvent dépendre de la méthode d'indexage. Pour les B-tree et les hash, les fonctions de support prennent les même types de données en entrée comme le font les opérateurs incluant dans la classe d'opérateur, bien que ce ne soit pas le cas pour la plupart des fonctions de support GIN et GiST.

35.14.4. Exemple

Maintenant que nous avons vu les idées, voici l'exemple promis de création d'une nouvelle classe d'opérateur. Cette classe d'opérateur encapsule les opérateurs qui trient les nombres complexes selon l'ordre de la valeur absolue, aussi avons-nous choisi le nom de `complex_abs_ops`. En premier lieu, nous avons besoin d'un ensemble d'opérateurs. La procédure pour définir des opérateurs a été discutée dans la Section 35.12, « Opérateurs définis par l'utilisateur ». Pour une classe d'opérateur sur les B-trees, nous avons besoin des opérateurs :

- valeur absolue less-than (stratégie 1) ;
- valeur absolue less-than-or-equal (stratégie 2) ;
- valeur absolue equal (stratégie 3) ;
- valeur absolue greater-than-or-equal (stratégie 4) ;
- valeur absolue greater-than (stratégie 5) ;

Le plus simple moyen de définie un ensemble d'opérateurs de comparaison est d'écrire en premier la fonction de comparaison B-tree, puis d'écrire les autres fonctions en tant que wrapper de la fonction de support. Ceci réduit les risques de résultats incohérents pour les cas spécifiques. En suivant cette approche, nous devons tout d'abord écrire :

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Maintenant, la fonction plus-petit-que ressemble à ceci :

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

Les quatre autres fonctions diffèrent seulement sur la façon dont ils comparent le résultat de la fonction interne au zéro.

Maintenant, déclarons en SQL les fonctions et les opérateurs basés sur ces fonctions :

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'nom_fichier', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

Il est important de spécifier les fonctions de sélectivité de restriction et de jointure, sinon l'optimiseur sera incapable de faire un usage effectif de l'index. Notez que les cas 'less-than', 'equal' et 'greater-than' doivent utiliser des fonctions différentes de sélectivité.

Voici d'autres choses importantes à noter :

- Il ne peut y avoir qu'un seul opérateur nommé, disons, = et acceptant un type complex pour ses deux opérandes. Dans le cas présent, nous n'avons aucun autre opérateur = pour complex mais, si nous construisons un type de donnée fonctionnel, nous aurions certainement désiré que = soit l'opération ordinaire d'égalité pour les nombres complexes (et non pour l'égalité de leurs valeurs absolues). Dans ce cas, nous aurions eu besoin d'utiliser un autre nom d'opérateur pour notre fonction `complex_abs_eq`.
- Bien que PostgreSQL™ puisse se débrouiller avec des fonctions ayant le même nom SQL, tant qu'elles ont en argument des types de données différents, en C il ne peut exister qu'une fonction globale pour un nom donné. Aussi ne devons-nous pas donner un nom simple comme `abs_eq`. Habituellement, inclure le nom du type de données dans le nom de la fonction C est une bonne habitude pour ne pas provoquer de conflit avec des fonctions pour d'autres types de donnée.
- Nous aurions pu faire de `abs_eq` le nom SQL de la fonction, en laissant à PostgreSQL™ le soin de la distinguer de toute autre fonction SQL de même nom par les types de données en argument. Pour la simplicité de l'exemple, nous donnerons à la fonction le même nom au niveau de C et au niveau de SQL.

La prochaine étape est l'enregistrement de la routine d'appui nécessaire pour les B-trees. Le code exemple C qui implémente ceci est dans le même fichier qui contient les fonctions d'opérateur. Voici comment déclarer la fonction :

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Maintenant que nous avons les opérateurs requis et la routine d'appui, nous pouvons enfin créer la classe d'opérateur.

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

Et c'est fait ! Il devrait être possible maintenant de créer et d'utiliser les index B-tree sur les colonnes complex.

Nous aurions pu écrire les entrées de l'opérateur de façon plus explicite comme dans :

```
        OPERATOR        1       < (complex, complex) ,
```

mais il n'y a pas besoin de faire ainsi quand les opérateurs prennent le même type de donnée que celui pour lequel la classe d'opérateur a été définie.

Les exemples ci-dessus supposent que vous voulez que cette nouvelle classe d'opérateur soit la classe d'opérateur B-tree par défaut pour le type de donnée complex. Si vous ne voulez pas, supprimez simplement le mot `DEFAULT`.

35.14.5. Classes et familles d'opérateur

Jusqu'à maintenant, nous avons supposé implicitement qu'une classe d'opérateur s'occupe d'un seul type de données. Bien qu'il ne peut y avoir qu'un seul type de données dans une colonne d'index particulière, il est souvent utile d'indexer les opérations qui comparent une colonne indexée à une valeur d'un type de données différent. De plus, s'il est intéressant d'utiliser un opérateur intertype en connexion avec une classe d'opérateur, souvent cet autre type de donnée a sa propre classe d'opérateur. Rendre explicite

les connexions entre classes en relation est d'une grande aide pour que le planificateur optimise les requêtes SQL (tout particulièrement pour les classes d'opérateur B-tree car le planificateur sait bien comme les utiliser).

Pour gérer ces besoins, PostgreSQL™ utilise le concept d'une *famille d'opérateur* . Une famille d'opérateur contient une ou plusieurs classes d'opérateur et peut aussi contenir des opérateurs indexables et les fonctions de support correspondantes appartenant à la famille entière mais pas à une classe particulière de la famille. Nous disons que ces opérateurs et fonctions sont « lâches » à l'intérieur de la famille, en opposition à être lié à une classe spécifique. Typiquement, chaque classe d'opérateur contient des opérateurs de types de données simples alors que les opérateurs inter-type sont lâches dans la famille.

Tous les opérateurs et fonctions d'une famille d'opérateurs doivent avoir une sémantique compatible où les pré-requis de la compatibilité sont dictés par la méthode indexage. Du coup, vous pouvez vous demander la raison pour s'embarrasser de distinguer les sous-ensembles de la famille en tant que classes d'opérateur. En fait, dans beaucoup de cas, les divisions en classe sont inutiles et la famille est le seul groupe intéressant. La raison de la définition de classes d'opérateurs est qu'ils spécifient à quel point la famille est nécessaire pour supporter un index particulier. S'il existe un index utilisant une classe d'opérateur, alors cette classe d'opérateur ne peut pas être supprimée sans supprimer l'index -- mais les autres parties de la famille d'opérateurs, donc les autres classes et les opérateurs lâches, peuvent être supprimées. Du coup, une classe d'opérateur doit être indiquée pour contenir l'ensemble minimum d'opérateurs et de fonctions qui sont raisonnablement nécessaire pour travailler avec un index sur un type de données spécifique, et ensuite les opérateurs en relation mais peuvent être ajoutés en tant que membres lâches de la famille d'opérateur.

Comme exemple, PostgreSQL™ a une famille d'opérateur B-tree interne `integer_ops`, qui inclut les classes d'opérateurs `int8_ops`, `int4_ops` et `int2_ops` pour les index sur les colonnes bigint (int8), integer (int4) et smallint (int2) respectivement. La famille contient aussi des opérateurs de comparaison inter-type permettant la comparaison de deux de ces types, pour qu'un index parmi ces types puisse être parcouru en utilisant une valeur de comparaison d'un autre type. La famille peut être dupliqué par ces définitions :

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- comparaisons int8 standard
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- comparaisons int4 standard
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- comparaisons int2 standard
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- comparaisons inter-types int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  -- comparaisons inter-types int8 vs int4
  OPERATOR 1 < (int8, int4) ,
```

```
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ,

  -- comparaisons inter-types int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- comparaisons inter-types int4 vs int8
  OPERATOR 1 < (int4, int8) ,
  OPERATOR 2 <= (int4, int8) ,
  OPERATOR 3 = (int4, int8) ,
  OPERATOR 4 >= (int4, int8) ,
  OPERATOR 5 > (int4, int8) ,
  FUNCTION 1 btint48cmp(int4, int8) ,

  -- comparaisons inter-types int2 vs int8
  OPERATOR 1 < (int2, int8) ,
  OPERATOR 2 <= (int2, int8) ,
  OPERATOR 3 = (int2, int8) ,
  OPERATOR 4 >= (int2, int8) ,
  OPERATOR 5 > (int2, int8) ,
  FUNCTION 1 btint28cmp(int2, int8) ,

  -- comparaisons inter-types int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

Notez que cette définition « surcharge » la stratégie de l'opérateur et les numéros de fonction support : chaque numéro survient plusieurs fois dans la famille. Ceci est autorisé aussi longtemps que chaque instance d'un numéro particulier a des types de données distincts en entrée. Les instances qui ont les deux types en entrée égalent au type en entrée de la classe d'opérateur sont les opérateurs primaires et les fonctions de support pour cette classe d'opérateur et, dans la plupart des cas, doivent être déclarées comme membre de la classe d'opérateur plutôt qu'en tant que membres lâches de la famille.

Dans une famille d'opérateur B-tree, tous les opérateurs de la famille doivent trier de façon compatible, ceci signifiant ques les lois transitives tiennent parmi tous les types de données supportés par la famille : « if A = B and B = C, then A = C » et « if A < B and B < C, then A < C ». Pour chaque opérateur de la famille, il doit y avoir une fonction de support pour les deux mêmes types de données en entrée que celui de l'opérateur. Il est recommandé qu'une famille soit complète, c'est-à-dire que pour chaque combinaison de types de données, tous les opérateurs sont inclus. Chaque classe d'opérateur doit juste inclure les opérateurs non inter-types et les fonctions de support pour ce type de données.

Pour construire une famille d'opérateurs de hachage pour plusieurs types de données, des fonctions de support de hachage compatibles doivent être créées pour chaque type de données supporté par la famille. Ici, compatibilité signifie que les fonctions sont garanties de renvoyer le même code de hachage pour toutes les paires de valeurs qui sont considérées égales par les opérateurs d'égalité de la famille, même quand les valeurs sont de type différent. Ceci est habituellement difficile à accomplir quand les types ont différentes représentations physiques, mais cela peut se faire dans la plupart des cas. Notez qu'il y a seulement une fonction de support par type de données, pas une par opérateur d'égalité. Il est recommandé qu'une famille soit terminée, c'est-à-dire fournit un opérateur d'égalité pour chaque combinaison de types de données. Chaque classe d'opérateur doit inclure l'opérateur d'égalité non inter-type et la fonction de support pour ce type de données.

Les index GIN et GiST n'ont pas de notion explicite d'opérations inter-types. L'ensemble des opérateurs supportés est simplement ce que les fonctions de support primaire peuvent supporter pour un opérateur donné.

Note

Avant PostgreSQL™ 8.3, le concept des familles d'opérateurs n'existait pas. Donc, tous les opérateurs inter-type dont le but était d'être utilisés avec un index étaient liés directement à la classe d'opérateur de l'index. Bien que cette approche fonctionne toujours, elle est obsolète car elle rend trop importantes les dépendances de l'index et

parce que le planificateur peut gérer des comparaisons inter-type avec plus d'efficacité que quand les typdes de données ont des opérateurs dans la même famille d'opérateur.

35.14.6. Dépendances du système pour les classes d'opérateur

PostgreSQL™ utilise les classe d'opérateur pour inférer les propriétés des opérateurs de plusieurs autres façons que le seul usage avec les index. Donc, vous pouvez créer des classes d'opérateur même si vous n'avez pas l'intention d'indexer une quelconque colonne de votre type de donnée.

En particulier, il existe des caractéristiques de SQL telles que `ORDER BY` et `DISTINCT` qui requièrent la comparaison et le tri des valeurs. Pour implémenter ces caractéristiques sur un type de donnée défini par l'utilisateur, PostgreSQL™ recherche la classe d'opérateur B-tree par défaut pour le type de donnée. Le membre « equals » de cette classe d'opérateur définit pour le système la notion d'égalité des valeurs pour `GROUP BY` et `DISTINCT`, et le tri ordonné imposé par la classe d'opérateur définit le `ORDER BY` par défaut.

La comparaison des tableaux de types définis par l'utilisateur repose sur les sémantiques définies par la classe d'opérateur B-tree par défaut.

S'il n'y a pas de classe d'opérateur B-tree par défaut pour le type de donnée, le système cherchera une classe d'opérateur de découpage. Mais puisque cette classe d'opérateur ne fournit que l'égalité, c'est en pratique seulement suffisant pour établir l'égalité de tableau.

Quand il n'y a pas de classe d'opérateur par défaut pour un type de donnée, vous obtenez des erreurs telles que « could not identify an ordering operator » si vous essayez d'utiliser ces caractéristiques SQL avec le type de donnée.

Note

Dans les versions de PostgreSQL™ antérieures à la 7.4, les opérations de tri et de groupement utilisaient implicitement les opérateurs nommés `=`, `<` et `>`. Le nouveau comportement qui repose sur les classes d'opérateurs par défaut évite d'avoir à faire une quelconque supposition sur le comportement des opérateurs avec des noms particuliers.

Un autre point important est qu'un opérateur apparaissant dans une famille d'opérateur de hachage est un candidat pour les jointures de hachage, les agrégations de hachage et les optimisations relatives. La famille d'opérateur de hachage est essentiel ici car elle identifie le(s) fonction(s) de hachage à utiliser.

35.14.7. Caractéristiques spéciales des classes d'opérateur

Il y a deux caractéristiques spéciales des classes d'opérateur dont nous n'avons pas encore parlées, essentiellement parce qu'elles ne sont pas utiles avec les méthodes d'index les plus communément utilisées.

Normalement, déclarer un opérateur comme membre d'une classe ou d'une famille d'opérateur signifie que la méthode d'indexation peut retrouver exactement l'ensemble de lignes qui satisfait la condition `WHERE` utilisant cet opérateur. Par exemple :

```
SELECT * FROM table WHERE colonne_entier < 4;
```

peut être accompli exactement par un index B-tree sur la colonne entière. Mais il y a des cas où un index est utile comme un guide inexact vers la colonne correspondante. Par exemple, si un index GiST enregistre seulement les rectangles limite des objets géométriques, alors il ne peut pas exactement satisfaire une condition `WHERE` qui teste le chevauchement entre des objets non rectangulaires comme des polygones. Cependant, nous pourrions utiliser l'index pour trouver des objets dont les rectangles limites chevauchent les limites de l'objet cible. Dans ce cas, l'index est dit être à perte pour l'opérateur. Les recherches par index à perte sont implémentées en ayant une méthode d'indexage qui renvoie un drapeau *recheck* quand une ligne pourrait ou non satisfaire la condition de la requête. Le système principal testera ensuite la condition originale de la requête sur la ligne récupérée pour s'assurer que la correspondance est réelle. Cette approche fonctionne si l'index garantit de renvoyer toutes les lignes requises, ainsi que quelques lignes supplémentaires qui pourront être éliminées par la vérification. Les méthodes d'indexage qui supportent les recherches à perte (actuellement GiST et GIN) permettent aux fonctions de support des classes individuelles d'opérateurs de lever le drapeau recheck, et donc c'est essentiellement une fonctionnalité pour les classes d'opérateur.

Considérons à nouveau la situation où nous gardons seulement dans l'index le rectangle délimitant un objet complexe comme un polygone. Dans ce cas, il n'est pas très intéressant de conserver le polygone entier dans l'index - nous pouvons aussi bien conserver seulement un objet simple du type box. Cette situation est exprimée par l'option `STORAGE` dans la commande **CREATE OPERATOR CLASS** : nous aurons à écrire quelque chose comme :

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

Actuellement, seule les méthodes d'indexation GIN et GiST supportent un type `STORAGE` qui soit différent du type de donnée de la colonne. Les routines d'appui de GiST pour la compression (`compress`) et la décompression (`decompress`) doivent s'occuper de la conversion du type de donnée quand `STORAGE` est utilisé. Avec GIN, le type `STORAGE` identifie le type des valeurs « key », qui est normalement différent du type de la colonne indexée -- par exemple, une classe d'opérateur pour des colonnes de tableaux d'entiers pourrait avoir des clés qui sont seulement des entiers. Les routines de support GIN `extractValue` et `extractQuery` sont responsables de l'extraction des clés à partir des valeurs indexées.

35.15. Utiliser le langage C++

Il est possible d'utiliser un compilateur C++ pour construire des extensions PostgreSQL™ en suivant les règles suivantes :

- Toutes les fonctions utilisées par le serveur doivent présentées une interface C au serveur ; ces fonctions C peuvent ensuite appeler des fonctions C++. Par exemple, le lien `extern C` est requis pour les fonctions utilisées par le serveur. Il est aussi nécessaire que chaque fonction passée comme pointeur entre le serveur et le code C++.
- Libérer la mémoire en utilisant la méthode appropriée de libération. Par exemple, la majorité de la mémoire serveur est allouée en utilisant la fonction `palloc()`, donc utilisez `pfree()` pour la libérer. Utiliser la fonction `delete()` de C++ pour ces cas échouera.
- Empêcher la propagation des exceptions dans le code C (utilisez un bloc capable de récupérer toutes les exceptions au plus haut niveau des fonctions `extern C`). Ceci est nécessaire même si le code C++ ne renvoie aucune exceptions car les événements comme les manques de mémoire renvoient toujours des exceptions. Toute exception doit être récupérée et une erreur appropriée doit être envoyée à l'interface C. Si possible, compilez C++ avec l'option `-fno-exceptions` pour complètement éliminer les exceptions ; dans ce cas, vous devez vérifier les erreurs dans votre code C++, par exemple vérifier le renvoi de NULL suite à l'appel de la méthode `new()`.
- S'il doit y avoir des appels aux fonctions du serveur à partir du code C++, assurez-vous que la pile d'appels C++ contient seulement l'ancienne structure de données (POD). Cela est nécessaire car des erreurs du serveur génèrent un appel distant à `longjmp()` qui ne dépile par proprement la pile d'appels C++ avec les objets autre que POD.

Pour résumé, il est préférable de placer le code C++ derrière un mur de fonctions `extern C` qui font l'interface avec le serveur, et d'éviter les fuites d'exception, de mémoire et de pile d'appels.

Chapitre 36. Déclencheurs (triggers)

Ce chapitre fournit des informations générales sur l'écriture des fonctions pour déclencheur. Les fonctions pour déclencheurs peuvent être écrites dans la plupart des langages de procédure disponibles incluant PL/pgSQL (Chapitre 39, PL/pgSQL - Langage de procédures SQL), PL/Tcl (Chapitre 40, PL/Tcl - Langage de procédures Tcl), PL/Perl (Chapitre 41, PL/Perl - Langage de procédures Perl) et PL/Python (Chapitre 42, PL/Python - Langage de procédures Python). Après avoir lu ce chapitre, vous devriez consulter le chapitre sur votre langage de procédure favori pour découvrir les spécificités de l'écriture de déclencheurs dans ce langage.

Il est aussi possible d'écrire une fonction déclencheur en C, bien que la plupart des gens trouvent plus facile d'utiliser un des langages de procédure. Il est actuellement impossible d'écrire une fonction déclencheur dans le langage de fonction simple SQL.

36.1. Aperçu du comportement des déclencheurs

Un déclencheur spécifie que la base de données doit exécuter automatiquement une fonction donnée chaque fois qu'un certain type d'opération est exécuté. Les fonctions déclencheur peuvent être définies pour s'exécuter avant ou après une commande **INSERT**, **UPDATE** ou **DELETE**, soit une fois par ligne modifiée, soit une fois par expression SQL. Les triggers **UPDATE** peuvent en plus être configurées pour n'être déclenchés que si certaines colonnes sont mentionnées dans la clause `SET` de l'instruction **UPDATE**. Les triggers peuvent aussi se déclencher pour des instructions **TRUNCATE**. Si un événement d'un trigger intervient, la fonction du trigger est appelée au moment approprié pour gérer l'événement.

La fonction déclencheur doit être définie avant que le déclencheur lui-même puisse être créé. La fonction déclencheur doit être déclarée comme une fonction ne prenant aucun argument et retournant un type `trigger` (la fonction déclencheur reçoit ses entrées via une structure TriggerData passée spécifiquement, et non pas sous la forme d'arguments ordinaires de fonctions).

Une fois qu'une fonction déclencheur est créée, le déclencheur (trigger) est créé avec CREATE TRIGGER(7). La même fonction déclencheur est utilisable par plusieurs déclencheurs.

PostgreSQL™ offre des déclencheurs *par ligne* et *par instruction*. Avec un déclencheur mode ligne, la fonction du déclencheur est appelée une fois pour chaque ligne affectée par l'instruction qui a lancé le déclencheur. Au contraire, un déclencheur mode instruction n'est appelé qu'une seule fois lorsqu'une instruction appropriée est exécutée, quelque soit le nombre de lignes affectées par cette instruction. En particulier, une instruction n'affectant aucune ligne résultera toujours en l'exécution de tout déclencheur mode instruction applicable. Ces deux types sont quelque fois appelés respectivement des *déclencheurs niveau ligne* et des *déclencheurs niveau instruction*. Les triggers sur **TRUNCATE** peuvent seulement être définis au niveau instruction.

Les déclencheurs sont aussi classifiés comme des déclencheurs *avant* et des déclencheurs *après*. Les déclencheurs avant au niveau instruction se déclenchent naturellement avant que l'instruction ne fasse quoi que ce soit alors que les déclencheurs après au niveau instruction sont exécutés à la fin de l'instruction. Les déclencheurs avant au niveau ligne s'exécutent immédiatement avant qu'une ligne particulière ne soit traitée alors que les déclencheurs après au niveau ligne s'exécutent à la fin de l'instruction (mais avant tout déclencheur après au niveau instruction).

Les fonctions déclencheurs appelées par des déclencheurs niveau instruction devraient toujours renvoyer NULL. Les fonctions déclencheurs appelées par des déclencheurs niveau ligne peuvent renvoyer une ligne de la table (une valeur de type HeapTuple) vers l'exécuteur appelant, s'ils le veulent. Un déclencheur niveau ligne exécuté avant une opération a les choix suivants :

- Il peut retourner un pointeur NULL pour sauter l'opération pour la ligne courante. Ceci donne comme instruction à l'exécuteur de ne pas exécuter l'opération niveau ligne qui a lancé le déclencheur (l'insertion ou la modification d'une ligne particulière de la table).
- Pour les déclencheurs **INSERT** et **UPDATE** de niveau ligne uniquement, la valeur de retour devient la ligne qui sera insérée ou remplacera la ligne en cours de mise à jour. Ceci permet à la fonction déclencheur de modifier la ligne en cours d'insertion ou de mise à jour.

Un déclencheur avant niveau ligne qui ne serait pas conçu pour avoir l'un de ces comportements doit prendre garde à retourner la même ligne que celle qui lui a été passée comme nouvelle ligne (c'est-à-dire : pour des déclencheurs **INSERT** et **UPDATE** : la nouvelle (`NEW`) ligne ,et pour les déclencheurs **DELETE**) : l'ancienne (`OLD`) ligne .

La valeur de retour est ignorée pour les déclencheurs niveau ligne lancés après une opération. Ils peuvent donc renvoyer la valeur NULL.

Si plus d'un déclencheur est défini pour le même événement sur la même relation, les déclencheurs seront lancés dans l'ordre alphabétique de leur nom. Dans le cas de déclencheurs avant, la ligne renvoyée par chaque déclencheur, qui a éventuellement été modifiée, devient l'argument du prochain déclencheur. Si un des déclencheurs avant renvoie un pointeur NULL, l'opération est abandonnée pour cette ligne et les déclencheurs suivants ne sont pas lancés.

Une définition de trigger peut aussi spécifier une condition booléenne `WHEN` qui sera testée pour savoir si le trigger doit bien

être déclenché. Dans les triggers de niveau ligne, la condition `WHEN` peut examiner l'ancienne et la nouvelle valeur des colonnes de la ligne. (les triggers de niveau instruction peuvent aussi avoir des conditions `WHEN` mais cette fonctionnalité est moins intéressante pour elles). Dans un trigger *avant*, la condition `WHEN` est évaluée juste avant l'exécution de la fonction, donc l'utilisation de `WHEN` n'est pas réellement différente du test de la même condition au début de la fonction trigger. Néanmoins, dans un tigger *après*, la condition `WHEN` est évaluée juste avant la mise à jour de la ligne et détermine si un événement va déclencher le trigger à la fin de l'instruction. Donc, quand la condition `WHEN` d'un trigger *après* ne renvoie pas true, il n'est pas nécessaire de mettre en queue un événement ou de récupérer de nouveau la ligne à la fin de l'instriction. Ceci permet une amélioration conséquente des performances pour les instructions qui modifient un grand nombre de lignes si le trigger a seulement besoin d'être exécuté que sur quelques lignes.

Les déclencheurs avant en mode ligne sont typiquement utilisés pour vérifier ou modifier les données qui seront insérées ou mises à jour. Par exemple, un déclencheur avant pourrait être utilisé pour insérer l'heure actuelle dans une colonne de type timestamp ou pour vérifier que deux éléments d'une ligne sont cohérents. Les déclencheurs après en mode ligne sont pour la plupart utilisés pour propager des mises à jour vers d'autres tables ou pour réaliser des tests de cohérence avec d'autres tables. La raison de cette division du travail est qu'un déclencheur après peut être certain qu'il voit la valeur finale de la ligne alors qu'un déclencheur avant ne l'est pas ; il pourrait exister d'autres déclencheurs avant qui seront exécutés après lui. Si vous n'avez aucune raison spéciale pour le moment du déclenchement, le cas avant est plus efficace car l'information sur l'opération n'a pas besoin d'être sauvegardée jusqu'à la fin du traitement.

Si une fonction déclencheur exécute des commandes SQL, alors ces commandes peuvent lancer à leur tour des déclencheurs. On appelle ceci un déclencheur en cascade. Il n'y a pas de limitation directe du nombre de niveaux de cascade. Il est possible que les cascades causent un appel récursif du même déclencheur ; par exemple, un déclencheur **INSERT** pourrait exécuter une commande qui insère une ligne supplémentaire dans la même table, entraînant un nouveau lancement du déclencheur **INSERT**. Il est de la responsabilité du programmeur d'éviter les récursions infinies dans de tels scénarios.

Quand un déclencheur est défini, des arguments peuvent être spécifiés pour lui. L'objectif de l'inclusion d'arguments dans la définition du déclencheur est de permettre à différents déclencheurs ayant des exigences similaires d'appeler la même fonction. Par exemple, il pourrait y avoir une fonction déclencheur généralisée qui prend comme arguments deux noms de colonnes et place l'utilisateur courant dans l'une et un horodatage dans l'autre. Correctement écrit, cette fonction déclencheur serait indépendante de la table particulière sur laquelle il se déclenche. Ainsi, la même fonction pourrait être utilisée pour des événements **INSERT** sur n'importe quelle table ayant des colonnes adéquates, pour automatiquement suivre les créations d'enregistrements dans une table de transactions par exemple. Elle pourrait aussi être utilisée pour suivre les dernières mises à jours si elle est définie comme un déclencheur **UPDATE**.

Chaque langage de programmation supportant les déclencheurs a sa propre méthode pour rendre les données en entrée disponible à la fonction du déclencheur. Cette donnée en entrée inclut le type d'événement du déclencheur (c'est-à-dire **INSERT** ou **UPDATE**) ainsi que tous les arguments listés dans **CREATE TRIGGER**. Pour un déclencheur niveau ligne, la donnée en entrée inclut aussi la ligne `NEW` pour les déclencheurs **INSERT** et **UPDATE** et/ou la ligne `OLD` pour les déclencheurs **UPDATE** et **DELETE**. Les déclencheurs niveau instruction n'ont actuellement aucun moyen pour examiner le(s) ligne(s) individuelle(s) modifiées par l'instruction.

36.2. Visibilité des modifications des données

Si vous exécutez des commandes SQL dans votre fonction SQL et que ces commandes accèdent à la table pour laquelle vous créez ce déclencheur, alors vous avez besoin de connaître les règles de visibilité des données car elles déterminent si les commandes SQL voient les modifications de données pour lesquelles est exécuté le déclencheur. En bref :

- Les déclencheurs niveau instruction suivent des règles de visibilité simples : aucune des modifications réalisées par une instruction n'est visible aux déclencheurs niveau instruction appelés avant l'instruction alors que toutes les modifications sont visibles aux déclencheurs après niveau instruction.
- Les modifications de données (insertion, mise à jour ou suppression) lançant le déclencheur ne sont naturellement *pas* visibles aux commandes SQL exécutées dans un déclencheur avant en mode ligne parce qu'elles ne sont pas encore survenues.
- Néanmoins, les commandes SQL exécutées par un déclencheur avant en mode ligne *verront* les effets des modifications de données pour les lignes précédemment traitées dans la même commande externe. Ceci requiert une grande attention car l'ordre des événements de modification n'est en général pas prévisible ; une commande SQL affectant plusieurs lignes pourrait visiter les lignes dans n'importe quel ordre.
- Quand un déclencheur après en mode ligne est exécuté, toutes les modifications de données réalisées par la commande externe sont déjà terminées et sont visibles par la fonction appelée par le déclencheur.

Si votre fonction trigger est écrite dans un des langages de procédures standard, alors les instructions ci-desus s'appliquent seulement si la fonction est déclarée `VOLATILE`. Les fonctions déclarées `STABLE` ou `IMMUTABLE` ne verront pas les modifications réalisées par la commande appelante dans tous les cas.

Il existe plus d'informations sur les règles de visibilité des données dans la Section 43.4, « Visibilité des modifications de données ». L'exemple dans la Section 36.4, « Un exemple complet de trigger » contient une démonstration de ces règles.

36.3. Écrire des fonctions déclencheurs en C

Cette section décrit les détails de bas niveau de l'interface d'une fonction déclencheur. Ces informations ne sont nécessaires que lors de l'écriture d'une fonction déclencheur en C. Si vous utilisez un langage de plus haut niveau, ces détails sont gérés pour vous. Dans la plupart des cas, vous devez considérer l'utilisation d'un langage de procédure avant d'écrire vos déclencheurs en C. La documentation de chaque langage de procédures explique comment écrire un déclencheur dans ce langage.

Les fonctions déclencheurs doivent utiliser la « version 1 » de l'interface du gestionnaire de fonctions.

Quand une fonction est appelée par le gestionnaire de déclencheur, elle ne reçoit aucun argument classique, mais un pointeur de « contexte » pointant sur une structure TriggerData. Les fonctions C peuvent vérifier si elles sont appelées par le gestionnaire de déclencheurs ou pas en exécutant la macro :

```
CALLED_AS_TRIGGER(fcinfo)
```

qui se décompose en :

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Si elle retourne la valeur vraie, alors il est bon de convertir `fcinfo->context` en type `TriggerData *` et de faire usage de la structure pointée TriggerData. La fonction *ne* doit *pas* modifier la structure TriggerData ou une donnée quelconque vers laquelle elle pointe.

struct TriggerData est définie dans `commands/trigger.h` :

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

où les membres sont définis comme suit :

type
: Toujours `T_TriggerData`.

tg_event
: Décrit l'événement pour lequel la fonction est appelée. Vous pouvez utiliser les macros suivantes pour examiner `tg_event` :

  `TRIGGER_FIRED_BEFORE(tg_event)`
  Renvoie vrai si le déclencheur est lancé avant l'opération.

  `TRIGGER_FIRED_AFTER(tg_event)`
  Renvoie vrai si le déclencheur est lancé après l'opération.

  `TRIGGER_FIRED_FOR_ROW(tg_event)`
  Renvoie vrai si le déclencheur est lancé pour un événement en mode ligne.

  `TRIGGER_FIRED_FOR_STATEMENT(tg_event)`
  Renvoie vrai si le déclencheur est lancé pour un événement en mode instruction.

  `TRIGGER_FIRED_BY_INSERT(tg_event)`
  Retourne vrai si le déclencheur est lancé par une commande **INSERT**.

  `TRIGGER_FIRED_BY_UPDATE(tg_event)`
  Retourne vrai si le déclencheur est lancé par une commande **UPDATE**.

  `TRIGGER_FIRED_BY_DELETE(tg_event)`
  Retourne vrai si le déclencheur est lancé par une commande **DELETE**.

  `TRIGGER_FIRED_BY_TRUNCATE(tg_event)`
  Renvoie true si le trigger a été déclenché par une commande **TRUNCATE**.

tg_relation
Un pointeur vers une structure décrivant la relation pour laquelle le déclencheur est lancé. Voir `utils/rel.h` pour les détails de cette structure. Les choses les plus intéressantes sont `tg_relation->rd_att` (descripteur de nuplets de la relation) et `tg_relation->rd_rel->relname` (nom de la relation ; le type n'est pas char* mais NameData ; utilisez `SPI_getrelname(tg_relation)` pour obtenir un char* si vous avez besoin d'une copie du nom).

tg_trigtuple
Un pointeur vers la ligne pour laquelle le déclencheur a été lancé. Il s'agit de la ligne étant insérée, mise à jour ou effacée. Si ce déclencheur a été lancé pour une commande **INSERT** ou **DELETE**, c'est cette valeur que la fonction doit retourner si vous ne voulez pas remplacer la ligne par une ligne différente (dans le cas d'un **INSERT**) ou sauter l'opération.

tg_newtuple
Un pointeur vers la nouvelle version de la ligne, si le déclencheur a été lancé pour un **UPDATE** et NULL si c'est pour un **INSERT** ou un **DELETE**. C'est ce que la fonction doit retourner si l'événement est un **UPDATE** et que vous ne voulez pas remplacer cette ligne par une ligne différente ou bien sauter l'opération.

tg_trigger
Un pointeur vers une structure de type Trigger, définie dans `utils/rel.h` :

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
} Trigger;
```

où *tgname* est le nom du déclencheur, *tgnargs* est le nombre d'arguments dans *tgargs* et *tgargs* est un tableau de pointeurs vers les arguments spécifiés dans l'expression contenant la commande **CREATE TRIGGER**. Les autres membres ne sont destinés qu'à un usage interne.

tg_trigtuplebuf
Le tampon contenant *tg_trigtuple* ou InvalidBuffer s'il n'existe pas une telle ligne ou si elle n'est pas stockée dans un tampon du disque.

tg_newtuplebuf
Le tampon contenant *tg_newtuple* ou InvalidBuffer s'il n'existe pas une telle ligne ou si elle n'est pas stockée dans un tampon du disque.

Une fonction déclencheur doit retourner soit un pointeur HeapTuple soit un pointeur NULL (*pas* une valeur SQL NULL, donc ne positionnez pas *isNull* à true). Faites attention de renvoyer soit un *tg_trigtuple* soit un *tg_newtuple*, comme approprié, si vous ne voulez pas changer la ligne en cours de modification.

36.4. Un exemple complet de trigger

Voici un exemple très simple de fonction déclencheur écrite en C (les exemples de déclencheurs écrits avec différents langages de procédures se trouvent dans la documentation de ceux-ci).

La fonction `trigf` indique le nombre de lignes de la table ttest et saute l'opération si la commande tente d'insérer une valeur NULL dans la colonne *x* (ainsi le déclencheur agit comme une contrainte non NULL mais n'annule pas la transaction).

Tout d'abord, la définition des tables :

```
CREATE TABLE ttest (
    x integer
);
```

Voici le code source de la fonction trigger :

```
#include "postgres.h"
#include "executor/spi.h"       /* nécessaire pour fonctionner avec SPI */
#include "commands/trigger.h"   /* ... et les déclencheurs */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checkNULL = false;
    bool        isNULL;
    int         ret, i;

    /* on s'assure que la fonction est appelée en tant que déclencheur */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* nuplet à retourner à l'exécuteur */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* vérification des valeurs NULL */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checkNULL = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connexion au gestionnaire SPI */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* obtient le nombre de lignes dans la table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) renvoie int8, prenez garde à bien convertir */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isNULL));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checkNULL)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isNULL);
```

```
        if (isNULL)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

Après avoir compilé le code source (voir Section 35.9.6, « Compiler et lier des fonctions chargées dynamiquement »), déclarez la fonction et les déclencheurs :

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'nomfichier'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

À présent, testez le fonctionnement du déclencheur :

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion supprimée et déclencheur APRES non exécuté

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                        ^^^^^^^^
                       souvenez-vous de ce que nous avons dit sur la visibilité.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                        ^^^^^^
                       souvenez-vous de ce que nous avons dit sur la visibilité.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
```

```
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                         ^^^^^^
                       souvenez-vous de ce que nous avons dit sur la visibilité.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Vous trouverez des exemples plus complexes dans `src/test/regress/regress.c` et dans `contrib/spi`.

Chapitre 37. Système de règles

Ce chapitre discute du système de règles dans PostgreSQL™. les systèmes de règles de production sont simples conceptuellement mais il existe de nombreux points subtils impliqués dans leur utilisation.

Certains autres systèmes de bases de données définissent des règles actives pour la base de données, conservées habituellement en tant que procédures stockées et déclencheurs. Avec PostgreSQL™, elles peuvent aussi être implémentées en utilisant des fonctions et des déclencheurs.

Le système de règles (plus précisément, le système de règles de réécriture de requêtes) est totalement différent des procédures stockées et des déclencheurs. Il modifie les requêtes pour prendre en considération les règles puis passe la requête modifiée au planificateur de requêtes pour planification et exécution. Il est très puissant et peut être utilisé pour beaucoup de choses comme des procédures en langage de requêtes, des vues et des versions. Les fondations théoriques et la puissance de ce système de règles sont aussi discutées dans Stonebraker et al, ACM, 1990 et Ong and Goh, 1990.

37.1. Arbre de requêtes

Pour comprendre comment fonctionne le système de règles, il est nécessaire de comprendre quand il est appelé et quelles sont ses entrées et sorties.

Le système de règles est situé entre l'analyseur et le planificateur. Il prend la sortie de l'analyseur, un arbre de requête et les règles de réécriture définies par l'utilisateur qui sont aussi des arbres de requêtes avec quelques informations supplémentaires, et crée zéro ou plusieurs arbres de requêtes comme résultat. Donc, son entrée et sortie sont toujours des éléments que l'analyseur lui-même pourrait avoir produit et, du coup, tout ce qu'il voit est représentable basiquement comme une instruction SQL.

Maintenant, qu'est-ce qu'un arbre de requêtes ? C'est une représentation interne d'une instruction SQL où les parties qui le forment sont stockées séparément. Ces arbres de requêtes sont affichables dans le journal de traces du serveur si vous avez configuré les paramètres `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan`. les actions de règles sont aussi enregistrées comme arbres de requêtes dans le catalogue système pg_rewrite. elles ne sont pas formatées comme la sortie de traces mais elles contiennent exactement la même information.

Lire un arbre de requête brut requiert un peu d'expérience. Mais comme les représentations SQL des arbres de requêtes sont suffisantes pour comprendre le système de règles, ce chapitre ne vous apprendra pas à les lire.

Lors de la lecture des représentations SQL des arbres de requêtes dans ce chapitre, il est nécessaire d'être capable d'identifier les morceaux cassés de l'instruction lorsqu'ils sont dans la structure de l'arbre de requête. Les parties d'un arbre de requêtes sont

le type de commande
: C'est une simple valeur indiquant quelle commande (**select**, **insert**, **update**, **delete**) l'arbre de requêtes produira.

la table d'échelle

: La table d'échelle est une liste des relations utilisées dans la requête. Dans une instruction **select**, ce sont les relations données après le mot clé `from`.

: Chaque entrée de la table d'échelle identifie une table ou une vue et indique par quel nom elle est désignée dans les autres parties de la requête. Dans l'arbre de requêtes, les entrées de la table d'échelle sont référencées par des numéros plutôt que par des noms. Il importe donc peu, ici, de savoir s'il y a des noms dupliqués comme cela peut être le cas avec une instruction SQL. Cela peut arriver après l'assemblage des tables d'échelle des règles. Les exemples de ce chapitre ne sont pas confrontés à cette situation.

la relation résultat
: C'est un index dans la table d'échelle qui identifie la relation où iront les résultats de la requête.

: Les requêtes **select** n'ont normalement pas de relation résultat. Le cas spécial d'un **select into** est pratiquement identique à un **create table** suivi par un `insert ... select` et n'est pas discuté séparément ici.

: Pour les commandes **insert**, **update** et **delete**, la relation de résultat est la table (ou vue !) où les changements prennent effet.

la liste cible

: La liste cible est une liste d'expressions définissant le résultat d'une requête. Dans le cas d'un **select**, ces expressions sont celles qui construisent la sortie finale de la requête. Ils correspondent aux expressions entre les mots clés **select** et **from** (`*` est seulement une abréviation pour tous les noms de colonnes d'une relation. Il est étendu par l'analyseur en colonnes individuelles, pour que le système de règles ne le voit jamais).

: Les commandes **delete** n'ont pas besoin d'une liste de colonnes car elles ne produisent aucun résultat. En fait, le planifica-

teur ajoutera une entrée spéciale ctid pour aller jusqu'à la liste de cibles mais c'est après le système de règles ; pour le système de règles, ce sera discuté après ; pour le système de règles, la liste de cibles est vide.

Pour les commandes **insert**, la liste cible décrit les nouvelles lignes devant aller dans la relation résultat. Elle consiste en des expressions de la clause `values` ou en celles de la clause **select** dans `insert ... SELECT`. la première étape du processus de réécriture ajoute les entrées de la liste cible pour les colonnes n'ont affectées par la commande originale mais ayant des valeurs par défaut. Toute colonne restante (avec soit une valeur donnée soit une valeur par défaut) sera remplie par le planificateur avec une expression NULL constante.

Pour les commandes **update**, la liste cible décrit les nouvelles lignes remplaçant les anciennes. Dans le système des règles, elle contient seulement les expressions de la partie `set colonne = expression` de la commande. le planificateur gèrera les colonnes manquantes en insérant des expressions qui copient les valeurs provenant de l'ancienne ligne dans la nouvelle. et elle ajoutera l'ancienne entrée ctid dans la nouvelle ligne comme pour le **delete**.

Chaque entrée de la liste cible contient une expression qui peut être une valeur constante, une variable pointant vers une colonne d'une des relations de la table d'échelle, un paramètre ou un arbre d'expressions réalisé à partir d'appels de fonctions, de constantes, de variables, d'opérateurs, etc.

la qualification
: La qualification de la requête est une expression ressemblant à une de celles contenues dans les entrées de la liste cible. La valeur résultant de cette expression est un booléen indiquant si l'opération (**insert**, **update**, **delete** ou **select**) pour la ligne de résultat final devrait être exécutée ou non. Elle correspond à la clause `where` d'une instruction SQL.

l'arbre de jointure
: L'arbre de jointure de la requête affiche la structure de la clause `from`. pour une simple requête comme `select ... from a, b, c`, l'arbre de jointure est une simple liste d'éléments de `from` parce que nous sommes autorisés à les joindre dans tout ordre. Mais quand des expressions `join`, et plus particulièrement les jointures externes, sont utilisées, nous devons les joindre dans l'ordre affiché par les jointures. Dans ce cas, l'arbre de jointure affiche la structure des expressions `join`. les restrictions associées avec ces clauses `join` particulières (à partir d'expressions `on` ou `using`) sont enregistrées comme des expressions de qualification attachées aux nœuds de l'arbre de jointure. Il s'avère agréable d'enregistrer l'expression de haut niveau `where` comme une qualification attachée à l'élément de l'arbre de jointure de haut niveau. Donc, réellement, l'arbre de jointure représente à la fois les clauses `from` et `where` d'un **select**.

le reste
: Les autres parties de l'arbre de requête comme la clause `order BY` n'ont pas d'intérêt ici. le système de règles substitue quelques entrées lors de l'application des règles mais ceci n'a pas grand chose à voir avec les fondamentaux du système de règles.

37.2. Vues et système de règles

Avec PostgreSQL™, les vues sont implémentées en utilisant le système de règles. En fait, il n'y a essentiellement pas de différences entre

```
CREATE VIEW ma_vue AS SELECT * FROM ma_table;
```

et ces deux commandes :

```
CREATE TABLE ma_vue (liste de colonnes identique à celle de ma_table);
CREATE RULE "_RETURN" AS ON SELECT TO ma_vue DO INSTEAD
    SELECT * FROM ma_table;
```

parce que c'est exactement ce que fait la commande **create VIEW** en interne. Cela présente quelques effets de bord. L'un d'entre eux est que l'information sur une vue dans les catalogues système PostgreSQL™ est exactement la même que celle d'une table. Donc, pour l'analyseur, il n'y a aucune différence entre une table et une vue. Elles représentent la même chose : des relations.

37.2.1. Fonctionnement des règles select

Les règles `on select` sont appliquées à toutes les requêtes comme la dernière étape, même si la commande donnée est un **insert**, **update** ou **delete**. et ils ont des sémantiques différentes à partir des règles sur les autres types de commandes dans le fait qu'elles modifient l'arbre de requêtes en place au lieu d'en créer un nouveau. Donc, les règles **select** sont décrites avant.

Actuellement, il n'existe qu'une action dans une règle `on SELECT` et elle doit être une action **select** inconditionnelle qui est `instead`. cette restriction était requise pour rendre les règles assez sûres pour les ouvrir aux utilisateurs ordinaires et cela restreint les règles `on select` à agir comme des vues.

Pour ce chapitre, les exemples sont deux vues jointes réalisant quelques calculs et quelques vues supplémentaires les utilisant à leur tour. Une des deux premières vues est personnalisée plus tard en ajoutant des règles pour des opérations **insert**, **update** et **delete** de façon à ce que le résultat final sera une vue qui se comporte comme une vraie table avec quelques fonctionnalités ma-

giques. Il n'existe pas un tel exemple pour commencer et ceci rend les choses plus difficiles à obtenir. Mais il est mieux d'avoir un exemple couvrant tous les points discutés étape par étape plutôt que plusieurs exemples, rendant la compréhension plus difficile.

Pour cet exemple, nous avons besoin d'une petite fonction `min` renvoyant la valeur la plus basse entre deux entiers. Nous la créons ainsi :

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$' LANGUAGE SQL STRICT;
```

Les tables réelles dont nous avons besoin dans les deux premières descriptions du système de règles sont les suivantes :

```
CREATE TABLE donnees_chaussure (
    nom_chaussure         text,     -- clé primaire
    dispo_chaussure       integer,  -- nombre de pairs disponibles
    couleur_chaussure     text,     -- couleur de lacet préférée
    long_min_chaussure    real,     -- longueur minimum du lacet
    long_max_chaussure    real,     -- longueur maximum du lacet
    unite_long_chaussure  text      -- unité de longueur
);

CREATE TABLE donnees_lacet (
    nom_lacet             text,     -- clé primaire
    dispo_lacet           integer,  -- nombre de pairs disponibles
    couleur_lacet         text,     -- couleur du lacet
    longueur_lacet        real,     -- longueur du lacet
    unite_lacet           text      -- unité de longueur
);

CREATE TABLE unite (
    nom_unite             text,     -- clé primaire
    facteur_unite         real      -- facteur pour le transformer en cm
);
```

Comme vous pouvez le constater, elles représentent les données d'un magasin de chaussures.

Les vues sont créées avec :

```
CREATE VIEW chaussure AS
    SELECT sh.nom_chaussure,
           sh.dispo_chaussure,
           sh.couleur_chaussure,
           sh.long_min_chaussure,
           sh.long_min_chaussure * un.facteur_unite AS long_min_chaussure_cm,
           sh.long_max_chaussure,
           sh.long_max_chaussure * un.facteur_unite AS long_max_chaussure_cm,
           sh.unite_long_chaussure
      FROM donnees_chaussure sh, unite un
     WHERE sh.unite_long_chaussure = un.nom_unite;

CREATE VIEW lacet AS
    SELECT s.nom_lacet,
           s.dispo_lacet,
           s.couleur_lacet,
           s.longueur_lacet,
           s.unite_lacet,
           s.longueur_lacet * u.facteur_unite AS longueur_lacet_cm
      FROM donnees_lacet s, unite u
     WHERE s.unite_lacet = u.nom_unite;

CREATE VIEW chaussure_prete AS
    SELECT rsh.nom_chaussure,
           rsh.dispo_chaussure,
           rsl.nom_lacet,
           rsl.dispo_lacet,
           min(rsh.dispo, rsl.dispo_lacet) AS total_avail
      FROM chaussure rsh, lacet rsl
     WHERE rsl.couleur_lacet = rsh.couleur
       AND rsl.longueur_lacet_cm >= rsh.long_min_chaussure_cm
       AND rsl.longueur_lacet_cm <= rsh.long_max_chaussure_cm;
```

La commande **create view** pour la vue `lacet` (qui est la plus simple que nous avons) écrira une relation `lacet` et une entrée

dans pg_rewrite indiquant la présence d'une règle de réécriture devant être appliquée à chaque fois que la relation `lacet` est référencée dans une table de la requête. La règle n'a aucune qualification de règle (discuté plus tard, avec les règles autres que **select** car les règles **select** ne le sont pas encore) et qu'il s'agit de `instead`. notez que les qualifications de règles ne sont pas identiques aux qualifications de requêtes. L'action de notre règle a une qualification de requête. L'action de la règle a un arbre de requête qui est une copie de l'instruction **select** dans la commande de création de la vue.

> **Note**
>
> Les deux entrées supplémentaires de la table d'échelle pour `new` et `old` que vous pouvez voir dans l'entrée de pg_rewrite ne sont d'aucun intérêt pour les règles **select**.

Maintenant, nous remplissons `unit`, `donnees_chaussure` et `donnees_lacet`, puis nous lançons une requête simple sur une vue :

```
INSERT INTO unite VALUES ('cm', 1.0);
INSERT INTO unite VALUES ('m', 100.0);
INSERT INTO unite VALUES ('inch', 2.54);

INSERT INTO donnees_chaussure VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO donnees_chaussure VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO donnees_chaussure VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO donnees_chaussure VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO donnees_lacet VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO donnees_lacet VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO donnees_lacet VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO donnees_lacet VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO donnees_lacet VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO donnees_lacet VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO donnees_lacet VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO donnees_lacet VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM lacet;

 nom_lacet   | dispo_lacet | couleur_lacet | longueur_lacet | unite_lacet |
longueur_lacet_cm
-------------+-------------+---------------+----------------+-------------+-----------------
 sl1         |           5 | black         |             80 | cm          |
80
 sl2         |           6 | black         |            100 | cm          |
100
 sl7         |           7 | brown         |             60 | cm          |
60
 sl3         |           0 | black         |             35 | inch        |
88.9
 sl4         |           8 | black         |             40 | inch        |
101.6
 sl8         |           1 | brown         |             40 | inch        |
101.6
 sl5         |           4 | brown         |              1 | m           |
100
 sl6         |           0 | brown         |            0.9 | m           |
90
(8 rows)
```

C'est la requête **select** la plus simple que vous pouvez lancer sur nos vues, donc nous prenons cette opportunité d'expliquer les bases des règles de vues. `select * from lacet` a été interprété par l'analyseur et a produit l'arbre de requête :

```
SELECT lacet.nom_lacet, lacet.dispo_lacet,
       lacet.couleur_lacet, lacet.longueur_lacet,
       lacet.unite_lacet, lacet.longueur_lacet_cm
  FROM lacet lacet;
```

et ceci est transmis au système de règles. Ce système traverse la table d'échelle et vérifie s'il existe des règles pour chaque relation. Lors du traitement d'une entrée de la table d'échelle pour `lacet` (la seule jusqu'à maintenant), il trouve la règle `_return` avec l'arbre de requête :

```
SELECT s.nom_lacet, s.dispo_lacet,
       s.couleur_lacet, s.longueur_lacet, s.unite_lacet,
```

```
       s.longueur_lacet * u.facteur_unite AS longueur_lacet_cm
  FROM lacet old, lacet new,
       donnees_lacet s, unit u
 WHERE s.unite_lacet = u.nom_unite;
```

Pour étendre la vue, la réécriture crée simplement une entrée de la table d'échelle de sous-requête contenant l'arbre de requête de l'action de la règle et substitue cette entrée avec l'original référencé dans la vue. L'arbre d'échelle résultant de la réécriture est pratiquement identique à celui que vous avez saisi :

```
SELECT lacet.nom_lacet, lacet.dispo_lacet,
       lacet.couleur_lacet, lacet.longueur_lacet,
       lacet.unite_lacet, lacet.longueur_lacet_cm
  FROM (SELECT s.nom_lacet,
               s.dispo_lacet,
               s.couleur_lacet,
               s.longueur_lacet,
               s.unite_lacet,
               s.longueur_lacet * u.facteur_unite AS longueur_lacet_cm
          FROM donnees_lacet s, unit u
         WHERE s.unite_lacet = u.nom_unite) lacet;
```

Néanmoins, il y a une différence : la table d'échelle de la sous-requête a deux entrées supplémentaires, `lacet old` et `lacet new`. ces entrées ne participent pas directement dans la requête car elles ne sont pas référencées par l'arbre de jointure de la sous-requête ou par la liste cible. La réécriture les utilise pour enregistrer l'information de vérification des droits d'accès qui étaient présents à l'origine dans l'entrée de table d'échelle référencée par la vue. De cette façon, l'exécution vérifiera toujours que l'utilisateur a les bons droits pour accéder à la vue même s'il n'y a pas d'utilisation directe de la vue dans la requête réécrite.

C'était la première règle appliquée. Le système de règles continuera de vérifier les entrées restantes de la table d'échelle dans la requête principale (dans cet exemple, il n'en existe pas plus), et il vérifiera récursivement les entrées de la table d'échelle dans la sous-requête ajoutée pour voir si une d'elle référence les vues. (Mais il n'étendra ni `old` ni `new` -- sinon nous aurions une récursion infinie !) Dans cet exemple, il n'existe pas de règles de réécriture pour `donnees_lacet` ou `unit`, donc la réécriture est terminée et ce qui est ci-dessus est le résultat final donné au planificateur.

Maintenant, nous voulons écrire une requête qui trouve les chaussures en magasin dont nous avons les lacets correspondants (couleur et longueur) et pour lesquels le nombre total de pairs correspondants exactement est supérieur ou égal à deux.

```
SELECT * FROM chaussure_prete WHERE total_avail >= 2;

 nom_chaussure | dispo | nom_lacet | dispo_lacet | total_avail
---------------+-------+-----------+-------------+-------------
 sh1           |     2 | sl1       |           5 |           2
 sh3           |     4 | sl7       |           7 |           4
(2 rows)
```

Cette fois, la sortie de l'analyseur est l'arbre de requête :

```
SELECT chaussure_prete.nom_chaussure, chaussure_prete.dispo,
       chaussure_prete.nom_lacet, chaussure_prete.dispo_lacet,
       chaussure_prete.total_avail
  FROM chaussure_prete chaussure_prete
 WHERE chaussure_prete.total_avail >= 2;
```

La première règle appliquée sera celle de la vue `chaussure_prete` et cela résultera en cet arbre de requête :

```
SELECT chaussure_prete.nom_chaussure, chaussure_prete.dispo,
       chaussure_prete.nom_lacet, chaussure_prete.dispo_lacet,
       chaussure_prete.total_avail
  FROM (SELECT rsh.nom_chaussure,
               rsh.dispo,
               rsl.nom_lacet,
               rsl.dispo_lacet,
               min(rsh.dispo, rsl.dispo_lacet) AS total_avail
          FROM chaussure rsh, lacet rsl
         WHERE rsl.couleur_lacet = rsh.couleur
          AND rsl.longueur_lacet_cm >= rsh.long_min_chaussure_cm
          AND rsl.longueur_lacet_cm <= rsh.long_max_chaussure_cm) chaussure_prete
 WHERE chaussure_prete.total_avail >= 2;
```

De façon similaire, les règles pour `chaussure` et `lacet` sont substituées dans la table d'échelle de la sous-requête, amenant à l'arbre de requête final à trois niveaux :

```
SELECT chaussure_prete.nom_chaussure, chaussure_prete.dispo,
       chaussure_prete.nom_lacet, chaussure_prete.dispo_lacet,
       chaussure_prete.total_avail
  FROM (SELECT rsh.nom_chaussure,
               rsh.dispo,
               rsl.nom_lacet,
               rsl.dispo_lacet,
               min(rsh.dispo, rsl.dispo_lacet) AS total_avail
          FROM (SELECT sh.nom_chaussure,
                       sh.dispo,
                       sh.couleur,
                       sh.long_min_chaussure,
                       sh.long_min_chaussure * un.facteur_unite AS
long_min_chaussure_cm,
                       sh.long_max_chaussure,
                       sh.long_max_chaussure * un.facteur_unite AS
long_max_chaussure_cm,
                       sh.unite_long_chaussure
                  FROM donnees_chaussure sh, unit un
                 WHERE sh.unite_long_chaussure = un.nom_unite) rsh,
               (SELECT s.nom_lacet,
                       s.dispo_lacet,
                       s.couleur_lacet,
                       s.longueur_lacet,
                       s.unite_lacet,
                       s.longueur_lacet * u.facteur_unite AS longueur_lacet_cm
                  FROM donnees_lacet s, unit u
                 WHERE s.unite_lacet = u.nom_unite) rsl
         WHERE rsl.couleur_lacet = rsh.couleur
           AND rsl.longueur_lacet_cm >= rsh.long_min_chaussure_cm
           AND rsl.longueur_lacet_cm <= rsh.long_max_chaussure_cm) chaussure_prete
 WHERE chaussure_prete.total_avail > 2;
```

Il s'avère que le planificateur réduira cet arbre en un arbre de requête à deux niveaux : les commandes **select** du bas seront « remontées » dans le **select** du milieu car il n'est pas nécessaire de les traiter séparément. Mais le **select** du milieu restera séparé du haut car il contient des fonctions d'agrégat. Si nous les avions monté, cela aurait modifié le comportement du **select** de haut niveau, ce qui n'est pas ce que nous voulons. Néanmoins, réduire l'arbre de requête est une optimisation qui ne concerne pas le système de réécriture.

37.2.2. Règles de vue dans des instructions autres que select

Deux détails de l'arbre de requête n'ont pas été abordés dans la description des règles de vue ci-dessus. Ce sont le type de commande et le relation résultante. En fait, les règles de vue n'ont pas besoin de cette information.

Il existe seulement quelques différences entre un arbre de requête pour un **select** et un pour une autre commande. de façon évidente, ils ont un type de commande différent et pour une commande autre qu' un **select**, la relation résultante pointe vers l'entrée de table d'échelle où le résultat devrait arriver. Tout le reste est absolument identique. Donc, avec deux tables `t1` et `t2` avec les colonnes `a` et `b`, les arbres de requêtes pour les deux commandes :

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

sont pratiquement identiques. En particulier :

- Les tables d'échelle contiennent des entrées pour les tables `t1` et `t2`.
- Les listes cibles contiennent une variable pointant vers la colonne `b` de l'entrée de la table d'échelle pour la table `t2`.
- Les expressions de qualification comparent les colonnes `a` des deux entrées de table d'échelle pour une égalité.
- Les arbres de jointure affichent une jointure simple entre `t1` et `t2`.

La conséquence est que les deux arbres de requête résultent en des plans d'exécution similaires : ce sont tous les deux des jointures sur les deux tables. Pour l'**update**, les colonnes manquantes de `t1` sont ajoutées à la liste cible par le planificateur et l'arbre de requête final sera lu de cette façon :

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

et, du coup, l'exécuteur lancé sur la jointure produira exactement le même résultat qu'un :

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

le ferait. Mais il y a un petit problème avec **update** : l'exécuteur ne fait pas attention au but du résultat de la jointure. Il produit simplement un ensemble de lignes composant le résultat. La différence entre une commande **select** et une commande **update** est gérée par celui qui a appelé l'exécuteur. L'appelant sait toujours (en regardant dans l'arbre de requêtes) qu'il s'agit d'un **update**, et il sait que le résultat ira dans la table `t1`. mais quelles lignes disponibles maintenant seront remplacées par les nouvelles lignes ?

Pour résoudre ce problème, une autre entrée est ajoutée dans la liste cible de l'**update** (et aussi dans les instructions **delete**) : l'identifiant actuel du tuple (ctid, acronyme de *current tuple ID*). cette colonne système contient le numéro de bloc du fichier et la position dans le bloc pour cette ligne. Connaissant la table, le ctid peut être utilisé pour récupérer la ligne originale de `t1` à mettre à jour. après avoir ajouté le ctid dans la liste cible, la requête ressemble à ceci :

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Maintenant, un autre détail de PostgreSQL™ entre en jeu. Les anciennes lignes de la table ne sont pas surchargées et cela explique pourquoi **rollback** est rapide. avec un **update**, la nouvelle ligne résultat est insérée dans la table (après avoir enlevé le ctid) et, dans le nouvel en-tête de ligne de l'ancienne ligne, vers où pointe le ctid, les entrées `cmax` et `xmax` sont configurées par le compteur de commande actuel et par l'identifiant de transaction actuel. Du coup, l'ancienne ligne est cachée et, après validation de la transaction, le nettoyeur (vacuum) peut réellement la supprimer.

Connaissant tout ceci, nous pouvons simplement appliquer les règles de vues de la même façon que toute autre commande. Il n'y a pas de différence.

37.2.3. Puissance des vues dans PostgreSQL™

L'exemple ci-dessus démontre l'incorporation des définitions de vues par le système de règles dans l'arbre de requête original. Dans le deuxième exemple, un simple **select** d'une vue a créé un arbre de requête final qui est une jointure de quatre tables (`unit` a été utilisé deux fois avec des noms différents).

Le bénéfice de l'implémentation des vues avec le système de règles est que le planificateur a toute l'information sur les tables à parcourir et sur les relations entre ces tables et les qualifications restrictives à partir des vues et les qualifications à partir de la requête originale dans un seule arbre de requête. Et c'est toujours la situation quand la requête originale est déjà une jointure sur des vues. Le planificateur doit décider du meilleur chemin pour exécuter la requête et plus le planificateur a d'informations, meilleure sera la décision. Le système de règles implémenté dans PostgreSQL™ s'en assure, c'est toute l'information disponible sur la requête à ce moment.

37.2.4. Mise à jour d'une vue

Qu'arrive-t'il si une vue est nommée comme la relation cible d'un **insert**, **update** ou **delete** ? après avoir réalisées les substitutions décrites ci-dessus, nous aurons un arbre de requête dans lequel la relation résultante pointe vers une entrée de la table d'échelle de la sous-requête. Ceci ne fonctionnera pas car la réécriture renvoie une erreur si elle aperçoit qu'elle a produit une telle chose.

Pour modifier ceci, nous pouvons définir des règles modifiant le comportement de ce type de commandes. C'est donc le thème de la prochaine section.

37.3. Règles sur insert, update et delete

Les règles définies sur **insert**, **update** et **delete** sont significativement différentes des règles de vue décrites dans la section précédente. Tout d'abord, leur commande **create rule** permet plus de choses :

- Elles peuvent n'avoir aucune action.
- Elles peuvent avoir plusieurs actions.
- Elles peuvent être de type `instead` ou `also` (valeur par défaut).
- Les pseudo relations `new` et `old` deviennent utiles.
- Elles peuvent avoir des qualifications de règles.

Ensuite, elles ne modifient pas l'arbre de requête en place. À la place, elles créent de nouveaux arbres de requêtes et peuvent abandonner l'original.

37.3.1. Fonctionnement des règles de mise à jour

Gardez en tête la syntaxe :

```
CREATE [ OR REPLACE ] RULE nom as on evenement
    TO table [ where condition ]
```

```
    DO [ ALSO | INSTEAD ] { NOTHING | commande | ( commande ; commande ... ) }
```

Dans la suite, *règles de mise à jour* signifie les règles qui sont définies sur **insert**, **update** ou **delete**.

Les règles de mise à jour sont appliquées par le système de règles lorsque la relation résultante et le type de commande d'un arbre de requête sont égaux pour l'objet et l'événement donné dans la commande **create RULE**. pour les règles de mise à jour, le système de règles crée une liste d'arbres de requêtes. Initialement, la liste d'arbres de requêtes est vide. Il peut y avoir aucune (mot clé `nothing`), une ou plusieurs actions. Pour simplifier, nous verrons une règle avec une action. Cette règle peut avoir une qualification et peut être de type `instead` ou `also` (valeur par défaut).

Qu'est-ce qu'une qualification de règle ? C'est une restriction indiquant le moment où doivent être réalisés les actions de la règle. Cette qualification peut seulement référencer les pseudo relations `new` et/ou `old`, qui représentent basiquement la relation qui a été donné comme objet (mais avec une signification spéciale).

Donc, nous avons trois cas qui produisent les arbres de requêtes suivants pour une règle à une seule action.

sans qualification avec soit `ALSO` soit `INSTEAD`
: l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de l'arbre de requête original

qualification donnée et `also`
: l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de la règle et de la qualification de l'arbre de requête original

qualification donnée avec `instead`
: l'arbre de requête à partir de l'action de la règle avec la qualification de la requête et la qualification de l'arbre de requête original ; et l'ajout de l'arbre de requête original avec la qualification inverse de la règle

Enfin, si la règle est `also`, l'arbre de requête original est ajouté à la liste. Comme seules les règles qualifiées `instead` ont déjà ajouté l'arbre de requête original, nous finissons avec un ou deux arbres de requête en sortie pour une règle avec une action.

Pour les règles `on insert`, la requête originale (si elle n'est pas supprimée par `instead`) est réalisée avant toute action ajoutée par les règles. Ceci permet aux actions de voir les lignes insérées. Mais pour les règles `on update` et `on delete`, la requête originale est réalisée après les actions ajoutées par les règles. Ceci nous assure que les actions pourront voir les lignes à mettre à jour ou à supprimer ; sinon, les actions pourraient ne rien faire parce qu'elles ne trouvent aucune ligne correspondant à leurs qualifications.

Les arbres de requêtes générés à partir des actions de règles sont envoyés de nouveau dans le système de réécriture et peut-être que d'autres règles seront appliquées résultant en plus ou moins d'arbres de requêtes. Donc, les actions d'une règle doivent avoir soit un type de commande différent soit une relation résultante différente de celle où la règle elle-même est active, sinon ce processus récursif se terminera dans une boucle infinie. (L'expansion récursive d'une règle sera détectée et rapportée comme une erreur.)

Les arbres de requête trouvés dans les actions du catalogue système pg_rewrite sont seulement des modèles. comme ils peuvent référencer les entrées de la table d'échelle pour `new` et `old`, quelques substitutions ont dû être faites avant qu'elles ne puissent être utilisées. Pour toute référence de `new`, une entrée correspondante est recherchée dans la liste cible de la requête originale. Si elle est trouvée, cette expression de l'entrée remplace la référence. Sinon, `new` signifie la même chose que `old` (pour un **update**) ou est remplacé par une valeur null (pour un **insert**). toute référence à `old` est remplacée par une référence à l'entrée de la table d'échelle qui est la relation résultante.

Après que le système a terminé d'appliquer des règles de mise à jour, il applique les règles de vues pour le(s) arbre(s) de requête produit(s). Les vues ne peuvent pas insérer de nouvelles actions de mise à jour, donc il n'est pas nécessaire d'appliquer les règles de mise à jour à la sortie d'une réécriture de vue.

37.3.1.1. Une première requête étape par étape

Disons que nous voulons tracer les modifications dans la colonne `dispo_lacet` de la relation `donnees_lacet`. donc, nous allons configurer une table de traces et une règle qui va écrire une entrée lorsqu'un **update** est lancé sur `donnees_lacet`.

```
CREATE TABLE lacet_log (
    nom_lacet    text,          -- modification de lacet
    dispo_lacet   integer,       -- nouvelle valeur disponible
    log_who    text,          -- qui l'a modifié
    log_when   timestamp      -- quand
);

CREATE RULE log_lacet AS ON UPDATE TO donnees_lacet
    WHERE NEW.dispo_lacet <> OLD.dispo_lacet
    DO INSERT INTO lacet_log VALUES (
                                     NEW.nom_lacet,
                                     NEW.dispo_lacet,
                                     current_user,
```

```
                                        current_timestamp
                                    );
```

Maintenant, quelqu'un exécute :

```
UPDATE donnees_lacet SET dispo_lacet = 6 WHERE nom_lacet = 'sl7';
```

et voici le contenu de la table des traces :

```
SELECT * FROM lacet_log;

 nom_lacet | dispo_lacet | log_who | log_when
-----------+-------------+---------+----------------------------------
 sl7       |           6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

C'est ce à quoi nous nous attendions. Voici ce qui s'est passé en tâche de fond. L'analyseur a créé l'arbre de requête :

```
UPDATE donnees_lacet SET dispo_lacet = 6
  FROM donnees_lacet donnees_lacet
 WHERE donnees_lacet.nom_lacet = 'sl7';
```

Il existe une règle `log_lacet` qui est `on UPDATE` avec l'expression de qualification de la règle :

```
NEW.dispo_lacet <> OLD.dispo_lacet
```

et l'action :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old;
```

(ceci semble un peu étrange car, normalement, vous ne pouvez pas écrire `insert ... values ... from`. ici, la clause `from` indique seulement qu'il existe des entrées de la table d'échelle dans l'arbre de requête pour `new` et `old`. elles sont nécessaires pour qu'elles puissent être référencées par des variables dans l'arbre de requête de la commande **insert**).

La règle est une règle qualifiée `also` de façon à ce que le système de règles doit renvoyer deux arbres de requêtes : l'action de la règle modifiée et l'arbre de requête original. Dans la première étape, la table d'échelle de la requête originale est incorporée dans l'arbre de requête d'action de la règle. Ceci a pour résultat :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet;
```

Pour la deuxième étape, la qualification de la règle lui est ajoutée, donc l'ensemble de résultat est restreint aux lignes où `dispo_lacet` a changé :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 where new.dispo_lacet <> old.dispo_lacet;
```

(Ceci semble encore plus étrange car `insert ... values` n'a pas non plus une clause `where` mais le planificateur et l'exécuteur n'auront pas de difficultés avec ça. Ils ont besoin de supporter cette même fonctionnalité pour `insert ... select`.)

À l'étape 3, la qualification de l'arbre de requête original est ajoutée, restreignant encore plus l'ensemble de résultats pour les seules lignes qui auront été modifiées par la requête originale :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 WHERE new.dispo_lacet <> old.dispo_lacet
   and donnees_lacet.nom_lacet = 'sl7';
```

La quatrième étape remplace les références à `new` par les entrées de la liste cible à partir de l'arbre de requête original ou par les

références de la variable correspondante à partir de la relation résultat :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, 6,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 WHERE 6 <> old.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';
```

L'étape 5 modifie les références `old` en référence de la relation résultat :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, 6,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 WHERE 6 <> donnees_lacet.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';
```

C'est tout. Comme la règle est de type `also`, nous affichons aussi l'arbre de requêtes original. En bref, l'affichage à partir du système de règles est une liste de deux arbres de requêtes est une liste de deux arbres de requêtes correspondant à ces instructions :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, 6,
       current_user, current_timestamp )
  FROM donnees_lacet
 WHERE 6 <> donnees_lacet.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';

UPDATE donnees_lacet SET dispo_lacet = 6
 WHERE nom_lacet = 'sl7';
```

Elles sont exécutées dans cet ordre et c'est exactement le but de la règle.

Les substitutions et les qualifications ajoutées nous assurent que, si la requête originale était :

```
UPDATE donnees_lacet SET couleur_lacet = 'green'
 WHERE nom_lacet = 'sl7';
```

aucune trace ne serait écrite. Dans ce cas, l'arbre de requête original ne contient pas une entrée dans la liste cible pour `dispo_lacet`, donc `new.dispo_lacet` sera remplacé par `donnees_lacet.dispo_lacet`. Du coup, la commande supplémentaire générée par la règle est :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, donnees_lacet.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet
 WHERE donnees_lacet.dispo_lacet <> donnees_lacet.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';
```

et la qualification ne sera jamais vraie.

Si la requête originale modifie plusieurs lignes, cela fonctionne aussi. Donc, si quelqu'un a lancé la commande :

```
UPDATE donnees_lacet SET dispo_lacet = 0
 WHERE couleur_lacet = 'black';
```

en fait, quatre lignes sont modifiées (`sl1`, `sl2`, `sl3` et `sl4`). mais `sl3` a déjà `dispo_lacet = 0`. dans ce cas, la qualification des arbres de requêtes originaux sont différents et cela produit un arbre de requête supplémentaire :

```
INSERT INTO lacet_log
SELECT donnees_lacet.nom_lacet, 0,
       current_user, current_timestamp
  FROM donnees_lacet
 WHERE 0 <> donnees_lacet.dispo_lacet
   AND donnees_lacet.couleur_lacet = 'black';
```

à générer par la règle. Cet arbre de requête aura sûrement inséré trois nouvelles lignes de traces. Et c'est tout à fait correct.

Ici, nous avons vu pourquoi il est important que l'arbre de requête original soit exécuté en premier. Si l'**update** a été exécuté avant, toutes les lignes pourraient aussi être initialisées à zéro, donc le **insert** tracé ne trouvera aucune ligne à `0 <> donnees_lacet.dispo_lacet`.

37.3.2. Coopération avec les vues

Une façon simple de protéger les vues d'une exécution d'**insert**, d'**update** ou de **delete** sur elles est de laisser s'abandonner ces arbres de requête. Donc, nous pourrions créer les règles :

```
CREATE RULE chaussure_ins_protect AS ON INSERT TO chaussure
    DO INSTEAD NOTHING;
CREATE RULE chaussure_upd_protect AS ON UPDATE TO chaussure
    DO INSTEAD NOTHING;
CREATE RULE chaussure_del_protect AS ON DELETE TO chaussure
    DO INSTEAD NOTHING;
```

Maintenant, si quelqu'un essaie de faire une de ces opérations sur la vue `chaussure`, le système de règles appliquera ces règles. Comme les règles n'ont pas d'action et sont de type `instead`, la liste résultante des arbres de requêtes sera vide et la requête entière deviendra vide car il ne reste rien à optimiser ou exécuter après que le système de règles en ait terminé avec elle.

Une façon plus sophistiquée d'utiliser le système de règles est de créer les règles qui réécrivent l'arbre de requête en un arbre faisant la bonne opération sur les vraies tables. Pour réaliser cela sur la vue `lacet`, nous créons les règles suivantes :

```
CREATE RULE lacet_ins AS ON INSERT TO lacet
    DO INSTEAD
    INSERT INTO donnees_lacet VALUES (
           NEW.nom_lacet,
           NEW.dispo_lacet,
           NEW.couleur_lacet,
           NEW.longueur_lacet,
           NEW.unite_lacet
    );

CREATE RULE lacet_upd AS ON UPDATE TO lacet
    DO INSTEAD
    UPDATE donnees_lacet
       SET nom_lacet = NEW.nom_lacet,
           dispo_lacet = NEW.dispo_lacet,
           couleur_lacet = NEW.couleur_lacet,
           longueur_lacet = NEW.longueur_lacet,
           unite_lacet = NEW.unite_lacet
     WHERE nom_lacet = OLD.nom_lacet;

CREATE RULE lacet_del AS ON DELETE TO lacet
    DO INSTEAD
    DELETE FROM donnees_lacet
     WHERE nom_lacet = OLD.nom_lacet;
```

Si vous voulez supporter les requêtes `RETURNING` sur la vue, vous devrez faire en sorte que les règles incluent les clauses `RETURNING` qui calcule les lignes de la vue. Ceci est assez simple pour des vues sur une seule table mais cela devient rapidement complexe pour des vues de jointure comme `lacet`. Voici un exemple pour le cas d'un INSERT :

```
CREATE RULE lacet_ins AS ON INSERT TO lacet
    DO INSTEAD
    INSERT INTO donnees_lacet VALUES (
           NEW.nom_lacet,
           NEW.dispo_lacet,
           NEW.couleur_lacet,
           NEW.longueur_lacet,
           NEW.unite_lacet
    )
    RETURNING
           donnees_lacet.*,
           (SELECT donnees_lacet.longueur_lacet * u.facteur_unite
            FROM unite u WHERE donnees_lacet.unite_lacet = u.nom_unite);
```

Notez que cette seule règle supporte à la fois les **INSERT** et les **INSERT RETURNING** sur la vue -- la clause `RETURNING` est tout simplement ignoré pour un **INSERT**.

Maintenant, supposons que, quelque fois, un paquet de lacets arrive au magasin avec une grosse liste. Mais vous ne voulez pas mettre à jour manuellement la vue `lacet` à chaque fois. à la place, nous configurons deux petites tables, une où vous pouvez insérer les éléments de la liste et une avec une astuce spéciale. Voici les commandes de création :

```
CREATE TABLE lacet_arrive (
    arr_name    text,
```

```
    arr_quant   integer
);

CREATE TABLE lacet_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE lacet_ok_ins AS ON INSERT TO lacet_ok
    DO INSTEAD
    UPDATE lacet
       SET dispo_lacet = dispo_lacet + NEW.ok_quant
     WHERE nom_lacet = NEW.ok_name;
```

Maintenant, vous pouvez remplir la table `lacet_arrive` avec les données de la liste :

```
SELECT * FROM lacet_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Jetez un œil rapidement aux données actuelles :

```
SELECT * FROM lacet;

 nom_lacet  | dispo_lacet | couleur_lacet | longueur_lacet | unite_lacet |
longueur_lacet_cm
------------+-------------+---------------+----------------+-------------+------------------
 sl1        |           5 | black         |             80 | cm          |
80
 sl2        |           6 | black         |            100 | cm          |
100
 sl7        |           6 | brown         |             60 | cm          |
60
 sl3        |           0 | black         |             35 | inch        |
88.9
 sl4        |           8 | black         |             40 | inch        |
101.6
 sl8        |           1 | brown         |             40 | inch        |
101.6
 sl5        |           4 | brown         |              1 | m           |
100
 sl6        |           0 | brown         |            0.9 | m           |
90
(8 rows)
```

Maintenant, déplacez les lacets arrivés dans :

```
INSERT INTO lacet_ok SELECT * FROM lacet_arrive;
```

et vérifiez le résultat :

```
SELECT * FROM lacet ORDER BY nom_lacet;

 nom_lacet  | dispo_lacet | couleur_lacet | longueur_lacet | unite_lacet |
longueur_lacet_cm
------------+-------------+---------------+----------------+-------------+------------------
 sl1        |           5 | black         |             80 | cm          |
80
 sl2        |           6 | black         |            100 | cm          |
100
 sl7        |           6 | brown         |             60 | cm          |
60
 sl4        |           8 | black         |             40 | inch        |
101.6
 sl3        |          10 | black         |             35 | inch        |
88.9
 sl8        |          21 | brown         |             40 | inch        |
```

```
101.6
 sl5        |           4 | brown         |              1 | m           |
100
 sl6        |          20 | brown         |            0.9 | m           |
90
(8 rows)

SELECT * FROM lacet_log;

 nom_lacet | dispo_lacet | log_who| log_when
-----------+-------------+--------+----------------------------------
 sl7       |           6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3       |          10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6       |          20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8       |          21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

C'est un long chemin du `insert ... select` à ces résultats. Et la description de la transformation de l'arbre de requêtes sera la dernière dans ce chapitre. Tout d'abord, voici la sortie de l'analyseur :

```
INSERT INTO lacet_ok
SELECT lacet_arrive.arr_name, lacet_arrive.arr_quant
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok;
```

Maintenant, la première règle `lacet_ok_ins` est appliquée et transforme ceci en :

```
UPDATE lacet
   SET dispo_lacet = lacet.dispo_lacet + lacet_arrive.arr_quant
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet
 WHERE lacet.nom_lacet = lacet_arrive.arr_name;
```

et jette l'**insert** actuel sur `lacet_ok`. la requête réécrite est passée de nouveau au système de règles et la seconde règle appliquée `lacet_upd` produit :

```
UPDATE donnees_lacet
   SET nom_lacet = lacet.nom_lacet,
       dispo_lacet = lacet.dispo_lacet + lacet_arrive.arr_quant,
       couleur_lacet = lacet.couleur_lacet,
       longueur_lacet = lacet.longueur_lacet,
       unite_lacet = lacet.unite_lacet
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet, lacet old,
       lacet new, donnees_lacet donnees_lacet
 WHERE lacet.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = lacet.nom_lacet;
```

De nouveau, il s'agit d'une règle `instead` et l'arbre de requête précédent est jeté. Notez que cette requête utilise toujours la vue `lacet`. mais le système de règles n'a pas fini cette étape, donc il continue et lui applique la règle `_return`. Nous obtenons :

```
UPDATE donnees_lacet
   SET nom_lacet = s.nom_lacet,
       dispo_lacet = s.dispo_lacet + lacet_arrive.arr_quant,
       couleur_lacet = s.couleur_lacet,
       longueur_lacet = s.longueur_lacet,
       unite_lacet = s.unite_lacet
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet, lacet old,
       lacet new, donnees_lacet donnees_lacet,
       lacet old, lacet new,
       donnees_lacet s, unit u
 WHERE s.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = s.nom_lacet;
```

Enfin, la règle `log_lacet` est appliquée, produisant l'arbre de requête supplémentaire :

```
INSERT INTO lacet_log
SELECT s.nom_lacet,
       s.dispo_lacet + lacet_arrive.arr_quant,
```

```
       current_user,
       current_timestamp
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet, lacet old,
       lacet new, donnees_lacet donnees_lacet,
       lacet old, lacet new,
       donnees_lacet s, unit u,
       donnees_lacet old, donnees_lacet new
       lacet_log lacet_log
 WHERE s.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = s.nom_lacet
   AND (s.dispo_lacet + lacet_arrive.arr_quant) <> s.dispo_lacet;
```

une fois que le système de règles tombe en panne de règles et renvoie les arbres de requêtes générés.

Donc, nous finissons avec deux arbres de requêtes finaux qui sont équivalents aux instructions SQL :

```
INSERT INTO lacet_log
SELECT s.nom_lacet,
       s.dispo_lacet + lacet_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM lacet_arrive lacet_arrive, donnees_lacet donnees_lacet,
       donnees_lacet s
 WHERE s.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = s.nom_lacet
   AND s.dispo_lacet + lacet_arrive.arr_quant <> s.dispo_lacet;

UPDATE donnees_lacet
   SET dispo_lacet = donnees_lacet.dispo_lacet + lacet_arrive.arr_quant
  FROM lacet_arrive lacet_arrive,
       donnees_lacet donnees_lacet,
       donnees_lacet s
 WHERE s.nom_lacet = lacet_arrive.nom_lacet
   AND donnees_lacet.nom_lacet = s.nom_lacet;
```

Le résultat est que la donnée provenant d'une relation insérée dans une autre, modifiée en mise à jour dans une troisième, modifiée en mise à jour dans une quatrième, cette dernière étant tracée dans une cinquième, se voit réduite à deux requêtes.

Il y a un petit détail assez horrible. En regardant les deux requêtes, nous nous apercevons que la relation `donnees_lacet` apparaît deux fois dans la table d'échelle où cela pourrait être réduit à une seule occurrence. Le planificateur ne gère pas ceci et, du coup, le plan d'exécution de la sortie du système de règles pour **insert** sera :

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on lacet_arrive
  ->  Seq Scan on donnees_lacet
```

alors qu'omettre la table d'échelle supplémentaire résulterait en un :

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on lacet_arrive
```

qui produit exactement les mêmes entrées dans la table des traces. Du coup, le système de règles a causé un parcours supplémentaire dans la table `donnees_lacet` qui n'est absolument pas nécessaire. et le même parcours redondant est fait une fois de plus dans l'**update**. mais ce fut réellement un travail difficile de rendre tout ceci possible.

Maintenant, nous faisons une démonstration finale du système de règles de PostgreSQL™ et de sa puissance. disons que nous ajoutons quelques lacets avec des couleurs extraordinaires à votre base de données :

```
INSERT INTO lacet VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
```

```
INSERT INTO lacet VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Nous voulons créer une vue vérifiant les entrées `lacet` qui ne correspondent à aucune chaussure pour la couleur. Voici la vue :

```
CREATE VIEW lacet_mismatch AS
    SELECT * FROM lacet WHERE NOT EXISTS
        (SELECT nom_chaussure FROM chaussure WHERE couleur = couleur_lacet);
```

Sa sortie est :

```
SELECT * FROM lacet_mismatch;

 nom_lacet | dispo_lacet | couleur_lacet | longueur_lacet | unite_lacet |
longueur_lacet_cm
-----------+-------------+---------------+----------------+-------------+-------------------
 sl9       |           0 | pink          |             35 | inch        |
88.9
 sl10      |        1000 | magenta       |             40 | inch        |
101.6
```

Maintenant, nous voulons la configurer pour que les lacets qui ne correspondent pas et qui ne sont pas en stock soient supprimés de la base de données. Pour rendre la chose plus difficile à PostgreSQL™, nous ne les supprimons pas directement. À la place, nous créons une vue supplémentaire :

```
CREATE VIEW lacet_can_delete AS
    SELECT * FROM lacet_mismatch WHERE dispo_lacet = 0;
```

et le faisons de cette façon :

```
DELETE FROM lacet WHERE EXISTS
    (SELECT * FROM lacet_can_delete
             WHERE nom_lacet = lacet.nom_lacet);
```

voilà :

```
SELECT * FROM lacet;

 nom_lacet | dispo_lacet | couleur_lacet | longueur_lacet | unite_lacet |
longueur_lacet_cm
-----------+-------------+---------------+----------------+-------------+-------------------
 sl1       |           5 | black         |             80 | cm          |
80
 sl2       |           6 | black         |            100 | cm          |
100
 sl7       |           6 | brown         |             60 | cm          |
60
 sl4       |           8 | black         |             40 | inch        |
101.6
 sl3       |          10 | black         |             35 | inch        |
88.9
 sl8       |          21 | brown         |             40 | inch        |
101.6
 sl10      |        1000 | magenta       |             40 | inch        |
101.6
 sl5       |           4 | brown         |              1 | m           |
100
 sl6       |          20 | brown         |            0.9 | m           |
90
(9 rows)
```

Un **delete** sur une vue, avec une qualification de sous-requête qui utilise au total quatre vues imbriquées/jointes, où l'une d'entre elles a une qualification de sous-requête contenant une vue et où les colonnes des vues calculées sont utilisées, est réécrite en un seul arbre de requête qui supprime les données demandées sur la vraie table.

Il existe probablement seulement quelques situations dans le vrai monde où une telle construction est nécessaire. Mais, vous vous sentez mieux quand cela fonctionne.

37.4. Règles et droits

À cause de la réécriture des requêtes par le système de règles de PostgreSQL™, d'autres tables/vues que celles utilisées dans la requête originale pourraient être accédées. Lorsque des règles de mise à jour sont utilisées, ceci peut inclure des droits d'écriture sur

les tables.

Les règles de réécriture n'ont pas de propriétaire séparé. Le propriétaire d'une relation (table ou vue) est automatiquement le propriétaire des règles de réécriture qui lui sont définies. Le système de règles de PostgreSQL™ modifie le comportement du système de contrôle d'accès par défaut. Les relations qui sont utilisées à cause des règles se voient vérifier avec les droits du propriétaire de la règle, et non avec ceux de l'utilisateur appelant cette règle. Ceci signifie qu'un utilisateur a seulement besoin des droits requis pour les tables/vues qu'il nomme explicitement dans ses requêtes.

Par exemple : un utilisateur a une liste de numéros de téléphone dont certains sont privés, les autres étant d'intérêt pour la secrétaire du bureau. Il peut construire de cette façon :

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

Personne sauf lui (et les superutilisateurs de la base de données) ne peut accéder à la table `phone_data`. mais, à cause du **grant**, la secrétaire peut lancer un **select** sur la vue `phone_number`. le système de règles réécrira le **select** sur `phone_number` en un **select** sur `phone_data`. Comme l'utilisateur est le propriétaire de `phone_number` et du coup le propriétaire de la règle, le droit de lecture de `phone_data` est maintenant vérifié avec ses propres privilèges et la requête est autorisée. La vérification de l'accès à `phone_number` est aussi réalisée mais ceci est fait avec l'utilisateur appelant, donc personne sauf l'utilisateur et la secrétaire ne peut l'utiliser.

Les droits sont vérifiés règle par règle. Donc, la secrétaire est actuellement la seule à pouvoir voir les numéros de téléphone publiques. Mais la secrétaire peut configurer une autre vue et autoriser l'accès au public. Du coup, tout le monde peut voir les données de `phone_number` via la vue de la secrétaire. ce que la secrétaire ne peut pas faire est de créer une vue qui accède directement à `phone_data` (en fait, elle le peut mais cela ne fonctionnera pas car tous les accès seront refusés lors de la vérification des droits). Dès que l'utilisateur s'en rendra compte, du fait que la secrétaire a ouvert la vue `phone_number` à tout le monde, il peut révoquer son accès. Immédiatement, tous les accès de la vue de la secrétaire échoueront.

Il pourrait être dit que cette vérification règle par règle est une brèche de sécurité mais ce n'est pas le cas. Si cela ne fonctionne pas de cette façon, la secrétaire pourrait copier une table avec les mêmes colonnes que `phone_number` et y copier les données une fois par jour. du coup, ce sont ces propres données et elle peut accorder l'accès à tout le monde si elle le souhaite. Une commande **grant** signifie « j'ai confiance en vous ». si quelqu'un en qui vous avez confiance se comporte ainsi, il est temps d'y réfléchir et d'utiliser **revoke**.

Notez que, bien que les vues puissent être utilisées pour cacher le contenu de certaines colonnes en utilisant la technique montrée ci-dessus, elles ne peuvent pas être utilisées de manière fiable pour cacher des données dans des lignes invisibles. Par exemple, la vue suivante n'est pas sécurisée :

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Cette vue peut sembler sécurisée car le système de règles va réécrire tout **SELECT** à partir de `phone_number` dans un **SELECT** à partir de `phone_data` et ajouter la qualification permettant de filter les enregistrements dont la colonne `phone` ne commence pas par 412. Mais si l'utilisateur peut créer ses propres fonctions, il n'est pas difficile de convaincre le planificateur d'exécuter la fonction définie par l'utilisateur avant l'expression `NOT LIKE`.

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;
SELECT * FROM phone_number WHERE tricky(person, phone);
```

Chaque personne et chaque numéro de téléphone de la table `phone_data` sera affiché dans un `NOTICE` car le planificateur choisira d'exécuter la procédure `tricky` avant le `NOT LIKE` car elle est moins coûteuse. Même si l'utilisateur ne peut pas définir des nouvelles fonctions, les fonctions internes peuvent être utilisées pour des attaques similaires. (Par exemple, les fonctions de conversions affichent leur entrée dans le message d'erreur qu'elles fournissent.)

Des considérations similaires s'appliquent aussi aux règles de mise à jour. Dans les exemples de la section précédente, le propriétaire des tables de la base de données d'exemple pourrait accorder les droits `select`, `insert`, `update` et `delete` sur la vue `lacet` à quelqu'un d'autre mais seulement `select` sur `lacet_log`. l'action de la règle pourrait écrire des entrées de trace qui seraient toujours exécutées avec succès et que l'autre utilisateur pourrait voir. Mais il ne peut pas créer d'entrées fausses, pas plus qu'il ne peut manipuler ou supprimer celles qui existent. Dans ce cas, il n'existe pas de possibilité de subvertir les règles en convaincant le planificateur de modifier l'ordre des opérations car la seule règle qui fait référence à `shoelace_log` est un `IN-`

SERT non qualifié. Ceci pourrait ne plus être vrai dans les scénarios complexes.

37.5. Règles et statut de commande

Le serveur PostgreSQL™ renvoie une chaîne de statut de commande, comme `insert 149592 1`, pour chaque commande qu'il reçoit. C'est assez simple lorsqu'il n'y a pas de règles impliquées. Mais qu'arrive-t'il lorsque la requête est réécrite par des règles ?

Les règles affectent le statut de la commande de cette façon :

- S'il n'y a pas de règle `instead` inconditionnelle pour la requête, alors la requête donnée originellement sera exécutée et son statut de commande sera renvoyé comme d'habitude. (Mais notez que s'il y avait des règles `instead` conditionnelles, la négation de leur qualifications sera ajouté à la requête initiale. Ceci pourrait réduire le nombre de lignes qu'il traite et, si c'est le cas, le statut rapporté en sera affecté.)
- S'il y a des règles `instead` inconditionnelles pour la requête, alors la requête originale ne sera pas exécutée du tout. Dans ce cas, le serveur renverra le statut de la commande pour la dernière requête qui a été insérée par une règle `instead` (conditionnelle ou non) et est du même type de commande (**insert**, **update** ou **delete**) que la requête originale. si aucune requête ne rencontrant ces pré-requis n'est ajoutée à une règle, alors le statut de commande renvoyé affiche le type de requête original et annule le compteur de ligne et le champ OID.

(Ce système a été établi pour PostgreSQL™ 7.3. Dans les versions précédentes, le statut de commande pouvait afficher des résultats différents lorsque les règles existaient.)

Le programmeur peut s'assurer que toute règle `instead` désirée est celle qui initialise le statut de commande dans le deuxième cas en lui donnant un nom de règle étant le dernier en ordre alphabétique parmi les règles actives pour qu'elle soit appliquée en dernier.

37.6. Règles contre déclencheurs

Beaucoup de choses pouvant se faire avec des déclencheurs peuvent aussi être implémentées en utilisant le système de règles de PostgreSQL™. un des points qui ne pourra pas être implémenté par les règles en certains types de contraintes, notamment les clés étrangères. Il est possible de placer une règle qualifiée qui réécrit une commande en `nothing` si la valeur d'une colonne n'apparaît pas dans l'autre table. Mais alors les données sont jetées et ce n'est pas une bonne idée. Si des vérifications de valeurs valides sont requises et dans le cas où il y a une erreur invalide, un message d'erreur devrait être généré et cela devra se faire avec un déclencheur.

D'un autre côté, un trigger ne peut pas être créé sur des vues car il n'y a pas de données réelles dans une vue ; néanmoins, des règles sur des INSERT, UPDATE et DELETE peuvent être créés pour des vues.

Pour les éléments qui peuvent être implémentés par les deux, ce qui sera le mieux dépend de l'utilisation de la base de données. Un déclencheur est exécuté une fois pour chaque ligne affectée. Une règle manipule la requête ou en génère une autre. Donc, si un grand nombre de lignes sont affectées pour une instruction, une règle lançant une commande supplémentaire sera certainement plus rapide qu'un déclencheur appelé pour chaque ligne et qui devra exécuter ces opérations autant de fois. Néanmoins, l'approche du déclencheur est conceptuellement plus simple que l'approche de la règle et est plus facile à utiliser pour les novices.

Ici, nous montrons un exemple où le choix d'une règle ou d'un déclencheur joue sur une situation. Voici les deux tables :

```
CREATE TABLE ordinateur (
    nom_hote        text,    -- indexé
    constructeur    text     -- indexé
);

CREATE TABLE logiciel (
    logiciel        text,    -- indexé
    nom_hote        text     -- indexé
);
```

Les deux tables ont plusieurs milliers de lignes et les index sur *`nom_hote`* sont uniques. la règle ou le déclencheur devrait implémenter une contrainte qui supprime les lignes de `logiciel` référençant un ordinateur supprimé. Le déclencheur utiliserait cette commande :

```
DELETE FROM logiciel WHERE nom_hote = $1;
```

Comme le déclencheur est appelé pour chaque ligne individuelle supprimée à partir de `ordinateur`, il peut préparer et sauvegarder le plan pour cette commande et passer la valeur *`nom_hote`* dans le paramètre. La règle devra être réécrite ainsi :

```
CREATE RULE ordinateur_del AS ON DELETE TO ordinateur
    DO DELETE FROM logiciel WHERE nom_hote = OLD.nom_hote;
```

Maintenant, nous apercevons différents types de suppressions. Dans le cas d'un :

```
DELETE FROM ordinateur WHERE nom_hote = 'mypc.local.net';
```

la table `ordinateur` est parcourue par l'index (rapide), et la commande lancée par le déclencheur pourrait aussi utiliser un parcours d'index (aussi rapide). La commande supplémentaire provenant de la règle serait :

```
DELETE FROM logiciel WHERE ordinateur.nom_hote = 'mypc.local.net'
                       AND logiciel.nom_hote = ordinateur.nom_hote;
```

Comme il y a une configuration appropriée des index, le planificateur créera un plan :

```
Nestloop
  ->  Index Scan using comp_hostidx on ordinateur
  ->  Index Scan using soft_hostidx on logiciel
```

Donc, il n'y aurait pas trop de différence de performance entre le déclencheur et l'implémentation de la règle.

Avec la prochaine suppression, nous voulons nous débarrasser des 2000 ordinateurs où *`nom_hote`* commence avec `old`. il existe deux commandes possibles pour ce faire. Voici l'une d'elle :

```
DELETE FROM ordinateur WHERE nom_hote >= 'old'
                       AND nom_hote <  'ole'
```

La commande ajoutée par la règle sera :

```
DELETE FROM logiciel WHERE ordinateur.nom_hote >= 'old'
                       AND ordinateur.nom_hote < 'ole'
                       AND logiciel.nom_hote = ordinateur.nom_hote;
```

avec le plan :

```
Hash Join
  ->  Seq Scan on logiciel
  ->  Hash
    ->  Index Scan using comp_hostidx on ordinateur
```

L'autre commande possible est :

```
DELETE FROM ordinateur WHERE nom_hote ~ '^old';
```

ce qui finira dans le plan d'exécution suivant pour la commande ajoutée par la règle :

```
Nestloop
  ->  Index Scan using comp_hostidx on ordinateur
  ->  Index Scan using soft_hostidx on logiciel
```

Ceci monte que le planificateur ne réalise pas que la qualification pour *`nom_hote`* dans `ordinateur` pourrait aussi être utilisée pour un parcours d'index sur `logiciel` quand il existe plusieurs expressions de qualifications combinées avec `and`, ce qui correspond à ce qu'il fait dans la version expression rationnelle de la commande. Le déclencheur sera appelé une fois pour chacun des 2000 anciens ordinateurs qui doivent être supprimées, et ceci résultera en un parcours d'index sur `ordinateur` et 2000 parcours d'index sur `logiciel`. l'implémentation de la règle le fera en deux commandes qui utilisent les index. Et cela dépend de la taille globale de la table `logiciel`, si la règle sera toujours aussi rapide dans la situation du parcours séquentiel. 2000 exécutions de commandes à partir du déclencheur sur le gestionnaire SPI prend un peu de temps, même si tous les blocs d'index seront rapidement dans le cache.

La dernière commande que nous regardons est :

```
DELETE FROM ordinateur WHERE constructeur = 'bim';
```

De nouveau, ceci pourrait résulter en de nombreuses lignes à supprimer dans `ordinateur`. donc, le déclencheur lancera de nouveau de nombreuses commandes via l'exécuteur. La commande générée par la règle sera :

```
DELETE FROM logiciel WHERE ordinateur.constructeur = 'bim'
                       AND logiciel.nom_hote = ordinateur.nom_hote;
```

Le plan pour cette commande sera encore la boucle imbriquée sur les deux parcours d'index, en utilisant seulement un index différent sur `ordinateur` :

```
Nestloop
  ->  Index Scan using comp_manufidx on ordinateur
  ->  Index Scan using soft_hostidx on logiciel
```

Dans chacun de ces cas, les commandes supplémentaires provenant du système de règles seront plus ou moins indépendantes du nombre de lignes affectées en une commande.

Voici le résumé, les règles seront seulement significativement plus lentes que les déclencheurs si leur actions résultent en des jointures larges et mal qualifiées, une situation où le planificateur échoue.

Chapitre 38. Langages de procédures

PostgreSQL™ permet l'écriture de fonctions et de procédures dans des langages différents du SQL et du C. Ces autres langages sont appelés génériquement des *langages de procédures* (LP, PL en anglais). Le serveur ne possède pas d'interpréteur interne des fonctions écrites dans un langage de procédures. La tâche est donc dévolue à un gestionnaire particulier qui, lui, connait les détails du langage. Le gestionnaire peut prendre en charge le travail de découpage, d'analyse syntaxique, d'exécution, etc., ou simplement servir de « colle » entre PostgreSQL™ et une implémentation existante d'un langage de programmation. Le gestionnaire est lui-même une fonction en langage C compilée dans une bibliothèque partagée et chargée à la demande, comme toute autre fonction C.

Il existe à ce jour quatre langages de procédures dans la distribution standard de PostgreSQL™ : PL/pgSQL (Chapitre 39, PL/pgSQL - Langage de procédures SQL), PL/Tcl (Chapitre 40, PL/Tcl - Langage de procédures Tcl), PL/Perl (Chapitre 41, PL/Perl - Langage de procédures Perl) et PL/Python (Chapitre 42, PL/Python - Langage de procédures Python).

Il existe d'autres langages de procédures qui ne sont pas inclus dans la distribution principale. L'Annexe G, Projets externes propose des pistes pour les trouver. De plus, d'autres langages peuvent être définis par les utilisateurs. Les bases de développement d'un nouveau langage de procédures sont couvertes dans le Chapitre 49, Écrire un gestionnaire de langage procédural.

38.1. Installation des langages de procédures

Un langage de procédures doit être « installé » dans toute base de données amenée à l'utiliser. Les langages de procédures installés dans la base de données `template1` sont automatiquement disponibles dans toutes les bases de données créées par la suite. **CREATE DATABASE** recopie en effet toutes les informations disponibles dans la base `template1`. Il est ainsi possible pour l'administrateur de définir, par base, les langages disponibles et d'en rendre certains disponibles par défaut.

Pour les langages fournis avec la distribution standard, l'installation dans la base courante se fait simplement par l'exécution de la commande **CREATE LANGUAGE** *`langage`*. On peut également utiliser le programme createlang(1) pour installer le langage en ligne de commande. Par exemple, pour installer le langage PL/Perl dans la base de données `template1`, on écrit :

```
createlang plperl template1
```

La procédure manuelle décrite ci-dessous n'est recommandée que pour installer des langages personnalisés que **CREATE LANGUAGE** ne connaît pas.

Procédure 38.1. Installation manuelle de langages de procédures

Un langage de procédures s'installe en cinq étapes effectuées obligatoirement par le superutilisateur des bases de données. Pour les langages connus par **CREATE LANGUAGE**, les deuxième, troisième et quatrième étapes peuvent être omises, parce qu'automatiquement réalisées.

1. La bibliothèque partagée du gestionnaire de langage doit être compilée et installée dans le répertoire de bibliothèques approprié. Cela se déroule comme la construction et l'installation de modules de classiques fonctions C utilisateur ; voir la Section 35.9.6, « Compiler et lier des fonctions chargées dynamiquement ». Il arrive souvent que le gestionnaire du langage dépende d'une bibliothèque externe fournissant le moteur de langage ; dans ce cas, elle doit aussi être installée.

2. Le gestionnaire doit être déclaré par la commande

   ```
   CREATE FUNCTION nom_fonction_gestionnaire()
       RETURNS gestionnaire_langage
       AS 'chemin-vers-objet-partagé'
       LANGUAGE C;
   ```

   Le type de retour spécial gestionnaire_langage indique au système que cette fonction ne renvoie pas un type de données SQL et n'est, de ce fait, pas utilisable directement dans des expressions SQL.

3. En option, le gestionnaire de langages peut fournir une fonction de gestion « en ligne » qui permet l'exécution de blocs de code anonyme (commandes DO(7)) écrits dans ce langage. Si une fonction de gestion en ligne est fourni par le langage, déclarez-le avec une commande comme

   ```
   CREATE FUNCTION nom_fonction_en_ligne(internal)
       RETURNS void
       AS 'chemin-vers-objet-partagé'
       LANGUAGE C;
   ```

4. En option, le gestionnaire de langages peut fournir une fonction de « validation » qui vérifie la définition d'une fonction

sans réellement l'exécuter. La fonction de validation, si elle existe, est appelée par **CREATE FUNCTION**. Si une telle fonction est fournie par le langage, elle sera déclarée avec une commande de la forme

```
CREATE FUNCTION nom_fonction_validation(oid)
    RETURNS void
    AS 'chemin-vers-objet-partagé'
    LANGUAGE C;
```

5. Le LP doit être déclaré par la commande

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE nom_langage
    HANDLER nom_fonction_gestionnaire
    [INLINE nom_fonction_en_ligne]
    [VALIDATOR nom_fonction_valideur] ;
```

Le mot clé optionnel `TRUSTED` (autrement dit, digne de confiance) indique que le langage n'autorise pas l'accès à des données normalement inaccessible à cet utilisateur. Les langages de confiance sont conçus pour les utilisateurs standards de la base de données, c'est-à-dire ceux qui ne sont pas superutilisateurs, et les autorisent à créer en tout sécurité des fonctions et des procédures pour triggers. Les fonctions en langage de procédures étant exécutées au sein du serveur, le paramètre `TRUSTED` ne devrait être positionné que pour les langages n'accédant pas aux organes internes du serveur ou au système de fichiers. Les langages PL/pgSQL, PL/Tcl, et PL/Perl sont considérés comme dignes de confiance ; les langages PL/TclU, PL/PerlU, et PL/PythonU sont conçus pour fournir des fonctionnalités illimitées et *ne* devraient *pas* être marqués dignes de confiance.

L'Exemple 38.1, « Installation manuelle de PL/Perl » présente le fonctionnement de la procédure d'installation manuelle du langage PL/Perl.

Exemple 38.1. Installation manuelle de PL/Perl

La commande suivante indique au serveur l'emplacement de la bibliothèque partagée pour la fonction de gestion des appels du langage PL/Perl.

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perl a une fonction de gestion en ligne et une fonction de validation, donc nous déclarons aussi celles-ci :

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;
```

La commande :

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

indique l'évocation des fonctions précédentes pour les fonctions et procédures de déclencheur lorsque l'attribut de langage est `plperl`.

Lors de l'installation par défaut de PostgreSQL™, le gestionnaire du langage PL/pgSQL est compilé et installé dans le répertoire des bibliothèques (« lib ») ; de plus, le langage PL/pgSQL est installé dans toutes les bases de données. Si le support de Tcl est configuré, les gestionnaires pour PL/Tcl et PL/TclU sont construits et installés dans le répertoire des bibliothèques mais le langage lui-même n'est pas installé par défaut dans les bases de données. De la même façon, les gestionnaires pour PL/Perl et PL/PerlU sont construits et installés si le support de Perl est configuré et le gestionnaire pour PL/PythonU est installé si le support de Python est configuré mais ces langages ne sont pas installés par défaut.

Chapitre 39. PL/pgSQL - Langage de procédures SQL

39.1. Aperçu

PL/pgSQL est un langage de procédures chargeable pour le système de bases de données PostgreSQL™. Les objectifs de la conception de PL/pgSQL ont été de créer un langage de procédures chargeable qui

- est utilisé pour créer des fonctions standards et triggers,
- ajoute des structures de contrôle au langage SQL,
- permet d'effectuer des traitements complexes,
- hérite de tous les types, fonctions et opérateurs définis par les utilisateurs,
- est défini comme digne de confiance par le serveur,
- est facile à utiliser.

Les fonctions PL/pgSQL acceptent un nombre variable d'arguments en utilisant le marqueur `VARIADIC`. Cela fonctionne exactement de la même façon pour les fonctions SQL, comme indiqué dans Section 35.4.5, « Fonctions SQL avec un nombre variables d'arguments ».

Les fonctions écrites en PL/pgSQL peuvent être utilisées partout où une fonction intégrée peut l'être. Par exemple, il est possible de créer des fonctions complexes de traitement conditionnel et, par la suite, de les utiliser pour définir des opérateurs ou de les utiliser dans des expressions d'index.

À partir de la version 9.0 de PostgreSQL™, PL/pgSQL est installé par défaut. Il reste toutefois un module chargeable et les administrateurs craignant pour la sécurité de leur instance pourront le retirer.

39.1.1. Avantages de l'utilisation de PL/pgSQL

SQL est le langage que PostgreSQL™ et la plupart des autres bases de données relationnelles utilisent comme langage de requête. Il est portable et facile à apprendre, mais chaque expression SQL doit être exécutée individuellement par le serveur de bases de données.

Cela signifie que votre application client doit envoyer chaque requête au serveur de bases de données, attendre que celui-ci la traite, recevoir et traiter les résultats, faire quelques calculs, et enfin envoyer d'autres requêtes au serveur. Tout ceci induit des communications interprocessus et induit aussi une surcharge du réseau si votre client est sur une machine différente du serveur de bases de données.

Grâce à PL/pgSQL vous pouvez grouper un bloc de traitement et une série de requêtes *au sein* du serveur de bases de données, et bénéficier ainsi de la puissance d'un langage de procédures, mais avec de gros gains en terme de communication client/serveur.

- Les allers/retours entre le client et le serveur sont éliminés
- Il n'est pas nécessaire de traiter ou transférer entre le client et le serveur les résultats intermédiaires dont le client n'a pas besoin
- Les va-et-vient des analyses de requêtes peuvent être évités

Ceci a pour résultat une augmentation considérable des performances en comparaison à une application qui n'utilise pas les procédures stockées.

Ainsi, avec PL/pgSQL vous pouvez utiliser tous les types de données, opérateurs et fonctions du SQL.

39.1.2. Arguments supportés et types de données résultats

Les fonctions écrites en PL/pgSQL peuvent accepter en argument n'importe quel type de données supporté par le serveur, et peuvent renvoyer un résultat de n'importe lequel de ces types. Elles peuvent aussi accepter ou renvoyer n'importe quel type composite (type ligne) spécifié par nom. Il est aussi possible de déclarer une fonction PL/pgSQL renvoyant un type record, signifiant que le résultat est un type ligne dont les colonnes sont déterminées par spécification dans la requête appelante (voir la Section 7.2.1.4, « Fonctions de table »).

Les fonctions PL/pgSQL acceptent en entrée et en sortie les types polymorphes anyelement, anyarray, anynonarray et anyenum. Le type de données réel géré par une fonction polymorphe peut varier d'appel en appel (voir la Section 35.2.5, « Types et fonctions polymorphes »). Voir l'exemple de la Section 39.3.1, « Déclarer des paramètres de fonctions ».

Les fonctions PL/pgSQL peuvent aussi renvoyer un ensemble de lignes (ou une table) de n'importe lequel des type de données dont les fonctions peuvent renvoyer une instance unique. Ces fonctions génèrent leur sortie en exécutant `RETURN NEXT` pour chaque élément désiré de l'ensemble résultat ou en utilisant **RETURN QUERY** pour afficher le résultat de l'évaluation d'une requête.

Enfin, une fonction PL/pgSQL peut être déclarée comme renvoyant void si elle n'a pas de valeur de retour utile.

Les fonctions PL/pgSQL peuvent aussi être déclarées avec des paramètres en sortie à la place de la spécification explicite du code de retour. Ceci n'ajoute pas de fonctionnalité fondamentale au langage mais c'est un moyen agréable principalement pour renvoyer plusieurs valeurs. La notation `RETURNS TABLE` peut aussi être utilisé à la place de `RETURNS SETOF`.

Des exemples spécifiques apparaissent dans la Section 39.3.1, « Déclarer des paramètres de fonctions » et la Section 39.6.1, « Retour d'une fonction ».

39.2. Structure de PL/pgSQL

PL/pgSQL est un langage structuré en blocs. Le texte complet de la définition d'une fonction doit être un *bloc*. Un bloc est défini comme :

```
[ <<label>> ]
[ DECLARE
    déclarations ]
BEGIN
    instructions
END [ label ];
```

Chaque déclaration et chaque expression au sein du bloc est terminé par un point-virgule. Un bloc qui apparaît à l'intérieur d'un autre bloc doit avoir un point-virgule après `END` (voir l'exemple ci-dessus) ; néanmoins, le `END` final qui conclut le corps d'une fonction n'a pas besoin de point-virgule.

Astuce

Une erreur habituelle est d'écrire un point-virgule immédiatement après `BEGIN`. C'est incorrect et a comme résultat une erreur de syntaxe.

Un *`label`* est seulement nécessaire si vous voulez identifier le bloc à utiliser dans une instruction `EXIT` ou pour qualifier les noms de variable déclarées dans le bloc. Si un label est écrit après `END`, il doit correspondre au label donné au début du bloc.

Tous les mots clés sont insensibles à la casse. Les identifiants sont convertis implicitement en minuscule sauf dans le cas de l'utilisation de guillemets doubles. Le comportement est donc identique à celui des commandes SQL habituelles.

Les commentaires fonctionnent de la même manière tant dans du PL/pgSQL que dans le code SQL. Un double tiret (`--`) commence un commentaire et celui-ci continue jusqu'à la fin de la ligne. Un `/*` commence un bloc de commentaire qui continue jusqu'au `*/` correspondant. Les blocs de commentaires peuvent imbriquer les uns dans les autres.

Chaque expression de la section expression d'un bloc peut être un *sous-bloc*. Les sous-blocs peuvent être utilisés pour des groupements logiques ou pour situer des variables locales dans un petit groupe d'instructions. Les variables déclarées dans un sous-bloc masquent toute variable nommée de façon similaire dans les blocs externes pendant toute la durée du sous-bloc. Cependant, vous pouvez accéder aux variables externes si vous qualifiez leur nom du label de leur bloc. Par exemple :

```
CREATE FUNCTION une_fonction() RETURNS integer AS $$
<< blocexterne >>
DECLARE
    quantite integer := 30;
BEGIN
    RAISE NOTICE 'quantité vaut ici %', quantite;  -- affiche 30
    quantite := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantite integer := 80;
    BEGIN
        RAISE NOTICE 'quantite vaut ici %', quantite;  -- affiche 80
```

```
        RAISE NOTICE 'la quantité externe vaut ici %', blocexterne.quantite;  --
affiche 50
    END;

    RAISE NOTICE 'quantité vaut ici %', quantite;  -- affiche 50

    RETURN quantite;
END;
$$ LANGUAGE plpgsql;
```

> **Note**
>
> Il existe un bloc externe caché entourant le corps de toute fonction PL/pgSQL. Ce bloc fournit la déclaration des paramètres de la fonction ainsi que quelques variables spéciales comme `FOUND` (voir la Section 39.5.5, « Obtention du statut du résultat »). Le bloc externe a pour label le nom de la fonction. Cela a pour conséquence que les paramètres et les variables spéciales peuvent être qualifiés du nom de la fonction.

Il est important de ne pas confondre l'utilisation de **BEGIN**/**END** pour grouper les instructions dans PL/pgSQL avec les commandes pour le contrôle des transactions. Les **BEGIN**/**END** de PL/pgSQL ne servent qu'au groupement ; ils ne débutent ni ne terminent une transaction. Les fonctions standards et les fonctions triggers sont toujours exécutées à l'intérieur d'une transaction établie par une requête extérieure -- ils ne peuvent pas être utilisés pour commencer ou valider une transaction car ils n'auraient pas de contexte pour s'exécuter. Néanmoins, un bloc contenant une clause `EXCEPTION` forme réellement une sous-transaction qui peut être annulée sans affecter la transaction externe. Pour plus d'informations sur ce point, voir la Section 39.6.5, « Récupérer les erreurs ».

39.3. Déclarations

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc. Les seules exceptions sont que la variable de boucle d'une boucle `FOR` effectuant une itération sur des valeurs entières est automatiquement déclarée comme variable entière (type integer), et de la même façon une variable de boucle `FOR` effectuant une itération sur le résultat d'un curseur est automatiquement déclarée comme variable de type record.

Les variables PL/pgSQL peuvent être de n'importe quel type de données tels que integer, varchar et char.

Quelques exemples de déclaration de variables :

```
id_utilisateur integer;
quantité numeric(5);
url varchar;
ma_ligne nom_table%ROWTYPE;
mon_champ nom_table.nom_colonne%TYPE;
une_ligne RECORD;
```

La syntaxe générale d'une déclaration de variable est :

```
nom [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

La clause `DEFAULT`, si indiquée, spécifie la valeur initiale affectée à la variable quand on entre dans le bloc. Si la clause `DEFAULT` n'est pas indiquée, la variable est initialisée à la valeur SQL NULL. L'option `CONSTANT` empêche la modification de la variable, de sorte que sa valeur reste constante pour la durée du bloc. Si `NOT NULL` est spécifié, l'affectation d'une valeur NULL aboutira à une erreur d'exécution. Les valeurs par défaut de toutes les variables déclarées `NOT NULL` doivent être précisées, donc non NULL.

La valeur par défaut d'une variable est évaluée et affectée à la variable à chaque entrée du bloc (pas seulement une fois lors de l'appel de la fonction). Ainsi, par exemple, l'affectation de `now()` à une variable de type timestamp donnera à la variable l'heure de l'appel de la fonction courante, et non l'heure au moment où la fonction a été précompilée.

Exemples :

```
quantité integer DEFAULT 32;
url varchar := 'http://mysite.com';
id_utilisateur CONSTANT integer := 10;
```

39.3.1. Déclarer des paramètres de fonctions

Les paramètres passés aux fonctions sont nommés par les identifiants `$1`, `$2`, etc. Éventuellement, des alias peuvent être déclarés

pour les noms de paramètres de type $*n* afin d'améliorer la lisibilité. L'alias ou l'identifiant numérique peuvent être utilisés indifféremment pour se référer à la valeur du paramètre.

Il existe deux façons de créer un alias. La façon préférée est de donner un nom au paramètre dans la commande **CREATE FUNCTION**, par exemple :

```
CREATE FUNCTION taxe_ventes(sous_total real) RETURNS real AS $$
BEGIN
    RETURN sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

L'autre façon, la seule disponible pour les versions antérieures à PostgreSQL™ 8.0, est de déclarer explicitement un alias en utilisant la syntaxe de déclaration :

```
nom ALIAS FOR $n;
```

Le même exemple dans ce style ressemble à ceci :

```
CREATE FUNCTION taxe_ventes(real) RETURNS real AS $$
DECLARE
    sous_total ALIAS FOR $1;
BEGIN
    RETURN sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Note

Ces deux exemples ne sont pas complètement identiques. Dans le premier cas, `sous_total` peut être référencé comme `taxe_ventes.sous_total`, alors que ce n'est pas possible dans le second cas. (Si nous avions attaché un label au bloc interne, `sous_total` aurait pu utiliser ce label à la place.)

Quelques exemples de plus :

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- quelques traitements utilisant ici v_string et index
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_champs_selectionnes(in_t un_nom_de_table) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Quand une fonction PL/pgSQL est déclarée avec des paramètres en sortie, ces derniers se voient attribués les noms $*n* et des alias optionnels de la même façon que les paramètres en entrée. Un paramètre en sortie est une variable qui commence avec la valeur NULL ; il devrait se voir attribuer une valeur lors de l'exécution de la fonction. La valeur finale du paramètre est ce qui est renvoyée. Par exemple, l'exemple taxe_ventes peut s'écrire de cette façon :

```
CREATE FUNCTION taxe_ventes(sous_total real, OUT taxe real) AS $$
BEGIN
    taxe := sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Notez que nous avons omis `RETURNS real`. Nous aurions pu l'inclure mais cela aurait été redondant.

Les paramètres en sortie sont encore plus utiles lors du retour de plusieurs valeurs. Un exemple trivial est :

```
CREATE FUNCTION somme_n_produits(x int, y int, OUT somme int, OUT produit int) AS $$
BEGIN
    somme := x + y;
    produit := x * y;
END;
```

```
$$ LANGUAGE plpgsql;
```

D'après ce qui a été vu dans la Section 35.4.4, « Fonctions SQL avec des paramètres en sortie », ceci crée réellement un type d'enregistrement anonyme pour les résultats de la fonction. Si une clause `RETURNS` est donnée, elle doit spécifier `RETURNS record`.

Voici une autre façon de déclarer une fonction PL/pgSQL, cette fois avec `RETURNS TABLE` :

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT quantity, quantity * price FROM sales
                 WHERE itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

C'est exactement équivalent à déclarer un ou plusieurs paramètres `OUT` et à spécifier `RETURNS SETOF` *`un_type`*.

Lorsque le type de retour d'une fonction PL/pgSQL est déclaré comme type polymorphe (anyelement, anyarray, anynonarray et anyenum), un paramètre spécial `$0` est créé. Son type de donnée est le type effectif de retour de la fonction, déduit d'après les types en entrée (voir la Section 35.2.5, « Types et fonctions polymorphes »). Ceci permet à la fonction d'accéder à son type de retour réel comme on le voit ici avec la Section 39.3.3, « Copie de types ». `$0` est initialisé à NULL et peut être modifié par la fonction, de sorte qu'il peut être utilisé pour contenir la variable de retour si besoin est, bien que cela ne soit pas requis. On peut aussi donner un alias à `$0`. Par exemple, cette fonction s'exécute comme un opérateur `+` pour n'importe quel type de données :

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    resultat ALIAS FOR $0;
BEGIN
    resultat := v1 + v2 + v3;
    RETURN resultat;
END;
$$ LANGUAGE plpgsql;
```

Le même effet peut être obtenu en déclarant un ou plusieurs paramètres polymorphes en sortie de types. Dans ce cas, le paramètre spécial `$0` n'est pas utilisé ; les paramètres en sortie servent ce même but. Par exemple :

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anyelement, v2 anyelement, v3 anyelement,
                                     OUT somme anyelement)
AS $$
BEGIN
    somme := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

39.3.2. `ALIAS`

```
nouveaunom ALIAS FOR anciennom;
```

La syntaxe `ALIAS` est plus générale que la section précédente pourrait faire croire : vous pouvez déclarer un alias pour n'importe quelle variable et pas seulement des paramètres de fonction. L'utilisation principale de cette instruction est l'attribution d'un autre nom aux variables aux noms prédéterminés, telles que `NEW` ou `OLD` au sein d'une procédure trigger.

Exemples:

```
DECLARE
  anterieur ALIAS FOR old;
  misajour ALIAS FOR new;
```

`ALIAS` créant deux manières différentes de nommer le même objet, son utilisation à outrance peut préter à confusion. Il vaut mieux ne l'utiliser uniquement pour se passer des noms prédéterminés.

39.3.3. Copie de types

```
variable%TYPE
```

`%TYPE` fournit le type de données d'une variable ou d'une colonne de table. Vous pouvez l'utiliser pour déclarer des variables qui contiendront des valeurs de base de données. Par exemple, disons que vous avez une colonne nommée `id_utilisateur` dans votre table `utilisateurs`. Pour déclarer une variable du même type de données que `utilisateurs.id_utilisateur`, vous pouvez écrire :

```
id_utilisateur utilisateurs.id_utilisateur%TYPE;
```

En utilisant `%TYPE` vous n'avez pas besoin de connaître le type de données de la structure à laquelle vous faites référence et, plus important, si le type de données de l'objet référencé change dans le futur (par exemple : vous changez le type de `id_utilisateur` de integer à real), vous pouvez ne pas avoir besoin de changer votre définition de fonction.

`%TYPE` est particulièrement utile dans le cas de fonctions polymorphes puisque les types de données nécessaires aux variables internes peuvent changer d'un appel à l'autre. Des variables appropriées peuvent être créées en appliquant `%TYPE` aux arguments de la fonction ou à la variable fictive de résultat.

39.3.4. Types ligne

```
nom nom_table%ROWTYPE;
nom nom_type_composite;
```

Une variable de type composite est appelée variable *ligne* (ou variable *row-type*). Une telle variable peut contenir une ligne entière de résultat de requête **SELECT** ou **FOR**, du moment que l'ensemble de colonnes de la requête correspond au type déclaré de la variable. Les champs individuels de la valeur row sont accessibles en utilisant la notation pointée, par exemple `varligne.champ`.

Une variable ligne peut être déclarée de façon à avoir le même type que les lignes d'une table ou d'une vue existante, en utilisant la notation *`nom_table`*`%ROWTYPE`. Elle peut aussi être déclarée en donnant un nom de type composite. Chaque table ayant un type de données associé du même nom, il importe peu dans PostgreSQL™ que vous écriviez `%ROWTYPE` ou pas. Cependant, la forme utilisant `%ROWTYPE` est plus portable.

Les paramètres d'une fonction peuvent être des types composites (lignes complètes de tables). Dans ce cas, l'identifiant correspondant `$`*n* sera une variable ligne à partir de laquelle les champs peuvent être sélectionnés avec la notation pointée, par exemple `$1.id_utilisateur`.

Seules les colonnes définies par l'utilisateur sont accessibles dans une variable de type ligne, et non l'OID ou d'autres colonnes systèmes (parce que la ligne pourrait être issue d'une vue). Les champs du type ligne héritent des tailles des champs de la table ou de leur précision pour les types de données tels que char(*n*).

Voici un exemple d'utilisation des types composites. table1 et table2 sont des tables ayant au moins les champs mentionnés :

```
CREATE FUNCTION assemble_champs(t_ligne table1) RETURNS text AS $$
DECLARE
    t2_ligne table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_ligne FROM table2 WHERE ... ;
    RETURN t_ligne.f1 || t2_ligne.f3 || t_ligne.f5 || t2_ligne.f7;
END;
$$ LANGUAGE plpgsql;

SELECT assemble_champs(t.*) FROM table1 t WHERE ... ;
```

39.3.5. Types record

```
nom RECORD;
```

Les variables record sont similaires aux variables de type ligne mais n'ont pas de structure prédéfinie. Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont affectées durant une commande **SELECT** ou **FOR**. La sous-structure d'une variable record peut changer à chaque fois qu'on l'affecte. Une conséquence de cela est qu'elle n'a pas de sous-structure jusqu'à ce qu'elle ait été affectée, et toutes les tentatives pour accéder à un de ses champs entraînent une erreur d'exécution.

Notez que `RECORD` n'est pas un vrai type de données mais seulement un paramètre fictif (placeholder). Il faut aussi réaliser que lorsqu'une fonction PL/pgSQL est déclarée renvoyer un type record, il ne s'agit pas tout à fait du même concept qu'une variable record, même si une telle fonction peut aussi utiliser une variable record pour contenir son résultat. Dans les deux cas, la structure réelle de la ligne n'est pas connue quand la fonction est écrite mais, dans le cas d'une fonction renvoyant un type record, la structure réelle est déterminée quand la requête appelante est analysée, alors qu'une variable record peut changer sa structure de ligne à la volée.

39.4. Expressions

Toutes les expressions utilisées dans les instructions PL/pgSQL sont traitées par l'exécuteur SQL classique du serveur. En effet, une requête comme

```
SELECT expression
```

est traité par le moteur SQL principal. Bien qu'utilisant la commande **SELECT**, tout nom de variable PL/pgSQL est remplacé par des paramètres (ceci est expliqué en détail dans la Section 39.10.1, « Substitution de variables »). Cela permet au plan de requête du **SELECT** d'être préparé une seule fois, puis d'être réutilisé pour les évaluations suivantes avec différentes valeurs des variables. Du coup, ce qui arrive réellement à la première utilisation d'une expression est simplement une commande **PREPARE**. Par exemple, si nous déclarons deux variables de type integer, `x` et `y`, et que nous écrivons :

```
IF x < y THEN ...
```

ce qui se passe en arrière plan est équivalent à :

```
PREPARE nom_instruction(integer, integer) AS SELECT $1 < $2;
```

puis cette instruction préparée est exécutée (via **EXECUTE**) pour chaque exécution de l'instruction **IF**, avec les valeurs actuelles des variables PL/pgSQL fournies en tant que valeurs des paramètres. Le plan de requête préparé de cette façon est sauvegardé pour toute la durée de la connexion à la base, comme le décrit la Section 39.10.2, « Mise en cache du plan ». Généralement, ces détails ne sont pas importants pour un utilisateur de PL/pgSQL, mais ils sont utiles à connaître pour diagnostiquer un problème.

39.5. Instructions de base

Dans cette section ainsi que les suivantes, nous décrirons tous les types d'instructions explicitement compris par PL/pgSQL. Tout ce qui n'est pas reconnu comme l'un de ces types d'instruction est présumé être une commande SQL et est envoyé au moteur principal de bases de données pour être exécutée comme décrit dans la Section 39.5.2, « Exécuter une commande sans résultats » et dans la Section 39.5.3, « Exécuter une requête avec une seule ligne de résultats ».

39.5.1. Affectation

L'affectation d'une valeur à une variable PL/pgSQL s'écrit ainsi :

```
variable := expression;
```

Comme expliqué précédemment, l'expression dans cette instruction est évaluée au moyen de la commande SQL **SELECT** envoyée au moteur principal de bases de données. L'expression ne doit manier qu'une seule valeur (éventuellement une valeur de rangée, si cette variable est une variable de rangée ou d'enrengistrement). La variable cible peut être une simple varible (éventuellement qualifiée avec un nom de bloc), un champ d'une rangée ou variable d'enrengistrement ou un élément de tableau qui se trouve être une simple variable ou champ.

Si le type de données du résultat de l'expression ne correspond pas au type de donnée de la variable, ou que la variable a une taille ou une précision (comme char(20)), la valeur résultat sera implicitement convertie par l'interpréteur PL/pgSQL en utilisant la fonction d'écriture (output-function) du type du résultat, et la fonction d'entrée (input-function) du type de la variable. Notez que cela peut conduire à des erreurs d'exécution générées par la fonction d'entrée si la forme de la chaîne de la valeur résultat n'est pas acceptable pour cette fonction.

Exemples :

```
taxe := sous_total * 0.06;
mon_enregistrement.id_utilisateur := 20;
```

39.5.2. Exécuter une commande sans résultats

Pour toute commande SQL qui ne renvoie pas de lignes, par exemple **INSERT** sans clause `RETURNING`, vous pouvez exécuter la commande à l'intérieur d'une fonction PL/pgSQL rien qu'en écrivant la commande.

Tout nom de variable PL/pgSQL apparaissant dans le texte de la commande est traité comme un paramètre, puis la valeur actuelle de la variable est fournie comme valeur du paramètre à l'exécution. C'est le traitement exact décrit précédemment pour les expressions. Pour les détails, voir la Section 39.10.1, « Substitution de variables ».

Lors de l'exécution d'une commande SQL de cette façon, PL/pgSQL planifie la commande une fois et ré-utilise ce plan lors des prochaines exécutions, pour la durée de vie de la connexion. Les implications de ceci sont discutées en détail dans la Section 39.10.2, « Mise en cache du plan ».

Parfois, il est utile d'évaluer une expression ou une requête **SELECT** mais sans récupérer le résultat, par exemple lors de l'appel d'une fonction qui a des effets de bord mais dont la valeur du résultat n'est pas utile. Pour faire cela en PL/pgSQL, utilisez l'instruction **PERFORM** :

```
PERFORM requête;
```

Ceci exécute la *requête* et ne tient pas compte du résultat. Écrivez la *requête* de la même façon que vous écririez une commande **SELECT** mais remplacez le mot clé initial **SELECT** avec **PERFORM**. Pour les requêtes **WITH**, utilisez **PERFORM** puis placez la requête entre parenthèses. Dans ce cas, la requête peut seulement renvoyer une ligne. Les variables PL/pgSQL seront substituées dans la requête comme pour les commandes qui ne renvoient pas de résultat. Le plan est mis en cache de la même façon. La variable spéciale `FOUND` est configurée à true si la requête a produit au moins une ligne, false dans le cas contraire (voir la Section 39.5.5, « Obtention du statut du résultat »).

> **Note**
>
> Vous pourriez vous attendre à ce que l'utilisation directe de **SELECT** aboutisse au même résultat mais, actuellement, la seule façon acceptée de le faire est d'utiliser **PERFORM**. Une commande SQL qui peut renvoyer des lignes comme **SELECT** sera rejetée comme une erreur si elle n'a pas de clause `INTO`, ce qui est discuté dans la section suivante.

Un exemple :

```
PERFORM creer_vuemat('cs_session_page_requests_mv', ma_requete);
```

39.5.3. Exécuter une requête avec une seule ligne de résultats

Le résultat d'une commande SQL ne ramenant qu'une seule ligne (mais avec une ou plusieurs colonnes) peut être affecté à une variable de type record, row ou à une liste de variables scalaires. Ceci se fait en écrivant la commande SQL de base et en ajoutant une clause `INTO`. Par exemple,

```
SELECT expressions_select INTO [STRICT] cible FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] cible;
UPDATE ... RETURNING expressions INTO [STRICT] cible;
DELETE ... RETURNING expressions INTO [STRICT] cible;
```

où *cible* peut être une variable de type record, row ou une liste de variables ou de champs record/row séparées par des virgules. Les variables PL/pgSQL seront substituées dans le reste de la requête, et le plan est mis en cache comme décrit ci-dessus pour les commandes qui ne renvoient pas de lignes. Ceci fonctionne pour **SELECT**, **INSERT**/**UPDATE**/**DELETE** avec `RETURNING`, et les commandes utilitaires qui renvoient des résultats de type rowset (comme **EXPLAIN**). Sauf pour la clause `INTO`, la commande SQL est identique à celle qui aurait été écrite en dehors de PL/pgSQL.

> **Astuce**
>
> Notez que cette interprétation de **SELECT** avec `INTO` est assez différente de la commande habituelle **SELECT INTO** où la cible `INTO` est une table nouvellement créée. Si vous voulez créer une table à partir du résultat d'un **SELECT** à l'intérieur d'une fonction PL/pgSQL, utilisez la syntaxe **CREATE TABLE ... AS SELECT**.

Si une ligne ou une liste de variables est utilisée comme cible, les colonnes du résultat de la requête doivent correspondre exactement à la structure de la cible (nombre de champs et types de données). Dans le cas contraire, une erreur sera rapportée à l'exécution. Quand une variable record est la cible, elle se configure automatiquement avec le type row des colonnes du résultat de la requête.

La clause `INTO` peut apparaître pratiquement partout dans la commande SQL. Elle est écrite soit juste avant soit juste après la liste d'*expressions_select* dans une commande **SELECT**, ou à la fin de la commande pour d'autres types de commande. Il est recommandé de suivre cette convention au cas où l'analyseur PL/pgSQL devient plus strict dans les versions futures.

Si `STRICT` n'est pas spécifié dans la clause `INTO`, alors *cible* sera configuré avec la première ligne renvoyée par la requête ou à NULL si la requête n'a renvoyé aucune ligne. (Notez que « la première ligne » n'est bien définie que si vous avez utilisé `ORDER BY`.) Toute ligne résultat après la première ligne est annulée. Vous pouvez vérifier la valeur de la variable spéciale `FOUND` (voir la Section 39.5.5, « Obtention du statut du résultat ») pour déterminer si une ligne a été renvoyée :

```
SELECT * INTO monrec FROM emp WHERE nom = mon_nom;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employé % introuvable', mon_nom;
END IF;
```

Si l'option `STRICT` est indiquée, la requête doit renvoyer exactement une ligne. Dans le cas contraire, une erreur sera rapportée à l'exécution, soit `NO_DATA_FOUND` (aucune ligne) soit `TOO_MANY_ROWS` (plus d'une ligne). Vous pouvez utiliser un bloc d'exception si vous souhaitez récupérer l'erreur, par exemple :

```
BEGIN
    SELECT * INTO STRICT monrec FROM emp WHERE nom = mon_nom;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employé % introuvable', mon_nom;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employé % non unique', mon_nom;
END;
```

Une exécution réussie de la commande avec `STRICT` renvoie toujours true pour `FOUND`.

Pour **INSERT**/**UPDATE**/**DELETE** avec `RETURNING`, PL/pgSQL rapporte une erreur si plus d'une ligne est renvoyée, même quand `STRICT` n'est pas spécifié. Ceci est dû au fait qu'il n'y a pas d'option comme `ORDER BY` qui pourrait déterminer la ligne à renvoyer.

> **Note**
>
> L'option `STRICT` correspond au comportement du **SELECT INTO** d'Oracle PL/SQL et des instructions relatives.

Pour gérer les cas où vous avez besoin de traiter plusieurs lignes de résultat à partir d'une requête SQL, voir la Section 39.6.4, « Boucler dans les résultats de requêtes ».

39.5.4. Exécuter des commandes dynamiques

Créer dynamique des requêtes SQL est un besoin habituel dans les fonctions PL/pgSQL, par exemple des requêtes qui impliquent différentes tables ou différents types de données à chaque fois qu'elles sont exécutées. Les tentatives normales de PL/pgSQL pour garder en cache les planifications des commandes (voir la Section 39.10.2, « Mise en cache du plan ») ne fonctionneront pas dans de tels scénarios. Pour gérer ce type de problème, l'instruction **EXECUTE** est proposée :

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ...] ];
```

où *`chaîne-commande`* est une expression manipulant une chaîne (de type text) contenant la commande à exécuter. La *`cible`* optionnelle est une variable record ou ligne ou même une liste de variables simples ou de champs de lignes/enregistrements séparées par des virgules, dans lesquels les résultats de la commande seront enregistrés. Les expressions `USING` optionnelles fournissent des valeurs à insérer dans la commande.

Aucune substitution des variables PL/pgSQL ne se fait dans la chaîne de commande calculée. Toutes les valeurs des variables requises doivent être insérées dans la chaîne de commande au moment de sa construction ; ou vous pouvez utiliser des paramètres comme décrits ci-dessous.

De plus, il n'y a pas mise en cache des commandes exécutées via **EXECUTE**. À la place, la commande est préparée à chaque fois que l'instruction est lancée. La chaîne commande peut être créée dynamiquement à l'intérieur de la fonction pour agir sur des tables ou colonnes différentes.

La clause `INTO` spécifie où devraient être affectés les résultats d'une commande SQL renvoyant des lignes. Si une ligne ou une liste de variable est fournie, elle doit correspondre exactement à la structure des résultats de la requête (quand une variable de type record est utilisée, elle sera automatiquement typée pour correspondre à la structure du résultat). Si plusieurs lignes sont renvoyées, alors seule la première sera assignée à la variable `INTO`. Si aucune ligne n'est renvoyée, NULL est affectée à la variable `INTO`. Si aucune clause `INTO` n'est spécifiée, les résultats de la requête sont ignorés.

Si l'option `STRICT` est indiquée, une erreur est rapportée sauf si la requête produit exactement une ligne.

La chaîne de commande peut utiliser des valeurs de paramètres, référencées dans la commande avec `$1`, `$2`, etc. Ces symboles font référence aux valeurs fournies dans la clause `USING`. Cette méthode est souvent préférable à l'insertion des valeurs en texte dans une chaîne de commande : cela évite la surcharge à l'exécution pour la conversion des valeurs en texte et vice-versa. C'est aussi moins sensible aux attaques par injection SQL car il n'est pas nécessaire de mettre entre guillemets ou d'échapper les valeurs. Voici un exemple :

```
EXECUTE 'SELECT count(*) FROM matable WHERE insere_par = $1 AND insere <= $2'
   INTO c
   USING utilisateur_verifie, date_verifiee;
```

Notez que les symboles de paramètres peuvent seulement être utilisés pour des valeurs de données -- si vous voulez utiliser des noms de tables et/ou colonnes déterminés dynamiquement, vous devez les insérer dans la chaîne de commande en texte. Par

exemple, si la requête précédente devait se faire avec une table sélectionnée dynamiquement, vous devriez faire ceci :

```
EXECUTE 'SELECT count(*) FROM '
    || tabname::regclass
    || ' WHERE insere_par = $1 AND insere <= $2'
   INTO c
   USING utilisateur_verifie, date_verifiee;
```

Une autre restriction sur les symboles de paramètres est qu'ils ne marchent que dans les commandes **SELECT**, **INSERT**, **UPDATE** et **DELETE**. Dans les autres types d'instructions (appellés de manière générique commandes utilitaires), vous devez insérer les valeurs sous forme de texte même si ce ne sont que des données.

Un **EXECUTE** avec une chaîne de commande constante et des paramètres `USING`, comme dans le premier exemple ci-dessus, est équivalent fonctionnellement à l'écriture simple d'une commande directement dans PL/pgSQL et permet le remplacement automatique des variables PL/pgSQL. La différence importante est que **EXECUTE** va planifier de nouveau la commande pour chaque exécution, générant un plan qui est spécifique aux valeurs actuelles des paramètres ; alors que PL/pgSQL crée habituellement un plan générique et le stocke pour le réutiliser. Dans des situations où le meilleur plan dépend fortement des valeurs des paramètres, **EXECUTE** peut être beaucoup plus rapide : alors que lorsque le plan n'est pas sensible aux valeurs des paramètres, la replanification sera une perte.

SELECT INTO n'est actuellement pas supporté à l'intérieur de **EXECUTE** ; à la place, exécutez une commande **SELECT** et spécifiez `INTO` comme faisant parti lui-même d'**EXECUTE**.

Note

L'instruction **EXECUTE** de PL/pgSQL n'a pas de relation avec l'instruction SQL EXECUTE(7) supportée par le serveur PostgreSQL™. L'instruction **EXECUTE** du serveur ne peut pas être utilisée directement dans les fonctions PL/pgSQL. En fait, elle n'est pas nécessaire.

Exemple 39.1. Mettre entre guillemets des valeurs dans des requêtes dynamiques

En travaillant avec des commandes dynamiques, vous aurez souvent à gérer des échappements de guillemets simples. La méthode recommandée pour mettre entre guillemets un texte fixe dans le corps de votre fonction est d'utiliser les guillemets dollar (si votre code n'utilise pas les guillemets dollar, référez-vous à l'aperçu dans la Section 39.11.1, « Utilisation des guillemets simples (quotes) », ce qui peut vous faire gagner des efforts lors du passage de ce code à un schéma plus raisonnable).

Les valeurs dynamiques qui sont à insérer dans la requête construite requièrent une gestion spéciale car elles pourraient elles-même contenir des guillemets. Un exemple (ceci suppose que vous utilisez les guillemets dollar pour la fonction dans sa globalité, du coup les guillemets n'ont pas besoin d'être doublés) :

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(nom_colonne)
    || ' = '
    || quote_literal(nouvelle_valeur)
    || ' WHERE cle = '
    || quote_literal(valeur_cle);
```

Cet exemple démontre l'utilisation des fonctions `quote_ident` et `quote_literal` (voir Section 9.4, « Fonctions et opérateurs de chaînes »). Pour plus de sûreté, les expressions contenant les identifiants des colonnes et des tables doivent être passées à la fonction `quote_ident` avant l'insertion dans une requête dynamique. Les expressions contenant des valeurs de type chaîne de caractères doivent être passées à `quote_literal`. Ce sont les étapes appropriées pour renvoyer le texte en entrée entouré par des guillemets doubles ou simples respectivement, en échappant tout caractère spécial.

Comme `quote_literal` est labelisé `STRICT`, elle renverra toujours NULL lorsqu'elle est appelée avec un argument NULL. Dans l'exemple ci-dessus, si `nouvelle_valeur` ou `valeur_clé` étaient NULL, la requête dynamique entière deviendrait NULL, amenant une erreur à partir du **EXECUTE**. Vous pouvez éviter ce problème en utilisant la fonction `quote_nullable` qui fonctionne de façon identique à `quote_literal` sauf si elle est appelée avec un argument NULL, elle renvoie la chaîne `NULL`. Par exemple,

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(nom_colonne)
        || ' = '
        || quote_nullable(nouvelle_valeur)
        || ' WHERE key = '
        || quote_nullable(valeur_clé);
```

Si vous travaillez avez des valeurs qui peuvent être NULL, vous devez utiliser `quote_nullable` à la place de `quote_literal`.

Comme toujours, il faut s'assurer que les valeurs NULL d'une requête ne ramènent pas des valeurs inattendues. Par exemple, la clause `WHERE`

```
'WHERE key = ' || quote_nullable(valeur_clé)
```

ne sera jamais vrai si `valeur_clé` est NULL car le résultat de l'opérateur d'égalité, `=`, avec au moins un des opérandes NULL est toujours NULL. Si vous souhaitez que NULL fonctionne comme toute autre valeur de clé ordinaire, vous devez ré-écrire la clause ci-dessus de cette façon :

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(Actuellement, `IS NOT DISTINCT FROM` est géré moins efficacement que `=`, donc ne l'utilisez pas sauf en cas d'extrême nécessité. Voir Section 9.2, « Opérateurs de comparaison » pour plus d'informations sur les NULL et `IS DISTINCT`.)

Notez que les guillemets dollar sont souvent utiles pour placer un texte fixe entre guillemets. Ce serait une très mauvaise idée d'écrire l'exemple ci-dessus de cette façon :

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(nom_colonne)
|| ' = $$'
|| nouvelle_valeur
|| '$$ WHERE cle = '
|| quote_literal(valeur_cle);
```

car cela casserait si le contenu de `nouvelle_valeur` pouvait contenir `$$`. La même objection s'applique à tout délimiteur dollar que vous pourriez choisir. Donc, pour mettre un texte inconnu entre guillemets de façon sûr, vous *devez* utiliser `quote_literal`, `quote_nullable` ou `quote_ident`, comme approprié.

Un exemple bien plus important d'une commande dynamique et d'**EXECUTE** est disponible dans l'Exemple 39.7, « Portage d'une fonction qui crée une autre fonction de PL/SQL vers PL/pgSQL », qui construit et exécute une commande **CREATE FUNCTION** pour définir une nouvelle fonction.

39.5.5. Obtention du statut du résultat

Il y a plusieurs moyens pour déterminer l'effet d'une commande. La première méthode est d'utiliser **GET DIAGNOSTICS** :

```
GET DIAGNOSTICS variable = élément [ , ... ] ;
```

Cette commande permet la récupération des indicateurs d'état du système. Chaque *`élément`* est un mot clé identifiant une valeur d'état devant être affectée à la variable indiquée (qui doit être du bon type de donnée pour que l'affectation puisse se faire sans erreur.) Les éléments d'état actuellement disponibles sont `ROW_COUNT`, le nombre de lignes traitées par la dernière commande SQL envoyée au moteur SQL, et `RESULT_OID`, l'OID de la dernière ligne insérée par la commande SQL la plus récente. Notez que `RESULT_OID` n'est utile qu'après une commande **INSERT** dans une table contenant des OID.

Exemple :

```
GET DIAGNOSTICS var_entier = ROW_COUNT;
```

La seconde méthode permettant de déterminer les effets d'une commande est la variable spéciale nommée `FOUND` de type boolean. La variable `FOUND` est initialisée à false au début de chaque fonction PL/pgSQL. Elle est positionnée par chacun des types d'instructions suivants :

- Une instruction **SELECT INTO** positionne `FOUND` à true si une ligne est affectée, false si aucune ligne n'est renvoyée.
- Une instruction **PERFORM** positionne `FOUND` à true si elle renvoie une ou plusieurs lignes, false si aucune ligne n'est produite.
- Les instructions **UPDATE**, **INSERT**, et **DELETE** positionnent `FOUND` à true si au moins une ligne est affectée, false si aucune ligne n'est affectée.
- Une instruction **FETCH** positionne `FOUND` à true si elle renvoie une ligne, false si aucune ligne n'est renvoyée.
- Une instruction **MOVE** initialise `FOUND` à true si elle repositionne le curseur avec succès. Dans le cas contraire, elle le positionne à false.

- La commande **FOR** positionne `FOUND` à true si elle effectue une itération une ou plusieurs fois, sinon elle renvoie false. Ceci s'applique aux quatre variantes de l'instruction **FOR** (boucles **FOR** integer, boucles **FOR** sur des ensembles d'enregistrements, boucles **FOR** sur des ensembles d'enregistrements dynamiques et boucles **FOR** sur des curseurs). `FOUND` n'est positionnée de cette façon que quand la boucle **FOR** s'achève ; dans l'exécution de la chaîne, `FOUND` n'est pas modifiée par l'instruction **FOR**, bien qu'elle puisse être modifié par l'exécution d'autres instructions situées dans le corps de la boucle.
- Les instructions **RETURN QUERY** et **RETURN QUERY EXECUTE** mettent à jour la variable `FOUND` à true si la requête renvoie au moins une ligne, et false si aucune ligne n'est renvoyée.

Les autres instructions PL/pgSQL ne changent pas l'état de `FOUND`. Notez que la commande **EXECUTE** modifie la sortie de **GET DIAGNOSTICS** mais ne change pas `FOUND`.

`FOUND` est une variable locale à l'intérieur de chaque fonction PL/pgSQL ; chaque changement qui y est fait n'affecte que la fonction en cours.

39.5.6. Ne rien faire du tout

Quelque fois, une instruction qui ne fait rien est utile. Par exemple, elle indique qu'une partie de la chaîne **IF/THEN/ELSE** est délibérément vide. Pour cela, utilisez l'instruction :

```
NULL;
```

Par exemple, les deux fragments de code suivants sont équivalents :

```
BEGIN
  y := x / 0;
  EXCEPTION
  WHEN division_by_zero THEN
    NULL;  -- ignore l'erreur
  END;
```

```
BEGIN
  y := x / 0;
  EXCEPTION
  WHEN division_by_zero THEN  -- ignore l'erreur
  END;
```

Ce qui est préférable est une question de goût.

> **Note**
>
> Dans le PL/SQL d'Oracle, les listes d'instructions vides ne sont pas autorisées et, du coup, les instructions **NULL** sont *requises* dans les situations telles que celles-ci. PL/pgSQL vous permet d'écrire simplement rien.

39.6. Structures de contrôle

Les structures de contrôle sont probablement la partie la plus utile (et importante) de PL/pgSQL. Grâce aux structures de contrôle de PL/pgSQL, vous pouvez manipuler les données PostgreSQL™ de façon très flexible et puissante.

39.6.1. Retour d'une fonction

Il y a deux commandes disponibles qui vous permettent de renvoyer des données d'une fonction : **RETURN** et **RETURN NEXT**.

39.6.1.1. RETURN

```
RETURN expression;
```

RETURN accompagné d'une expression termine la fonction et renvoie le valeur de l'*`expression`* à l'appelant. Cette forme doit être utilisée avec des fonctions PL/pgSQL qui ne renvoient pas d'ensemble de valeurs.

Lorsqu'elle renvoie un type scalaire, n'importe quelle expression peut être utilisée. Le résultat de l'expression sera automatiquement converti vers le type de retour de la fonction, comme décrit pour les affectations. Pour renvoyer une valeur composite (ligne), vous devez écrire une variable record ou ligne comme *`expression`*.

Si vous déclarez la fonction avec des paramètres en sortie, écrivez seulement **RETURN** sans expression. Les valeurs courantes des paramètres en sortie seront renvoyées.

Si vous déclarez que la fonction renvoie void, une instruction **RETURN** peut être utilisée pour quitter rapidement la fonction ; mais n'écrivez pas d'expression après **RETURN**.

La valeur de retour d'une fonction ne peut pas être laissée indéfinie. Si le contrôle atteint la fin du bloc de haut niveau de la fonction, sans parvenir à une instruction **RETURN**, une erreur d'exécution survient. Néanmoins, cette restriction ne s'applique pas aux fonctions sans paramètre de sortie et aux fonctions renvoyant void. Dans ces cas, une instruction **RETURN** est automatiquement exécutée si le bloc de haut niveau est terminé.

39.6.1.2. RETURN NEXT et RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY requete;
RETURN QUERY EXECUTE command-string [ USING expression [, ...] ];
```

Quand une fonction PL/pgSQL déclare renvoyer `SETOF un_certain_type`, la procédure à suivre est un peu différente. Dans ce cas, les éléments individuels à renvoyer sont spécifiés par une séquence de commandes **RETURN NEXT** ou **RETURN QUERY**, suivies de la commande finale **RETURN** sans argument qui est utilisée pour indiquer la fin de l'exécution de la fonction. **RETURN NEXT** peut être utilisé avec des types de données scalaires comme composites ; avec un type de résultat composite, une « table » entière de résultats sera renvoyée. **RETURN QUERY** ajoute les résultats de l'exécution d'une requête à l'ensemble des résultats de la fonction. **RETURN NEXT** et **RETURN QUERY** peuvent être utilisés dans la même fonction, auquel cas leurs résultats seront concaténées.

RETURN NEXT et **RETURN QUERY** ne quittent pas réellement la fonction -- elles ajoutent simplement zéro ou plusieurs lignes à l'ensemble de résultats de la fonction. L'exécution continue ensuite avec l'instruction suivante de la fonction PL/pgSQL. Quand plusieurs commandes **RETURN NEXT** et/ou **RETURN QUERY** successives sont exécutées, l'ensemble de résultats augmente. Un **RETURN**, sans argument, permet de quitter la fonction mais vous pouvez aussi continuer jusqu'à la fin de la fonction.

RETURN QUERY dispose d'une variante **RETURN QUERY EXECUTE**, qui spécifie la requête à exécuter dynamiquement. Les expressions de paramètres peuvent être insérées dans la chaîne calculée via `USING`, de la même façon que le fait la commande **EXECUTE**.

Si vous déclarez la fonction avec des paramètres en sortie, écrivez **RETURN NEXT** sans expression. À chaque exécution, les valeurs actuelles des variables paramètres en sortie seront sauvegardées pour un renvoi éventuel en tant que résultat en sortie. Notez que vous devez déclarer la fonction en tant que `SETOF record` quand il y a plusieurs paramètres en sortie, ou `SETOF un_certain_type` quand il y a un seul paramètre en sortie, et de type `un_certain_type`, pour créer une fonction SRF avec des paramètres en sortie.

Voici un exemple d'une fonction utilisant **RETURN NEXT** :

```
CREATE TABLE truc (id_truc INT, sousid_truc INT, nom_truc TEXT);
INSERT INTO truc VALUES (1, 2, 'trois');
INSERT INTO truc VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION obtenirTousLesTrucs() RETURNS SETOF foo AS
$BODY$
DECLARE
    r truc%rowtype;
BEGIN
    FOR r IN SELECT * FROM truc
    WHERE id_truc > 0
    LOOP
        -- quelques traitements
        RETURN NEXT r; -- renvoie la ligne courante du SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE 'plpgsql' ;

SELECT * FROM obtenirTousLesTrucs();
```

Note

L'implémentation actuelle de **RETURN NEXT** et de **RETURN QUERY** pour PL/pgSQL récupère la totalité de l'ensemble des résultats avant d'effectuer le retour de la fonction, comme vu plus haut. Cela signifie que si une

fonction PL/pgSQL produit une structure résultat très grande, les performances peuvent être faibles : les données seront écrites sur le disque pour éviter un épuisement de la mémoire mais la fonction en elle-même ne renverra rien jusqu'à ce que l'ensemble complet des résultats soit généré. Une version future de PL/pgSQL permettra aux utilisateurs de définir des fonctions renvoyant des ensembles qui n'auront pas cette limitation. Actuellement, le point auquel les données commencent à être écrites sur le disque est contrôlé par la variable de configuration work_mem. Les administrateurs ayant une mémoire suffisante pour enregistrer des ensembles de résultats plus importants en mémoire doivent envisager l'augmentation de ce paramètre.

39.6.2. Contrôles conditionnels

Les instructions **IF** et **CASE** vous permettent d'exécuter des commandes basées sur certaines conditions. PL/pgSQL a trois formes de `IF` :

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`

et deux formes de **CASE** :

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

39.6.2.1. `IF-THEN`

```
IF expression-booleenne THEN
    instructions
END IF;
```

Les instructions `IF-THEN` sont la forme la plus simple de `IF`. Les instructions entre `THEN` et `END IF` seront exécutées si la condition est vraie. Autrement, elles seront ignorées.

Exemple :

```
IF v_id_utilisateur <> 0 THEN
    UPDATE utilisateurs SET email = v_email WHERE id_utilisateur = v_id_utilisateur;
END IF;
```

39.6.2.2. `IF-THEN-ELSE`

```
IF expression-booleenne THEN
    instructions
ELSE
    instructions
END IF;
```

Les instructions `IF-THEN-ELSE` s'ajoutent au `IF-THEN` en vous permettant de spécifier un autre ensemble d'instructions à exécuter si la condition n'est pas vraie (notez que ceci inclut le cas où la condition s'évalue à NULL.).

Exemples :

```
IF id_parent IS NULL OR id_parent = ''
THEN
    RETURN nom_complet;
ELSE
    RETURN hp_true_filename(id_parent) || '/' || nom_complet;
END IF;
```

```
IF v_nombre > 0 THEN
    INSERT INTO nombre_utilisateurs (nombre) VALUES (v_nombre);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

39.6.2.3. IF-THEN-ELSIF

```
IF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
    ...
]
]
[ ELSE
    instructions ]
END IF;
```

Quelques fois, il existe plus de deux alternatives. `IF-THEN-ELSIF` fournit une méthode agréable pour vérifier différentes alternatives. Les conditions `IF` sont testées successivement jusqu'à trouver la bonne. Alors les instructions associées sont exécutées, puis le contrôle est passé à la prochaine instruction après `END IF`. (Toute autre condition `IF` n'est *pas* testée.) Si aucune des conditions `IF` n'est vraie, alors le bloc `ELSE` (s'il y en a un) est exécuté.

Voici un exemple :

```
IF nombre = 0 THEN
    resultat := 'zero';
ELSIF nombre > 0 THEN
    resultat := 'positif';
ELSIF nombre < 0 THEN
    resultat := 'negatif';
ELSE
    -- hmm, la seule possibilité est que le nombre soit NULL
    resultat := 'NULL';
END IF;
```

Le mot clé `ELSIF` peut aussi s'écrire `ELSEIF`.

Une façon alternative d'accomplir la même tâche est d'intégrer les instructions `IF-THEN-ELSE`, comme dans l'exemple suivant :

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

Néanmoins, cette méthode requiert d'écrire un `END IF` pour chaque `IF`, donc c'est un peu plus compliqué que d'utiliser `ELSIF` quand il y a beaucoup d'autres alternatives.

39.6.2.4. CASE simple

```
CASE expression_recherche
    WHEN expression [, expression [ ... ]] THEN
      instructions
  [ WHEN expression [, expression [ ... ]] THEN
      instructions
    ... ]
  [ ELSE
      instructions ]
END CASE;
```

La forme simple de **CASE** fournit une exécution conditionnelle basée sur l'égalité des opérandes. L'*`expression-recherche`* est évaluée (une fois) puis comparée successivement à chaque *`expression`* dans les clauses `WHEN`. Si une correspondance est trouvée, alors les *`instructions`* correspondantes sont exécutées, puis le contrôle est passé à la prochaine instruction après `END CASE`. (Les autres expressions `WHEN` ne sont pas testées.) Si aucune correspondance n'est trouvée, les *`instructions`* du bloc `ELSE` sont exécutées ; s'il n'y a pas de bloc `ELSE`, une exception `CASE_NOT_FOUND` est levée.

Voici un exemple simple :

```
CASE x
    WHEN 1, 2 THEN
        msg := 'un ou deux';
    ELSE
        msg := 'autre valeur que un ou deux';
END CASE;
```

39.6.2.5. CASE recherché

```
CASE
    WHEN expression_booléenne THEN
      instructions
  [ WHEN expression_booléenne THEN
      instructions
    ... ]
  [ ELSE
      instructions ]
END CASE;
```

La forme recherché de **CASE** fournit une exécution conditionnelle basée sur la vérification d'expressions booléennes. Chaque *expression-booléenne* de la clause WHEN est évaluée à son tour jusqu'à en trouver une qui est validée (true). Les *instructions* correspondantes sont exécutées, puis le contrôle est passé à la prochaine instruction après END CASE. (Les expressions WHEN suivantes ne sont pas testées.) Si aucun résultat vrai n'est trouvé, les *instructions* du bloc ELSE sont exécutées. Si aucun bloc ELSE n'est présent, une exception CASE_NOT_FOUND est levée.

Voici un exemple :

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'valeur entre zéro et dix';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'valeur entre onze et vingt';
END CASE;
```

Cette forme de **CASE** est entièrement équivalente à IF-THEN-ELSIF, sauf pour la règle qui dit qu'atteindre une clause ELSE omise résulte dans une erreur plutôt que ne rien faire.

39.6.3. Boucles simples

Grâce aux instructions LOOP, EXIT, CONTINUE, WHILE et FOR, vous pouvez faire en sorte que vos fonctions PL/pgSQL répètent une série de commandes.

39.6.3.1. LOOP

```
[<<label>>]
LOOP
    instructions
END LOOP [ label ];
```

LOOP définit une boucle inconditionnelle répétée indéfiniment jusqu'à ce qu'elle soit terminée par une instruction EXIT ou **RETURN**. Le *label* optionnel peut être utilisé par les instructions EXIT et CONTINUE dans le cas de boucles imbriquées pour définir la boucle impliquée.

39.6.3.2. EXIT

```
EXIT [ label ] [ WHEN expression-booléenne ];
```

Si aucun *label* n'est donné, la boucle la plus imbriquée se termine et l'instruction suivant END LOOP est exécutée. Si un *label* est donné, ce doit être le label de la boucle, du bloc courant ou d'un niveau moins imbriqué. La boucle ou le bloc nommé se termine alors et le contrôle continue avec l'instruction située après le END de la boucle ou du bloc correspondant.

Si WHEN est spécifié, la sortie de boucle ne s'effectue que si *expression-booléenne* est vraie. Sinon, le contrôle passe à l'instruction suivant le EXIT.

`EXIT` peut être utilisé pour tous les types de boucles ; il n'est pas limité aux boucles non conditionnelles.

Lorsqu'il est utilisé avec un bloc `BEGIN`, `EXIT` passe le contrôle à la prochaine instruction après la fin du bloc. Notez qu'un label doit être utilisé pour cela ; un `EXIT` sans label n'est jamais pris en compte pour correspondre à un bloc `BEGIN`. (Ceci est un changement de la version 8.4 de PostgreSQL™. Auparavant, il était permis de faire correspondre un `EXIT` sans label avec un bloc `BEGIN`.)

Exemples :

```
LOOP
    -- quelques traitements
    IF nombre > 0 THEN
        EXIT;  -- sortie de boucle
    END IF;
END LOOP;

LOOP
    -- quelques traitements
    EXIT WHEN nombre > 0;
END LOOP;

<<un_bloc>>
BEGIN
    -- quelques traitements
    IF stocks > 100000 THEN
        EXIT un_bloc;  -- cause la sortie (EXIT) du bloc BEGIN
    END IF;
    -- les traitements ici seront ignorés quand stocks > 100000
END;
```

39.6.3.3. `CONTINUE`

```
CONTINUE [ label ] [ WHEN expression-booléenne ];
```

Si aucun *`label`* n'est donné, la prochaine itération de la boucle interne est commencée. C'est-à-dire que toutes les instructions restantes dans le corps de la boucle sont ignorées et le contrôle revient à l'expression de contrôle de la boucle pour déterminer si une autre itération de boucle est nécessaire. Si le *`label`* est présent, il spécifie le label de la boucle dont l'exécution va être continuée.

Si `WHEN` est spécifié, la prochaine itération de la boucle est commencée seulement si l'*`expression-booléenne`* est vraie. Sinon, le contrôle est passé à l'instruction après `CONTINUE`.

`CONTINUE` peut être utilisé avec tous les types de boucles ; il n'est pas limité à l'utilisation des boucles inconditionnelles.

Exemples :

```
LOOP
    -- quelques traitements
    EXIT WHEN nombre > 100;
    CONTINUE WHEN nombre < 50;
    -- quelques traitements pour nombre IN [50 .. 100]
END LOOP;
```

39.6.3.4. `WHILE`

```
[<<label>>]
WHILE expression-booléenne LOOP
    instructions
END LOOP [ label ];
```

L'instruction `WHILE` répète une séquence d'instructions aussi longtemps que *`expression-booléenne`* est évaluée à vrai. L'expression est vérifiée juste avant chaque entrée dans le corps de la boucle.

Par exemple :

```
WHILE montant_possede > 0 AND balance_cadeau > 0 LOOP
    -- quelques traitements ici
END LOOP;
```

```
WHILE NOT termine LOOP
    -- quelques traitements ici
END LOOP;
```

39.6.3.5. FOR (variante avec entier)

```
[<<label>>]
FOR nom IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    instruction
END LOOP [ label ];
```

Cette forme de FOR crée une boucle qui effectue une itération sur une plage de valeurs entières. La variable *nom* est automatiquement définie comme un type integer et n'existe que dans la boucle (toute définition de la variable est ignorée à l'intérieur de la boucle). Les deux expressions donnant les limites inférieures et supérieures de la plage sont évaluées une fois en entrant dans la boucle. Si la clause BY n'est pas spécifiée, l'étape d'itération est de 1, sinon elle est de la valeur spécifiée dans la clause BY, qui est évaluée encore une fois à l'entrée de la boucle. Si REVERSE est indiquée, alors la valeur de l'étape est soustraite, plutôt qu'ajoutée, après chaque itération.

Quelques exemples de boucles FOR avec entiers :

```
FOR i IN 1..10 LOOP
    -- prend les valeurs 1,2,3,4,5,6,7,8,9,10 dans la boucle
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- prend les valeurs 10,9,8,7,6,5,4,3,2,1 dans la boucle
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- prend les valeurs 10,8,6,4,2 dans la boucle
END LOOP;
```

Si la limite basse est plus grande que la limite haute (ou moins grande dans le cas du REVERSE), le corps de la boucle n'est pas exécuté du tout. Aucune erreur n'est renvoyée.

Si un *label* est attaché à la boucle FOR, alors la variable entière de boucle peut être référencée avec un nom qualifié en utilisant ce *label*.

39.6.4. Boucler dans les résultats de requêtes

En utilisant un type de FOR différent, vous pouvez itérer au travers des résultats d'une requête et par là-même manipuler ces données. La syntaxe est la suivante :

```
[<<label>>]
FOR cible IN requête LOOP
    instructions
END LOOP [ label ];
```

La *cible* est une variable de type record, row ou une liste de variables scalaires séparées par une virgule. La *cible* est affectée successivement à chaque ligne résultant de la *requête* et le corps de la boucle est exécuté pour chaque ligne. Voici un exemple :

```
CREATE FUNCTION cs_rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    vues_mat RECORD;
BEGIN
    PERFORM cs_log('Rafraichissement des vues matérialisées...');

    FOR vues_mat IN SELECT * FROM cs_vues_materialisees ORDER BY cle_tri LOOP

        -- À présent vues_mat contient un enregistrement de cs_vues_materialisees

            PERFORM cs_log('Rafraichissement de la vue matérialisée '
                           || quote_ident(vues_mat.vm_nom) || '...');
            EXECUTE 'TRUNCATE TABLE  ' || quote_ident(vues_mat.vm_nom);
            EXECUTE 'INSERT INTO '
                    || quote_ident(vues_mat.vm_nom) || ' '
                    || vues_mat.vm_requete;
    END LOOP;
```

```
    PERFORM cs_log('Fin du rafraichissement des vues matérialisées.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Si la boucle est terminée par une instruction `EXIT`, la dernière valeur ligne affectée est toujours accessible après la boucle.

La *`requête`* utilisée dans ce type d'instruction `FOR` peut être toute commande SQL qui renvoie des lignes à l'appelant : **SELECT** est le cas le plus commun mais vous pouvez aussi utiliser **INSERT**, **UPDATE** ou **DELETE** avec une clause `RETURNING`. Certaines commandes comme **EXPLAIN** fonctionnent aussi.

Les variables PL/pgSQL sont substituées dans le texte de la requête et le plan de requête est mis en cache pour une réutilisation possible. C'est couvert en détail dans la Section 39.10.1, « Substitution de variables » et dans la Section 39.10.2, « Mise en cache du plan ».

L'instruction `FOR-IN-EXECUTE` est un moyen d'itérer sur des lignes :

```
[<<label>>]
FOR target IN EXECUTE text_expression [ USING expression [, ...] ] LOOP
    instructions
END LOOP [ label ];
```

Ceci est identique à la forme précédente, à ceci près que l'expression de la requête source est spécifiée comme une expression chaîne, évaluée et replanifiée à chaque entrée dans la boucle `FOR`. Ceci permet au développeur de choisir entre la vitesse d'une requête préplanifiée et la flexibilité d'une requête dynamique, uniquement avec l'instruction **EXECUTE**. Comme avec **EXECUTE**, les valeurs de paramètres peuvent être insérées dans la commande dynamique via `USING`.

Une autre façon de spécifier la requête dont les résultats devront être itérés est de la déclarer comme un curseur. Ceci est décrit dans Section 39.7.4, « Boucler dans les résultats d'un curseur ».

39.6.5. Récupérer les erreurs

Par défaut, toute erreur survenant dans une fonction PL/pgSQL annule l'exécution de la fonction mais aussi de la transaction qui l'entoure. Vous pouvez récupérer les erreurs en utilisant un bloc **BEGIN** avec une clause `EXCEPTION`. La syntaxe est une extension de la syntaxe habituelle pour un bloc **BEGIN** :

```
  [ <<label>> ]
  [ DECLARE
    declarations ]
  BEGIN
  instructions
  EXCEPTION
  WHEN condition [ OR condition ... ] THEN
  instructions_gestion_erreurs
  [ WHEN condition [ OR condition ... ] THEN
    instructions_gestion_erreurs
    ... ]
  END;
```

Si aucune erreur ne survient, cette forme de bloc exécute simplement toutes les *`instructions`* puis passe le contrôle à l'instruction suivant `END`. Mais si une erreur survient à l'intérieur des *`instructions`*, le traitement en cours des *`instructions`* est abandonné et le contrôle est passé à la liste d'`EXCEPTION`. Une recherche est effectuée sur la liste pour la première *`condition`* correspondant à l'erreur survenue. Si une correspondance est trouvée, les *`instructions_gestion_erreurs`* correspondantes sont exécutées puis le contrôle est passé à l'instruction suivant le `END`. Si aucune correspondance n'est trouvée, l'erreur se propage comme si la clause `EXCEPTION` n'existait pas du tout : l'erreur peut être récupérée par un bloc l'enfermant avec `EXCEPTION` ou, s'il n'existe pas, elle annule le traitement de la fonction.

Les noms des *`condition`* sont indiquées dans l'Annexe A, Codes d'erreurs de PostgreSQL™. Un nom de catégorie correspond à toute erreur contenue dans cette catégorie. Le nom de condition spéciale `OTHERS` correspond à tout type d'erreur sauf `QUERY_CANCELED` (il est possible, mais pas recommandé, de récupérer `QUERY_CANCELED` par son nom). Les noms des conditions ne sont pas sensibles à la casse. De plus, une condition d'erreur peut être indiquée par un code `SQLSTATE` ; par exemple, ces deux cas sont équivalents :

```
        WHEN division_by_zero THEN ...
        WHEN SQLSTATE '22012' THEN ...
```

Si une nouvelle erreur survient à l'intérieur des *`instructions_gestion_erreurs`* sélectionnées, elle ne peut pas être récu-

pérée par cette clause `EXCEPTION` mais est propagée en dehors. Une clause `EXCEPTION` l'englobant pourrait la récupérer.

Quand une erreur est récupérée par une clause `EXCEPTION`, les variables locales de la fonction PL/pgSQL restent dans le même état qu'au moment où l'erreur est survenue mais toutes les modifications à l'état persistant de la base de données à l'intérieur du bloc sont annulées. Comme exemple, considérez ce fragment :

```
INSERT INTO mon_tableau(prenom, nom) VALUES('Tom', 'Jones');
BEGIN
  UPDATE mon_tableau SET prenom = 'Joe' WHERE nom = 'Jones';
  x := x + 1;
  y := x / 0;
  EXCEPTION
    WHEN division_by_zero THEN
      RAISE NOTICE 'récupération de l''erreur division_by_zero';
RETURN x;
END;
```

Quand le contrôle parvient à l'affectation de `y`, il échouera avec une erreur `division_by_zero`. Elle sera récupérée par la clause `EXCEPTION`. La valeur renvoyée par l'instruction **RETURN** sera la valeur incrémentée de `x` mais les effets de la commande **UPDATE** auront été annulés. La commande **INSERT** précédant le bloc ne sera pas annulée, du coup le résultat final est que la base de données contient `Tom Jones` et non pas `Joe Jones`.

Astuce

Un bloc contenant une clause `EXCEPTION` est significativement plus coûteuse en entrée et en sortie qu'un bloc sans. Du coup, n'utilisez pas `EXCEPTION` sans besoin.

À l'intérieur d'un gestionnaire d'exceptions, la variable `SQLSTATE` contient le code d'erreur correspondant à l'exception qui a été levée (référez-vous au Tableau A.1, « Codes d'erreur de PostgreSQL™ » pour une liste des codes d'erreurs possibles). La variable `SQLERRM` contient le message d'erreur associé avec l'exception. Ces variables sont indéfinies à l'extérieur des gestionnaires d'exceptions.

Exemple 39.2. Exceptions avec UPDATE/INSERT

Cet exemple utilise un gestionnaire d'exceptions pour réaliser soit un **UPDATE** soit un **INSERT**, comme approprié :

```
CREATE TABLE base (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION fusionne_base(cle INT, donnee TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- commençons par tenter la mise à jour de la clé
        UPDATE base SET b = donnee WHERE a = cle;
        IF found THEN
            RETURN;
        END IF;

        -- si elle n'est pas dispo, tentons l'insertion de la clé
        -- si quelqu'un essaie d'insérer la même clé en même temps,
        -- il y aura une erreur pour violation de clé unique
        BEGIN
            INSERT INTO base(a,b) VALUES (cle, donnee);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- ne rien faire, et tente de nouveau la mise à jour
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT fusionne_base(1, 'david');
SELECT fusionne_base(1, 'dennis');
```

39.7. Curseurs

Plutôt que d'exécuter la totalité d'une requête à la fois, il est possible de créer un *curseur* qui encapsule la requête, puis en lit le résultat quelques lignes à la fois. Une des raisons pour faire de la sorte est d'éviter les surcharges de mémoire quand le résultat contient un grand nombre de lignes (cependant, les utilisateurs PL/pgSQL n'ont généralement pas besoin de se préoccuper de cela puisque les boucles `FOR` utilisent automatiquement un curseur en interne pour éviter les problèmes de mémoire). Un usage plus intéressant est de renvoyer une référence à un curseur qu'une fonction a créé, permettant à l'appelant de lire les lignes. C'est un moyen efficace de renvoyer de grands ensembles de lignes à partir des fonctions.

39.7.1. Déclaration de variables curseur

Tous les accès aux curseurs dans PL/pgSQL se font par les variables curseur, qui sont toujours du type de données spécial refcursor. Un des moyens de créer une variable curseur est de simplement la déclarer comme une variable de type refcursor. Un autre moyen est d'utiliser la syntaxe de déclaration de curseur qui est en général :

```
nom [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR requête;
```

(`FOR` peut être remplacé par `IS` pour la compatibilité avec Oracle™). Si `SCROLL` est spécifié, le curseur sera capable d'aller en sens inverse ; si `NO SCROLL` est indiqué, les récupérations en sens inverses seront rejetées ; si rien n'est indiqué, cela dépend de la requête. *`arguments`* est une liste de paires de *`nom type-de-donnée`* qui définit les noms devant être remplacés par les valeurs des paramètres dans la requête donnée. La valeur effective à substituer pour ces noms sera indiquée plus tard lors de l'ouverture du curseur.

Quelques exemples :

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (cle integer) IS SELECT * FROM tenk1 WHERE unique1 = cle;
```

Ces variables sont toutes trois du type de données refcursor mais la première peut être utilisée avec n'importe quelle requête alors que la seconde a une requête complètement spécifiée qui lui est déjà *liée*, et la dernière est liée à une requête paramétrée (`cle` sera remplacée par un paramètre de valeur entière lors de l'ouverture du curseur). La variable `curs1` est dite *non liée* puisqu'elle n'est pas liée à une requête particulière.

39.7.2. Ouverture de curseurs

Avant qu'un curseur puisse être utilisé pour rapatrier des lignes, il doit être *ouvert* (c'est l'action équivalente de la commande SQL **DECLARE CURSOR**). PL/pgSQL dispose de trois formes pour l'instruction **OPEN**, dont deux utilisent des variables curseur non liées et la dernière une variable curseur liée.

> **Note**
>
> Les variables des curseurs liés peuvent aussi être utilisés sans les ouvrir explicitement, via l'instruction **FOR** décrite dans Section 39.7.4, « Boucler dans les résultats d'un curseur ».

39.7.2.1. OPEN FOR *`requête`*

```
OPEN var_curseur_nonlie [ [ NO ] SCROLL ] FOR requete;
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert, et il doit avoir été déclaré comme une variable de curseur non lié (c'est-à-dire comme une simple variable refcursor). La requête doit être un **SELECT** ou quelque chose d'autre qui renvoie des lignes (comme **EXPLAIN**). La requête est traitée de la même façon que les autres commandes SQL dans PL/pgSQL : les noms de variables PL/pgSQL sont substitués et le plan de requête est mis en cache pour une possible ré-utilisation. Quand une variable PL/pgSQL est substituée dans une requête de type curseur, la valeur qui est substituée est celle qu'elle avait au moment du **OPEN** ; les modifications ultérieures n'auront pas affectées le comportement du curseur. Les options `SCROLL` et `NO SCROLL` ont la même signification que pour un curseur lié.

Exemple :

```
OPEN curs1 FOR SELECT * FROM foo WHERE cle = ma_cle;
```

39.7.2.2. OPEN FOR EXECUTE

```
OPEN var_curseur_nonlie [ [ NO ] SCROLL ] FOR EXECUTE requete
                                        [ USING expression [, ... ] ];
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert et il doit avoir été

déclaré comme une variable de curseur non lié (c'est-à-dire comme une simple variable refcursor). La requête est spécifiée comme une expression chaîne de la même façon que dans une commande **EXECUTE**. Comme d'habitude, ceci donne assez de flexibilité pour que le plan de la requête puisse changer d'une exécution à l'autre (voir la Section 39.10.2, « Mise en cache du plan »), et cela signifie aussi que la substitution de variable n'est pas faite sur la chaîne de commande. Comme avec la commande **EXECUTE**, les valeurs de paramètre peuvent être insérées dans la commande dynamique avec `USING`. Les options `SCROLL` et `NO SCROLL` ont la même signification que pour un curseur lié.

Exemple :

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
                                        || ' WHERE col1 = $1' USING keyvalue;
```

Dans cet exemple, le nom de la table est insérée dans la requête textuellement et l'utilisation de `quote_ident()` est recommandé afin de se prémunir contre des injections SQL. La valeur de comparaison pour `col1` esr insérée avec la paramètre `USING` et n'a donc pas besoin d'être protégé.

39.7.2.3. Ouverture d'un curseur lié

```
OPEN var_curseur_lié [ ( arguments ) ];
```

Cette forme d'**OPEN** est utilisée pour ouvrir une variable curseur à laquelle la requête est liée au moment de la déclaration. Le curseur ne peut pas être déjà ouvert. Une liste des expressions arguments doit apparaître si et seulement si le curseur a été déclaré comme acceptant des arguments. Ces valeurs seront remplacées dans la requête. Le plan de requête pour un curseur lié est toujours considéré comme pouvant être mis en cache ; il n'y a pas d'équivalent de la commande **EXECUTE** dans ce cas. Notez que `SCROLL` et `NO SCROLL` ne peuvent pas être indiqués car le comportement du curseur était déjà déterminé.

Notez que, parce que la substitution de variables est faite sur la requête du curseur lié, il y a deux façon de passer des valeurs au curseur : soit explicitement avec un argument pour **OPEN** soit implicitement en référençant une variable PL/pgSQL dans la requête. Néanmoins, seules les variables déclarées avant la déclaration du curseur lié pourront être substituées. De toute façon, la valeur à passer est déterminé au moment de l'exécution de **OPEN**.

Exemples :

```
OPEN curs2;
OPEN curs3(42);
```

39.7.3. Utilisation des curseurs

Une fois qu'un curseur a été ouvert, il peut être manipulé grâce aux instructions décrites ci-dessous.

Ces manipulations n'ont pas besoin de se dérouler dans la même fonction que celle qui a ouvert le curseur. Vous pouvez renvoyer une valeur refcursor à partir d'une fonction et laisser l'appelant opérer sur le curseur (d'un point de vue interne, une valeur refcursor est simplement la chaîne de caractères du nom d'un portail contenant la requête active pour le curseur. Ce nom peut être passé à d'autres, affecté à d'autres variables refcursor et ainsi de suite, sans déranger le portail).

Tous les portails sont implicitement fermés à la fin de la transaction. C'est pourquoi une valeur refcursor est utilisable pour référencer un curseur ouvert seulement jusqu'à la fin de la transaction.

39.7.3.1. `FETCH`

```
FETCH [ direction { FROM | IN } ] curseur INTO cible;
```

FETCH récupère la prochaine ligne à partir d'un curseur et la place dans une cible, qui peut être une variable ligne, une variable record ou une liste de variables simples séparées par des virgules, comme dans un **SELECT INTO**. S'il n'y a pas de ligne suivante, la cible est mise à NULL. Comme avec **SELECT INTO**, la variable spéciale `FOUND` peut être lue pour voir si une ligne a été récupérée.

La clause *`direction`* peut être une des variantes suivantes autorisées pour la commande SQL FETCH(7) sauf celles qui peuvent récupérer plus d'une ligne ; nommément, cela peut être `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *`nombre`*, `RELATIVE` *`nombre`*, `FORWARD` ou `BACKWARD`. Omettre *`direction`* est identique à spécifier `NEXT`. Les valeurs *`direction`* qui nécessitent d'aller en sens inverse risquent d'échouer sauf si le curseur a été déclaré ou ouvert avec l'option `SCROLL`.

`curseur` doit être le nom d'une variable refcursor qui référence un portail de curseur ouvert.

Exemples :

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
```

```
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

39.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] curseur;
```

MOVE repositionne un curseur sans récupérer de données. **MOVE** fonctionne exactement comme la commande **FETCH** sauf qu'elle ne fait que repositionner le curseur et ne renvoie donc pas les lignes du déplacement. Comme avec **SELECT INTO**, la variable spéciale `FOUND` peut être lue pour vérifier s'il y avait bien les lignes correspondant au déplacement.

La clause de *`direction`* peut être l'une des variantes autorisées dna sla commande SQL FETCH(7), nommément `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *`nombre`*, `RELATIVE` *`nombre`*, `ALL`, `FORWARD` [*`nombre`* | `ALL`], ou `BACKWARD` [*`nombre`* | `ALL`]. Omettre *`direction`* est identique à spécifier `NEXT`. Les valeurs *`direction`* qui nécessitent de se déplacer en arrière risquent d'échouer sauf si le curseur a été déclaré ou ouvert avec l'option `SCROLL`.

Exemples :

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

39.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF curseur;
DELETE FROM table WHERE CURRENT OF curseur;
```

Quand un curseur est positionné sur une ligne d'une table, cette ligne peut être mise à jour ou supprimée en utilisant le curseur qui identifie la ligne. Il existe des restrictions sur ce que peut être la requête du curseur (en particulier, pas de regroupement) et il est mieux d'utiliser `FOR UPDATE` dans le curseur. Pour des informations supplémentaires, voir la page de référence DECLARE(7).

Un exemple :

```
UPDATE foo SET valdonnee = mavaleur WHERE CURRENT OF curs1;
```

39.7.3.4. CLOSE

```
CLOSE curseur;
```

CLOSE ferme le portail sous-tendant un curseur ouvert. Ceci peut être utilisé pour libérer des ressources avant la fin de la transaction ou pour libérer la variable curseur pour pouvoir la réouvrir.

Exemple :

```
CLOSE curs1;
```

39.7.3.5. Renvoi de curseurs

Les fonctions PL/pgSQL peuvent renvoyer des curseurs à l'appelant. Ceci est utile pour renvoyer plusieurs lignes ou colonnes, spécialement avec des ensembles de résultats très grands. Pour cela, la fonction ouvre le curseur et renvoie le nom du curseur à l'appelant (ou simplement ouvre le curseur en utilisant un nom de portail spécifié par ou autrement connu par l'appelant). L'appelant peut alors récupérer les lignes à partir du curseur. Le curseur peut être fermé par l'appelant ou il sera fermé automatiquement à la fin de la transaction.

Le nom du portail utilisé pour un curseur peut être spécifié par le développeur ou peut être généré automatiquement. Pour spécifier un nom de portail, affectez simplement une chaîne à la variable refcursor avant de l'ouvrir. La valeur de la variable refcursor sera utilisée par **OPEN** comme nom du portail sous-jacent. Néanmoins, si la variable refcursor est NULL, **OPEN** génère automatiquement un nom qui n'entre pas en conflit avec tout portail existant et l'affecte à la variable refcursor.

Note

> Une variable curseur avec limites est initialisée avec la valeur de la chaîne représentant son nom, de façon à ce que le nom du portail soit identique au nom de la variable curseur, sauf si le développeur le surcharge par affectation avant d'ouvrir le curseur. Mais, une variable curseur sans limite aura par défaut la valeur NULL, dont il reçoit un nom unique généré automatiquement sauf s'il est surchargé.

L'exemple suivant montre une façon de fournir un nom de curseur par l'appelant :

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION fonction_reference(refcursor) RETURNS refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
$$ LANGUAGE plpgsql;

BEGIN;
SELECT fonction_reference('curseur_fonction');
FETCH ALL IN curseur_fonction;
COMMIT;
```

L'exemple suivant utilise la génération automatique du nom du curseur :

```
CREATE FUNCTION fonction_reference2() RETURNS refcursor AS $$
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
$$ LANGUAGE plpgsql;

-- Il faut être dans une transaction pour utiliser les curseurs.
BEGIN;
SELECT fonction_reference2();

   fonction_reference2
---------------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

L'exemple suivant montre une façon de renvoyer plusieurs curseurs à une seule fonction :

```
CREATE FUNCTION ma_fonction(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- doit être dans une transaction pour utiliser les curseurs.
BEGIN;

SELECT * FROM ma_fonction('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

39.7.4. Boucler dans les résultats d'un curseur

C'est une variante de l'instruction **FOR** qui permet l'itération sur les lignes renvoyées par un curseur. La syntaxe est :

```
[ <<label>> ]
FOR var_record IN var_curseur_lié [ ( valeurs_argument ) ] LOOP
    instructions
END LOOP [ label ];
```

La variable curseur doit avoir été liée à une requête lors de sa déclaration et il *ne peut pas* être déjà ouvert. L'instruction **FOR** ouvre automatiquement le curseur, et il ferme le curseur en sortie de la boucle. Une liste des expressions de valeurs des arguments doit apparaître si et seulement si le curseur a été déclaré prendre des arguments. Ces valeurs seront substitutées dans la requête, de la même façon que lors d'un **OPEN**. La variable variable *var_record* est définie automatiquement avec le type record et existe seulement dans la boucle (toute définition existante d'un nom de variable est ignorée dans la boucle). Chaque ligne renvoyée par le curseur est successivement affectée à la variable d'enregistrement et le corps de la boucle est exécuté.

39.8. Erreurs et messages

Utilisez l'instruction **RAISE** pour rapporter des messages et lever des erreurs.

```
RAISE [ niveau ] 'format' [, expression [, ...]] [ USING option = expression [, ... ]
];
RAISE [ niveau ] nom_condition [ USING option = expression [, ... ] ];
RAISE [ niveau ] SQLSTATE 'état_sql' [ USING option = expression [, ... ] ];
RAISE [ niveau ] USING option = expression [, ... ];
RAISE ;
```

L'option *niveau* indique la sévérité de l'erreur. Les niveaux autorisés sont DEBUG, LOG, INFO, NOTICE, WARNING et EXCEPTION, ce dernier étant la valeur par défaut. EXCEPTION lève une erreur (ce qui annule habituellement la transaction en cours). Les autres niveaux ne font que générer des messages aux différents niveaux de priorité. Les variables de configuration log_min_messages et client_min_messages contrôlent l'envoi de messages dans les traces, au client ou aux deux. Voir le Chapitre 18, Configuration du serveur pour plus d'informations.

Après *niveau*, vous pouvez écrire un *format* (qui doit être une chaîne litérale, pas une expression). La chaîne format indique le texte du message d'erreur à rapporter. Elle peut être suivie par des expressions optionnelles à insérer dans le message. Dans la chaîne, % est remplacé par la représentation de la valeur du prochain argument. Écrivez %% pour saisir un % litéral.

Dans cet exemple, la valeur de v_job_id remplace le % dans la chaîne.

```
RAISE NOTICE 'Appel de cs_creer_job(%)', v_job_id;
```

Vous pouvez attacher des informations supplémentaires au rapport d'erreur en écrivant USING suivi par des éléments *option* = *expression*. Les mots clés autorisées pour *option* sont MESSAGE, DETAIL, HINT et ERRCODE, alors que chaque *expression* peut être une expression de type chaîne. MESSAGE configure le texte de l'erreur (cette option ne peut pas être utilisée sous la forme de **RAISE** qui inclut une chaîne de format avant USING). DETAIL fournit un message détaillé de l'erreur, HINT propose une astuce. ERRCODE indique le code d'erreur (SQLSTATE) à rapporter, soit par le nom de la condition comme indiquée dans Annexe A, Codes d'erreurs de PostgreSQL™, soit directement sous la forme d'un code SQLSTATE à cinq caractères.

Cet exemple annulera la transaction avec le message d'erreur et l'astuce donnés :

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user id';
```

Ces deux exemples affichent des façons équivalents pour initialiser SQLSTATE :

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

Il existe une deuxième syntaxe **RAISE** pour laquelle l'argument principale est le nom de la condition ou le SQLSTATE à rapporter, par exemple :

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Dans cette syntaxe, USING peut être utilisé pour fournir un message d'erreur, un détail ou une astuce personnalisé. Voici une autre façon de faire l'exemple précédent :

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

Une autre variante est d'écrire `RAISE USING` ou `RAISE niveau USING` et de placer tout le reste dans la liste `USING`.

La dernière variante de **RAISE** n'a aucun paramètre. Cette forme peut seulement être utilisée dans un bloc `BEGIN` d'une clause `EXCEPTION` ; cela fait que l'erreur est renvoyée au prochain bloc englobant.

Si aucun nom de condition ou SQLSTATE n'est indiqué dans une commande **RAISE EXCEPTION**, la valeur par défaut est d'utiliser `RAISE_EXCEPTION` (`P0001`). Si aucun message texte n'est indiqué, la valeur par défaut est d'utiliser le nom de la condition ou le SQLSTATE comme texte de message.

> **Note**
>
> Lors de la spécification du code d'erreur par un code SQLSTATE, vous n'êtes pas limité aux codes d'erreur prédéfinis, mais pouvez sélectionner tout code d'erreur consistant en cinq chiffres et/ou des lettres ASCII majuscules, autre que `00000`. Il est recommandé d'éviter d'envoyer des codes d'erreur qui se terminent avec trois zéros car il y a des codes de catégorie, et peuvent seulement être récupérés en filtrant la catégorie complète.

39.9. Procédures trigger

PL/pgSQL peut être utilisé pour définir des procédures trigger. Une procédure trigger est créée grâce à la commande **CREATE FUNCTION** utilisée comme fonction sans arguments ayant un type de retour trigger. Notez que la fonction doit être déclarée avec aucun argument même si elle s'attend à recevoir les arguments spécifiés dans **CREATE TRIGGER** -- les arguments trigger sont passés via `TG_ARGV`, comme décrit plus loin.

Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Ce sont :

`NEW`
: Type de données RECORD ; variable contenant la nouvelle ligne de base de données pour les opérations **INSERT**/**UPDATE** dans les triggers de niveau ligne. Cette variable est NULL dans un trigger de niveau instruction et pour les opérations **DELETE**.

`OLD`
: Type de données RECORD ; variable contenant l'ancienne ligne de base de données pour les opérations **UPDATE**/**DELETE** dans les triggers de niveau ligne. Cette variable est NULL dans les triggers de niveau instruction et pour les opérations **INSERT**.

`TG_NAME`
: Type de données name ; variable qui contient le nom du trigger réellement lancé.

`TG_WHEN`
: Type de données text ; une chaîne, soit `BEFORE` soit `AFTER`, selon la définition du trigger.

`TG_LEVEL`
: Type de données text ; une chaîne, soit `ROW` soit `STATEMENT`, selon la définition du trigger.

`TG_OP`
: Type de données text ; une chaîne, `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE` indiquant pour quelle opération le trigger a été lancé.

`TG_RELID`
: Type de données oid ; l'ID de l'objet de la table qui a causé le déclenchement du trigger.

`TG_RELNAME`
: Type de données name ; le nom de la table qui a causé le déclenchement. C'est obsolète et pourrait disparaître dans une prochaine version. À la place, utilisez `TG_TABLE_NAME`.

`TG_TABLE_NAME`
: Type de données name ; le nom de la table qui a déclenché le trigger.

`TG_TABLE_SCHEMA`
: Type de données name ; le nom du schéma de la table qui a appelé le trigger.

`TG_NARGS`
: Type de données integer ; le nombre d'arguments donnés à la procédure trigger dans l'instruction **CREATE TRIGGER**.

`TG_ARGV[]`
: Type de donnée text ; les arguments de l'instruction **CREATE TRIGGER**. L'index débute à 0. Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à `tg_nargs`) auront une valeur NULL.

Une fonction trigger doit renvoyer soit NULL soit une valeur record ayant exactement la structure de la table pour laquelle le trigger a été lancé.

Les triggers de niveau ligne lancés `BEFORE` peuvent renvoyer NULL pour indiquer au gestionnaire de trigger de sauter le reste de l'opération pour cette ligne (les triggers suivants ne sont pas lancés, et les **INSERT**/**UPDATE**/**DELETE** ne se font pas pour cette ligne). Si une valeur non NULL est renvoyée alors l'opération se déroule avec cette valeur ligne. Renvoyer une valeur ligne différente de la valeur originale de `NEW` modifie la ligne qui sera insérée ou mise à jour. De ce fait, si la fonction de trigger veut que l'action réussise sans modifier la valeur de rangée, `NEW` (ou une valeur égale) doit être renvoyée. Pour modifier la rangée à être stockée, il est possible de remplacer les valeurs directement dans `NEW` et renvoyer le `NEW` modifié ou de générer un nouvel enregistrement à renvoyer. Dans le cas d'un before-trigger sur une commande **DELETE**, la valeur renvoyée n'a aucun effet direct mais doit être non-nulle pour permettre à l'action trigger de continuer. Notez que `NEW` est nul dans le cadre des triggers **DELETE** et que renvoyer ceci n'est pas recommandé dans les cas courants. Une pratique utile dans des triggers **DELETE** serait de renvoyer `OLD`.

La valeur de retour d'un trigger de niveau rangée déclenché `AFTER` ou un trigger de niveau instruction déclenché `BEFORE` ou `AFTER` est toujours ignoré ; il pourrait aussi bien être NULL. Néanmoins, tous les types de triggers peuvent toujours annuler l'opération complète en envoyant une erreur.

L'Exemple 39.3, « Une procédure trigger PL/pgSQL » montre un exemple d'une procédure trigger dans PL/pgSQL.

Exemple 39.3. Une procédure trigger PL/pgSQL

Cet exemple de trigger assure qu'à chaque moment où une ligne est insérée ou mise à jour dans la table, le nom de l'utilisateur courant et l'heure sont estampillés dans la ligne. Et cela vous assure qu'un nom d'employé est donné et que le salaire est une valeur positive.

```
CREATE TABLE emp (
    nom_employe text,
    salaire integer,
    date_dermodif timestamp,
    utilisateur_dermodif text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Verifie que nom_employe et salary sont donnés
        IF NEW.nom_employe IS NULL THEN
            RAISE EXCEPTION 'nom_employe ne peut pas être NULL';
        END IF;
        IF NEW.salaire IS NULL THEN
            RAISE EXCEPTION '% ne peut pas avoir un salaire', NEW.nom_employe;
        END IF;

        -- Qui travaille pour nous si la personne doit payer pour cela ?
        IF NEW.salaire < 0 THEN
            RAISE EXCEPTION '% ne peut pas avoir un salaire négatif', NEW.nom_employe;
        END IF;

        -- Rappelons-nous qui a changé le salaire et quand
        NEW.date_dermodif := current_timestamp;
        NEW.utilisateur_dermodif := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Une autre façon de tracer les modifications sur une table implique la création d'une nouvelle table qui contient une ligne pour chaque insertion, mise à jour ou suppression qui survient. Cette approche peut être vue comme un audit des modifications sur une table. L'Exemple 39.4, « Une procédure d'audit par trigger en PL/pgSQL » montre un exemple d'une procédure d'audit par trigger en PL/pgSQL.

Exemple 39.4. Une procédure d'audit par trigger en PL/pgSQL

Cet exemple de trigger nous assure que toute insertion, modification ou suppression d'une ligne dans la table `emp` est enregistrée dans la table `emp_audit`. L'heure et le nom de l'utilisateur sont conservées dans la ligne avec le type d'opération réalisé.

```
CREATE TABLE emp (
    nom_employe       text NOT NULL,
    salaire           integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    tampon            timestamp NOT NULL,
    id_utilisateur    text      NOT NULL,
    nom_employe       text      NOT NULL,
    salaire           integer
);

CREATE OR REPLACE FUNCTION audit_employe() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Ajoute une ligne dans emp_audit pour refléter l'opération réalisée
    -- sur emp,
    -- utilise la variable spéciale TG_OP pour cette opération.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- le résultat est ignoré car il s'agit d'un trigger AFTER
END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE audit_employe();
```

Une utilisation des triggers est le maintien d'une table résumée d'une autre table. Le résumé résultant peut être utilisé à la place de la table originale pour certaines requêtes -- souvent avec des temps d'exécution bien réduits. Cette technique est souvent utilisée pour les statistiques de données où les tables de données mesurées ou observées (appelées des tables de faits) peuvent être extrêmement grandes. L'Exemple 39.5, « Une procédure trigger PL/pgSQL pour maintenir une table résumée » montre un exemple d'une procédure trigger en PL/pgSQL maintenant une table résumée pour une table de faits dans un système de données (data warehouse).

Exemple 39.5. Une procédure trigger PL/pgSQL pour maintenir une table résumée

Le schéma détaillé ici est partiellement basé sur l'exemple du *Grocery Store* provenant de *The Data Warehouse Toolkit* par Ralph Kimball.

```
--
-- Tables principales - dimension du temps de ventes.
--
CREATE TABLE time_dimension (
  time_key                    integer NOT NULL,
  day_of_week                 integer NOT NULL,
  day_of_month                integer NOT NULL,
  month                       integer NOT NULL,
  quarter                     integer NOT NULL,
  year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
```

```
  time_key                    integer NOT NULL,
  product_key                 integer NOT NULL,
  store_key                   integer NOT NULL,
  amount_sold                 numeric(12,2) NOT NULL,
  units_sold                  integer NOT NULL,
  amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Table résumé - ventes sur le temps.
--
CREATE TABLE sales_summary_bytime (
  time_key                    integer NOT NULL,
  amount_sold                 numeric(15,2) NOT NULL,
  units_sold                  numeric(12) NOT NULL,
  amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Fonction et trigger pour amender les colonnes résumées
-- pour un UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
DECLARE
  delta_time_key          integer;
  delta_amount_sold       numeric(15,2);
  delta_units_sold        numeric(12);
  delta_amount_cost       numeric(15,2);
BEGIN

  -- Travaille sur l'ajout/la suppression de montant(s).
  IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
    delta_units_sold = -1 * OLD.units_sold;
    delta_amount_cost = -1 * OLD.amount_cost;

  ELSIF (TG_OP = 'UPDATE') THEN

    -- interdit les mises à jour qui modifient time_key -
    -- (probablement pas trop cher, car DELETE + INSERT est la façon la plus
    -- probable de réaliser les modifications).
    IF ( OLD.time_key != NEW.time_key) THEN
      RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                      OLD.time_key, NEW.time_key;
    END IF;

    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

  ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;

  END IF;


  -- Insertion ou mise à jour de la ligne de résumé avec les nouvelles valeurs.
  <<insert_update>>
  LOOP
```

```
  UPDATE sales_summary_bytime
  SET amount_sold = amount_sold + delta_amount_sold,
    units_sold = units_sold + delta_units_sold,
    amount_cost = amount_cost + delta_amount_cost
    WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;

    BEGIN
      INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost)
        VALUES (
        delta_time_key,
        delta_amount_sold,
        delta_units_sold,
        delta_amount_cost
        );
      EXIT insert_update;

      EXCEPTION
      WHEN UNIQUE_VIOLATION THEN
      -- do nothing
      END;
      END LOOP insert_update;

  RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
  AFTER INSERT OR UPDATE OR DELETE ON sales_fact
  FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

39.10. Les dessous de PL/pgSQL

Cette section discute des détails d'implémentation les plus importants à connaître pour les utilisateurs de PL/pgSQL.

39.10.1. Substitution de variables

Les instructions et expressions SQL au sein d'une fonction PL/pgSQL peuvent faire appel aux variables et paramètres d'une fonction. En coulisses, PL/pgSQL remplace les paramètres de requêtes par des références. Les paramètres ne seront remplacés qu'aux endroits où un paramètre ou une référence de colonne sont autorisés par la syntaxe. Pour un cas extrême, considerez cet exemple de mauvaise programmation :

```
        INSERT INTO foo (foo) VALUES (foo);
```

La première occurrence de `foo` doit être un nom de table, d'après la syntaxe et ne sera donc pas remplacée, même si la fonction a une variable nommée `foo`. La deuxième occurrence doit être le nom d'une colonne de la table et ne sera donc pas remplacée non plus. Seule la troisième occurrence peuvent être une référence à la variable de la fonction.

Note

> Les versions de PostgreSQL™ avant la 9.0 remplaçaient la variable dans les trois cas, donnant lieu à des erreurs de syntaxe.

Les noms de variables n'étant pas différents des noms de colonnes, d'après la syntaxe, il peut y avoir ambuiguité dans les instructions qui font référence aux deux : un nom donné fait-il référence à un nom de colonne ou à une variable ? Modifions l'exemple précédent.

```
        INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Ici, `dest` et `src` doivent être des noms de table et `col` doit être une colonne de `dest` mais `foo` et `bar` peuvent être aussi bien des variables de la fonction que des colonnes de `src`.

Par défait, PL/pgSQL signalera une erreur si un nom dans une requête SQL peut faire référence à la fois à une variable et à une colonne. Vous pouvez corriger ce problème en renommant la variable ou colonne, en qualifiant la référence ambigüe ou en précisant à PL/pgSQL quelle est l'interpretation à privilégier.

Le choix le plus simple est de renommer la variable ou colonne. Une règle de codage récurrente est d'utiliser une convention de nommage différente pour les variables de PL/pgSQL que pour les noms de colonne. Par exemple, si vous utilisez toujours des variables de fonctions en `v_quelquechose` tout en vous assurant qu'aucun nom de colonne ne commence par `v_`, aucun conflit ne sera possible.

Autrement, vous pouvez qualifier les références ambigües pour les rendre plus claires. Dans l'exemple ci-dessus, `src.foo` serait une référence sans amigüité à une colonne de table. Pour créer une référence sans amigüité à une variable, déclarez-la dans un bloc nommé et utilisez le nom du bloc (voir Section 39.2, « Structure de PL/pgSQL »). Par exemple,

```
    <<bloc>>
    DECLARE
        foo int;
    BEGIN
        foo := ...;
        INSERT INTO dest (col) SELECT bloc.foo + bar FROM src;
```

Ici, `bloc.foo` désigne la variable même s'il existe une colonne `foo` dans la base `src`. Les paramètres de fonction, ainsi que les variables spéciales tel que `FOUND`, peuvent être qualifiés par le nom de la fonction, parce qu'ils sont implicitement déclarés dans un bloc exterieur portant le nom de la fonction.

Quelque fois, il n'est pas envisageable de lever toutes les ambigüités dans une grande quantité de code PL/pgSQL. Dans ces cas-ci, vous pouvez spécifier à PL/pgSQL qu'il doit traiter les références ambigües comme étant une variable (ce qui est compatible avec le comportement de PL/pgSQL avant PostgreSQL™ 9.0) ou comme étant la colonne d'une table (ce qui est compatible avec d'autres systèmes tels que Oracle™).

Pour modifier ce comportement dans toute l'instance, mettez le paramètre de configuration `plpgsql.variable_conflict` à l'un de `error`, `use_variable` ou `use_column` (où `error` est la valeur par défaut). Ce paramètre agit sur les compilations posterieures d'instructions dans les fonctions PL/pgSQL mais pas les instructions déjà compilées dans la session en cours. Pour fixer ce paramètre avant le chargement de PL/pgSQL, vous devez ajouter « `plpgsql` » à la liste custom_variable_classes dans `postgresql.conf`. Cette modification pouvant affecter de manière inattendue le comportement des fonctions PL/pgSQL, elle ne peut être faïte que par un administrateur.

Vous pouvez modifier ce comportement fonction par fonction, en insérant l'une de ces commandes spéciales au début de la fonction :

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

Ces commandes n'agissent que sur les fonctions qui les contient et surchargent la valeur de `plpgsql.variable_conflict`. Un exemple est

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
```

```
$$ LANGUAGE plpgsql;
```

Dans la commande `UPDATE`, `curtime`, `comment`, et `id` font référence aux variables et paramètres de la fonction, que la table `users` ait ou non des colonnes portant ces noms. Notez qu'il a fallu qualifier la référence à `users.id` dans la clause `WHERE` pour qu'elle fasse référence à la colonne. Mais nous ne qualifions pas la référence à `comment` comme cible dans la liste `UPDATE` car, d'après la syntaxe, elle doit être une colonne de `users`. Nous pourrions écrire la même fonction sans dépendre de la valeur de `variable_conflict` de cette manière :

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

La substitution de variable n'arrive pas dans la chaîne de commande donnée à **EXECUTE** ou une de ces variantes. Si vous avez besoin d'insérer une valeur dans une telle commande, faites-le lors de la construction d'une valeur de chaîne, illustrée dans la Section 39.5.4, « Exécuter des commandes dynamiques », ou utilisez `USING`.

La substitution de variable fonctionne seulement dans les commandes **SELECT**, **INSERT**, **UPDATE** et **DELETE** parce que le moteur SQL principal autorise les paramètres de la requête seulement dans ces commandes. Pour utiliser un nom variable ou une valeur dans les autres types d'instructions (généralement appelées des instructions utilitaires), vous devez construire l'instruction en question comme une chaîne et l'exécuter via **EXECUTE**.

39.10.2. Mise en cache du plan

L'interpréteur PL/pgSQL analyse le source d'une fonction et produit un arbre binaire interne d'instructions la première fois que la fonction est appelée (à l'intérieur de chaque session). L'arbre des instructions se traduit complètement par la structure d'instructions PL/pgSQL mais les expressions et les commandes SQL individuelles utilisées dans la fonction ne sont pas traduites immédiatement.

Au moment où chaque expression et commande SQL est exécutée en premier lieu dans la fonction, l'interpréteur PL/pgSQL crée un plan d'exécution préparée (en utilisant les fonctions `SPI_prepare` et `SPI_saveplan` du gestionnaire SPI). Les appels suivants à cette expression ou commande réutilisent le plan préparé. Du coup, une fonction avec du code conditionnel contenant de nombreuses instructions pour lesquels les plans d'exécutions pourraient être requis vont seulement préparer et sauvegarder ces plans qui sont utilisés tout au long de la durée de vie de la connexion à la base de données. Ceci peut réduire substantiellement le temps total requis pour analyser et générer les plans d'exécution pour les instructions d'une fonction PL/pgSQL. Un inconvénient est que les erreurs dans une expression ou commande spécifique ne peuvent pas être détectées avant que la fonction a atteint son exécution. (Les erreurs de syntaxe triviales seront détectées à la première passe d'analyse mais quelque chose de plus complexe ne sera pas détecté avant son exécution.)

Un plan sauvegardé sera regénéré automatiquement s'il y a des changements dans le schéma de n'importe quelle table utilisée dans la requête ou si une fonction utilisée dans la requête et définie par un utilisateur est redefinie Ceci rend la re-utilisation d'un plan préparé transparent dans la plupart des cas mais il existe des cas particuliers ou un plan obsolète peut être re-utilisé. Par exemple, la suppression et la re-création d'un opérateur defini par un utilisateur n'affectera pas les plans déjà en cache; ils continueront d'appeler la fonction sous-jacente de l'opérateur d'origine, si celle-ci n'a pas été modifiée. Lorsque c'est nécessaure, le cache peut être vidé en commençant une nouvelle session.

Comme PL/pgSQL sauvegarde les plans d'exécution de cette façon, les commandes SQL qui apparaissent directement dans une fonction PL/pgSQL doivent faire référence aux même tables et aux mêmes colonnes à chaque exécution ; c'est-à-dire que vous ne pouvez pas utiliser un paramètre comme le nom d'une table ou d'une colonne dans une commande SQL. Pour contourner cette restriction, vous pouvez construire des commandes dynamiques en utilisant l'instruction **EXECUTE** de PL/pgSQL -- au prix de la construction d'un nouveau plan d'exécution à chaque exécution.

Un autre point important est que les plans préparés utilisent des paramètres pour permettre le changement des valeurs des variables PL/pgSQL entre chaque exécution, comme indiqué en détail ci-dessus. Quelque fois, cela signifie qu'un plan est moins efficace que s'il avait été généré avec une valeur spécifique. Comme exemple, regardez :

```
SELECT * INTO mon_enregistrement FROM dictionnaire WHERE mot LIKE terme_recherche;
```

où `terme_recherche` est une variable PL/pgSQL. Le plan caché pour cette requête n'utilisera jamais un index sur une variable. Le plan caché pour cette requête n'utilisera jamais un index sur un *`mot`* car le planificateur ne peut pas savoir si le modèle `LIKE` comprendra une ancre à gauche à l'exécution. Pour utiliser un index, la requête doit être planifiée avec un modèle spécifique

pour `LIKE`. C'est un autre cas où **EXECUTE** peut être utilisé pour forcer la génération d'un nouveau plan à chaque exécution.

La nature muable des variables de type record présente un autre problème dans cette connexion. Quand les champs d'une variable record sont utilisés dans les expressions ou instructions, les types de données des champs ne doivent pas modifier d'un appel de la fonction à un autre car chaque expression sera planifiée en utilisant le type de données qui est présent quand l'expression est atteinte en premier. **EXECUTE** peut être utilisé pour contourner ce problème si nécessaire.

Si la même fonction est utilisée comme trigger pour plus d'une table, PL/pgSQL prépare et met en cache les plans indépendament pour chacune de ses tables -- c'est-à-dire qu'il y a un cache pour chaque combinaison fonction trigger/table, pas uniquement pour chaque fonction. Ceci diminue certains des problèmes avec les types de données variables ; par exemple, une fonction trigger pourra fonctionner correctement avec une colonne nommée `cle` même si cette colonne a différents types dans différentes tables.

De la même façon, les fonctions ayant des types polymorphiques pour les arguments ont un cache séparé des plans pour chaque combinaison des types d'argument réél avec lesquels elles ont été appelées, donc les différences de type de données ne causent pas d'échecs inattendus.

La mise en cache du plan peut parfois avoir des effets surprenants sur l'interprétation des valeurs sensibles à l'heure. Par exemple, il y a une différence entre ce que font ces deux fonctions :

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

et :

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

Dans le cas de `logfunc1`, l'analyseur principal de PostgreSQL™ sait que, au moment de la préparation du plan pour **INSERT**, la chaîne `'now'` devra être interprétée comme une valeur de type timestamp car la colonne cible de `logtable` est de ce type. Du coup, `'now'` sera converti en une constante quand **INSERT** est planifié, puis utilisé dans tous les appels à `logfunc1` lors de la durée de vie de la session. Il n'est pas nécessaire de préciser que ce n'est pas le souhait du développeur.

Dans le cas de `logfunc2`, l'analyseur principal de PostgreSQL™ ne connaît pas le type que deviendra `'now'` et, du coup, il renvoie une valeur de type text contenant la chaîne `now`. Lors de l'affectation à la variable `curtime` locale, l'interpréteur PL/pgSQL convertie cette chaîne dans le type timestamp en appelant les fonctions `text_out` et `timestamp_in` pour la conversion. Du coup, l'heure calculée est mise à jour à chaque exécution comme le suppose le développeur.

39.11. Astuces pour développer en PL/pgSQL

Un bon moyen de développer en PL/pgSQL est d'utiliser l'éditeur de texte de votre choix pour créer vos fonctions, et d'utiliser psql dans une autre fenêtre pour charger et tester ces fonctions. Si vous procédez ainsi, une bonne idée est d'écrire la fonction en utilisant **CREATE OR REPLACE FUNCTION**. De cette façon vous pouvez simplement recharger le fichier pour mettre à jour la définition de la fonction. Par exemple :

```
CREATE OR REPLACE FUNCTION fonction_test(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

Pendant que psql s'exécute, vous pouvez charger ou recharger des définitions de fonction avec :

```
\i nom_fichier.sql
```

puis immédiatement soumettre des commandes SQL pour tester la fonction.

Un autre bon moyen de développer en PL/pgSQL est d'utiliser un outil d'accès à la base de données muni d'une interface graphique qui facilite le développement dans un langage de procédures. Un exemple d'un tel outil est pgAdmin, bien que d'autres existent. Ces outils fournissent souvent des fonctionnalités pratiques telles que la détection des guillemets ouverts et facilitent la re-création et le débogage des fonctions.

39.11.1. Utilisation des guillemets simples (quotes)

Le code d'une fonction PL/pgSQL est spécifié dans la commande **CREATE FUNCTION** comme une chaîne de caractères. Si vous écrivez la chaîne littérale de la façon ordinaire en l'entourant de guillemets simples, alors tout guillemet simple dans le corps de la fonction doit être doublé ; de la même façon, les antislashs doivent être doublés (en supposant que la syntaxe d'échappement de chaînes est utilisée). Doubler les guillemets devient rapidement difficile et, dans la plupart des cas compliqués, le code peut devenir rapidement incompréhensible parce que vous pouvez facilement vous trouver avec une douzaine, voire plus, de guillemets adjacents. À la place, il est recommandé d'écrire le corps de la fonction en tant qu'une chaîne littérale « avec guillemets dollar » (voir la Section 4.1.2.4, « Constantes de chaînes avec guillemet dollar »). Dans cette approche, vous ne doublez jamais les marques de guillemets mais vous devez faire attention à choisir un délimiteur dollar différent pour chaque niveau d'imbrication dont vous avez besoin. Par exemple, vous pouvez écrire la commande **CREATE FUNCTION** en tant que :

```
CREATE OR REPLACE FUNCTION fonction_test(integer) RETURNS integer AS $PROC$
   ....
$PROC$ LANGUAGE plpgsql;
```

À l'intérieur de ceci, vous pouvez utiliser des guillemets pour les chaînes littérales simples dans les commandes SQL et `$$` pour délimiter les fragments de commandes SQL que vous assemblez comme des chaînes. Si vous avez besoin de mettre entre guillemets du texte qui inclut `$$`, vous pouvez utiliser `$Q$`, et ainsi de suite.

Le graphe suivant montre ce que vous devez faire lors de l'écriture de guillemets simples sans guillemets dollar. Cela pourrait être utile lors de la traduction de code avec guillemets simples en quelque chose de plus compréhensible.

1 guillemet simple
: Pour commencer et terminer le corps de la fonction, par exemple :

```
CREATE FUNCTION foo() RETURNS integer AS '
        .....
' LANGUAGE plpgsql;
```

Partout au sein du corps de la fonction entouré de guillemets simples, les guillemets simples *doivent* aller par paires.

2 guillemets simples
: Pour les chaînes de caractères à l'intérieur du corps de la fonction, par exemple :

```
une_sortie := ''Blah'';
SELECT * FROM utilisateurs WHERE f_nom=''foobar'';
```

Dans l'approche du guillemet dollar, vous devriez juste écrire :

```
une_sortie := 'Blah';
SELECT * FROM utilisateurs WHERE f_nom='foobar';
```

ce qui serait exactement ce que l'analyseur PL/pgSQL verrait dans les deux cas.

4 guillemets simples
: Quand vous avez besoin d'un guillemet simple dans une chaîne constante à l'intérieur du corps de la fonction, par exemple :

```
une_sortie := une_sortie || '' AND nom LIKE ''''foobar'''' AND xyz''
```

La valeur effectivement concaténée à `une_sortie` est : `AND nom LIKE 'foobar' AND xyz`.

Dans l'approche du guillemet dollar, vous auriez écrit :

```
une_sortie := une_sortie || $$ AND nom LIKE 'foobar' AND xyz$$
```

Faites attention que chaque délimiteur en guillemet dollar ne soient pas simplement `$$`.

6 guillemets simples
: Quand un simple guillemet dans une chaîne à l'intérieur du corps d'une fonction est adjacent à la fin de cette chaîne constante, par exemple :

```
une_sortie := une_sortie || '' AND nom LIKE ''''foobar''''''
```

La valeur effectivement concaténée à `une_sortie` est alors : `AND nom LIKE 'foobar'`.

Dans l'approche guillemet dollar, ceci devient :

```
une_sortie := une_sortie || $$ AND nom LIKE 'foobar'$$
```

10 guillemets simples
: Lorsque vous voulez deux guillemets simples dans une chaîne constante (qui compte pour huit guillemets simples) et qu'elle est adjacente à la fin de cette chaîne constante (deux de plus). Vous n'aurez probablement besoin de ceci que si vous écrivez une fonction qui génère d'autres fonctions comme dans l'Exemple 39.7, « Portage d'une fonction qui crée une autre fonction

de PL/SQL vers PL/pgSQL ». Par exemple :

```
une_sortie := une_sortie || '' if v_'' ||
    referrer_keys.kind || '' like '''''''''
    || referrer_keys.key_string || '''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

La valeur de `une_sortie` sera alors :

```
if v_... like ''...'' then return ''...''; end if;
```

Dans l'approche du guillemet dollar, ceci devient :

```
une_sortie := une_sortie || $$ if v_$$ || referrer_keys.kind || $$ like '$$
|| referrer_keys.key_string || $$'
then return '$$  || referrer_keys.referrer_type
|| $$'; end if;$$;
```

où nous supposons que nous avons seulement besoin de placer des marques de guillemets simples dans `une_sortie` parce que les guillemets seront recalculés avant utilisation.

39.12. Portage d'Oracle™ PL/SQL

Cette section explicite les différences entre le PL/pgSQL de PostgreSQL™ et le langage PL/SQL d'Oracle, afin d'aider les développeurs qui portent des applications d'Oracle® vers PostgreSQL™.

PL/pgSQL est similaire à PL/SQL sur de nombreux aspects. C'est un langage itératif structuré en blocs et toutes les variables doivent être déclarées. Les affectations, boucles, conditionnelles sont similaires. Les principales différences que vous devez garder à l'esprit quand vous portez de PL/SQL vers PL/pgSQL sont:

- Si un nom utilisé dans une commande SQL peut être soit un nom de colonne d'une table soit une référence à une variable de la fonction, PL/SQL le traite comme un nom de commande. Cela correspond au comportement de PL/pgSQL lorsque `plpgsql.variable_conflict` = `use_column`, ce qui n'est pas la valeur par défaut, comme expliqué dans Section 39.10.1, « Substitution de variables ». Il est préféreable d'éviter de tels ambigüités des le début mais si vous devez migrer une grande quantité de code qui dépend de ce comportement, paramétrer `variable_conflict` peut s'avérer être la meilleure solution.
- Dans PostgreSQL™, le corps de la fonction doit être écrit comme une chaîne litérale. Du coup, vous avez besoin d'utiliser les guillemets dollar ou l'échappement des simples guillemets dans le corps de la fonction. Voir la Section 39.11.1, « Utilisation des guillemets simples (quotes) ».
- À la place des paquetages, utilisez des schémas pour organiser vos fonctions en groupes.
- Comme il n'y a pas de paquetages, il n'y a pas non plus de variables au niveau paquetage. Ceci est un peu ennuyant. Vous pourriez être capable de conserver un état par session dans les tables temporaires à la place.
- Les boucles **FOR** d'entiers en ordre inverse (`REVERSE`) fonctionnent différemment ; PL/SQL compte du second numéro jusqu'au premier alors que PL/pgSQL compte du premier jusqu'au second, ceci réclamant que les limites de la boucle soient échangées lors du portage. Cette incompatibilité est malheureuse mais a peu de chance d'être changée. (Voir Section 39.6.3.5, « `FOR` (variante avec entier) ».)
- Les boucles **FOR** sur des requêtes (autres que des curseurs) fonctionnent aussi différemment : la variable cible doit avoir été déclarée alors que PL/SQL les déclare toujours implicitement. Un avantage de ceci est que les valeurs des variables sont toujours accessibles à la sortie de la boucle.
- Il existe plusieurs différences de notation pour l'utilisation des variables curseurs.

39.12.1. Exemples de portages

L'Exemple 39.6, « Portage d'une fonction simple de PL/SQL vers PL/pgSQL » montre comment porter une simple fonction de PL/SQL vers PL/pgSQL.

Exemple 39.6. Portage d'une fonction simple de PL/SQL vers PL/pgSQL

Voici une fonction en PL/SQL Oracle™ :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar, v_version varchar)
RETURN varchar IS
```

```
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Parcourons cette fonction et voyons les différences avec PL/pgSQL :

- Le mot clé `RETURN` dans le prototype de la fonction (pas dans le corps de la fonction) devient `RETURNS` dans PostgreSQL. De plus, `IS` devient `AS` et vous avez besoin d'ajouter une clause `LANGUAGE` parce que PL/pgSQL n'est pas le seul langage de procédures disponible.
- Dans PostgreSQL™, le corps de la fonction est considéré comme une chaîne littérale, donc vous avez besoin d'utiliser les guillemets simples ou les guillemets dollar tout autour. Ceci se substitue au `/` de fin dans l'approche d'Oracle.
- La commande `show errors` n'existe pas dans PostgreSQL™ et n'est pas nécessaire car les erreurs sont rapportées automatiquement.

Voici de quoi aurait l'air cette fonction portée sous PostgreSQL™ :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar, v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

L'Exemple 39.7, « Portage d'une fonction qui crée une autre fonction de PL/SQL vers PL/pgSQL » montre comment porter une fonction qui crée une autre fonction et comment gérer les problèmes de guillemets résultants.

Exemple 39.7. Portage d'une fonction qui crée une autre fonction de PL/SQL vers PL/pgSQL

La procédure suivante récupère des lignes d'une instruction **SELECT** et construit une grande fonction dont les résultats sont dans une instruction `IF` pour favoriser l'efficacité.

Voici la version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
        v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
        ' IF v_' || referrer_key.kind
        || ' LIKE ''' || referrer_key.key_string
        || ''' THEN RETURN ''' || referrer_key.referrer_type
        || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Voici comment la fonction serait dans PostgreSQL™ :

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN' ;

    FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
        func_body := func_body ||
        ' IF v_' || referrer_key.kind
        || ' LIKE ' || quote_literal(referrer_key.key_string)
        || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
        || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
        'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
        v_domain varchar,
        v_url varchar)
        RETURNS varchar AS '
        || quote_literal(func_body)
        || ' LANGUAGE plpgsql;' ;

        EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Notez comment le corps de la fonction est construit séparément et est passé au travers de `quote_literal` pour doubler tout symbole guillemet qu'il peut contenir. Cette technique est nécessaire parce que nous ne pouvons pas utiliser à coup sûr les guillemets dollar pour définir la nouvelle fonction : nous ne sommes pas sûr de savoir quelle chaîne sera interpolée à partir du champ *`referrer_key.key_string`* (nous supposons ici que ce *`referrer_key.kind`* vaut à coup sûr `host`, `domain` ou `url` mais *`referrer_key.key_string`* pourrait valoir autre chose, il pourrait contenir en particulier des signes dollar). Cette fonction est en fait une amélioration de l'original Oracle parce qu'il ne génèrera pas de code cassé quand *`referrer_key.key_string`* ou *`referrer_key.referrer_type`* contient des guillemets.

L'Exemple 39.8, « Portage d'une procédure avec manipulation de chaînes et paramètres `OUT` de PL/SQL vers PL/pgSQL » montre comment porter une fonction ayant des paramètres `OUT` et effectuant des manipulations de chaînes. PostgreSQL™ n'a pas de fonction `instr` intégrée mais vous pouvez en créer une en utilisant une combinaison d'autres fonctions. Dans la Section 39.12.3, « Annexe », il y a une implémentation PL/pgSQL d'`instr` que vous pouvez utiliser pour faciliter votre portage.

Exemple 39.8. Portage d'une procédure avec manipulation de chaînes et paramètres `OUT` de PL/SQL vers PL/pgSQL

La procédure Oracle™ suivante est utilisée pour analyser une URL et renvoyer plusieurs éléments (hôte, chemin et requête). Les fonctions PL/pgSQL ne peuvent renvoyer qu'une seule valeur.

Voici la version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- Celle-ci sera passée en retour
    v_path OUT VARCHAR,  -- Celle-là aussi
    v_query OUT VARCHAR) -- Et celle-là
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');
```

```
    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Voici une traduction possible en PL/pgSQL :

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

Cette fonction pourrait être utilisée ainsi :

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

L'Exemple 39.9, « Portage d'une procédure de PL/SQL vers PL/pgSQL » montre comment porter une procédure qui utilise de nombreuses fonctionnalités spécifiques à Oracle.

Exemple 39.9. Portage d'une procédure de PL/SQL vers PL/pgSQL

La version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock
        raise_application_error(-20000, 'Unable to create a new job: a job is currently
running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
        EXCEPTION
          WHEN dup_val_on_index THEN NULL; -- ne vous inquietez pas si cela existe déjà
    END;
    COMMIT;
END;
/
show errors
```

Les procédures comme celles-ci peuvent être aisément converties en fonctions PostgreSQL™ renvoyant un void. Cette procédure en particulier est intéressante parce qu'elle peut nous apprendre diverses choses :

1. Il n'y a pas d'instruction `PRAGMA` dans PostgreSQL™.
2. Si vous faites un **LOCK TABLE** dans PL/pgSQL, le verrou ne sera pas libéré jusqu'à ce que la transaction appelante soit terminée.
3. Vous ne pouvez pas lancer un **COMMIT** dans une fonction PL/pgSQL. La fonction est lancée à l'intérieur d'une transaction externe et, du coup, un **COMMIT** impliquerait simplement la fin de l'exécution de la fonction. Néanmoins, dans ce cas particulier, ce n'est de toute façon pas nécessaire parce que le verrou obtenu par **LOCK TABLE** sera libéré lors de la levée de l'erreur.

Voici comment nous pourrions porter cette procédure vers PL/pgSQL :

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
  a_running_job_count integer;
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

  SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

  IF a_running_job_count > 0 THEN
    RAISE EXCEPTION 'Unable to create a new job: a job is currently running';
  END IF;

  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
  EXCEPTION
    WHEN unique_violation THEN
    -- ne vous inquietez pas si cela existe déjà
  END;
```

```
END;
$$ LANGUAGE plpgsql;
```

1. La syntaxe de `RAISE` est considérablement différente de l'instruction Oracle similaire, bien que le cas basique du `RAISE` *`nom_exception`* fonctionne de façon similaire.
2. Les noms d'exceptions supportées par PL/pgSQL sont différents de ceux d'Oracle. L'ensemble de noms d'exceptions intégré est plus important (voir l'Annexe A, Codes d'erreurs de PostgreSQL™). Il n'existe actuellement pas de façon de déclarer des noms d'exceptions définis par l'utilisateur, bien que vous puissiez aussi ignorer les valeurs SQLSTATE choisies par l'utilisateur.

La principale différence fonctionnelle entre cette procédure et l'équivalent Oracle est que le verrou exclusif sur la table `cs_jobs` sera détenu jusqu'à la fin de la transaction appelante. De plus, si l'appelant annule plus tard (par exemple à cause d'une erreur), les effets de cette procédure seront annulés.

39.12.2. Autres choses à surveiller

Cette section explique quelques autres choses à surveiller quand on effectue un portage de fonctions PL/SQL Oracle vers PostgreSQL.

39.12.2.1. Annulation implicite après une exception

Dans PL/pgSQL, quand une exception est récupérée par une clause `EXCEPTION`, toutes les modifications de la base de données depuis le bloc `BEGIN` sont automatiquement annulées. C'est-à-dire que le comportement est identique à celui obtenu à partir d'Oracle avec :

```
BEGIN
SAVEPOINT s1;
... code ici ...
EXCEPTION
WHEN ... THEN
ROLLBACK TO s1;
... code ici ...
WHEN ... THEN
ROLLBACK TO s1;
... code ici ...
END;
```

Si vous traduisez une procédure d'Oracle qui utilise **SAVEPOINT** et **ROLLBACK TO** dans ce style, votre tâche est facile : omettez **SAVEPOINT** et **ROLLBACK TO**. Si vous avez une procédure qui utilise **SAVEPOINT** et **ROLLBACK TO** d'une façon différente, alors un peu de réflexion supplémentaire sera nécessaire.

39.12.2.2. EXECUTE

La version PL/pgSQL d'**EXECUTE** fonctionne de façon similaire à la version PL/SQL mais vous devez vous rappeler d'utiliser `quote_literal` et `quote_ident` comme décrit dans la Section 39.5.4, « Exécuter des commandes dynamiques ». Les constructions de type `EXECUTE 'SELECT * FROM $1';` ne fonctionneront pas de façon fiable à moins d'utiliser ces fonctions.

39.12.2.3. Optimisation des fonctions PL/pgSQL

PostgreSQL™ vous donne deux modificateurs de création de fonctions pour optimiser l'exécution : la « volatilité » (la fonction renvoie toujours le même résultat quand on lui donne les mêmes arguments) et la « rigueur » (une fonction renvoie NULL si tous ses arguments sont NULL). Consultez la page de référence de CREATE FUNCTION(7) pour les détails.

Pour faire usage de ces attributs d'optimisation, votre instruction **CREATE FUNCTION** devrait ressembler à ceci :

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

39.12.3. Annexe

Cette section contient le code d'un ensemble de fonctions `instr` compatible Oracle que vous pouvez utiliser pour simplifier vos efforts de portage.

```
--
-- fonctions instr qui reproduisent la contrepartie Oracle
```

```
-- Syntaxe: instr(string1, string2, [n], [m]) où [] signifie paramètre optionnel.
--
-- Cherche string1 en commençant par le  n-ième caractère pour la m-ième occurrence
-- de string2.  Si n est négatif, cherche en sens inverse. Si m n'est pas fourni
-- suppose 1 (la recherche commence au premier caractère).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_nombre integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
```

```
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_nombre := occur_nombre + 1;

                IF occur_nombre = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Chapitre 40. PL/Tcl - Langage de procédures Tcl

PL/Tcl est un langage de procédures chargeable pour le système de bases de données PostgreSQL™, activant l'utilisation du *langage Tcl* pour l'écriture de fonctions de procédures déclencheurs.

40.1. Aperçu

PL/Tcl offre un grand nombre de fonctionnalités qu'un codeur de fonctions dispose avec le langage C, avec quelques restrictions et couplé à de puissantes bibliothèques de traitement de chaînes de caractères disponibles pour Tcl.

Une *bonne* restriction est que tout est exécuté dans le contexte de l'interpréteur Tcl. En plus de l'ensemble sûr de commandes limitées de Tcl, seules quelques commandes sont disponibles pour accéder à la base via SPI et pour envoyer des messages via `elog()`. PL/Tcl ne fournit aucun moyen pour accèder aux internes du serveur de bases ou pour gagner un accès au niveau système d'exploitation avec les droits du processus serveur PostgreSQL™ comme le fait une fonction C. Du coup, les utilisateurs de la base, sans droits, peuvent utiliser ce langage en toute confiance ; il ne leur donne pas une autorité illimitée.

L'autre restriction d'implémentation est que les fonctions Tcl ne peuvent pas être utilisées pour créer des fonctions d'entrées/sorties pour les nouveaux types de données.

Quelques fois, il est préférable d'écrire des fonctions Tcl non restreintes par le Tcl sûr. Par exemple, vous pourriez vouloir une fonction Tcl pour envoyer un courrier électronique. Pour gérer ces cas, il existe une variante de PL/Tcl appelée `PL/TclU` (Tcl non accrédité). C'est exactement le même langage sauf qu'un interpréteur Tcl complet est utilisé. *Si PL/TclU est utilisé, il doit être installé comme langage de procédures non accrédité* de façon à ce que seuls les superutilisateurs de la base de données puissent créer des fonctions avec lui. Le codeur d'une fonction PL/TclU doit faire attention au fait que la fonction ne pourra pas être utilisé pour faire autre chose que son but initial, car il sera possible de faire tout ce qu'un administrateur de la base de données peut faire.

Le code de l'objet partagé pour les gestionnaires d'appel PL/Tcl et PL/TclU est automatiquement construit et installé dans le répertoire des bibliothèques de PostgreSQL™ si le support de Tcl est spécifié dans l'étape de configuration de la procédure d'installation. Pour installer PL/Tcl et/ou PL/TclU dans une base de données particulière, utilisez le programme **createlang**, par exemple `createlang pltcl` *`nom_base`* ou `createlang pltclu` *`nom_base`*.

40.2. Fonctions et arguments PL/Tcl

Pour créer une fonction dans le langage PL/Tcl, utilisez la syntaxe standard de CREATE FUNCTION(7) :

```
CREATE FUNCTION nom_fonction (types_arguments) RETURNS
type_en_retour AS $$
    # corps de la fonction PL/Tcl
$$ LANGUAGE pltcl;
```

PL/TclU est identique sauf que le langage doit être `pltclu`.

Le corps de la fonction est simplement un bout de script Tcl. Quand la fonction est appelée, les valeurs des arguments sont passées en tant que variables `$1` ... `$`*`n`* au script Tcl. Le résultat est renvoyé à partir du code Tcl de la façon habituelle avec une instruction `return`.

Par exemple, une fonction renvoyant le plus grand de deux valeurs entières pourrait être définie ainsi :

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Notez la clause `STRICT` qui nous permet d'éviter de penser aux valeurs NULL en entrées : si une valeur NULL est passée, la fonction ne sera pas appelée du tout mais renverra automatiquement un résultat nul.

Dans une fonction non stricte, si la valeur réelle d'un argument est NULL, la variable `$`*`n`* correspondante sera initialisée avec une chaîne vide. Pour détecter si un argument particulier est NULL, utilisez la fonction `argisnull`. Par exemple, supposez que nous voulons `tcl_max` avec un argument NULL et un non NULL pour renvoyer l'argument non NULL plutôt que NULL :

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
```

```
    return $2
$$ LANGUAGE pltcl;
```

Comme indiqué ci-dessus, pour renvoyer une valeur NULL à partir d'une fonction PL/Tcl, exécutez `return_null`. Ceci peut être fait que la fonction soit stricte ou non.

Les arguments de type composé sont passés à la fonction comme des tableaux Tcl. Les noms des éléments du tableau sont les noms d'attribut du type composite. Si un attribut dans la ligne passée a la valeur NULL, il n'apparaîtra pas dans le tableau. Voici un exemple :

```
CREATE TABLE employe (
    nom text,
    salaire integer,
    age integer
);

CREATE FUNCTION surpaye(employe) RETURNS boolean AS $$
    if {200000.0 < $1(salaire)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salaire)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

Il n'y a actuellement aucun support pour le retour d'une valeur résultat de type composé et pour le retour d'ensembles.

PL/Tcl n'a pas actuellement du support complet pour les types de domaine : il traite un domaine de la même façon que le type scalaire sous-jacent. Cela signifie que les contraintes associées avec le domaine ne seront pas forcées. Ce n'est pas un problème pour les arguments de la fonction mais c'est hasardeux de déclarer une fonction PL/Tcl renvoyant un type domaine.

40.3. Valeurs des données avec PL/Tcl

Les valeurs des arguments fournies au code d'une fonction PL/Tcl sont simplement les arguments en entrée convertis au format texte (comme s'ils avaient été affichés par une instruction **SELECT**). De même, la commande `return` acceptera toute chaîne acceptable dans le format d'entrée du type de retour déclaré pour la fonction. Donc, à l'intérieur de la fonction PL/Tcl, toutes les valeurs de données sont simplement des chaînes de texte.

40.4. Données globales avec PL/Tcl

Quelque fois, il est utile d'avoir des données globales qui sont conservées entre deux appels à une fonction ou qui sont partagées entre plusieurs fonctions. Ceci peut être facilement obtenu car toutes les fonctions PL/Tcl exécutées dans une session partagent le même interpréteur Tcl sûr. Donc, toute variable globale Tcl est accessible aux appels de fonctions PL/Tcl et persisteront pour la durée de la session SQL (notez que les fonctions PL/TclU partagent de la même façon les données globales mais elles sont dans un interpréteur Tcl différent et ne peuvent pas communiquer avec les fonctions PL/Tcl). C'est facile à faire en PL/Tcl mais il existe quelques restrictions qui doivent être comprises.

Pour des raisons de sécurité, PL/Tcl exécute les fonctions appelées par tout rôle SQL dans un interpréteur Tcl séparé pour ce rôle. Ceci empêche une interférence accidentelle ou malicieuse d'un utilisateur avec le comportement des fonctions PL/Tcl d'un autre utilisateur. Chaque interpréteur aura ses propres valeurs pour toutes les variables globales Tcl. Du coup, deux fonctions PL/Tcl partageront les mêmes variables globales si et seulement si elles sont exécutées par le même rôle SQL. Dans une application où une seule session exécute du code sous plusieurs rôles SQL(via des fonctions `SECURITY DEFINER`, l'utilisation de **SET ROLE**, etc), vous pouvez avoir besoin de mettre des étapes explicites pour vous assurer que les fonctions PL/Tcl peuvent partager des données. Pour cela, assurez-vous que les fonctions qui doivent communiques ont pour propriétaire le même utilisateur et marquez-les avec l'option `SECURITY DEFINER`. Bien sûr, vous devez faire attention à ce que de telles fonctions ne puissent pas être utilisées pour faire des choses non souhaitées.

Toutes les fonctions PL/TclU utilisées dans une session s'exécutent avec le même interpréteur Tcl, qui est bien sûr différent des interpréteurs utilisées pour les fonctions PL/Tcl. Donc les données globales sont automatiquement partagées entre des fonctions PL/TclU. Ceci n'est pas considéré comme un risque de sécurité parce que toutes les fonctions PL/TclU sexécutent dans le même niveau de confiance, celui d'un super-utilisateur.

Pour aider à la protection des fonctions PL/Tcl sur les interférences non intentionnelles, un tableau global est rendu disponible pour chaque fonction via la commande `upvar`. Le nom global de cette variable est le nom interne de la fonction alors que le nom local est `GD`. Il est recommandé d'utiliser `GD` pour les données privées persistantes d'une fonction. Utilisez les variables globales

Tcl uniquement pour les valeurs que vous avez l'intention de partager avec les autres fonctions. (Notez que les tableaux `GD` sont seulement globaux à l'intérieur d'un interpréteur particulier, pour qu'ils ne franchissent pas les restrictions de sécurité mentionnées ci-dessus.)

Un exemple de l'utilisation de `GD` apparaît dans l'exemple `spi_execp` ci-dessous.

40.5. Accès à la base de données depuis PL/Tcl

Les commandes suivantes sont disponibles pour accéder à la base de données depuis le corps d'une fonction PL/Tcl :

`spi_exec [-count n] [-array name] command [loop-body]`

Exécute une commande SQL donnée en tant que chaîne. Une erreur dans la commande lève une erreur. Sinon, la valeur de retour de `spi_exec` est le nombre de lignes intéressées dans le processus (sélection, insertion, mise à jour ou suppression) par la commande ou zéro si la commande est une instruction utilitaire. De plus, si la commande est une instruction **SELECT**, les valeurs des données sélectionnées sont placées dans des variables Tcl décrites ci-dessous.

La valeur optionnelle `-count` indique à `spi_exec` le nombre maximum de lignes à travailler dans la commande. L'effet de ceci est comparable à l'initialisation d'une requête en tant que curseur et de dire `FETCH n`.

Si la commande est une instruction **SELECT**, les valeurs des colonnes de résultat sont placées dans les variables Tcl nommées d'après les colonnes. Si l'option `-array` est donnée, les valeurs de colonnes sont stockées à la place dans un tableau associatif nommé, avec les noms des colonnes utilisés comme index du tableau.

Si la commande est une instruction **SELECT** et qu'aucun script *`loop-body`* n'est donné, alors seule la première ligne de résultats est stockée dans des variables Tcl ; les lignes suivantes sont ignorées. Aucun stockage n'intervient si la requête ne renvoie pas de ligne (ce cas est détectable avec le résultat de la fonction `spi_exec`). Par exemple :

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

initialisera la variable Tcl `$cnt` avec le nombre de lignes dans le catalogue système pg_proc.

Si l'argument *`loop-body`* optionnel est donné, il existe un morceau de script Tcl qui est exécuté une fois pour chaque ligne du résultat de la requête (*`loop-body`* est ignoré si la commande donnée n'est pas un **SELECT**). Les valeurs des colonnes de la ligne actuelle sont stockées dans des variables Tcl avant chaque itération. Par exemple :

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

affichera un message de trace pour chaque ligne de `pg_class`. Cette fonctionnalité travaille de façon similaire aux autres constructions de boucles de Tcl ; en particulier, `continue` et `break` fonctionnent de la même façon à l'intérieur de *`loop-body`*.

Si une colonne d'un résultat de la requête est NULL, la variable cible est « dés-initialisée » plutôt qu'initialisée.

`spi_prepare query typelist`

Prépare et sauvegarde un plan de requête pour une exécution future. Le plan sauvegardé sera conservé pour la durée de la session actuelle.

La requête peut utiliser des paramètres, c'est-à-dire des emplacements pour des valeurs à fournir lorsque le plan sera réellement exécuté. Dans la chaîne de requête, faites référence aux paramètres avec les symboles `$1` ... `$n`. Si la requête utilise les paramètres, les noms des types de paramètre doivent être donnés dans une liste Tcl (écrivez une liste vide pour *`typelist`* si aucun paramètre n'est utilisé).

La valeur de retour de `spi_prepare` est l'identifiant de la requête à utiliser dans les appels suivants à `spi_execp`. Voir `spi_execp` pour un exemple.

`spi_execp [-count n] [-array name] [-nulls string] queryid [value-list] [loop-body]`

Exécute une requête préparée précédemment avec `spi_prepare`. *`queryid`* est l'identifiant renvoyé par `spi_prepare`. Si la requête fait référence à des paramètres, une liste de valeurs (*`value-list`*) doit être fournie. C'est une liste Tcl des valeurs réelles des paramètres. La liste doit être de la même longueur que la liste de types de paramètres donnée précédemment lors de l'appel à `spi_prepare`. Oubliez-la si la requête n'a pas de paramètres.

La valeur optionnelle pour `-nulls` est une chaîne d'espaces et de caractères `'n'` indiquant à `spi_execp` les paramètres nuls. Si indiqué, elle doit avoir exactement la même longueur que *`value-list`*. Si elle est omise, toutes les valeurs de paramètres sont non NULL.

Sauf si la requête et ses paramètres sont spécifiés, `spi_execp` fonctionne de la même façon que `spi_exec`. Les options `-count`, `-array` et *`loop-body`* sont identiques. Du coup, la valeur du résultat l'est aussi.

Voici un exemple d'une fonction PL/Tcl utilisant un plan préparé :

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prépare le plan sauvegardé au premier appel
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

Nous avons besoin des antislashs à l'intérieur de la chaîne de la requête passée à `spi_prepare` pour s'assurer que les marqueurs `$n` sont passés au travers de `spi_prepare` sans transformation et ne sont pas remplacés avec la substitution de variables de Tcl.

`spi_lastoid`

Renvoie l'OID de la ligne insérée par le dernier appel à `spi_exec` ou `spi_execp`, si la commande était un **INSERT** d'une seule ligne et que la table modifiée contenait des OID (sinon, vous obtenez zéro).

`quote` *`string`*

Double toutes les occurrences de guillemet simple et d'antislash dans la chaîne donnée. Ceci peut être utilisé pour mettre entre guillemets des chaînes de façon sûr et pour qu'elles puissent être insérées dans des commandes SQL passées à `spi_exec` ou `spi_prepare`. Par exemple, pensez à une chaîne de commande SQL comme :

```
"SELECT '$val' AS ret"
```

où la variable Tcl `val` contient actuellement le mot `doesn't`. Ceci finirait avec la chaîne de commande :

```
SELECT 'doesn't' AS ret
```

qui va causer une erreur d'analyse lors de `spi_exec` ou de `spi_prepare`. Pour fonctionner correctement, la commande soumise devrait contenir :

```
SELECT 'doesn''t' AS ret
```

qui peut-être créé avec PL/Tcl en utilisant :

```
"SELECT '[ quote $val ]' AS ret"
```

Un avantage de `spi_execp` est que vous n'avez pas à mettre entre guillemets des valeurs de paramètres comme ceux-ci car les paramètres ne sont jamais analysés comme faisant partie de la chaîne de la commande SQL.

`elog` *`level msg`*

Émet une trace ou un message d'erreur. Les niveaux possibles sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR` et `FATAL`. `ERROR` élève une condition d'erreur ; si elle n'est pas récupérée par le code Tcl, l'erreur est propagée à la requête appelante, causant l'annulation de la transaction ou sous-transaction en cours. Ceci est en fait identique à la commande `error`. `FATAL` annule la transaction et fait que la session courante s'arrête (il n'existe probablement aucune raison d'utiliser ce niveau d'erreur dans les fonctions PL/Tcl mais il est fourni pour que tous les messages soient tout de même disponibles). Les autres niveaux génèrent seulement des messages de niveaux de priorité différent. Le fait que les messages d'un niveau de priorité particulier sont reportés au client, écrit dans les journaux du serveur ou les deux à la fois, est contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir le Chapitre 18, Configuration du serveur pour plus d'informations.

40.6. Procédures pour déclencheurs en PL/Tcl

Les procédures pour déclencheurs peuvent être écrites en PL/Tcl. PostgreSQL™ requiert qu'une procédure, devant être appelée en tant que déclencheur, doit être déclarée comme une fonction sans arguments et retourner une valeur de type `trigger`.

L'information du gestionnaire de déclencheur est passée au corps de la procédure avec les variables suivantes :

`$TG_name`

Nom du déclencheur provenant de l'instruction **CREATE TRIGGER**.

`$TG_relid`

L'identifiant objet de la table qui est à la cause du lancement du déclencheur.

`$TG_table_name`

Le nom de la table qui est à la cause du lancement du déclencheur.

`$TG_table_schema`
: Le schéma de la table qui est à la cause du lancement du déclencheur.

`$TG_relatts`
: Une liste Tcl des noms des colonnes de la table, préfixée avec un élément de liste vide. Donc, rechercher un nom de colonne dans la liste avec la commande `lsearch` de Tcl renvoie le numéro de l'élément, en commençant à 1 pour la première colonne, de la même façon que les colonnes sont numérotées personnellement avec PostgreSQL™.

`$TG_when`
: La chaîne `BEFORE` ou `AFTER` suivant le type de l'événement du déclencheur.

`$TG_level`
: La chaîne `ROW` ou `STATEMENT` suivant le type de l'événement du déclencheur.

`$TG_op`
: La chaîne `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE` suivant le type de l'événement du déclencheur.

`$NEW`
: Un tableau associatif contenant les valeurs de la nouvelle ligne de la table pour les actions **INSERT** ou **UPDATE** ou vide pour **DELETE**. Le tableau est indexé par nom de colonne. Les colonnes NULL n'apparaissent pas dans le tableau. Ce paramètre n'est pas initialisé pour les triggers de niveau instruction.

`$OLD`
: Un tableau associatif contenant les valeurs de l'ancienne ligne de la table pour les actions **UPDATE** or **DELETE** ou vide pour **INSERT**. Le tableau est indexé par nom de colonne. Les colonnes NULL n'apparaissent pas dans le tableau. Ce paramètre n'est pas initialisé pour les triggers de niveau instruction.

`$args`
: Une liste Tcl des arguments de la procédure ainsi que l'instruction **CREATE TRIGGER**. Ces arguments sont aussi accessibles par `$1` ... `$`*n* dans le corps de la procédure.

Le code de retour d'une procédure déclencheur peut être faite avec une des chaînes `OK` ou `SKIP` ou une liste renvoyée par la commande Tcl `array get`. Si la valeur de retour est `OK`, l'opération (**INSERT**/**UPDATE**/**DELETE**) qui a lancé le déclencheur continuera normalement. `SKIP` indique au gestionnaire de déclencheurs de supprimer silencieusement l'opération pour cette ligne. Si une liste est renvoyée, elle indique à PL/Tcl de renvoyer la ligne modifiée au gestionnaire de déclencheurs qui l'insérera au lieu de celle données par `$NEW` (ceci fonctionne seulement pour **INSERT** et **UPDATE**). Inutile de dire que tout ceci est vraiment significatif lorsque le déclencheur est `BEFORE` et **FOR EACH ROW** ; sinon le code de retour est ignoré.

Voici un petit exemple de procédure déclencheur qui force une valeur entière dans une table pour garder trace du nombre de mises à jour réalisées sur la ligne. Pour les nouvelles lignes insérées, la valeur est initialisée à 0 puis incrémentée à chaque opération de mise à jour.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Notez que la procédure déclencheur elle-même ne connaît pas le nom de la colonne ; c'est fourni avec les arguments du déclencheur. Ceci permet à la procédure déclencheur d'être ré-utilisée avec différentes tables.

40.7. Les modules et la commande unknown

PL/Tcl dispose du support de chargement automatique de code Tcl lorsqu'il est utilisé. Il reconnaît une table spéciale, `plt-`

`cl_modules`, qui est présumée contenir les modules de code Tcl. Si cette table existe, le module `unknown` est récupéré de la table et chargé immédiatement dans l'interpréteur Tcl avant la première exécution d'une fonction PL/Tcl dans une session. (Ceci survient séparément pour chaque interpréteur Tcl, si plus d'un est utilisé dans une session ; voir Section 40.4, « Données globales avec PL/Tcl ».)

Alors que le module `unknown` pourrait réellement contenir tout script d'initialisation dont vous avez besoin, il définit normalement une procédure Tcl `unknown` qui est appelée lorsque Tcl ne reconnaît pas le nom de la procédure appelée. La version standard de PL/Tcl essaie de trouver un module dans `pltcl_modules` qui définira la procédure requis. Si une procédure est trouvée, elle est chargée dans l'interpréteur, puis l'exécution est permise avec l'appel original de la procédure. Une deuxième table `pltcl_modfuncs` fournit un index des fonctions et des modules qui les définissent, de façon à ce que la recherche soit rapide.

La distribution PostgreSQL™ inclut les scripts de support pour maintenir ces tables : **pltcl_loadmod**, **pltcl_listmod**, **pltcl_delmod** ainsi que le source pour le module standard `unknown` dans `share/unknown.pltcl`. Ce module doit être chargeable dans chaque base de données initialement pour supporter le mécanisme de chargement automatique.

Les tables `pltcl_modules` et `pltcl_modfuncs` doivent être lisibles par tous mais il est conseillé de les laisser modifiables uniquement par le propriétaire, administrateur de la base de données. Pour des raisons de sécurité, PL/Tcl ignorera `pltcl_modules` (et donc n'essaiera pas de charger le module `unknown`) sauf s'il appartient à un superutilisateur. Cependant, les droits de modification sur cette table peuvent être données à d'autres utilisateurs si vous avez suffisamment confiance en eux.

40.8. Noms de procédure Tcl

Avec PostgreSQL™, le même nom de fonction peut être utilisé par plusieurs fonctions tant que le nombre d'arguments ou leurs types diffèrent. Néanmoins, Tcl requiert que les noms de procédure soient distincts. PL/Tcl gère ceci en faisant en sorte que les noms de procédures Tcl internes contiennent l'identifiant de l'objet de la fonction depuis la table système pg_proc. Du coup, les fonctions PostgreSQL™ avec un nom identique et des types d'arguments différents seront aussi des procédures Tcl différentes. Ceci ne concerne normalement pas le développeur PL/Tcl mais cela pourrait apparaître dans une session de débogage.

Chapitre 41. PL/Perl - Langage de procédures Perl

PL/Perl est un langage de procédures chargeable qui vous permet d'écrire des fonctions PostgreSQL™ dans le *langage de programmation Perl*.

Le principal avantage habituellement cité quant à l'utilisation de Perl est que cela permet l'utilisation des nombreux opérateurs et fonctions de « gestion de chaînes » disponibles grâce à Perl dans des procédures stockées. L'analyse de chaînes complexes se trouve facilité par l'utilisation de Perl et des fonctions et structures de contrôles fournies dans PL/pgSQL.

Pour installer PL/Perl dans une base de données spécifique, utilisez la commande `createlang plperl` *`nom_base`*.

> **Astuce**
>
> Si un langage est installé dans `template1`, toutes les bases de données créées ultérieurement disposeront automatiquement de ce langage.

> **Note**
>
> Les utilisateurs des paquetages sources doivent explicitement autoriser la construction de PL/Perl pendant le processus d'installation (se référer à la Chapitre 15, Procédure d'installation de PostgreSQL™ du code source pour plus d'informations). Les utilisateurs des paquetages binaires peuvent trouver PL/Perl dans un sous-paquetage séparé.

41.1. Fonctions et arguments PL/Perl

Pour créer une fonction dans le langage PL/Perl, utilisez la syntaxe standard CREATE FUNCTION(7) :

```
CREATE FUNCTION nom_fonction (types-arguments) RETURNS
type-retour AS $$
    # Corps de la fonction PL/Perl
$$ LANGUAGE plperl;
```

Le corps de la fonction est du code Perl normal. En fait, le code supplémentaire PL/Perl l'emballe dans une sous-routine Perl. Une fonction PL/Perl est appelée dans un contexte scalaire, il ne peut donc pas retourner une liste. Vous pouvez retourner des valeurs non scalaire par référence comme indiqué ci-dessous.

PL/Perl peut aussi être utilisé au sein de blocs de procédures anonymes avec l'ordre DO(7) :

```
DO $$
    # PL/Perl code
$$ LANGUAGE plperl;
```

Un bloc de procédure anonyme ne prend pas d'arguments et toute valeur retourné est ignorée. Ceci mis à part, il se comporte comme une fonction classique.

> **Note**
>
> L'utilisation de sous-routines nommées est dangereux en Perl, spécialement si elles font références à des variables lexicales dans la partie englobante. Comme une fonction PL/Perl est englobée dans une sous-routine, toute sous-routine nommée que vous y créez sera englobée. En général, il est bien plus sûr de créer des sous-routines anonymes que vous appellerez via un coderef. Pour de plus amples détails, voir les entrées `Variable "%s" will not stay shared` et `Variable "%s" is not available` dans le manuel perldiag, ou recherchez « perl nested named subroutine » sur internet.

La syntaxe de la commande **CREATE FUNCTION** requiert que le corps de la fonction soit écrit comme une constante de type chaîne. Il est habituellement plus agréable d'utiliser les guillemets dollar (voir la Section 4.1.2.4, « Constantes de chaînes avec guillemet dollar ») pour cette constante. Si vous choisissez d'utiliser la syntaxe d'échappement des chaînes `E''`, vous devez doubler les marques de guillemets simples (`'`) et les antislashs (\) utilisés dans le corps de la fonction (voir la Section 4.1.2.1, « Constantes de chaînes »).

Les arguments et les résultats sont manipulés comme dans n'importe quel routine Perl : les arguments sont passés au tableau `@_` et une valeur de retour est indiquée par `return` ou par la dernière expression évaluée dans la fonction.

Par exemple, une fonction retournant le plus grand de deux entiers peut être définie comme suit :

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

Si une valeur NULL en SQL est passée à une fonction, cet argument apparaîtra comme « undefined » en Perl. La fonction définie ci-dessus ne se comportera pas correctement avec des arguments NULL (en fait, tout se passera comme s'ils avaient été des zéros). Nous aurions pu ajouter `STRICT` à la définition de la fonction pour forcer PostgreSQL™ à faire quelque chose de plus raisonnable : si une valeur NULL est passée en argument, la fonction ne sera pas du tout appelée mais retournera automatiquement un résultat NULL. D'une autre façon, nous aurions pu vérifier dans le corps de la fonction la présence d'arguments NULL. Par exemple, supposons que nous voulions que `perl_max` avec un argument NULL et un autre non NULL retourne une valeur non NULL plutôt qu'une valeur NULL, on aurait écrit :

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

Comme le montre l'exemple ci-dessus, passer une valeur NULL en SQL à une fonction en PL/Perl retourne une valeur non définie. Et ceci, que la fonction soit déclarée stricte ou non.

Dans un argument de fonction, tout ce qui n'est pas une référence est une chaîne qui est dans la représentation texte externe standard de PostgreSQL™ pour ce type de données. Dans le cas de types numériques ou texte, Perl fera ce qu'il faut et le programmeur n'aura pas à s'en soucier. Néanmoins, dans d'autres cas, l'argument aura besoin d'être converti dans une forme qui est plus utilisable que Perl. Par exemple, la fonction `decode_bytea` peut-être utilisée pour convertir un argument de type bytea en données binaires non échappées.

De façon similaire, les valeurs renvoyées à PostgreSQL™ doivent être dans le format textuel. Par exemple, la fonction `encode_bytea` peut être utilisée pour échapper des données binaires en retournant une valeur de type bytea.

Perl peut renvoyer des tableaux PostgreSQL™ comme référence à des tableaux Perl. Voici un exemple :

```
CREATE OR REPLACE function renvoit_tableau()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select renvoit_tableau();
```

Les arguments de type composite sont passés à la fonction en tant que références d'un tableau de découpage, les clés du tableau de découpage étant les noms des attributs du type composé. Voici un exemple :

```
CREATE TABLE employe (
    nom text,
    basesalaire integer,
    bonus integer
);

CREATE FUNCTION empcomp(employe) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalaire} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT nom, empcomp(employe.*) FROM employe;
```

Une fonction PL/Perl peut renvoyer un résultat de type composite en utilisant la même approche : renvoyer une référence à un hachage qui a les attributs requis. Par exemple

```
      CREATE TYPE testligneperl AS (f1 integer, f2 text, f3 text);

      CREATE OR REPLACE FUNCTION perl_ligne() RETURNS test_ligne_perl AS $$
      return {f2 => 'hello', f1 => 1, f3 => 'world'};
```

```
      $$ LANGUAGE plperl;

      SELECT * FROM perl_row();
```

Toute colonne dans le type de données déclaré du résultat qui n'est pas présente dans le hachage sera renvoyée NULL.

Les fonctions PL/Perl peuvent aussi renvoyer des ensembles de types scalaires ou composites. Habituellement, vous voulez renvoyer une ligne à la fois, à la fois pour améliorer le temps de démarrage et pour éviter d'allonger la queue de l'ensemble des résultats en mémoire. Vous pouvez faire ceci avec `return_next` comme indiqué ci-dessous. Notez qu'après le dernier `return_next`, vous devez placer soit `return` soit (encore mieux) `return undef`.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF test_ligne_perl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

Pour les petits ensembles de résultats, vous pouvez renvoyer une référence à un tableau contenant soit des scalaires, soit des références à des tableaux soit des références à des hachages de types simples, de types tableaux ou de types composites. Voici quelques exemples simples pour renvoyer l'ensemble complet du résultant en tant que référence de tableau :

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
      return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testligneperl AS $$
return [
      { f1 => 1, f2 => 'Bonjour', f3 => 'Monde' },
      { f1 => 2, f2 => 'Bonjour', f3 => 'PostgreSQL' },
      { f1 => 3, f2 => 'Bonjour', f3 => 'PL/Perl' }
      ];
$$  LANGUAGE plperl;

SELECT * FROM perl_set();
```

Si vous souhaitez utiliser le pragma `strict` dans votre code, vous avez plusieurs options. Pour une utilisation temporaire globale vous pouvez positionner (**SET**) `plperl.use_strict` à « true ». Ce paramètre affectera les compilations suivantes de fonctions PL/Perl, mais pas les fonctions déjà compilées dans la session en cours. Pour une utilisation globale permanente, vous pouvez positionner `plperl.use_strict` à « true » dans le fichier `postgresql.conf`.

Pour une utilisation permanente dans des fonctions spécifiques, vous pouvez simplement placer:

```
use strict;
```

en haut du corps de la fonction.

Le pragma `feature` est aussi disponible avec `use` si votre version de Perl est 5.10.0 ou supérieur.

41.2. Valeurs en PL/Perl

Les valeurs des arguments fournis au code d'une fonction PL/Perl sont simplement les arguments d'entrée convertis en tant que texte (comme s'ils avaient été affichés par une commande **SELECT**). Inversement, les commandes `return` and `return_next` accepterons toute chaîne qui a un format d'entrée acceptable pour le type de retour déclaré de la fonction.

41.3. Fonction incluses

41.3.1. Accès à la base de données depuis PL/Perl

L'accès à la base de données à l'intérieur de vos fonctions écrites en Perl peut se faire à partir des fonctions suivantes :

`spi_exec_query(query [, max-rows])`

`spi_exec_query` exécute une commande SQL et renvoie l'ensemble complet de la ligne comme une référence à un table de références hachées. *Vous ne devez utiliser cette commande que lorsque vous savez que l'ensemble de résultat sera relativement petit.* Voici un exemple d'une requête (commande **SELECT**) avec le nombre optionnel maximum de lignes :

```
$rv = spi_exec_query('SELECT * FROM ma_table', 5);
```

Ceci entrevoit cinq lignes au maximum de la table `ma_table`. Si `ma_table` a une colonne `ma_colonne`, vous obtenez la valeur de la ligne `$i` du résultat de cette façon :

```
$foo = $rv->{rows}[$i]->{ma_colonne};
```

Le nombre total des lignes renvoyées d'une requête **SELECT** peut être accédé de cette façon :

```
$nrows = $rv->{processed}
```

Voici un exemple en utilisant un type de commande différent :

```
$query = "INSERT INTO ma_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

Ensuite, vous pouvez accéder au statut de la commande (c'est-à-dire, `SPI_OK_INSERT`) de cette façon :

```
$res = $rv->{status};
```

Pour obtenir le nombre de lignes affectées, exécutez :

```
$nrows = $rv->{processed};
```

Voici un exemple complet :

```
CREATE TABLE test (
    i int,
    v varchar
    );

INSERT INTO test (i, v) VALUES (1, 'première ligne');
INSERT INTO test (i, v) VALUES (2, 'deuxième ligne');
INSERT INTO test (i, v) VALUES (3, 'troisième ligne');
INSERT INTO test (i, v) VALUES (4, 'immortel');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`spi_query(command)`, `spi_fetchrow(cursor)`, `spi_cursor_close(cursor)`

`spi_query` et `spi_fetchrow` fonctionnent ensemble comme une paire d'ensembles de lignes pouvant être assez importants ou pour les cas où vous souhaitez renvoyer les lignes dès qu'elles arrivent. `spi_fetchrow` fonctionne *seulement* avec `spi_query`. L'exemple suivant illustre comment vous les utilisez ensemble :

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
```

```
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

Habituellement, `spi_fetchrow` devra être répété jusqu'à ce qu'il renvoie `undef`, indiquant qu'il n'y a plus de lignes à lire. Le curseur renvoyé par `spi_query` est automatiquement libéré quand `spi_fetchrow` renvoie `undef`. Si vous ne souhaitez pas lire toutes les lignes, appelez à la place `spi_cursor_close` pour libérer le curseur. Un échec ici résultera en des pertes mémoire.

`spi_prepare(`*`command`*`, `*`argument types`*`)`, `spi_query_prepared(`*`plan`*`, `*`arguments`*`)`, `spi_exec_prepared(`*`plan [, attributes], arguments`*`)`, `spi_freeplan(`*`plan`*`)`

`spi_prepare`, `spi_query_prepared`, `spi_exec_prepared` et `spi_freeplan` implémentent la même fonctionnalité, mais pour des requêtes préparées. `spi_prepare` accepte une chaîne pour la requête avec des arguments numérotés ($1, $2, etc) et une liste de chaînes indiquant le type des arguments :

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2', 'INTEGER',
'TEXT');
```

Une fois qu'un plan est préparé suite à un appel à `spi_prepare`, le plan peut être utilisé à la place de la requête, soit dans `spi_exec_prepared`, où le résultat est identique à celui renvoyé par `spi_exec_query`, soit dans `spi_query_prepared` qui renvoi un curseur exactement comme le fait `spi_query`, qui peut ensuite être passé à `spi_fetchrow`. Le deuxième paramètre, optionnel, de `spi_exec_prepared` est une référence hachée des attributs ; le seul attribut actuellement supporté est `limit`, qui configure le nombre maximum de lignes renvoyées par une requête.

L'avantage des requêtes préparées est que cela rend possible l'utilisation d'un plan préparé par plusieurs exécutions de la requête. Une fois que le plan n'est plus utile, il peut être libéré avec `spi_freeplan` :

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare( 'SELECT (now() + $1)::date AS now',
'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();
```

```
  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

Notez que l'indice du paramètre dans `spi_prepare` est défini via $1, $2, $3, etc, donc évitez de déclarer des chaînes de requêtes qui pourraient aisément amener des bogues difficiles à trouver et corriger.

Cet autre exemple illustre l'utilisation d'un paramètre optionnel avec `spi_exec_prepared` :

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address FROM
generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts WHERE address << $1',
'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

41.3.2. Fonctions utiles en PL/Perl

`elog(`*`level, msg`*`)`

Produit un message de trace ou d'erreur. Les niveaux possibles sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` et `ERROR`. `ERROR` lève une condition d'erreur ; si elle n'est pas récupérée par le code Perl l'entourant, l'erreur se propage à l'extérieur de la requête appelante, causant l'annulation de la transaction ou sous-transaction en cours. Ceci est en fait identique à la commande `die` de Perl. Les autres niveaux génèrent seulement des messages de niveaux de priorité différents. Le fait que les messages d'un niveau de priorité particulier soient rapportés au client, écrit dans les journaux du serveur, voire les deux, est contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir le Chapitre 18, Configuration du serveur pour plus d'informations.

`quote_literal(`*`string`*`)`

Retourne la chaîne donnée convenablement placé entre simple guillemets pour être utilisée comme une chaîne littérale au sein d'une chaîne représentant un ordre SQL. Les simples guillemets et antislashes de la chaîne sont correctement doublés Notez que `quote_literal` retourne undef avec une entrée undef ; si l'argument peut être undef, `quote_nullable` est souvent plus approprié.

`quote_nullable(`*`string`*`)`

Retourne la chaîne donnée convenablement placé entre simple guillemets pour être utilisée comme une chaîne littérale au sein d'une chaîne représentant un ordre SQL. Si l'argument d'entrée est undef, retourne la chaîne "NULL" sans simple guillemet. Les simples guillemets et antislashes de la chaîne sont correctement doublés

`quote_ident(string)`
Retourne la chaîne donnée convenablement placé entre guillemets pour être utilisée comme un identifiant au sein d'une chaîne représentant un ordre SQL. Les guillemets sont ajoutées seulement si cela est nécessaire (i.e. si la chaîne contient des caractères non-identifiant ou est en majuscule). Les guillemets de la chaîne seront convenablement doublés.

`decode_bytea(string)`
Retourne les données binaires non échappé représentées par le contenu de la chaîne donnée, qui doit être encodé au format bytea.

`encode_bytea(string)`
Retourne sous la forme d'un bytea le contenu binaire dans la chaîne passé en argument.

`encode_array_literal(array)`, `encode_array_literal(array, delimiter)`
Retourne le contenu de tableau passé par référence sous forme d'une chaîne littérale. (voir Section 8.14.2, « Saisie de valeurs de type tableau »). Retourne la valeur de l'argument non altérée si ce n'est pas une référence à un tableau. Le délimiteur utilisé entre les éléments du tableau sous forme littérale sera par défaut `", "` si aucun délimiteur n'est spécifié ou s'il est undef.

`encode_array_constructor(array)`
Retourne le contenu de tableau passé par référence sous forme d'une chaîne permettant de construire un tableau en SQL. (voir Section 4.2.11, « Constructeurs de tableaux »). Chaque élément est entouré de simple guillemets par `quote_nullable`. Retourne la valeur de l'argument, entouré de simple guillemets par `quote_nullable`, si ce n'est pas une référence à un tableau.

`looks_like_number(string)`
Retourne une valeur vraie si le contenu de la chaîne passée ressemble à un nombre, selon l'interprétation de Perl, et faux dans le cas contraire. Retourne undef si undef est passé en argument. Tout espace en début et fin de chaîne sont ignorés. `Inf` et `Infinity` sont vu comme des nombres.

41.4. Valeurs globales dans PL/Perl

Vous pouvez utiliser le hachage global `%_SHARED` pour stocker les données, incluant les références de code, entre les appels de fonction pour la durée de vie de la session en cours.

Voici un exemple simple pour des données partagées :

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
if ($_SHARED{$_[0]} = $_[1]) {
      return 'ok';
} else {
      return "Ne peux pas initialiser la variable partagée $_[0] à $_[1]";
}
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
      return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Bonjour, PL/Perl ! Comment va ?');
SELECT get_var('sample');
```

Voici un exemple légèrement plus compliqué utilisant une référence de code :

```
CREATE OR REPLACE FUNCTION ma_fonction() RETURNS void AS $$
$_SHARED{myquote} = sub {
      my $arg = shift;
      $arg =~ s/(['\\])/\\$1/g;
      return "'$arg'";
      };
$$ LANGUAGE plperl;

SELECT ma_fonction(); /* initialise la fonction */

/* Initialise une fonction qui utilise la fonction quote */

CREATE OR REPLACE FUNCTION utilise_quote(TEXT) RETURNS text AS $$
      my $text_to_quote = shift;
      my $qfunc = $_SHARED{myquote};
      return &$qfunc($text_to_quote);
```

```
$$ LANGUAGE plperl;
```

(Vous pouviez avoir remplacé le code ci-dessus avec la seule ligne `return $_SHARED{myquote}->($_[0]);` au prix d'une mauvaise lisibilité.)

Pour des raisons de sécurité, PL/Perl exécute des fonctions appelées par un rôle SQL dans un interpréteur Perl séparé pour ce rôle. Ceci empêche l'interférence accidentelle ou malicieuse d'un utilisateur avec le comportement des fonctions PL/Perl d'un autre utilisateur. Chaque interprétateur a sa propre valeur de la variable `%_SHARED` et des autres états globaux. Du coup, deux fonctions PL/Perl partageront la même valeur de `%_SHARED` si et seulement si elles sont exécutées par le même rôle SQL. Dans une application où une session seule exécute du code sous plusieurs rôles SQL (via des fonctions `SECURITY DEFINER`, l'utilisation de **SET ROLE**, etc), vous pouvez avoir besoin de mettre en place des étapes explicites pour vous assurer que les fonctions PL/Perl peuvent partager des données `%_SHARED`. Pour cela, assurez-vous que les fonctions qui doivent communiquer ont pour propriétaire le même utilisateur et marquez les comme `SECURITY DEFINER`. Bien sûr, vous devez faire attention à ce que ces fonctions ne puissent pas être utilisées pour faire des choses qu'elles ne sont pas sensées faire.

41.5. Niveaux de confiance de PL/Perl

Normalement, PL/Perl est installé en tant que langage de programmation de « confiance », de nom `plperl`. Durant cette installation, certaines commandes Perl sont désactivées pour préserver la sécurité. En général, les commandes qui interagissent avec l'environnement sont restreintes. Cela inclut les commandes sur les descripteurs de fichiers, `require` et `use` (pour les modules externes). Il n'est pas possible d'accéder aux fonctions et variables internes du processus du serveur de base de données ou d'obtenir un accès au niveau du système d'exploitation avec les droits du processus serveur, tel qu'une fonction C peut le faire. Ainsi, n'importe quel utilisateur sans droits sur la base de données est autorisé à utiliser ce langage.

Voici l'exemple d'une fonction qui ne fonctionnera pas car les commandes système ne sont pas autorisées pour des raisons de sécurité :

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

La création de cette fonction échouera car le validateur détectera l'utilisation par cette fonction d'une opération interdite.

Il est parfois souhaitable d'écrire des fonctions Perl qui ne sont pas restreintes. Par exemple, on peut souhaiter vouloir envoyer des courriers électroniques. Pour supporter ce cas de figure, PL/Perl peut aussi être installé comme un langage « douteux » (habituellement nommé PL/PerlU). Dans ce cas, la totalité du langage Perl est accessible. Si la commande **createlang** est utilisée pour installer le langage, le nom du langage `plperlu` sélectionnera la version douteuse de PL/Perl.

Les auteurs des fonctions PL/PerlU doivent faire attention au fait que celles-ci ne puissent être utilisées pour faire quelque chose de non désiré car cela donnera la possibilité d'agir comme si l'on possédait les privilèges d'administrateur de la base de données. Il est à noter que le système de base de données ne permet qu'aux super-utilisateurs de créer des fonctions dans un langage douteux.

Si la fonction ci-dessus a été créée par un super-utilisateur en utilisant le langage `plperlu`, l'exécution de celle-ci réussira.

De la même façon, les blocs de procédure anonymes écris en perl peuvent utiliser les opérations restreintes si le langage est spécifié comme `plperlu` plutôt que `plperl`, mais l'appelant doit être un super-utilisateur.

Note

While PL/Perl functions run in a separate Perl interpreter for each SQL role, all PL/PerlU functions executed in a given session run in a single Perl interpreter (which is not any of the ones used for PL/Perl functions). This allows PL/PerlU functions to share data freely, but no communication can occur between PL/Perl and PL/PerlU functions.

Note

Perl cannot support multiple interpreters within one process unless it was built with the appropriate flags, namely either `usemultiplicity` or `useithreads`. (`usemultiplicity` is preferred unless you actually need to use threads. For more details, see the perlembed man page.) If PL/Perl is used with a copy of Perl that was not built this way, then it is only possible to have one Perl interpreter per session, and so any one session can only execute either PL/PerlU functions, or PL/Perl functions that are all called by the same SQL role.

41.6. Déclencheurs PL/Perl

PL/Perl peut être utilisé pour écrire des fonctions pour déclencheurs. Dans une fonction déclencheur, la référence hachée `$_TD` contient des informations sur l'événement du déclencheur en cours. `$_TD` est une variable globale qui obtient une valeur locale séparée à chaque appel du déclencheur. Les champs de la référence de hachage `$_TD` sont :

`$_TD->{new}{foo}`
: Valeur `NEW` de la colonne `foo`

`$_TD->{old}{foo}`
: Valeur `OLD` de la colonne `foo`

`$_TD->{name}`
: Nom du déclencheur appelé

`$_TD->{event}`
: Événement du déclencheur : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` ou `UNKNOWN`

`$_TD->{when}`
: Quand le déclencheur a été appelé : `BEFORE` (avant), `AFTER` (après) ou `UNKNOWN` (inconnu)

`$_TD->{level}`
: Le niveau du déclencheur : `ROW` (ligne), `STATEMENT` (instruction) ou `UNKNOWN` (inconnu)

`$_TD->{relid}`
: L'OID de la table sur lequel le déclencheur a été exécuté

`$_TD->{table_name}`
: Nom de la table sur lequel le déclencheur a été exécuté

`$_TD->{relname}`
: Nom de la table sur lequel le déclencheur a été exécuté. Elle est obsolète et pourrait être supprimée dans une prochaine version. Utilisez $_TD->{table_name} à la place.

`$_TD->{table_schema}`
: Nom du schéma sur lequel le déclencheur a été exécuté.

`$_TD->{argc}`
: Nombre d'arguments de la fonction déclencheur

`@{$_TD->{args}}`
: Arguments de la fonction déclencheur. N'existe pas si `$_TD->{argc}` vaut 0.

Les déclencheurs niveau ligne peuvent renvoyer un des éléments suivants :

`return;`
: Exécute l'opération

`"SKIP"`
: N'exécute pas l'opération

`"MODIFY"`
: Indique que la ligne `NEW` a été modifiée par la fonction déclencheur

Voici un exemple d'une fonction déclencheur illustrant certains points ci-dessus :

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # passe la commande INSERT/UPDATE
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # modifie la ligne et exécute la commande INSERT/UPDATE
    } else {
        return;           # exécute la commande INSERT/UPDATE
    }
```

```
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

41.7. PL/Perl sous le capot

41.7.1. Configuration

Cette section liste les paramètres de configuration de PL/Perl. Pour paramétrer n'importe lequel d'entre eux avant que PL/Perl ne soit chargé, il est nécessaire d'avoir ajouté « `plperl` » à la liste custom_variable_classes dans `postgresql.conf`.

`plperl.on_init` (string)

Spécifie un code perl à exécuter lorsque l'interpréteur Perl est initialisé pour la première fois et avant qu'il soit spécialisé pour être utilisé par `plperl` ou `plperlu`. Les fonction SPI ne sont pas disponible lorsque ce code est exécuté. Si le code lève une erreur, il interrompra l'initialisation de l'interpréteur et la propagera à la requête originale, provoquant ainsi l'annulation de la transaction ou sous-transaction courante.

Le code Perl est limité à une seule ligne. Un code plus long peut être placé dans un module et chargé par `on_init`. Exemples:

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

Tous les modules chargés par `plperl.on_init`, directement ou indirectement, seront disponibles depuis `plperl`. Cela entraîne un problème de sécurité potentiel. Pour consulter la liste des modules chargés, vous pouvez utiliser :

```
DO 'elog(WARNING, join ", ", sort keys %INC)' language plperl;
```

L'initialisation aura lieu au sein du postmaster si la librairie plperl est incluse dans le paramètre shared_preload_libraries), auquel cas une plus grande attention doit être portée au risque de déstabiliser ce dernier. The principal reason for making use of this feature is that Perl modules loaded by `plperl.on_init` need be loaded only at postmaster start, and will be instantly available without loading overhead in individual database sessions. However, keep in mind that the overhead is avoided only for the first Perl interpreter used by a database session -- either PL/PerlU, or PL/Perl for the first SQL role that calls a PL/Perl function. Any additional Perl interpreters created in a database session will have to execute `plperl.on_init` afresh. Also, on Windows there will be no savings whatsoever from preloading, since the Perl interpreter created in the postmaster process does not propagate to child processes.

Ce paramètre ne peut être positionné que dans le fichier postgresql.conf ou depuis la ligne de commande de démarrage du serveur.

`plperl.on_plperl_init` (string), `plperl.on_plperlu_init` (string)

These parameters specify Perl code to be executed when a Perl interpreter is specialized for `plperl` or `plperlu` respectively. This will happen when a PL/Perl or PL/PerlU function is first executed in a database session, or when an additional interpreter has to be created because the other language is called or a PL/Perl function is called by a new SQL role. This follows any initialization done by `plperl.on_init`. The SPI functions are not available when this code is executed. The Perl code in `plperl.on_plperl_init` is executed after « locking down » the interpreter, and thus it can only perform trusted operations.

Si le code lève une erreur, il interrompra l'initialisation et la propagera à la requête originale, provoquant ainsi l'annulation de la transaction ou sous-transaction courante. Any actions already done within Perl won't be undone; however, that interpreter won't be used again. If the language is used again the initialization will be attempted again within a fresh Perl interpreter.

Only superusers can change these settings. Although these settings can be changed within a session, such changes will not affect Perl interpreters that have already been used to execute functions.

`plperl.use_strict` (boolean)

Lorsqu'il est positionné à « true », les compilations des fonction PL/Perl suivantes auront le pragma `strict` activé. Ce paramètre n'affecte pas les fonctions déjà compilées au sein de la session courante.

41.7.2. Limitations et fonctionnalités absentes

Les fonctionnalités suivantes ne sont actuellement pas implémentées dans PL/Perl, mais peuvent faire l'objet de contributions généreuses de votre part.

- Les fonctions PL/Perl ne peuvent pas s'appeler entre elles.
- SPI n'est pas complètement implémenté.
- Si vous récupérez des ensembles de données très importants en utilisant `spi_exec_query`, vous devez être conscient qu'ils iront tous en mémoire. Vous pouvez l'éviter en utilisant `spi_query`/`spi_fetchrow` comme montré précédemment.

  Un problème similaire survient si une fonction renvoyant un ensemble passe un gros ensemble de lignes à PostgreSQL via `return`. Vous pouvez l'éviter aussi en utilisant à la place `return_next` pour chaque ligne renvoyée, comme indiqué précédemment.
- Lorsque'une session se termine normalement, et pas à cause d'une erreur fatale, tous les blocs `END` qui ont été définis sont exécutés. Actuellement, aucune autre action ne sont réalisées. Spécifiquement, les descripteurs de fichiers ne sont pas vidés automatiquement et les objets ne sont pas détruits automatiquement.

Chapitre 42. PL/Python - Langage de procédures Python

Le langage de procédures PL/Python permet l'écriture de fonctions PostgreSQL™ avec le *langage Python* (mais voir aussi Section 42.1, « Python 2 et Python 3 »).

Pour installer PL/Python dans une base de données particulières, utilisez `createlang plpythonu` *`nom_base`*.

> **Astuce**
>
> Si un langage est installé dans `template1`, toutes les bases nouvellement créées se verront installées ce langage automatiquement.

Depuis PostgreSQL™ 7.4, PL/Python est seulement disponible en tant que langage « sans confiance » (ceci signifiant qu'il n'offre aucun moyen de restreindre ce que les utilisateurs en font). Il a donc été renommé en `plpythonu`. La variante de confiance `plpython` pourrait être de nouveau disponible dans le futur, si un nouveau mécanisme sécurisé d'exécution est développé dans Python. Le codeur d'une fonction dans PL/Python sans confiance doit faire attention à ce que cette fonction ne puisse pas être utilisée pour réaliser quelque chose qui n'est pas prévue car il sera possible de faire tout ce que peut faire un utilisateur connecté en tant qu'administrateur de la base de données. Seuls les superutilisateurs peuvent créer des fonctions dans des langages sans confiance comme `plpythonu`.

> **Note**
>
> Les utilisateurs des paquets sources doivent activer spécifiquement la construction de PL/Python lors des étapes d'installation (référez-vous aux instructions d'installation pour plus d'informations). Les utilisateurs de paquets binaires pourront trouver PL/Python dans un paquet séparé.

42.1. Python 2 et Python 3

PL/Python accepte à la fois les versions 2 et 3 de Python. (Les instructions d'installation de PostgreSQL peuvent contenir des informations plus précises sur les versions mineures précisément supportées de Python.) Comme les variantes Python 2 et Python 3 sont incompatibles pour certaines parties très importantes, le schéma de nommage et de transition suivant est utilisé par PL/Python pour éviter de les mixer :

- Le langage PostgreSQL nommé `plpython2u` implémente PL/Python sur la variante Python 2 du langage.
- Le langage PostgreSQL nommé `plpython3u` implémente PL/Python sur la variante Python 3 du langage.
- Le langage nommé `plpythonu` implémente PL/Python suivant la variante par défaut du langage Python, qui est actuellement Python 2. (Cette valeur par défaut est indépendante de ce que toute installations locales de Python pourrait considérer comme la valeur par « défaut », par exemplece que pourrait être `/usr/bin/python`.) La valeur par défaut sera probablement changée avec Python 3 dans une prochaine version de PostgreSQL, suivant les progrès de la migration à Python 3 dans la communauté Python.

Cela dépend de la configuration lors de la compilation ou des paquets installés si PL/Python pour Python 2 ou Python 3 ou les deux sont disponibles.

> **Astuce**
>
> La variante construite dépend de la version de Python trouvée pendant l'installation ou de la version sélectionnée explicitement en configurant la variable d'environnement `PYTHON` ; voir Section 15.5, « Procédure d'installation ». Pour que les deux variantes de PL/Python soient disponibles sur une installation, le répertoire des sources doit être configuré et construit deux fois.

Ceci a pour résultat la stratégie suivante d'utilisation et de migration :

- Les utilisateurs existants et ceux qui ne sont pas actuellement intéressés par Python 3 utilisent le nom `plpythonu` et n'ont rien à changer pour l'instant. Il est recommandé de « s'assurer » graduellement de migrer le code vers Python 2.6/2.7 pour simplifier une migration éventuelle vers Python 3.

  En pratique, beaucoup de fonctions PL/Python seront migrées à Python 3 avec peu, voire par du tout, de modifications.

- Les utilisateurs sachant d'avance qu'ils ont du code reposant massivement sur Python 2 et ne planifient pas de changer peuvent utiliser le nom `plpython2u`. Cela continuera de fonctionner, y compris dans un futur lointain, jusqu'à ce que le support de Python 2 soit complètement supprimée de PostgreSQL.
- Les utilisateurs qui veulent utiliser Python 3 peuvent utiliser le nom `plpython3u`, qui continuera à fonctionner en permanence avec les standards actuels. Dans le futur, quand Python 3 deviendra la version par défaut du langage, ils pourront supprimer le chiffre « 3 », principalement pour des raisons esthétiques.
- Les intrépides qui veulent construire un système d'exploitation utilisant seulement Python-3, peuvent modifier les scripts de compilation pour rendre `plpythonu` équivalent à `plpython3u`, en gardant en tête que cela rend leur installation incompatible avec la majorité de ce qui existe dans ce monde.

Voir aussi le document *What's New In Python 3.0* pour plus d'informations sur le portage vers Python 3.

Il n'est pas permis d'utiliser PL/Python basé sur Python 2 et PL/Python basé sur Python 3 dans la même session car les symbôles dans les modules dynamiques entreraient en conflit, ce qui pourrait résulter en des arrêts brutaux du processus serveur PostgreSQL. Une vérification est ajoutée pour empêcher ce mélange de versions majeures Python dans une même sessio. Cette vérification aura pour effet d'annuler la session si une différence est détectée. Néanmoins, il est possible d'utiliser les deux variantes de PL/Python dans une même base de données à condition que ce soit dans des sessions séparées.

42.2. Fonctions PL/Python

Les fonctions PL/Python sont déclarées via la syntaxe standard CREATE FUNCTION(7) :

```
CREATE FUNCTION nom_fonction (liste-arguments)
  RETURNS return-type
AS $$
  # corps de la fonction PL/Python
$$ LANGUAGE plpythonu;
```

Le corps d'une fonction est tout simplement un script Python. Quand la fonction est appelée, ses arguments sont passés au script Python comme des éléments de la liste `args` ; les arguments nommés sont en plus passés comme des variables ordinaires. L'utilisation des arguments nommés est beaucoup plus lisible. Le résultat est renvoyé par le code Python de la façon habituelle, avec `return` ou `yield` (dans le cas d'une instruction avec un ensemble de résultats). Si vous ne fournissez pas une valeur de retour, Python renvoie la valeur par défaut None. PL/Python traduit la valeur None de Python comme une valeur NULL SQL.

Par exemple, une fonction renvoyant le plus grand de deux entiers peut être définie ainsi :

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

Le code Python donné comme corps de la définition de fonction est transformé en fonction Python. Par exemple, le code ci-dessus devient :

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

en supposant que 23456 est l'OID affecté à la fonction par PostgreSQL™.

Les arguments sont définis comme des variables globales. Conséquence subtile des règles sur la portée de variables dans Python, il n'est pas possible de réaffecter une variable à l'intérieur d'une fonction en conservant son nom, sauf si elle est préalablement déclarée comme globale à l'intérieur du bloc. Ainsi, l'exemple suivant ne fonctionnera pas :

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpythonu;
```

car affecter la variable `x` la transforme en variable locale pour ce bloc et que, par conséquent, la variable `x` de l'expression de

droite fait référence à une variable locale `x` non encore définie, et non pas au paramètre de la fonction PL/Python. L'utilisation du mot-clé `global` permet de résoudre le problème :

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # ok now
  return x
$$ LANGUAGE plpythonu;
```

Cependant, il vaut mieux ne pas trop s'appuyer sur ce détail d'implémentation de PL/Python. Il est préférable de traiter les paramètres de fonction comme étant en lecture seule.

42.3. Valeur des données avec PL/Python

De manière générale, le but de PL/Python est de fournir une relation « naturelle » entre PostgreSQL et le monde Python. Ces règles relationelles sont décrites ci-dessous.

42.3.1. Type de données

Les paramètres de fonctions sont convertis de leur type PostgreSQL vers un type correspondant en Python :

- Le type boolean PostgreSQL est converti en bool Python.
- Les smallint et int de PostgreSQL sont convertis en int Python. Le bigint PostgreSQL est converti en long pour Python 2 et en int pour Python 3.
- Les real, double et numeric de PostgreSQL sont convertis en float Python. Notez que pour numeric, cela entraine une perte d'information et peut aboutir à des résulats incorrects. Cela devrait être corrigé dans une future version.
- Le bytea PostgreSQL est converti en str pour Python 2 et en bytes pour Python 3. Avec Python 2, la chaîne devrait être traitée comme une séquence d'octets sans encodage.
- Tous les autres types de données, y compris les chaînes de caractères PostgreSQL, sont convertis en str Python. En Python 2, ces chaînes auront le même encodage de caractères que le serveur. En Python 3, ce seront des chaînes Unicode comme les autres.
- Pour les données non scalaires, voir ci-dessous.

Les valeurs renvoyées par les fonctions sont converties en types de retour PostgreSQL comme suit:

- Quand le type de la valeur PostgreSQL renvoyée est boolean, la valeur de retrour sera évaluée en fonction des règles *Python*. Ainsi, les 0 et les chaines vides sont fausses, mais la valeur `'f'` est vraie.
- Quand le type de la valeur PostgreSQL renvoyée est bytea, la valeur de retour sera convertie en chaine de caractères (Python 2) ou en octets (Python 3) en utilisant les mécanismes Python correspondants, le résultat étant ensuite converti en bytea.
- Pour tous les autres types de retrour PostgreSQL, la valeur renvoyée par Python est convertie en chaine de caractères en utilisant la méthode Python `str`, et et le résultat est transmis à la fonction d'entrée du type de données PostgreSQL.

  Les chaines de caractères en Python 2 doivent être transmises dans le même encodage que celui du serveur PostgreSQL. Les chaines invalides dans l'encodage du serveur entraineront la levée d'une erreur, mais toutes les erreurs d'encodage ne sont pas detectées, ce qui peut aboutir à une corruption des données lorsque ces règles ne sont pas respéctée. Les chaines Unicode sont automatiquement converties dans le bon encodage, il est donc plus prudent de les utiliser. Dans Python 3, toutes les chaines sont en Unicode.
- Pour les données non scalaires, voire ci dessous.

Notez que les erreurs logiques entre le type de retour déclaré dans PostgreSQL et le type de l'objet Python renvoyé ne sont pas détectées. La valeur sera convertie dans tous les cas.

Astuce

Les fonctions PL/Python ne peuvent renvoyer les types RECORD ou SETOF RECORD. Une solution de contournement est d'écrire une fonction PL/pgSQL qui crée une table temporaire, d'appeler la fonction en PL/Python pour remplir cette table puis de laisser à la fonction PL/pgSQL le soin de renvoyer le type générique RECORD de cette table temporaire.

42.3.2. Null, None

Si une valeur SQL NULL est passée à une fonction, la valeur de l'argument apparaîtra comme None au niveau de Python. Par exemple, la définition de la fonction `pymax` montrée dans Section 42.2, « Fonctions PL/Python » renverra la mauvaise réponse pour les valeurs NULL. Nous pouvons ajouter `STRICT` à la définition de la fonction pour faire en sorte que PostgreSQL™ fasse quelque-chose de plus raisonnable : si une valeur NULL est passée, la fonction ne sera pas appelée du tout mais renverra juste un résultat NULL automatiquement. Sinon, vous pouver vérifier les entrées NULL dans le corps de la fonction :

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

Comme montré ci-dessus, pour renvoyer une valeur SQL NULL à partir d'une fonction PL/Python, renvoyez la valeur None. Ceci peut se faire que la fonction soit stricte ou non.

42.3.3. Tableaux, Listes

Les valeurs de type tableaux SQL sont passées via PL/Python comme des listes Python. Pour renvoyer une valeur de type tableau SQL par une fonction PL/Python, renvoyez une séquence Python, par exemple une liste ou un tuple :

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Notez que, avec Python, les chaînes sont des séquences, ce qui peut avoir des effets indésirables qui peuvent être familiers aux codeurs Python :

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

42.3.4. Types composites

Les arguments de type composite sont passés à la fonction via une correspondance Python. Les noms d'élément de la correspondance sont les noms d'attribut du type composite. Si un attribut a une valeur NULL dans la ligne traitée; il a la valeur NULL dans sa correspondance. Voici un exemple :

```
CREATE TABLE employe (
  nom text,
  salaire integer,
  age integer
);

CREATE FUNCTION trop_paye (e employe)
  RETURNS boolean
AS $$
```

```
  if e["salaire"] > 200000:
    return True
  if (e["age"] < 30) and (e["salaire"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

Il existe plusieurs façon de renvoyer une ligne ou des types composites à partir d'une fonction Python. Les exemples suivants supposent que nous avons :

```
CREATE TABLE valeur_nommee (
  nom   text,
  valeur  integer
);
```

ou

```
CREATE TYPE valeur_nommee AS (
  nom   text,
  valeur  integer
);
```

Une valeur composite peut être renvoyé comme :

Un type séquence (ligne ou liste), mais pas un ensemble parce que ce n'est pas indexable

Les objets séquences renvoyés doivent avoir le même nombre d'éléments que le type composite a de champs. L'élément d'index 0 est affecté au premier champ du type composite, 1 au second et ainsi de suite. Par exemple :

```
CREATE FUNCTION cree_paire (nom text, valeur integer)
  RETURNS valeur_nommee
AS $$
  return [ nom, valeur ]
  # ou autrement, en tant que ligne : return ( nom, valeur )
$$ LANGUAGE plpythonu;
```

Pour renvoyer NULL dans une colonne, insérez None à la position correspondante.

Correspondance (dictionnaire)

La valeur de chaque colonne du type résultat est récupérée à partir de la correspondance avec le nom de colonne comme clé. Exemple :

```
CREATE FUNCTION cree_paire (nom text, valeur integer)
  RETURNS valeur_nommee
AS $$
  return { "nom": nom, "valeur": valeur }
$$ LANGUAGE plpythonu;
```

Des paires clés/valeurs supplémentaires du dictionnaire sont ignorées. Les clés manquantes sont traitées comme des erreurs. Pour renvoyer NULL comme une colonne, insérez None avec le nom de la colonne correspondante comme clé.

Objet (tout objet fournissant la méthode `__getattr__`)

Ceci fonctionne de la même façon qu'une correspondance. Exemple :

```
CREATE FUNCTION cree_paire (nom text, valeur integer)
  RETURNS valeur_nommee
AS $$
  class valeur_nommee:
    def __init__ (self, n, v):
      self.nom = n
      self.valeur = v
  return valeur_nommee(nom, valeur)

  # ou simplement
  class nv: pass
  nv.nom = nom
  nv.valeur = valeur
  return nv
$$ LANGUAGE plpythonu;
```

42.3.5. Set-Returning Functions

Une fonction PL/Python peut aussi renvoyer des ensembles scalaires ou des types composites. Il existe plusieurs façon de faire ceci parce que l'objet renvoyé est transformé en interne en itérateur. Les exemples suivants supposent que nous avons le type composite :

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

Un résultat ensemble peut être renvoyé à partir de :

Un type séquence (ligne, liste, ensemble)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # renvoie la ligne contenant les listes en tant que types composites
  # toutes les autres combinaisons fonctionnent aussi
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

L'itérateur (tout objet fournissant les méthodes __iter__ et next)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

Le générateur (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

Avertissement

À cause du *bogue #1483133* de Python, certaines versions de débogage de Python 2.4 (configuré et compilé avec l'option `--with-pydebug`) sont connues pour arrêter brutalement le serveur PostgreSQL™ lors de l'utilisation d'un itérateur pour renvoyer un résultat ensemble. Les versions non corrigées de Fedora 4 contiennent ce bogue. Cela n'arrive pas dans les versions de production de Python et sur les versions corrigées de Fedora 4.

42.4. Sharing Data

Le dictionnaire global `SD` est disponible pour stocker des données entres les appels de fonctions. Cette variable est une donnée statique privée. Le dictionnaire global `GD` est une donnée publique disponible pour toutes les fonctions Python à l'intérieur d'une session. À utiliser avec précaution.

Chaque fonction obtient son propre environnement d'exécution dans l'interpréteur Python, de façon à ce que les données globales et les arguments de fonction provenant de `ma_fonction` ne soient pas disponibles depuis `ma_fonction2`. L'exception concerne les données du dictionnaire `GD` comme indiqué ci-dessus.

42.5. Blocs de code anonymes

PL/Python accepte aussi les blocs de code anonymes appelés avec l'instruction DO(7) :

```
DO $$
    # Code PL/Python
$$ LANGUAGE plpythonu;
```

Un bloc de code anonyme ne reçoit aucun argument et, quelque soit la valeur renvoyée, elle est ignorée. Sinon, ce bloc se comporte exactement comme n'importe quelle fonction.

42.6. Fonctions de déclencheurs

Quand une fonction est utilisée par un trigger, le dictionnaire `TD` contient les valeurs relatives au trigger :

`TD["event"]`
: contient l'événement sous la forme d'une chaîne : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` ou `UNKNOWN`.

`TD["when"]`
: contient une chaîne valant soit `BEFORE`, soit `AFTER` soit `UNKNOWN`.

`TD["level"]`
: contient une chaîne valant soit `ROW` soit `STATEMENT` soit `UNKNOWN`.

`TD["new"]`, `TD["old"]`
: pour un trigger au niveau ligne, ces champs contiennent les lignes du trigger, l'ancienne version et la nouvelle version ; les deux champs ne sont pas forcément disponibles, ceci dépendant de l'événement qui a déclenché le trigger

`TD["name"]`
: contient le nom du trigger.

`TD["table_name"]`
: contient le nom de la table sur laquelle le trigger a été déclenché

`TD["table_schema"]`
: contient le schéma de la table sur laquelle le trigger a été déclenché

`TD["relid"]`
: contient l'OID de la table sur laquelle le trigger a été déclenché

`TD["args"]`
: si la commande **CREATE TRIGGER** comprend des arguments, ils sont disponibles dans les variables allant de `TD["args"][0]` à `TD["args"][n-1]`.

Si `TD["when"]` vaut `BEFORE` et si `TD["level"]` vaut `ROW`, vous pourriez renvoyer `None` ou `"OK"` à partir de la fonction Python pour indiquer que la ligne n'est pas modifiée, `"SKIP"` pour annuler l'événement ou `"MODIFY"` pour indiquer que vous avez modifié la ligne. Sinon la valeur de retour est ignorée.

42.7. Accès à la base de données

Le module du langage PL/Python importe automatiquement un module Python appelé `plpy`. Les fonctions et constantes de ce module vous sont accessibles dans le code Python via `plpy.`*`foo`*.

Le module `plpy` propose deux fonctions appelées `execute` et `prepare`. Appeler `plpy.execute` avec une requête sous forme de chaîne de caractères et un argument optionnel de limite fait que la requête est exécutée et le résultat renvoyé dans un objet résultat. Cet objet émule un objet liste ou dictionnaire. L'accès aux résultats se fait par numéro de ligne et nom de colonne. Deux méthodes supplémentaires sont utilisables : `nrows` qui renvoit le nombre de lignes renvoyées par la requête, et `status` qui correspond à la valeur de retour de `SPI_execute()`. L'objet résultat est modifiable.

Par exemple :

```
rv = plpy.execute("SELECT * FROM ma_table", 5)
```

renvoit jusqu'à cinq lignes de `ma_table`. Si `ma_table` a une colonne `ma_colonne`, son contenu peut être récupéré ainsi :

```
foo = rv[i]["ma_colonne"]
```

la seconde fonction, `plpy.prepare`, prépare le plan d'exécution d'une requête. Il utilise comme arguments une chaîne de caractères pour la requête et une liste des types de paramètres si des références de paramètres sont indiquées dans la requête. Par exemple :

```
plan = plpy.prepare("SELECT nom FROM mes_utilisateurs WHERE prenom = $1", [ "text" ])
```

`text` est le type de la variable que vous devrez passer pour `$1`. Après avoir préparé une requête, vous devez utiliser la fonction `plpy.execute` pour l'exécuter :

```
rv = plpy.execute(plan, [ "nom" ], 5)
```

Le troisième argument, optionnel, est la limite.

Les paramètres de requêtes et les champs de résultats sont convertis entre PostgreSQL et les types de données Python comme indiqué dans Section 42.3, « Valeur des données avec PL/Python ». L'exception est que les types composites ne sont pas actuellement supportés : ils sont rejetés dans le cas des paramètres de requête et convertis en chaînes de caractères quand ils apparaissent dans le résultat d'une requête. Pour contourner ce deuxième cas, la requête peut être quelque fois écrite de façon à ce que le type composite apparaisse comme une ligne de résultat plutôt que comme le champ d'une ligne du résultat. Autrement, la chaîne résultante peut être analysée manuellement mais cette approche n'est pas recommendée car une version future pourrait demander de refaire l'analyse de la chaîne en retour.

Quand vous préparez un plan en utilisant le module PL/Python, il est automatiquement sauvegardé. Lisez la documentation SPI (Chapitre 43, Interface de programmation serveur) pour une description complète. Pour en avoir une utilisation réelle via des appels de fonctions, vous avez besoin d'utiliser un dictionnaire de stockage persistent `SD` ou `GD` (voir Section 42.4, « Sharing Data »). Par exemple :

```
CREATE FUNCTION utilise_plan_sauvegarde() RETURNS trigger AS $$
    if SD.has_key("plan"):
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # reste de la fonction
$$ LANGUAGE plpythonu;
```

42.8. Fonctions outils

Le module `plpy` fournit aussi les fonctions `plpy.debug(msg)`, `plpy.log(msg)`, `plpy.info(msg)`, `plpy.notice(msg)`, `plpy.warning(msg)`, `plpy.error(msg)` et `plpy.fatal(msg)`. `plpy.error` et `plpy.fatal("msg")` lèvent une exception Python qui, si non attrapée, se propage à la requête appelante causant l'annulation de la transaction ou sous-transaction en cours. `raise plpy.Error(msg)` et `raise plpy.Fatal(msg)` sont équivalent à appeler, respectivement, `plpy.error` et `plpy.fatal`. Les autres fonctions génèrent uniquement des messages de niveaux de priorité différents. Que les messages d'une priorité particulière soient reportés au client, écrit dans les journaux du serveur ou les deux, cette configuration est contrôlée par les variables log_min_messages et client_min_messages. Voir le Chapitre 18, Configuration du serveur pour plus d'informations.

42.9. Variables d'environnement

Certaines des variables d'environnement qui sont acceptées par l'interpréteur Python peuvent aussi être utilisées pour modifier le comportement de PL/Python. Elles doivent être configurées dans l'environnement du processus serveur PostgreSQL principal, par exemple dans le script de démarrage. Les variables d'environnement disponibles dépendent de la version de Python ; voir la documentation de Python pour les détails. Au moment de l'écriture de ce chapitre, les variables d'environnement suivantes avaient un comportement sur PL/Python, à condition d'utiliser une version adéquate de Python :

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`

- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`
- `PYTHONVERBOSE`
- `PYTHONCASEOK`
- `PYTHONDONTWRITEBYTECODE`
- `PYTHONIOENCODING`
- `PYTHONUSERBASE`

(Cela semble être un détail d'implémentation de Python, en dehors du contrôle de PL/Python, qui fait que certaines variables d'environnement listées dans la page man de **python** sont seulement utilisables avec l'interpréteur en ligne de commande et non avec un interpréteur Python embarqué.)

Chapitre 43. Interface de programmation serveur

L'*interface de programmation serveur* (SPI) donne aux auteurs de fonctions C la capacité de lancer des commandes SQL au sein de leurs fonctions. SPI est une série de fonctions d'interface simplifiant l'accès à l'analyseur, au planificateur et au lanceur. SPI fait aussi de la gestion de mémoire.

> **Note**
>
> Les langages procéduraux disponibles donnent plusieurs moyens de lancer des commandes SQL à partir de procédures. La plupart est basée à partir de SPI. Cette documentation présente donc également un intérêt pour les utilisateurs de ces langages.

Pour assurer la compréhension, nous utiliserons le terme de « fonction » quand nous parlerons de fonctions d'interface SPI et « procédure » pour une fonction C définie par l'utilisateur et utilisant SPI.

Notez que si une commande appelée via SPI échoue, alors le contrôle ne sera pas redonné à votre procédure. Au contraire, la transaction ou sous-transaction dans laquelle est exécutée votre procédure sera annulée. (Ceci pourrait être surprenant étant donné que les fonctions SPI ont pour la plupart des conventions documentées de renvoi d'erreur. Ces conventions s'appliquent seulement pour les erreurs détectées à l'intérieur des fonctions SPI.) Il est possible de récupérer le contrôle après une erreur en établissant votre propre sous-transaction englobant les appels SPI qui pourraient échouer. Ceci n'est actuellement pas documenté parce que les mécanismes requis sont toujours en flux.

Les fonctions SPI renvoient un résultat positif en cas de succès (soit par une valeur de retour entière, soit dans la variable globale `SPI_result` comme décrit ci-dessous). En cas d'erreur, un résultat négatif ou NULL sera retourné.

Les fichiers de code source qui utilisent SPI doivent inclure le fichier d'en-tête `executor/spi.h`.

43.1. Fonctions d'interface

Nom

SPI_connect — connecter une procédure au gestionnaire SPI

Synopsis

```
int SPI_connect(void)
```

Description

`SPI_connect` ouvre une connexion au gestionnaire SPI lors de l'appel d'une procédure. Vous devez appeler cette fonction si vous voulez lancer des commandes au travers du SPI. Certaines fonctions SPI utilitaires peuvent être appelées à partir de procédures non connectées.

Si votre procédure est déjà connectée, `SPI_connect` retournera le code d'erreur SPI_ERROR_CONNECT. Cela peut arriver si une procédure qui a appelé `SPI_connect` appelle directement une autre procédure qui appelle `SPI_connect`. Bien que des appels récursifs au gestionnaire SPI soient permis lorsqu'une commande SQL appelée au travers du SPI invoque une autre fonction qui utilise SPI, les appels directement intégrés à `SPI_connect` et `SPI_finish` sont interdits (mais voir `SPI_push` et `SPI_pop`).

Valeur de retour

SPI_OK_CONNECT
: en cas de succès

SPI_ERROR_CONNECT
: en cas d'échec

Nom

SPI_finish — déconnecter une procédure du gestionnaire SPI

Synopsis

```
int SPI_finish(void)
```

Description

`SPI_finish` ferme une connexion existante au gestionnaire SPI. Vous devez appeler cette fonction après avoir terminé les opérations SPI souhaitées pendant l'invocation courante de votre procédure. Vous n'avez pas à vous préoccuper de ceci, sauf si vous terminez la transaction via `elog(ERROR)`. Dans ce cas, SPI terminera automatiquement.

Si `SPI_finish` est appelée sans avoir une connexion valable, elle retournera SPI_ERROR_UNCONNECTED. Il n'y a pas de problème fondamental avec cela ; le gestionnaire SPI n'a simplement rien à faire.

Valeur de retour

SPI_OK_FINISH
: si déconnectée correctement

SPI_ERROR_UNCONNECTED
: si appel à partir d'une procédure non connectée

Nom

SPI_push — pousse la pile SPI pour autoriser une utilisation récursive de SPI

Synopsis

```
void SPI_push(void)
```

Description

`SPI_push` devrait être appelé avant d'exécuter une autre procédure qui pourrait elle-même souhaiter utiliser SPI. Après `SPI_push`, SPI n'est plus dans un état « connecté » et les appels de fonction SPI seront rejetés sauf si un nouveau `SPI_connect` est exécuté. Ceci nous assure une séparation propre entre l'état SPI de votre procédure et celui d'une autre procédure que vous appelez. Après le retour de cette dernière, appelez `SPI_pop` pour restaurer l'accès à votre propre état SPI.

Notez que `SPI_execute` et les fonctions relatives font automatiquement l'équivalent de `SPI_push` avant de repasser le contrôle au moteur d'exécution SQL, donc il n'est pas nécessaire de vous inquiéter de cela lors de l'utilisation de ces fonctions. Vous aurez besoin d'appeler `SPI_push` et `SPI_pop` seulement quand vous appelez directement un code arbitraire qui pourrait contenir des appels `SPI_connect`.

Nom

SPI_pop — récupère la pile SPI pour revenir de l'utilisation récursive de SPI

Synopsis

```
void SPI_pop(void)
```

Description

`SPI_pop` enlève l'environnement précédent de la pile d'appel SPI. Voir `SPI_push`.

Nom

SPI_execute — exécute une commande

Synopsis

```
int SPI_execute(const char * command, bool read_only, long count)
```

Description

`SPI_exec` lance la commande SQL spécifiée pour *`count`* lignes. Si *`read_only`* est `true`, la commande doit être en lecture seule et la surcharge de l'exécution est quelque peu réduite.

Cette fonction ne devrait être appelée qu'à partir d'une procédure connectée.

Si *`count`* vaut zéro, alors la commande est exécutée pour toutes les lignes auxquelles elle s'applique. Si *`count`* est supérieur à 0, alors pas plus de *`count`* lignes seront récupérées. L'exécution s'arrêtera quand le compte est atteint, un peu comme l'ajout d'une clause `LIMIT` à une requête. Par exemple :

```
SPI_execute("SELECT * FROM foo", true, 5);
```

récupérera 5 lignes tout au plus à partir de la table. Notez qu'une telle limite n'est efficace qu'à partir du moment où la requête renvoie des lignes. Par exemple :

```
 SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

insérera toutes les lignes de bar, en ignorant le paramètre *`count`*. Cependant, avec

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

au plus cinq lignes seront insérées car l'exécution s'arrêtera après la cinquième ligne renvoyée par `RETURNING`.

Vous pourriez passer plusieurs commandes dans une chaîne, mais ces commandes ne peuvent pas dépendre d'objets créés plus tôt dans la chaîne car toute la chaîne est analysée et planifiée avant le début de l'exécution. `SPI_execute` renvoie le résultat pour la dernière commande exécutée. La limite *`count`* s'applique à chaque commande séparément (même si seul le dernier résultat sera renvoyé). La limite n'est pas appliquée à toute commande cachée générée par les règles.

Quand *`read_only`* vaut `false`, `SPI_execute` incrémente le compteur de la commande et calcule une nouvelle *image* avant d'exécuter chaque commande dans la chaîne. L'image n'est pas réellement modifiée si le niveau d'isolation de la transaction en cours est `SERIALIZABLE` mais, en mode `READ COMMITTED`, la mise à jour de l'image permet à chaque commande de voir les résultats des transactions nouvellement validées à partir des autres sessions. Ceci est essentiel pour un comportement cohérent quand les commandes modifient la base de données.

Quand *`read_only`* vaut `true`, `SPI_execute` ne met à jour ni l'image ni le compteur de commandes, et il autorise seulement les commandes **SELECT** dans la chaîne des commandes. Elles sont exécutées en utilisant l'image précédemment établie par la requête englobante. Ce mode d'exécution est un peu plus rapide que le mode lecture/écriture à cause de l'élimination de la surcharge par commande. Il autorise aussi directement la construction des fonctions *stable* comme les exécutions successives utiliseront toutes la même image, il n'y aura aucune modification dans les résultats.

Il n'est généralement pas conseillé de mixer les commandes en lecture seule et les commandes en lecture/écriture à l'intérieur d'une seule fonction utilisant SPI ; ceci pourrait causer un comportement portant confusion car les requêtes en mode lecture seule devraient ne pas voir les résultats de toute mise à jour de la base de données effectuées par les requêtes en lecture/écriture.

Le nombre réel de lignes pour lesquelles la (dernière) commande a été lancée est retourné dans la variable globale `SPI_processed`. Si la valeur de retour de la fonction est SPI_OK_SELECT, SPI_OK_INSERT_RETURNING, SPI_OK_DELETE_RETURNING ou SPI_OK_UPDATE_RETURNING, alors vous pouvez utiliser le pointeur global `SPITupleTable *SPI_tuptable` pour accéder aux lignes de résultat. Quelques commandes (comme **EXPLAIN**) renvoient aussi des ensembles de lignes et `SPI_tuptable` contiendra aussi le résultat dans ces cas.

La structure SPITupleTable est définie comme suit :

```
typedef struct
{
    MemoryContext tuptabcxt;    /* contexte mémoire de la table de résultat */
    uint32      alloced;        /* nombre de valeurs allouées */
    uint32      free;           /* nombre de valeurs libres */
    TupleDesc   tupdesc;        /* descripteur de rangées */
    HeapTuple  *vals;           /* rangées */
} SPITupleTable;
```

vals est un tableau de pointeurs vers des lignes (le nombre d'entrées valables est donné par SPI_processed). *tupdesc* est un descripteur de ligne que vous pouvez passer aux fonctions SPI qui traitent des lignes. *tuptabcxt*, *alloced* et *free* sont des champs internes non conçus pour être utilisés par des routines SPI appelantes.

SPI_finish libère tous les SPITupleTables allouées pendant la procédure courante. Vous pouvez libérer une table de résultats donnée plus tôt, si vous en avez terminé avec elle, en appelant SPI_freetuptable.

Arguments

const char * *command*
: chaîne contenant la commande à exécuter

bool *read_only*
: true en cas d'exécution en lecture seule

long *count*
: nombre maximum de lignes à traiter ou 0 pour aucune limite

Valeur de retour

Si l'exécution de la commande a réussi, alors l'une des valeurs (positives) suivantes sera renvoyée :

SPI_OK_SELECT
: si un **SELECT** (mais pas **SELECT INTO**) a été lancé

SPI_OK_SELINTO
: si un **SELECT INTO** a été lancé

SPI_OK_INSERT
: si un **INSERT** a été lancé

SPI_OK_DELETE
: si un **DELETE** a été lancé

SPI_OK_UPDATE
: si un **UPDATE** a été lancé

SPI_OK_INSERT_RETURNING
: si un **INSERT RETURNING** a été lancé

SPI_OK_DELETE_RETURNING
: si un **DELETE RETURNING** a été lancé

SPI_OK_UPDATE_RETURNING
: si un **UPDATE RETURNING** a été lancé

SPI_OK_UTILITY
: si une commande utilitaire (c'est-à-dire **CREATE TABLE**) a été lancée

SPI_OK_REWRITTEN
: si la commande a été réécrite dans un autre style de commande (c'est-à-dire que **UPDATE** devient un **INSERT**) par une règle.

Sur une erreur, l'une des valeurs négatives suivante est renvoyée :

SPI_ERROR_ARGUMENT
: si *command* est NULL ou *count* est inférieur à 0

SPI_ERROR_COPY
: si **COPY TO stdout** ou **COPY FROM stdin** ont été tentés

SPI_ERROR_TRANSACTION
: Si une commande de manipulation de transaction a été tentée (**BEGIN**, **COMMIT**, **ROLLBACK**, **SAVEPOINT**, **PREPARE TRANSACTION**, **COMMIT PREPARED**, **ROLLBACK PREPARED** ou toute variante de ces dernières)

SPI_ERROR_OPUNKNOWN
: si le type de commande est inconnu (ce qui ne devrait pas arriver)

SPI_ERROR_UNCONNECTED
: si appel à partir d'une procédure non connectée

Notes

Toutes les fonctions d'exécution de requêtes SPI changent à la fois `SPI_processed` et `SPI_tuptable` (juste le pointeur, pas le contenu de la structure). Sauvegardez ces deux variables globales dans des variables locales de procédures si vous voulez accéder à la table des résultats de `SPI_execute` ou d'une fonction d'exécution de requêtes sur plusieurs appels.

Nom

SPI_exec — exécute une commande en lecture/écriture

Synopsis

```
int SPI_exec(const char * command, long count)
```

Description

`SPI_exec` est identique à `SPI_execute`, mais le paramètre *`read_only`* de ce dernier est bloqué sur la valeur `false`.

Arguments

`const char *` *`command`*
: chaîne contenant la commande à exécuter

`long` *`count`*
: nombre maximum de lignes à renvoyer ou `0` pour aucune limite

Valeur de retour

Voir `SPI_execute`.

Nom

SPI_execute_with_args — exécute une commande avec des paramètres hors ligne

Synopsis

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

Description

`SPI_execute_with_args` exécute une commande qui pourrait inclure des références à des paramètres fournis en externe. Le texte de commande fait référence à un paramètre avec `$n` et l'appel spécifie les types et valeurs des données pour chaque symbole de ce type. *`read_only`* et *`count`* ont la même interprétation que dans `SPI_execute`.

Le principal avantage de cette routine comparé à `SPI_execute` est que les valeurs de données peuvent être insérées dans la commande sans mise entre guillemets et échappements, et donc avec beaucoup moins de risques d'attaques du type injection SQL.

Des résultats similaires peuvent être réalisés avec `SPI_prepare` suivi par `SPI_execute_plan` ; néanmoins, lors de l'utilisation de cette fonction, le plan de requête est personnalisé avec les valeurs de paramètres spécifiques fournies. Pour une exécution simple, cette fonction doit être préférée. Si la même commande doit être exécutée avec plusieurs paramètres différents, chaque méthode peut être la plus rapide, le coût de la planification pouvant contre-balancer les bénéfices des plans personnalisés.

Arguments

`const char *` *`command`*
: chaîne de commande

`int` *`nargs`*
: nombre de paramètres en entrée (`$1`, `$2`, etc.)

`Oid *` *`argtypes`*
: un tableau contenant les OID des types de données des paramètres

`Datum *` *`values`*
: un tableau des valeurs réelles des paramètres

`const char *` *`nulls`*
: un tableau décrivant les paramètres NULL

  Si *`nulls`* vaut NULL, alors `SPI_execute_with_args` suppose qu'aucun paramètre n'est NULL.

`bool` *`read_only`*
: `true` pour les exécutions en lecture seule

`long` *`count`*
: nombre maximum de lignes à renvoyer ou `0` pour aucune limite

Valeur de retour

La valeur de retour est identique à celle de `SPI_execute`.

`SPI_processed` et `SPI_tuptable` sont configurés comme dans `SPI_execute` en cas de succès.

Nom

SPI_prepare — prépare un plan pour une commande sans l'exécuter tout de suite

Synopsis

```
SPIPlanStr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

Description

`SPI_prepare` crée et retourne un plan d'exécution pour la commande spécifiée mais ne lance pas la commande. Cette fonction ne peut être appelée que depuis une procédure connectée.

Lorsque la même commande ou une commande semblable doit être lancée à plusieurs reprises, il peut être intéressant de ne faire la planification que d'une seule fois. `SPI_prepare` convertit une chaîne de commande en un plan d'exécution qui peut être lancé plusieurs fois en utilisant `SPI_executeplan`.

Une commande préparée peut être généralisée en utilisant les paramètres (`$1`, `$2`, etc.) en lieu et place de ce qui serait des constantes dans une commande normale. Les valeurs actuelles des paramètres sont alors spécifiées lorsque `SPI_executeplan` est appelée. Ceci permet à la commande préparée d'être utilisée sur une plage plus grande de situations que cela ne serait possible sans paramètres.

Le plan renvoyé par `SPI_prepare` ne peut être utilisé que dans l'invocation courante de la procédure puisque `SPI_finish` libère la mémoire allouée pour le plan. Mais un plan peut être sauvegardé plus longtemps par l'utilisation de la fonction `SPI_saveplan`.

Arguments

`const char * command`
: chaîne contenant la commande à planifier

`int nargs`
: nombre de paramètres d'entrée (`$1`, `$2`, etc.)

`Oid * argtypes`
: pointeur vers un tableau contenant les OID des types de données des paramètres

Valeurs de retour

`SPI_prepare` retourne un pointeur non nul vers un plan d'exécution. En cas d'erreur, NULL sera retourné et `SPI_result` sera positionnée à un des mêmes codes d'erreur utilisés par `SPI_execute` sauf qu'il est positionné à SPI_ERROR_ARGUMENT si *`command`* est NULL ou si *`nargs`* est inférieur à 0 ou si *`nargs`* est supérieur à 0 et *`typesargs`* est NULL.

Notes

SPIPlanPtr est déclaré comme un pointeur vers un type de structure opaque dans `spi.h`. Il est déconseillé d'essayer d'accéder à son contenu directement car cela rend votre code plus fragile aux futures versions de PostgreSQL™.

Il y a un inconvénient à utiliser les paramètres : puisque le planificateur ne connaît pas les valeurs qui seront utilisées pour les paramètres, il effectuera des choix pires que ceux qu'il prendrait pour une commande normale avec toutes les constantes visibles.

Nom

SPI_prepare_cursor — prépare un plan pour une commande, sans l'exécuter pour l'instant

Synopsis

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs, Oid * argtypes, int
cursorOptions)
```

Description

`SPI_prepare_cursor` est identique à `SPI_prepare`, sauf qu'il permet aussi la spécification du paramètre des « options du curseur » du planificateur. Il s'agit d'un champ de bits dont les valeurs sont indiquées dans `nodes/parsenodes.h` pour le champ *`options`* de DeclareCursorStmt. `SPI_prepare` utilise zéro pour les options du curseur.

Arguments

`const char *` *`command`*
: chaîne commande

`int` *`nargs`*
: nombre de paramètres en entrée (`$1`, `$2`, etc.)

`Oid *` *`argtypes`*
: pointeur vers un tableau contenant l'OID des types de données des paramètres

`int` *`cursorOptions`*
: champ de bits précisant les options du curseur ; zéro est le comportement par défaut

Valeur de retour

`SPI_prepare_cursor` a les mêmes conventions pour la valeur de retour que `SPI_prepare`.

Notes

Les bits utiles pour *`cursorOptions`* incluent CURSOR_OPT_SCROLL, CURSOR_OPT_NO_SCROLL et CURSOR_OPT_FAST_PLAN. Notez en particulier que CURSOR_OPT_HOLD est ignoré.

Nom

SPI_prepare_params — prépare un plan pour une commande, mais sans l'exécuter

Synopsis

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

Description

`SPI_prepare_params` crée et renvoie un plan d'exécution pour la commande indiquée mais n'exécute pas la commande. Cette fonction est équivalente à `SPI_prepare_cursor` avec en plus le fait que l'appelant peut indiquer des fonctions pour contrôler l'analyse de références de paramètres externes.

Arguments

`const char *` *`command`*
: chaîne correspondant à la commande

`ParserSetupHook` *`parserSetup`*
: fonction de configuration de l'analyseur

`void *` *`parserSetupArg`*
: argument passé à *`parserSetup`*

`int` *`cursorOptions`*
: masque de bits des options du curseur, sous la forme d'un entier ; zéro indique le comportement par défaut

Code de retour

`SPI_prepare_params` a les mêmes conventions de retour que `SPI_prepare`.

Nom

SPI_getargcount — renvoie le nombre d'arguments nécessaires au plan préparé par `SPI_prepare`

Synopsis

```
int SPI_getargcount(SPIPlanPtr plan)
```

Description

`SPI_getargcount` renvoie le nombre d'arguments nécessaires pour exécuter un plan préparé par `SPI_prepare`.

Arguments

`SPIPlanPtr` *`plan`*
: plan d'exécution (renvoyé par `SPI_prepare`)

Code de retour

Le nombre d'arguments attendus par le *`plan`*. Si *`plan`* est NULL ou invalide, `SPI_result` est initialisé à SPI_ERROR_ARGUMENT et `-1` est renvoyé.

Nom

SPI_getargtypeid — renvoie l'OID du type de données pour un argument du plan préparé par `SPI_prepare`

Synopsis

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

Description

`SPI_getargtypeid` renvoie l'OID représentant l'identifiant du type pour le *`argIndex`*-ième argument d'un plan préparé par `SPI_prepare`. Le premier argument se trouve à l'index zéro.

Arguments

`SPIPlanPtr` *`plan`*
: plan d'exécution (renvoyé par `SPI_prepare`)

`int` *`argIndex`*
: index de l'argument (à partir de zéro)

Code de retour

L'identifiant du type de l'argument à l'index donné. Si le *`plan`* est NULL ou invalide, ou *`argIndex`* inférieur à 0 ou pas moins que le nombre d'arguments déclaré pour le *`plan`*, `SPI_result` est initialisé à SPI_ERROR_ARGUMENT et InvalidOid est renvoyé.

Nom

SPI_is_cursor_plan — renvoie true si le plan préparé par `SPI_prepare` peut être utilisé avec `SPI_cursor_open`

Synopsis

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

Description

`SPI_is_cursor_plan` renvoie true si un plan préparé par `SPI_prepare` peut être passé comme un argument à `SPI_cursor_open` ou false si ce n'est pas le cas. Les critères sont que le *`plan`* représente une seule commande et que cette commande renvoit des lignes à l'appelant ; par l'exemple, **SELECT** est autorisé sauf s'il contient une clause `INTO` et **UPDATE** est autorisé seulement s'il contient un `RETURNING`

Arguments

`SPIPlanPtr` *`plan`*
: plan d'exécution (renvoyé par `SPI_prepare`)

Return Value

true ou false pour indiquer si *`plan`* peut produire un curseur ou non, avec `SPI_result` initialisé à zéro. S'il nest pas possible de déterminer la réponse (par exemple, si le *`plan`* vaut NULL ou est invalide, ou s'il est appelé en étant déconnecté de SPI), alors `SPI_result` est configuré avec un code d'erreur convenable et false est renvoyé.

Nom

SPI_execute_plan — exécute un plan préparé par `SPI_prepare`

Synopsis

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls, bool
read_only, long count)
```

Description

`SPI_execute_plan` exécute un plan préparé par `SPI_prepare`. *`read_only`* et *`count`* ont la même interprétation que dans `SPI_execute`.

Arguments

`SPIPlanPtr` *`plan`*
: plan d'exécution (retourné par `SPI_prepare`)

`Datum *`*`values`*
: Un tableau des vraies valeurs des paramètres. Doit avoir la même longueur que le nombre d'arguments du plan.

`const char *` *`nulls`*
: Un tableau décrivant les paramètres nuls. Doit avoir la même longueur que le nombre d'arguments du plan. `n` indique une valeur NULL (l'entrée correspondante dans *`values`* sera ignorée) ; un espace indique une valeur non NULL (l'entrée correspondante dans *`values`* est valide).

  Si *`nulls`* est NULL, alors `SPI_executeplan` part du principe qu'aucun paramètre n'est nul.

`bool` *`read_only`*
: `true` pour une exécution en lecture seule

`long` *`count`*
: nombre maximum de lignes à renvoyer ou `0` pour aucune ligne à renvoyer

Valeur de retour

La valeur de retour est la même que pour `SPI_execute` avec les résultats d'erreurs (négatif) possibles :

SPI_ERROR_ARGUMENT
: si *`plan`* est NULL ou invalide ou *`count`* est inférieur à 0

SPI_ERROR_PARAM
: si *`values`* est NULL et *`plan`* est préparé avec des paramètres

`SPI_processed` et `SPI_tuptable` sont positionnés comme dans `SPI_execute` en cas de réussite.

Nom

SPI_execute_plan_with_paramlist — exécute un plan préparé par `SPI_prepare`

Synopsis

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

Description

`SPI_execute_plan_with_paramlist` exécute un plan préparé par `SPI_prepare`. Cette fonction est l'équivalent de `SPI_execute_plan`, sauf que les informations sur les valeurs des paramètres à passer à la requête sont présentées différemment. La représentation `ParamListInfo` peut être utilse pour passer des valeurs qui sont déjà disponibles dans ce format. Elle supporte aussi l'utilisation d'ensemble de paramètres dynamiques indiqués via des fonctions dans `ParamListInfo`.

Arguments

`SPIPlanPtr` *`plan`*
: plan d'exécution (renvoyé par `SPI_prepare`)

`ParamListInfo` *`params`*
: structure de données contenant les types et valeurs de paramètres ; NULL si aucune structure

`bool` *`read_only`*
: `true` pour une exécution en lecture seule

`long` *`count`*
: nombre maximum de lignes à renvoyer ou `0` pour aucune ligne à renvoyer

Code de retour

La valeur de retour est identique à celle de `SPI_execute_plan`.

`SPI_processed` et `SPI_tuptable` sont initialisés de la même façon que pour `SPI_execute_plan` en cas de réussite.

Nom

SPI_execp — exécute un plan en mode lecture/écriture

Synopsis

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

Description

`SPI_execp` est identique à `SPI_execute_plan` mais le paramètre *`read_only`* de ce dernier vaut toujours `false`.

Arguments

`SPIPlanPtr` *`plan`*
: plan d'exécution (renvoyé par `SPI_prepare`)

`Datum *` *`values`*
: Un tableau des vraies valeurs de paramètre. Doit avoir la même longueur que le nombre d'arguments du plan.

`const char *` *`nulls`*
: Un tableau décrivant les paramètres NULL. Doit avoir la même longueur que le nombre d'arguments du plan. `n` indique une valeur NULL (l'entrée dans *`values`* sera ignorée) ; un espace indique une valeur non NULL (l'entrée dans *`values`* est valide).

  Si *`nulls`* est NULL, alors `SPI_execp` suppose qu'aucun paramètre n'est NULL.

`long` *`count`*
: nombre maximum de lignes à renvoyer ou `0` pour aucune ligne à renvoyer

Valeur de retour

Voir `SPI_execute_plan`.

`SPI_processed` et `SPI_tuptable` sont initialisées comme dans `SPI_execute` en cas de succès.

Nom

SPI_cursor_open — met en place un curseur en utilisant un plan créé avec `SPI_prepare`

Synopsis

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
Datum * values, const char * nulls,
bool read_only)
```

Description

`SPI_cursor_open` met en place un curseur (en interne, un portail) qui lancera un plan préparé par `SPI_prepare`. Les paramètres ont la même signification que les paramètres correspondant à `SPI_execute_plan`.

Utiliser un curseur au lieu de lancer le plan directement a deux avantages. Premièrement, les lignes de résultats peuvent être récupérées un certain nombre à la fois, évitant la saturation de mémoire pour les requêtes qui retournent trop de lignes. Deuxièmement, un portail peut survivre à la procédure courante (elle peut, en fait, vivre jusqu'à la fin de la transaction courante). Renvoyer le nom du portail à l'appelant de la procédure donne un moyen de retourner une série de ligne en tant que résultat.

Les données passées seront copiées dans le portail du curseur, donc il peut être libéré alors que le curseur existe toujours.

Arguments

`const char * name`
: nom pour le portail ou NULL pour laisser le système choisir un nom

`SPIPlanPtr plan`
: plan d'exécution (retourné par `SPI_prepare`)

`Datum * values`
: Un tableau des valeurs de paramètres actuelles. Doit avoir la même longueur que le nombre d'arguments du plan.

`const char *nulls`
: Un tableau décrivant quels paramètres sont NULL. Doit avoir la même longueur que le nombre d'arguments du plan. `n` indique une valeur NULL (l'entrée correspondante dans `values` sera ignorée) ; un espace indique une valeur non NULL (l'entrée correspondante dans `values` est valide).

  Si `nulls` est NULL, alors `SPI_cursor_open` part du principe qu'aucun paramètre n'est nul.

`bool read_only`
: `true` pour les exécutions en lecture seule

Valeur de retour

Pointeur vers le portail contenant le curseur. Notez qu'il n'y a pas de convention pour le renvoi d'une erreur ; toute erreur sera rapportée via `elog`.

Nom

SPI_cursor_open_with_args — ouvre un curseur en utilisant une requête et des paramètres

Synopsis

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

Description

`SPI_cursor_open_with_args` initialise un curseur (en interne, un portail) qui exécutera la requête spécifié. La plupart des paramètres ont la même signification que les paramètres correspondant de `SPI_prepare_cursor` et `SPI_cursor_open`.

Pour une exécution seule, cette fonction sera préférée à `SPI_prepare_cursor` suivie de `SPI_cursor_open`. Si la même commande doit être exécutée avec plusieurs paramètres différents, il n'y a pas de différences sur les deux méthode, la replanification a un coût mais bénéficie de plans personnalisés.

Les données passées seront copiées dans le portail du curseur, donc elles seront libérées alors que le curseur existe toujours.

Arguments

`const char * name`
: nom du portail, ou NULL pour que le système sélectionne un nom de lui-même

`const char * command`
: chaîne de commande

`int nargs`
: nombre de paramètres en entrée (`$1`, `$2`, etc.)

`Oid * argtypes`
: un tableau contenant les OID des types de données des paramètres

`Datum * values`
: un tableau des valeurs actuelles des paramètres

`const char * nulls`
: un tableau décrivant les paramètres NULL

  Si *`nulls`* vaut NULL, alors `SPI_cursor_open_with_args` suppose qu'aucun paramètre n'est NULL.

`bool read_only`
: `true` pour une exécution en lecture seule

`int cursorOptions`
: masque de bits des options du curseur : zéro cause le comportement par défaut

Valeur de retour

Pointeur du portail contenant le curseur. Notez qu'il n'y a pas de convention pour le renvoi des erreurs ; toute erreur sera rapportée par `elog`.

Nom

SPI_cursor_open_with_paramlist — ouvre un curseur en utilisant les paramètres

Synopsis

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

Description

`SPI_cursor_open_with_paramlist` prépare un curseur (en interne un portail), qui exécutera un plan préparé par `SPI_prepare`. Cette fonction est équivalente à `SPI_cursor_open` sauf que les informations sur les valeurs des paramètres passées à la requête sont présentées différemment. La représentation de `ParamListInfo` peut être utile pour fournir des valeurs déjà disponibles dans ce format. Elle supporte aussi l'utilisation d'ensemble de paramètres dynamiques via des fonctions spécifiées dans `ParamListInfo`.

Les données passées en paramètre seront copiées dans le portail du curseur et peuvent donc être libérées alors que le curseur existe toujours.

Arguments

`const char * name`
: nom d'un portail ou NULL pour que le système en choisisse un lui-même

`SPIPlanPtr plan`
: plan d'exécution (renvoyé par `SPI_prepare`)

`ParamListInfo params`
: structure de données contenant les types et valeurs de paramètres ; NULL sinon

`bool read_only`
: `true` pour une exécution en lecture seule

Valeur de retour

Pointeur vers le portail contenant le curseur. Notez qu'il n'existe pas de convention pour le retour d'erreur ; toute erreur sera renvoyée via `elog`.

Nom

SPI_cursor_find — recherche un curseur existant par nom

Synopsis

```
Portal SPI_cursor_find(const char * name)
```

Description

`SPI_cursor_find` recherche un portail par nom. Ceci est principalement utile pour résoudre un nom de curseur renvoyé en tant que texte par une autre fonction.

Arguments

`const char * name`
: nom du portail

Valeur de retour

Pointeur vers le portail portant le nom spécifié ou NULL si aucun n'a été trouvé

Nom

SPI_cursor_fetch — extrait des lignes à partir d'un curseur

Synopsis

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_fetch` extrait des lignes à partir d'un curseur. Ceci est équivalent à un sous-ensemble de la commande SQL **FETCH** (voir `SPI_scroll_cursor_fetch` pour plus de détails).

Arguments

`Portal` *`portal`*
: portail contenant le curseur

`bool` *`forward`*
: vrai pour une extraction en avant, faux pour une extraction en arrière

`long` *`count`*
: nombre maximum de lignes à récupérer

Valeur de retour

`SPI_processed` et `SPI_tuptable` sont positionnés comme dans `SPI_execute` en cas de réussite.

Notes

Récupérer en sens inverse pourrait échouer si le plan du curseur n'était pas créé avec l'option CURSOR_OPT_SCROLL.

Nom

SPI_cursor_move — déplace un curseur

Synopsis

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_move` saute un certain nombre de lignes dans un curseur. Ceci est équivalent à un sous-ensemble de la commande SQL **MOVE** (voir `SPI_scroll_cursor_move` pour plus de détails).

Arguments

`Portal` *`portal`*
: portail contenant le curseur

`bool` *`forward`*
: vrai pour un saut en avant, faux pour un saut en arrière

`long` *`count`*
: nombre maximum de lignes à déplacer

Notes

Se déplacer en sens inverse pourrait échouer si le plan du curseur n'a pas été créé avec l'option CURSOR_OPT_SCROLL option.

Nom

SPI_scroll_cursor_fetch — récupère quelques lignes à partir d'un curseur

Synopsis

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction, long count)
```

Description

`SPI_scroll_cursor_fetch` récupère quelques lignes à partir d'un curseur. C'est équivalent à la commande SQL **FETCH**.

Arguments

`Portal` *`portal`*
: portail contenant le curseur

`FetchDirection` *`direction`*
: un parmi FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE ou FETCH_RELATIVE

`long` *`count`*
: nombre de lignes à récupérer pour FETCH_FORWARD ou FETCH_BACKWARD ; nombre de lignes absolu à récupérer pour FETCH_ABSOLUTE ; ou nombre de lignes relatif à récupérer pour FETCH_RELATIVE

Valeur de retour

`SPI_processed` et `SPI_tuptable` sont configurés comme `SPI_execute` en cas de succès.

Notes

Voir la commande SQL FETCH(7) pour des détails sur l'interprétation des paramètres *`direction`* et *`count`*.

Les valeurs de direction autres que FETCH_FORWARD peuvent échouer si le plan du curseur n'a pas été créé avec l'option CURSOR_OPT_SCROLL.

Nom

SPI_scroll_cursor_move — déplacer un curseur

Synopsis

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction, long count)
```

Description

`SPI_scroll_cursor_move` ignore un certain nombre de lignes dans un curseur. C'est l'équivalent de la commande SQL **MOVE**.

Arguments

`Portal` *`portal`*
: portail contenant le curseur

`FetchDirection` *`direction`*
: un parmi FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE et FETCH_RELATIVE

`long` *`count`*
: nombre de lignes à déplacer pour FETCH_FORWARD ou FETCH_BACKWARD ; nombre de lignes absolu à déplacer pour FETCH_ABSOLUTE ; ou nombre de lignes relatif à déplacer pour FETCH_RELATIVE

Valeur de retour

`SPI_processed` est configuré comme `SPI_execute` en cas de succès. `SPI_tuptable` est configuré à NULL car aucune ligne n'est renvoyée par cette fonction.

Notes

Voir la commande SQL FETCH(7) pour des détails sur l'interprétation des paramètres *`direction`* et *`count`*.

Les valeurs de direction autres que FETCH_FORWARD peuvent échouer si le plan du curseur n'a pas été créé avec l'option CURSOR_OPT_SCROLL.

Nom

SPI_cursor_close — ferme un curseur

Synopsis

```
void SPI_cursor_close(Portal portal)
```

Description

`SPI_cursor_close` ferme un curseur créé précédemment et libère la mémoire du portail.

Tous les curseurs ouverts sont fermés automatiquement à la fin de la transaction. `SPI_cursor_close` n'a besoin d'être invoqué que s'il est désirable de libérer les ressources plus tôt.

Arguments

`Portal` *`portal`*
: portail contenant le curseur

Nom

SPI_saveplan — sauvegarde un plan

Synopsis

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

Description

`SPI_saveplan` sauvegarde un plan validé (préparé par `SPI_prepare`) dans une zone de mémoire qui ne sera pas libérée par `SPI_finish` et par le gestionnaire de transactions et retourne le pointeur vers le plan sauvegardé. Ceci vous donne la possibilité de réutiliser les plans préparés lors des invocations suivantes de votre procédure dans la session courante.

Arguments

`SPIPlanPtr` *`plan`*
: le plan à sauvegarder

Valeur de retour

Pointeur vers le plan sauvegardé ; NULL en cas d'échec. En cas d'erreur, `SPI_result` est positionnée comme suit :

SPI_ERROR_ARGUMENT
: si *`plan`* est NULL ou invalide

SPI_ERROR_UNCONNECTED
: si appelé d'une procédure non connectée

Notes

Le plan passé n'est pas libéré, donc vous pouvez souhaiter exécuter `SPI_freeplan` sur ce dernier pour éviter des pertes mémoire jusqu'à `SPI_finish`.

Si l'un des objets (une table, une fonction, etc.) référencés par le plan préparé est supprimé ou redéfini, alors les exécutions futures de `SPI_execute_plan` pourrait échouer ou renvoyer des résultats différents de ce que le plan indique initialement.

43.2. Fonctions de support d'interface

Les fonctions décrites ici donnent une interface pour extraire les informations des séries de résultats renvoyés par `SPI_execute` et les autres fonctions SPI.

Toutes les fonctions décrites dans cette section peuvent être utilisées par toutes les procédures, connectées et non connectées.

Nom

SPI_fname — détermine le nom de colonne pour le numéro de colonne spécifié

Synopsis

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_fname` retourne une copie du nom de colonne d'une colonne spécifiée (vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin).

Arguments

`TupleDesc` *`rowdesc`*
: description de rangée d'entrée

`int` *`colnumber`*
: nombre de colonne (le compte commence à 1)

Valeur de retour

Le nom de colonne ; NULL si *`colnumber`* est hors de portée. `SPI_result` est positionnée à SPI_ERROR_NOATTRIBUTE en cas d'échec.

Nom

SPI_fnumber — détermine le numéro de colonne pour le nom de colonne spécifiée

Synopsis

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Description

`SPI_fnumber` renvoie le numéro de colonne pour la colonne portant le nom spécifié.

Si *`colname`* réfère à une colonne système (c'est-à-dire `oid`), alors le numéro de colonne négatif approprié sera renvoyé. L'appelant devra faire attention à tester la valeur de retour pour égalité exacte à SPI_ERROR_NOATTRIBUTE pour détecter une erreur ; tester le résultat pour une valeur inférieure ou égale à 0 n'est pas correcte sauf si les colonnes systèmes doivent être rejetées.

Arguments

`TupleDesc` *`rowdesc`*
: description de la rangée d'entrée

`const char *` *`colname`*
: nom de colonne

Valeur de retour

Numéro de colonne (le compte commence à 1) ou SPI_ERROR_NOATTRIBUTE si la colonne nommée n'est trouvée.

Nom

SPI_getvalue — renvoie la valeur de chaîne de la colonne spécifiée

Synopsis

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

Description

`SPI_getvalue` retourne la représentation chaîne de la valeur de la colonne spécifiée.

Le résultat est retourné en mémoire allouée en utilisant `palloc` (vous pouvez utiliser `pfree` pour libérer la mémoire lorsque vous n'en avez plus besoin).

Arguments

`HeapTuple` *`row`*
: ligne d'entrée à examiner

`TupleDesc` *`rowdesc`*
: description de la ligne en entrée

`int` *`colnumber`*
: numéro de colonne (le compte commence à 1)

Valeur de retour

Valeur de colonne ou NULL si la colonne est NULL, si *`colnumber`* est hors de portée (`SPI_result` est positionnée à SPI_ERROR_NOATTRIBUTE) ou si aucune fonction de sortie n'est disponible (`SPI_result` est positionnée à SPI_ERROR_NOOUTFUNC).

Nom

SPI_getbinval — retourne la valeur binaire de la colonne spécifiée

Synopsis

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber, bool * isNULL)
```

Description

`SPI_getbinval` retourne la valeur de la colonne spécifiée dans le format interne (en tant que type Datum).

Cette fonction n'alloue pas de nouvel espace pour le datum. Dans le cas d'un type de données passé par référence, la valeur de retour sera un pointeur dans la ligne passée.

Arguments

`HeapTuple` *`row`*
: ligne d'entrée à examiner

`TupleDesc` *`rowdesc`*
: description de la ligne d'entrée

`int` *`colnumber`*
: numéro de colonne (le compte commence à 1)

`bool *` *`isNULL`*
: indique une valeur NULL dans la colonne

Valeur de retour

La valeur binaire de la colonne est retournée. La variable vers laquelle pointe *`isNULL`* est positionnée à vrai si la colonne est NULL et sinon à faux.

`SPI_result` est positionnée à SPI_ERROR_NOATTRIBUTE en cas d'erreur.

Nom

SPI_gettype — retourne le nom du type de donnée de la colonne spécifiée

Synopsis

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_gettype` retourne une copie du nom du type de donnée de la colonne spécifiée (vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin).

Arguments

`TupleDesc` *`rowdesc`*
: description de ligne d'entrée

`int` *`colnumber`*
: numéro de colonne (le compte commence à 1)

Valeur de retour

Le nom de type de donnée de la colonne spécifiée ou NULL en cas d'erreur. `SPI_result` est positionnée à SPI_ERROR_NOATTRIBUTE en cas d'erreur.

Nom

SPI_gettypeid — retourne l'OID de type de donnée de la colonne spécifiée

Synopsis

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_gettypeid` retourne l'OID du type de donnée de la colonne spécifiée.

Arguments

`TupleDesc` *`rowdesc`*
: description de ligne d'entrée

`int` *`colnumber`*
: numéro de colonne (le compte commence à 1)

Valeur de retour

L'OID du type de donnée de la colonne spécifiée ou InvalidOid en cas d'erreur. En cas d'erreur, `SPI_result` est positionnée à SPI_ERROR_NOATTRIBUTE.

Nom

SPI_getrelname — retourne le nom de la relation spécifiée

Synopsis

```
char * SPI_getrelname(Relation rel)
```

Description

`SPI_getrelname` retourne une copie du nom de la relation spécifiée (vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin).

Arguments

`Relation` *`rel`*
: relation d'entrée

Valeur de retour

Le nom de la relation spécifiée.

Nom

SPI_getnspname — renvoie l'espace de noms de la relation spécifiée

Synopsis

```
char * SPI_getnspname(Relation rel)
```

Description

`SPI_getnspname` renvoie une copie du nom de l'espace de nom auquel appartient la Relation spécifiée. Ceci est équivalent au schéma de la relation. Vous devriez libérer (`pfree`) la valeur de retour de cette fonction lorsque vous en avez fini avec elle.

Arguments

`Relation rel`
: relation en entrée

Valeur de retour

Le nom de l'espace de noms de la relation spécifiée.

43.3. Gestion de la mémoire

PostgreSQL™ alloue de la mémoire dans des *contextes mémoire* qui donnent une méthode pratique pour gérer les allocations faîtes dans plusieurs endroits qui ont besoin de vivre pour des durées différentes. Détruire un contexte libère toute la mémoire qui y était allouée. Donc, il n'est pas nécessaire de garder la trace des objets individuels pour éviter les fuites de mémoire ; à la place, seul un petit nombre de contextes doivent être gérés. `palloc` et les fonctions liées allouent de la mémoire du contexte « courant ».

`SPI_connect` crée un nouveau contexte mémoire et le rend courant. `SPI_finish` restaure le contexte mémoire précédant et détruit le contexte créé par `SPI_connect`. Ces actions garantissent que les allocations temporaires de mémoire faîtes dans votre procédure soient réclamées lors de la sortie de la procédure, évitant les fuites de mémoire.

En revanche, si votre procédure a besoin de renvoyer un objet dans de la mémoire allouée (tel que la valeur d'un type de donné passé par référence), vous ne pouvez pas allouer cette mémoire en utilisant `palloc`, au moins pas tant que vous êtes connecté à SPI. Si vous essayez, l'objet sera désalloué par `SPI_finish` et votre procédure ne fonctionnera pas de manière fiable. Pour résoudre ce problème, utilisez `SPI_palloc` pour allouer de la mémoire pour votre objet de retour. `SPI_palloc` alloue de la mémoire dans le « contexte de mémoire courant », c'est-à-dire le contexte de mémoire qui était courant lorsque `SPI_connect` a été appelée, ce qui est précisément le bon contexte pour une valeur renvoyée à partir de votre procédure.

Si `SPI_palloc` est appelé pendant que la procédure n'est pas connectée à SPI, alors il agit de la même manière qu'un `palloc` normal. Avant qu'une procédure ne se connecte au gestionnaire SPI, toutes les allocations faîtes par la procédure via `palloc` ou par une fonction utilitaire SPI sont faîtes dans le contexte de mémoire courant.

Quand `SPI_connect` est appelée, le contexte privé de la procédure, qui est créée par `SPI_connect`, est nommé le contexte courant. Toute allocation faîte par `palloc`, `repalloc` ou une fonction utilitaire SPI (à part pour `SPI_copytuple`, `SPI_returntuple`, `SPI_modifytuple`, et `SPI_palloc`) sont faîtes dans ce contexte. Quand une procédure se déconnecte du gestionnaire SPI (via `SPI_finish`), le contexte courant est restauré au contexte de mémoire courant et toutes les allocations faîtes dans le contexte de mémoire de la procédure sont libérées et ne peuvent plus être utilisées.

Toutes les fonctions couvertes dans cette section peuvent être utilisées par des procédures connectées comme non connectées. Dans une procédure non connectée, elles agissent de la même façon que les fonctions serveur sous-jacentes (`palloc`, etc.).

Nom

SPI_palloc — alloue de la mémoire dans le contexte de mémoire courant

Synopsis

```
void * SPI_palloc(Size size)
```

Description

`SPI_palloc` alloue de la mémoire dans le contexte de mémoire courant.

Arguments

`Size` *`size`*
: taille en octets du stockage à allouer

Valeur de retour

Pointeur vers le nouvel espace de stockage de la taille spécifiée

Nom

SPI_repalloc — ré-alloue de la mémoire dans le contexte de mémoire courant

Synopsis

```
void * SPI_repalloc(void * pointer, Size size)
```

Description

`SPI_repalloc` change la taille d'un segment de mémoire alloué auparavant en utilisant `SPI_palloc`.

Cette fonction n'est plus différente du `repalloc` standard. Elle n'est gardée que pour la compatibilité du code existant.

Arguments

`void * pointer`
: pointeur vers l'espace de stockage à modifier

`Size size`
: taille en octets du stockage à allouer

Valeur de retour

Pointeur vers le nouvel espace de stockage de taille spécifiée avec le contenu copié de l'espace existant

Nom

SPI_pfree — libère de la mémoire dans le contexte de mémoire courant

Synopsis

```
void SPI_pfree(void * pointer)
```

Description

`SPI_pfree` libère de la mémoire allouée auparavant par `SPI_palloc` ou `SPI_repalloc`.

Cette fonction n'est plus différente du `pfree` standard. Elle n'est conservée que pour la compatibilité du code existant.

Arguments

`void * pointer`
: pointeur vers l'espace de stockage à libérer

Nom

SPI_copytuple — effectue une copie d'une ligne dans le contexte de mémoire courant

Synopsis

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Description

`SPI_copytuple` crée une copie d'une ligne dans le contexte de mémoire courant. Ceci est normalement utilisé pour renvoyer une ligne modifiée à partir d'un déclencheur. Dans une fonction déclarée pour renvoyer un type composite, utilisez `SPI_returntuple` à la place.

Arguments

`HeapTuple` *`row`*
: ligne à copier

Valeur de retour

la ligne copiée ; NULL seulement si *`row`* est NULL

Nom

SPI_returntuple — prépare le renvoi d'une ligne en tant que Datum

Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Description

`SPI_returntuple` crée une copie d'une ligne dans le contexte de l'exécuteur supérieur, la renvoyant sous la forme d'une ligne de type Datum. Le pointeur renvoyé a seulement besoin d'être converti en Datum via `PointerGetDatum` avant d'être renvoyé.

Notez que ceci devrait être utilisé pour les fonctions qui déclarent renvoyer des types composites. Ce n'est pas utilisé pour les déclencheurs ; utilisez pour renvoyer une ligne modifiée dans un déclencheur.

Arguments

`HeapTuple` *`row`*
: ligne à copier

`TupleDesc` *`rowdesc`*
: descripteur pour la ligne (passez le même descripteur chaque fois pour un cache plus efficace)

Valeur de retour

HeapTupleHeader pointant vers la ligne copiée ; NULL seulement si *`row`* ou *`rowdesc`* est NULL

Nom

SPI_modifytuple — crée une ligne en remplaçant les champs sélectionnés d'une ligne donnée

Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, ncols, colnumber, Datum *
values, const char * nulls)
```

Description

`SPI_modifytuple` crée une nouvelle ligne en retirant les nouvelles valeurs pour les colonnes sélectionnées et en copiant les colonnes de la ligne d'origine à d'autres positions. La ligne d'entrée n'est pas modifiée.

Arguments

`Relation` *`rel`*
: Utilisé seulement en tant que source du descripteur de ligne pour la ligne (passez une relation plutôt qu'un descripteur de ligne est une erreur).

`HeapTuple` *`row`*
: rangée à modifier

`int` *`ncols`*
: nombre de numéros de colonnes dans le tableau *`colnumber`*

`int *` *`colnumber`*
: tableau des numéros de colonnes à modifier (le numéro des colonnes commence à 1)

`Datum *` *`values`*
: nouvelles valeurs pour les colonnes spécifiées

`const char *` *`nulls`*
: quelles nouvelles valeurs sont NULL, si elles existent (voir `SPI_executeplan` pour le format)

Valeur de retour

nouvelle ligne avec modifications, allouée dans le contexte de mémoire courant ; NULL seulement si *`row`* est NULL

En cas d'erreur, `SPI_result` est positionnée comme suit :

SPI_ERROR_ARGUMENT
: si *`rel`* est NULL ou si *`row`* est NULL ou si *`ncols`* est inférieur ou égal à 0 ou si *`nocolonne`* est NULL ou si *`values`* est NULL.

SPI_ERROR_NOATTRIBUTE
: si *`nocolonne`* contient un numéro de colonne invalide (inférieur ou égal à 0 ou supérieur au numéro de colonne dans *`row`*)

Nom

SPI_freetuple — libère une ligne allouée dans le contexte de mémoire courant

Synopsis

```
void SPI_freetuple(HeapTuple row)
```

Description

`SPI_freetuple` libère une rangée allouée auparavant dans le contexte de mémoire courant.

Cette fonction n'est plus différente du standard `heap_freetuple`. Elle est gardée juste pour la compatibilité du code existant.

Arguments

`HeapTuple` *`row`*
: rangée à libérer

Nom

SPI_freetuptable — libère une série de lignes créée par `SPI_execute` ou une fonction semblable

Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Description

`SPI_freetuptable` libère une série de lignes créée auparavant par une fonction d'exécution de commandes SPI, tel que `SPI_execute`. Par conséquent, cette fonction est souvent appelée avec la variable globale `SPI_tupletable` comme argument.

Cette fonction est utile si une procédure SPI a besoin d'exécuter de multiples commandes et ne veut pas garder les résultats de commandes précédentes en mémoire jusqu'à sa fin. Notez que toute série de lignes non libérées est libérée quand même lors de `SPI_finish`.

Arguments

`SPITupleTable *` *`tuptable`*
: pointeur vers la série de lignes à libérer

Nom

SPI_freeplan — libère un plan sauvegardé auparavant

Synopsis

```
int SPI_freeplan(SPIPlanPtr plan)
```

Description

`SPI_freeplan` libère un plan de commandes d'exécution retourné auparavant par `SPI_prepare` ou sauvegardé par `SPI_saveplan`.

Arguments

`SPIPlanPtr` *`plan`*
: pointeur vers le plan à libérer

Valeur de retour

SPI_ERROR_ARGUMENT si *`plan`* est NULL ou invalide.

43.4. Visibilité des modifications de données

Les règles suivantes gouvernent la visibilité des modifications de données dans les fonctions qui utilisent SPI (ou tout autre fonction C) :

- Pendant l'exécution de la commande SQL, toute modification de données faite par la commande est invisible à la commande. Par exemple, dans la commande :

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  les lignes insérées sont invisibles à la partie **SELECT**.
- Les modifications effectuées par une commande C sont visibles par toutes les commandes qui sont lancées après C, peu importe qu'elles soient lancées à l'intérieur de C (pendant l'exécution de C) ou après que C soit terminée.
- Les commandes exécutées via SPI à l'intérieur d'une fonction appelée par une commande SQL (soit une fonction ordinaire soit un déclencheur) suivent une des règles ci-dessus suivant le commutateur lecture/écriture passé à SPI. Les commandes exécutées en mode lecture seule suivent la première règle : elles ne peuvent pas voir les modifications de la commande appelante. Les commandes exécutées en mode lecture/écriture suivent la deuxième règle : elles peuvent voir toutes les modifications réalisées jusqu'à maintenant.
- Tous les langages standards de procédures initialisent le mode lecture/écriture suivant l'attribut de volatilité de la fonction. Les commandes des fonctions `STABLE` et `IMMUTABLE` sont réalisées en mode lecture seule alors que les fonctions `VOLATILE` sont réalisées en mode lecture/écriture. Alors que les auteurs de fonctions C sont capables de violer cette convention, il est peu probable que cela soit une bonne idée de le faire.

La section suivante contient un exemple qui illustre l'application de ces règles.

43.5. Exemples

Cette section contient un exemple très simple d'utilisation de SPI. La procédure `execq` prend une commande SQL comme premier argument et un compteur de lignes comme second, exécute la commande en utilisant `SPI_exec` et renvoie le nombre de lignes qui ont été traitées par la commande. Vous trouverez des exemples plus complexes pour SPI dans l'arborescence source dans `src/test/regress/regress.c` et dans `contrib/spi`.

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
```

```
#endif

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /* Convertir l'objet texte donné en chaîne C */
    command = text_to_cstring(sql);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * Si des lignes ont été récupérées,
     * alors les afficher via elog(INFO).
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    return (proc);
}
```

(Cette fonction utilisera la convention d'appel version 0 pour rendre l'exemple plus simple à comprendre. Dans des applications réelles, vous devriez utiliser la nouvelle interface version 1.)

Voici comment déclarer la fonction après l'avoir compilée en une bibliothèque partagée (les détails sont dans Section 35.9.6, « Compiler et lier des fonctions chargées dynamiquement ») :

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Voici une session d'exemple :

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inséré par execq
```

```
INFO:  EXECQ:  1    -- retourné par execq et inséré par l'INSERT précédant

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, une seule ligne insérée - comme spécifié

 execq
-------
     3              -- 10 est la valeur max seulement, 3 est le nombre réel de rangées
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- aucune rangée dans a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- il y a une rangée dans a + 1
(2 rows)

-- Ceci montre la règle de visibilité de modifications de données :

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 rangées * 1 (x dans la première rangée)
 6                  -- 3 rangées (2 + 1 juste insérée) * 2 (x dans la deuxième rangée)
(4 rows)                 ^^^^^^^
                         rangées visible à execq() dans des invocations différentes
```

Partie VI. Référence

Les points abordés dans ce référentiel ont pour objectif de fournir, de manière concise, un résumé précis, complet, formel et faisant autorité sur leurs sujets respectifs. Des informations complémentaires sur l'utilisation de PostgreSQL™ sont présentées, dans d'autres parties de cet ouvrage, sous la forme de descriptions, de tutoriels ou d'exemples. On pourra se reporter à la liste de références croisées disponible sur chaque page de référence.

Les entrées du référentiel sont également disponibles sous la forme de pages « man » traditionnelles.

Commandes SQL

Cette partie regroupe les informations de référence concernant les commandes SQL reconnues par PostgreSQL™. Généralement, on désigne par « SQL » le langage ; toute information sur la structure et la compatibilité standard de chaque commande peut être trouvée sur les pages référencées.

Nom

ABORT — Interrompre la transaction en cours

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Description

ABORT annule la transaction en cours et toutes les mises à jour effectuées pendant cette transaction. Cette commande a un comportement identique à la commande SQL ROLLBACK(7). Elle n'est présente que pour des raisons historiques.

Paramètres

`WORK`, `TRANSACTION`
: Mots-clé optionnels. Ils n'ont aucun effet.

Notes

COMMIT(7) est utilisé pour terminer avec succès une transaction.

Lancer **ABORT** à l'extérieur de toute transaction ne cause aucun dégât, mais provoque un message d'avertissement.

Exemples

Annuler toutes les modifications :

```
ABORT;
```

Compatibilité

Cette commande est une extension PostgreSQL™ présente pour des raisons historiques. **ROLLBACK** est la commande équivalente du standard SQL.

Voir aussi

BEGIN(7), COMMIT(7), ROLLBACK(7)

Nom

ALTER AGGREGATE — Modifier la définition d'une fonction d'agrégat

Synopsis

```
ALTER AGGREGATE nom ( type [ , ... ] ) RENAME TO nouveau_nom
ALTER AGGREGATE nom ( type [ , ... ] ) OWNER TO nouveau_proprietaire
ALTER AGGREGATE nom ( type [ , ... ] ) SET SCHEMA nouveau_schema
```

Description

ALTER AGGREGATE change la définition d'une fonction d'agrégat.

Seul le propriétaire de la fonction d'agrégat peut utiliser **ALTER AGGREGATE**. Pour modifier le schéma d'une fonction d'agrégat, il est nécessaire de posséder le droit `CREATE` sur le nouveau schéma. Pour modifier le propriétaire de la fonction, il faut être un membre direct ou indirect du nouveau rôle propriétaire, rôle qui doit en outre posséder le droit `CREATE` sur le schéma de la fonction d'agrégat. Ces restrictions assurent que la modification du propriétaire ne permet pas d'aller au-delà de ce que permet la suppression et la recréation d'une fonction d'agrégat. Toutefois, un superutilisateur peut modifier la possession de n'importe quelle fonction d'agrégat.

Paramètres

nom
: Le nom (éventuellement qualifié du nom du schéma) de la fonction d'agrégat.

type
: Un type de données en entrée sur lequel la fonction d'agrégat opère. Pour référencer une fonction d'agrégat sans argument, écrivez `*` à la place de la liste des types de données en entrée.

nouveau_nom
: Le nouveau nom de la fonction d'agrégat.

nouveau_propriétaire
: Le nouveau propriétaire de la fonction d'agrégat.

nouveau_schema
: Le nouveau schéma de la fonction d'agrégat.

Exemples

Renommer la fonction d'agrégat `mamoyenne` de type integer en `ma_moyenne` :

```
ALTER AGGREGATE mamoyenne(integer) RENAME TO ma_moyenne;
```

Changer le propriétaire de la fonction d'agrégat `mamoyenne` de type integer en `joe` :

```
ALTER AGGREGATE mamoyenne(integer) OWNER TO joe;
```

Déplacer la fonction d'agrégat `mamoyenne` du type integer dans le schéma `monschema` :

```
ALTER AGGREGATE mamoyenne(integer) SET SCHEMA monschema;
```

Compatibilité

Il n'y a pas de commande **ALTER AGGREGATE** dans le standard SQL.

Voir aussi

CREATE AGGREGATE(7), DROP AGGREGATE(7)

Nom

ALTER CONVERSION — Modifier la définition d'une conversion

Synopsis

```
ALTER CONVERSION nom RENAME TO nouveau_nom
ALTER CONVERSION nom OWNER TO nouveau_propriétaire
```

Description

ALTER CONVERSION modifie la définition d'une conversion.

Seul le propriétaire de la conversion peut utiliser **ALTER CONVERSION**. Pour changer le propriétaire, il faut aussi être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit `CREATE` sur le schéma de la conversion. Ces restrictions assurent que le changement de propriétaire ne va pas au-delà de ce qui peut être obtenu en supprimant et en re-créant la conversion. Toutefois, un superutilisateur peut changer le propriétaire de n'importe quelle conversion.

Paramètres

nom
: Le nom de la conversion.

nouveau_nom
: Le nouveau nom de la conversion.

nouveau_propriétaire
: Le nouveau propriétaire de la conversion.

Exemples

Renommer la conversion `iso_8859_1_to_utf8` en `latin1_to_unicode` :

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

Changer le propriétaire de la conversion `iso_8859_1_to_utf8` en `joe` :

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibilité

Il n'y a pas d'instruction **ALTER CONVERSION** dans le standard SQL.

Voir aussi

CREATE CONVERSION(7), DROP CONVERSION(7)

Nom

ALTER DATABASE — Modifier une base de données

Synopsis

```
ALTER DATABASE nom [ [ WITH ] option [ ... ] ]

où option peut être :

    CONNECTION LIMIT limite_connexion

ALTER DATABASE nom RENAME TO nouveau_nom
ALTER DATABASE nom OWNER TO nouveau_propriétaire

ALTER DATABASE nom SET TABLESPACE nouveau_tablespace

ALTER DATABASE nom SET paramètre { TO | = } { valeur | DEFAULT }
ALTER DATABASE nom SET paramètre FROM CURRENT
ALTER DATABASE nom RESET paramètre
ALTER DATABASE nom RESET ALL
```

Description

ALTER DATABASE modifie les attributs d'une base de données.

La première forme modifie certains paramètres d'une base de données (voir ci-dessous pour les détails). Seul le propriétaire de la base de données ou un superutilisateur peut modifier ces paramètres.

La deuxième forme permet de renommer la base. Seul le propriétaire ou un superutilisateur peut renommer une base. Un propriétaire qui n'est pas superutilisateur doit en outre posséder le droit `CREATEDB`. La base en cours d'utilisation ne peut pas être renommée (on se connectera à une base différente pour réaliser cette opération).

La troisième forme change le propriétaire de la base de données. Pour changer le propriétaire, il faut être propriétaire de la base de données et membre direct ou indirect du nouveau rôle propriétaire. Le droit `CREATEDB` est également requis (les superutilisateurs ont automatiquement tous ces droits).

La quatrième forme change le tablespace par défaut de la base de données. Seuls le propriétaire de la base de données et un superutilisateur peuvent le faire ; vous devez aussi avoir le droit CREATE pour le nouveau tablespace. Cette commande déplace physiquement toutes tables et index actuellement dans l'ancien tablespace par défaut de la base de données vers le nouveau tablespace. Notez que les tables et index placés dans d'autres tablespaces ne sont pas affectés.

Les formes restantes modifient la valeur par défaut d'un paramètre de configuration pour une base PostgreSQL™. Par la suite, à chaque fois qu'une nouvelle session est lancée, la valeur spécifique devient la valeur par défaut de la session. Les valeurs par défaut de la base deviennent les valeurs par défaut de la session. En fait, elles surchargent tout paramètre présent dans `postgresql.conf` ou indiqué sur la ligne de commande de **postgres**. Seul le propriétaire de la base de données ou un superutilisateur peut modifier les valeurs par défaut de la session pour une base. Certaines variables ne peuvent pas être configurées de cette façon pour une base de données ou peuvent seulement être configurées par un superutilisateur.

Paramètres

nom
: Le nom de la base dont les attributs sont à modifier.

limite_connexion
: Le nombre de connexions concurrentes sur la base de données. -1 signifie aucune limite.

nouveau_nom
: Le nouveau nom de la base.

nouveau_propriétaire
: Le nouveau propriétaire de la base.

nouveau_tablespace
: Le nouveau tablespace par défaut de la base de données.

paramètre, *valeur*

Configure cette valeur comme valeur par défaut de la base pour le paramètre de configuration précisée. Si *valeur* indique DEFAULT ou, de façon équivalente, si RESET est utilisé, le paramètrage en cours pour cette base est supprimée, donc la valeur système est utilisée pour les nouvelles sessions. Utiliser RESET ALL permet de supprimer tous les paramètres spécifiques de cette base. SET FROM CURRENT sauvegarde la valeur actuelle du paramètre en tant que valeur spécifique de la base.

Voir SET(7) et Chapitre 18, Configuration du serveur pour plus d'informations sur les noms de paramètres et valeurs autorisées.

Notes

Il est possible de lier une valeur de session par défaut à un rôle plutôt qu'à une base. Voir ALTER ROLE(7) à ce propos. En cas de conflit, les configurations spécifiques au rôle l'emportent sur celles spécifiques à la base.

Exemples

Désactiver les parcours d'index par défaut de la base test :

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Compatibilité

La commande **ALTER DATABASE** est une extension PostgreSQL™.

Voir aussi

CREATE DATABASE(7), DROP DATABASE(7), SET(7), CREATE TABLESPACE(7)

Nom

ALTER DEFAULT PRIVILEGES — définit les droits d'accès par défaut

Synopsis

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } cible_rôle [, ...] ]
    [ IN SCHEMA nom_schéma [, ...] ]
    grant_ou_revoke_réduit

where grant_ou_revoke_réduit is one of:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [,...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [,...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

Description

ALTER DEFAULT PRIVILEGES vous permet de configurer les droits qui seront appliqués aux objets qui seront créés dans le futur. (Cela ne modifie pas les droits affectés à des objets déjà existants.) Actuellement, seuls les droits pour les tables (ceci incluant les vues), les séquences et les fonctions peuvent être modifiés.

Vous pouvez modifier les droits par défaut seulement pour les objets qui seront créés par vous ou par des rôles dont vous êtes membres. Les droits peuvent être configurés de manière globale (c'est-à-dire pour tous les objets de la base de données) ou pour les objets des schémas indiqués. Les droits par défaut spécifiques par schéma sont ajoutés aux droits par défaut globaux pour le type d'objet particulier.

Comme indiqué dans GRANT(7), les droits par défaut de tout type d'objet donnent tous les droits au propriétaire de l'objet et peut aussi donner certains droits à `PUBLIC`. Néanmoins, ce comportement peut être changé par une modification des droits par défaut globaux avec **ALTER DEFAULT PRIVILEGES**.

Paramètres

cible_rôle

Le nom d'un rôle existant dont le rôle actuel est membre. Si `FOR ROLE` est omis, le rôle courant est utilisé.

`nom_schéma`
Le nom d'un schéma existant. S'il est indiqué, les droits par défaut sont modifiés pour les objets créés après coup dans ce schéma. Si `IN SCHEMA` est omis, les droits globaux par défaut sont modifiés.

`nom_rôle`
Le nom d'un rôle existant pour donner ou reprendre les droits. Ce paramètre, et tous les autres paramètres dans *`grant_ou_revoke_réduit`*, agissent de la façon décrite dans GRANT(7) ou REVOKE(7), sauf qu'un permet de configurer les droits pour une classe complète d'objets plutôt que pour des objets nommés spécifiques.

Notes

Utilisez la commande **\ddp** de psql(1) pour otbenir des informations sur les droits par défaut. La signification des valeurs de droit est identique à celles utilisées par **\dp** et est expliqué dans GRANT(7).

Si vous souhaitez supprimer un rôle dont les droits par défaut ont été modifiés, il est nécessaire d'inverser les modifications dans ses droits par défaut ou d'utiliser **DROP OWNED BY** pour supprimer l'entrée des droits par défaut pour le rôle.

Examples

Donner le droit SELECT à tout le monde pour toutes les tables (et vues) que vous pourriez créer plus tard dans le schéma `mon_schema`, et permettre au rôle `webuser` d'utiliser en plus INSERT :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA mon_schema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA mon_schema GRANT INSERT ON TABLES TO webuser;
```

Annuler ce qui a été fait ci-dessus, pour que les tables créées par la suite n'aient pas plus de droits qu'en standard :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

Supprimer le droit publique EXECUTE qui est normalement donné aux fonctions, pour toutes les fonctions créées après coup par le rôle `admin` :

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Compatibilité

Il n'existe pas d'instruction **ALTER DEFAULT PRIVILEGES** dans le standard SQL.

Voir aussi

GRANT(7), REVOKE(7)

Nom

ALTER DOMAIN — Modifier la définition d'un domaine

Synopsis

```
ALTER DOMAIN nom
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN nom
    { SET | DROP } NOT NULL
ALTER DOMAIN nom
    ADD contrainte_de_domaine
ALTER DOMAIN nom
    DROP CONSTRAINT nom_de_contrainte [ RESTRICT | CASCADE ]
ALTER DOMAIN nom
    OWNER TO nouveau_propriétaire
ALTER DOMAIN nom
    SET SCHEMA nouveau_schema
```

Description

ALTER DOMAIN modifie la définition d'un domaine. Il existe plusieurs sous-formes :

SET/DROP DEFAULT
: Ces formes positionnent ou suppriment la valeur par défaut d'un domaine. Les valeurs par défaut ne s'appliquent qu'aux commandes **INSERT** ultérieures ; les colonnes d'une table qui utilise déjà le domaine ne sont pas affectées.

SET/DROP NOT NULL
: Ces formes agissent sur l'acceptation ou le rejet des valeurs NULL par un domaine. `SET NOT NULL` ne peut être utilisé que si les colonnes qui utilisent le domaine contiennent des valeurs non nulles.

ADD *`contrainte de domaine`*
: Cette forme ajoute une nouvelle contrainte à un domaine avec la même syntaxe que CREATE DOMAIN(7). Ceci ne fonctionne que lorsque toutes les colonnes qui utilisent le domaine satisfont à la nouvelle contrainte.

DROP CONSTRAINT
: Cette forme supprime les contraintes sur un domaine.

OWNER
: Cette forme change le propriétaire du domaine.

SET SCHEMA
: Cette forme change le schéma du domaine. Toute contrainte associée au domaine est déplacée dans le nouveau schéma.

Seul le propriétaire de la fonction d'agrégat peut utiliser **ALTER AGGREGATE**.

Seul le propriétaire du domaine peut utiliser **ALTER DOMAIN**. Pour modifier le schéma d'un domaine, le droit `CREATE` sur le nouveau schéma est également requis. Pour modifier le propriétaire, il faut être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit `CREATE` sur le schéma du domaine. Ces restrictions assurent que la modification du propriétaire n'agissent pas au-delà de ce qui est réalisable en supprimant et en re-créant le domaine. Toutefois, un superutilisateur peut modifier le propriétaire de n'importe quel domaine.

Paramètres

`nom`
: Le nom du domaine à modifier.

`contrainte_de_domaine`
: Nouvelle contrainte de domaine pour le domaine.

`nom_de_contrainte`
: Le nom d'une contrainte à supprimer.

`CASCADE`
: Les objets qui dépendent de la contrainte sont automatiquement supprimés.

`RESTRICT`
La contrainte n'est pas supprimée si des objets en dépendent. C'est le comportement par défaut.

`nouveau_propriétaire`
Le nom de l'utilisateur nouveau propriétaire du domaine.

`nouveau_schema`
Le nouveau schéma du domaine.

Notes

Actuellement, **ALTER DOMAIN ADD CONSTRAINT** et **ALTER DOMAIN SET NOT NULL** échoueront si le domaine nommé ou tout domaine dérivé est utilisé pour une colonne de type composite dans toute table de la base de données. Il se pourrait que cela soit amélioré pour vérifier la nouvelle contrainte sur ce type de colonnes intégrées.

Exemples

Ajouter une contrainte `NOT NULL` à un domaine :

```
ALTER DOMAIN codezip SET NOT NULL;
```

Supprimer une contrainte `NOT NULL` d'un domaine :

```
ALTER DOMAIN codezip DROP NOT NULL;
```

Ajouter une contrainte de contrôle à un domaine :

```
ALTER DOMAIN codezip ADD CONSTRAINT verif_zip CHECK (char_length(VALUE) = 5);
```

Supprimer une contrainte de contrôle d'un domaine :

```
ALTER DOMAIN codezip DROP CONSTRAINT verif_zip;
```

Déplacer le domaine dans un schéma différent :

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibilité

ALTER DOMAIN se conforme au standard SQL, à l'exception des variantes `OWNER` et `SET SCHEMA`, qui sont des extensions PostgreSQL™.

Voir aussi

CREATE DOMAIN(7), DROP DOMAIN(7)

Nom

ALTER FOREIGN DATA WRAPPER — modifier la définition d'un wrapper de données distantes

Synopsis

```
ALTER FOREIGN DATA WRAPPER nom
    [ VALIDATOR fonction_validation | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER nom OWNER TO nouveau_propriétaire
```

Description

ALTER FOREIGN DATA WRAPPER modifie la définition d'un wrapper de données distantes. La première forme de la commande modifie la bibliothèque ou les options génériques du wrapper de données distantes (au moins une clause est nécessaire). La seconde forme modifie le propriétaire du wrapper de données distantes.

Seuls les superutilisateurs peuvent modifier les wrappers de données distantes. De plus, seuls les superutilisateurs peuvent être propriétaire de wrappers de données distantes.

Paramètres

nom
: Le nom d'un wrapper de données distantes existant.

`VALIDATOR` *fonction_validation*
: Indique une fonction de validation pour le wrapper de données distantes.

  Notez qu'il est possible que les options des wrappers de données distantes, des serveurs et des correspondances utilisateur deviennent invalides une fois le validateur changé. C'est à l'utilisateur de s'assurer que ces options sont correctes avant d'utiliser le wrapper de données distantes.

`NO VALIDATOR`
: Cette option est utilisée pour spécifier que le wrapper de données distantes n'aura plus de fonction de validation.

`OPTIONS ( [ ADD | SET | DROP ]` *option* `['`*valeur*`'] [, ... ] )`
: Modifie les options du wrapper de données distantes. `ADD`, `SET` et `DROP` spécifient l'action à réaliser. `ADD` est pris par défaut si aucune opération n'est explicitement spécifiée. Les noms des options doivent être uniques ; les noms et valeurs sont validés en utilisant la bibliothèque de wrapper de données distantes.

Exemples

Modifier wrapper de données distantes `dbi`, ajouter l'option `foo`, supprimer `bar` :

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

Modifier la fonction de validation du wrapper de données distantes `dbi` en `bob.myvalidator` :

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Compatibilité

ALTER FOREIGN DATA WRAPPER se conforme à ISO/IEC 9075-9 (SQL/MED). Le standard ne spécifie pas les variantes `VALIDATOR` et `OWNER TO` de la commande.

Voir aussi

CREATE FOREIGN DATA WRAPPER(7), DROP FOREIGN DATA WRAPPER(7)

Nom

ALTER FUNCTION — Modifier la définition d'une fonction

Synopsis

```
ALTER FUNCTION nom ( [ [ modearg ] [ nomarg ] typearg [, ...] ] )
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION nom ( [ [ modearg ] [ nomarg ] typearg [, ...] ] )
    RENAME TO nouveau_nom
ALTER FUNCTION nom ( [ [ modearg ] [ nomarg ] typearg [, ...] ] )
    OWNER TO new_owner
ALTER FUNCTION nom ( [ [ modearg ] [ nomarg ] typearg [, ...] ] )
    SET SCHEMA nouveau_schema

où action peut être :

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    COST cout_execution
    ROWS nb_lignes_resultat
    SET parametre { TO | = } { valeur | DEFAULT }
    SET parametre FROM CURRENT
    RESET parametre
    RESET ALL
```

Description

ALTER FUNCTION modifie la définition d'une fonction.

Seul le propriétaire de la fonction peut utiliser **ALTER FUNCTION**. Le privilège `CREATE` sur le nouveau schéma est requis pour pouvoir changer le schéma de la fonction. Pour modifier le propriétaire, il est nécessaire d'être membre direct ou indirect du nouveau rôle propriétaire. Ce dernier doit posséder le droit `CREATE` sur le schéma de la fonction. Ces restrictions assurent que la modification du propriétaire n'a pas d'effets autres que ceux obtenus par la suppression et la re-création de la fonction ; toutefois, un superutilisateur peut modifier le propriétaire de n'importe quelle fonction.

Paramètres

nom
: Le nom de la fonction.

modearg
: Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. En cas d'omission, la valeur par défaut est `IN`. **ALTER FUNCTION** ne tient pas compte des arguments `OUT`, car seuls les arguments en entrée sont nécessaire pour déterminer l'identité de la fonction. Les arguments `IN`, `INOUT` et `VARIADIC` sont donc suffisants.

nomarg
: Le nom d'un argument. **ALTER FUNCTION** ne tient pas compte des noms des arguments, car seuls les types de données des arguments sont nécessaires pour déterminer l'identité d'une fonction.

typearg
: Le(s) type(s) de données des arguments de la fonction (éventuellement qualifié(s) du nom du schéma).

nouveau_nom
: Le nouveau nom de la fonction.

nouveau_proprietaire
: Le nouveau propriétaire de la fonction. Si cette fonction est marquée `SECURITY DEFINER`, elle s'exécute par la suite sous cette identité.

nouveau_schema
: Le nouveau schéma de la fonction.

`CALLED ON NULL INPUT`, `RETURNS NULL ON NULL INPUT`, `STRICT`
: `CALLED ON NULL INPUT` modifie la fonction pour qu'elle puisse être appelée avec des arguments NULL. `RETURNS NULL ON NULL INPUT` et `STRICT` modifie la fonction pour qu'elle ne soit pas appelée si un des arguments est NULL ;

un résultat NULL est alors automatiquement déterminé. Voir CREATE FUNCTION(7) pour plus d'informations.

`IMMUTABLE`, `STABLE`, `VOLATILE`
Modifie la volatilité de la fonction. Voir CREATE FUNCTION(7) pour plus d'informations.

`[ EXTERNAL ] SECURITY INVOKER`, `[ EXTERNAL ] SECURITY DEFINER`
Précise si la fonction doit être appelée avec les droits de l'utilisateur qui l'a créée. Le mot clé `EXTERNAL`, ignoré, existe pour des raisons de compatibilité SQL. Voir CREATE FUNCTION(7) pour plus d'informations.

`COST` *`cout_execution`*
Modifie l'estimation du coût d'exécution de la fonction. Voir CREATE FUNCTION(7) pour plus d'informations.

`ROWS` *`nb_lignes_resultat`*
Modifie l'estimation du nombre de lignes renvoyées par une fonction SRF. Voir CREATE FUNCTION(7) pour plus d'informations.

`parametre`, *`valeur`*
Ajoute ou modifie l'initialisation d'un paramètre de configuration lorsque la fonction est appelée. Si *`valeur`* est `DEFAULT` ou, de façon équivalente, si `RESET` est utilisé, le paramètre local de la fonction est supprimée pour que la fonction s'exécute avec la valeur par défaut du paramètre. Utiliser `RESET ALL` supprime tous les valeurs spécifiques des paramètres pour cette fonction. `SET FROM CURRENT` sauvegarde la valeur courante comme valeur du paramètre à appliquer lors de l'exécution de la fonction.

Voir SET(7) et Chapitre 18, Configuration du serveur pour plus d'informations sur les noms des paramètres et les valeurs autorisés.

`RESTRICT`
Ignoré, présent pour des raisons de conformité avec le standard SQL.

Exemples

Renommer la fonction `sqrt` pour le type integer en `square_root` :

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

Changer le propriétaire de la fonction `sqrt` pour le type integer en `joe` :

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

Modifier le schéma de la fonction `sqrt` du type integer par `maths` :

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

Pour ajuster automatiquement le chemin de recherche des schémas pour une fonction :

```
ALTER FUNCTION verifie_motdepasse(text) SET search_path = admin, pg_temp;
```

Pour désactiver le paramètre `search_path` d'une fonction :

```
ALTER FUNCTION verifie_motdepasse(text) RESET search_path;
```

La fonction s'exécutera maintenant avec la valeur de la session pour cette variable.

Compatibilité

La compatibilité de cette instruction avec l'instruction **ALTER FUNCTION** du standard SQL est partielle. Le standard autorise la modification d'un plus grand nombre de propriétés d'une fonction mais ne laisse pas la possibilité de renommer une fonction, de placer le commutateur `SECURITY DEFINER` sur la fonction, d'y attacher des valeurs de paramètres ou d'en modifier le propriétaire, le schéma ou la volatilité. Le standard requiert le mot clé `RESTRICT` ; il est optionnel avec PostgreSQL™.

Voir aussi

CREATE FUNCTION(7), DROP FUNCTION(7)

Nom

ALTER GROUP — Modifier le nom d'un rôle ou la liste de ses membres

Synopsis

```
ALTER GROUP nom_groupe ADD USER nom utilisateur [, ... ]
ALTER GROUP nom_groupe DROP USER nom_utilisateur [, ... ]

ALTER GROUP nom_groupe RENAME TO nouveau_nom
```

Description

ALTER GROUP modifie les attributs d'un groupe d'utilisateurs Cette commande est obsolète, mais toujours acceptée pour des raisons de compatibilité ascendante. Les groupes (et les utilisateurs) ont été remplacés par le concept plus général de rôles.

Les deux premières formes ajoutent des utilisateurs à un groupe ou en suppriment. Tout rôle peut être ici « utilisateur » ou « groupe ». Ces variantes sont réellement équivalentes à la promotion ou la révocation de l'appartenance au rôle nommé « groupe » ; il est donc préférable d'utiliser GRANT(7) et REVOKE(7) pour le faire.

La troisième forme change le nom du groupe. Elle est strictement équivalente au renommage du rôle par ALTER ROLE(7).

Paramètres

nom_groupe
: Le nom du groupe (rôle) à modifier.

nom_utilisateur
: Les utilisateurs (rôles) à ajouter au groupe ou à en enlever. Les utilisateurs doivent préalablement exister ; **ALTER GROUP** ne crée pas et ne détruit pas d'utilisateur.

nouveau_nom
: Le nouveau nom du groupe.

Exemples

Ajouter des utilisateurs à un groupe :

```
ALTER GROUP staff ADD USER karl, john;
```

Supprimer des utilisateurs d'un groupe :

```
ALTER GROUP workers DROP USER beth;
```

Compatibilité

Il n'existe pas de relation **ALTER GROUP** en SQL standard.

Voir aussi

GRANT(7), REVOKE(7), ALTER ROLE(7)

Nom

ALTER INDEX — Modifier la définition d'un index

Synopsis

```
ALTER INDEX nom RENAME TO nouveau_nom
ALTER INDEX nom SET TABLESPACE nom_espacelogique
ALTER INDEX nom SET ( parametre_stockage = valeur [, ... ] )
ALTER INDEX nom RESET ( parametre_stockage [, ... ] )
```

Description

ALTER INDEX modifie la définition d'un index. Il existe plusieurs formes de l'instruction :

`RENAME`
: La forme `RENAME` modifie le nom de l'index. Cela n'a aucun effet sur les données stockées.

`SET TABLESPACE`
: Cette forme remplace le tablespace de l'index par le tablespace spécifié et déplace le(s) fichier(s) de données associé(s) à l'index dans le nouveau tablespace. Voir aussi CREATE TABLESPACE(7).

`SET ( paramètre_stockage = valeur [, ... ] )`
: Cette forme modifie un ou plusieurs paramètres spécifiques à la méthode d'indexage de cet index. Voir CREATE INDEX(7) pour les détails sur les paramètres disponibles. Notez que le contenu de l'index ne sera pas immédiatement modifié par cette commande ; suivant le paramètre, vous pouvez avoir besoin de reconstruire l'index avec REINDEX(7) pour obtenir l'effet désiré.

`RESET ( paramètre_stockage [, ... ] )`
: Cette forme réinitialise un ou plusieurs paramètres de stockage spécifiques à la méthode d'indexage à leurs valeurs par défaut. Comme avec `SET`, un `REINDEX` peut être nécessaire pour mettre à jour l'index complètement.

Paramètres

nom
: Le nom de l'index à modifier (éventuellement qualifié du nom du schéma).

nouveau_nom
: Le nouveau nom de l'index.

nom_espacelogique
: Le nom du tablespace dans lequel déplacer l'index.

paramètre_stockage
: Le nom du paramètre de stockage spécifique à la méthode d'indexage.

valeur
: La nouvelle valeur du paramètre de stockage spécifique à la méthode d'indexage. Cette valeur peut être un nombre ou une chaîne suivant le paramètre.

Notes

Ces opérations sont aussi possibles en utilisant ALTER TABLE(7). **ALTER INDEX** n'est en fait qu'un alias pour les formes d'**ALTER TABLE** qui s'appliquent aux index.

Auparavant, il existait une variante **ALTER INDEX OWNER** mais elle est maintenant ignorée (avec un message d'avertissement). Un index ne peut pas avoir un propriétaire différent de celui de la table. Modifier le propriétaire de la table modifie automatiquement celui de l'index.

Il est interdit de modifier toute partie d'un index du catalogue système.

Exemples

Renommer un index existant :

```
ALTER INDEX distributeurs RENAME TO fournisseurs;
```

Déplacer un index dans un autre tablespace :

```
ALTER INDEX distributeurs SET TABLESPACE espacelogiquerapide;
```

Pour modifier le facteur de remplissage d'un index (en supposant que la méthode d'indexage le supporte) :

```
ALTER INDEX distributeurs SET (fillfactor = 75);
REINDEX INDEX distributeurs;
```

Compatibilité

ALTER INDEX est une extension PostgreSQL™.

Voir aussi

CREATE INDEX(7), REINDEX(7)

Nom

ALTER LANGUAGE — Modifier la définition d'un langage procédural

Synopsis

```
ALTER LANGUAGE nom RENAME TO nouveau_nom
```

```
ALTER LANGUAGE nom OWNER TO nouveau_proprietaire
```

Description

ALTER LANGUAGE modifie la définition d'un langage. Les seules fonctionnalités disponibles sont le renommage du langage et son changement de propriétaire. Vous devez être soit un super-utilisateur soit le propriétaire du langage pour utiliser **ALTER LANGUAGE**.

Paramètres

nom
: Le nom du langage.

nouveau_nom
: Le nouveau nom du langage.

new_owner
: Le nouveau propriétaire du langage

Compatibilité

Il n'existe pas de relation **ALTER LANGUAGE** dans le standard SQL.

Voir aussi

CREATE LANGUAGE(7), DROP LANGUAGE(7)

Nom

ALTER LARGE OBJECT — Modifier la définition d'un Large Object

Synopsis

```
ALTER LARGE OBJECT oid_large_object OWNER TO nouveau_propriétaire
```

Description

ALTER LARGE OBJECT modifie la définition d'un « Large Object ». La seule fonctionnalité disponible est l'affectation d'un nouveau propriétaire. Vous devez être un superutilisateur ou le propriétaire du « Large Object » pour utiliser **ALTER LARGE OBJECT**.

Paramètres

oid_large_object
: OID d'un « Large Object » à modifier

nouveau_propriétaire
: Le nouveau propriétaire du « Large Object »

Compatibilité

Il n'existe pas d'instruction **ALTER LARGE OBJECT** dans le standard SQL.

Voir aussi

Chapitre 32, Objets larges

Nom

ALTER OPERATOR — Modifier la définition d'un opérateur

Synopsis

```
ALTER OPERATOR name ( { type_gauche | NONE } , { type_droit | NONE } ) OWNER TO
nouveau_propriétaire
```

Description

ALTER OPERATOR modifie la définition d'un opérateur. La seule fonctionnalité disponible est le changement de propriétaire d'un opérateur.

Seul le propriétaire de l'opérateur peut utiliser **ALTER OPERATOR**. Pour modifier le propriétaire, il est nécessaire d'être un membre direct ou indirect du nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma de l'opérateur. Ces restrictions assurent que la modification du propriétaire produise le même résultat que la suppression et la re-création de l'opérateur ; néanmoins, un superutilisateur peut modifier le propriétaire de n'importe quel opérateur.

Paramètres

nom
: Le nom de l'opérateur (éventuellement qualifié du nom du schéma).

type_gauche
: Le type de données de l'opérande gauche de l'opérateur ; `NONE` si l'opérateur n'a pas d'opérande gauche.

type_droit
: Le type de données de l'opérande droit de l'opérateur ; `NONE` si l'opérateur n'a pas d'opérande droit.

nouveau_propriétaire
: Le nouveau propriétaire de l'opérateur.

Exemples

Modifier le propriétaire d'un opérateur personnalisé `a @@ b` pour le type text :

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Compatibilité

Il n'existe pas d'instructions **ALTER OPERATOR** dans le standard SQL.

Voir aussi

CREATE OPERATOR(7), DROP OPERATOR(7)

Nom

ALTER OPERATOR CLASS — Modifier la définition d'une classe d'opérateur

Synopsis

```
ALTER OPERATOR CLASS nom USING méthode_indexage RENAME TO nouveau_nom
ALTER OPERATOR CLASS nom USING méthode_indexage OWNER TO nouveau_propriétaire
```

Description

ALTER OPERATOR CLASS modifie la définition d'une classe d'opérateur.

Seul le propriétaire de la classe d'opérateur peut utiliser **ALTER OPERATOR CLASS**. Pour modifier le propriétaire, il est obligatoire d'être un membre direct ou indirect du nouveau rôle propriétaire. Ce rôle doit posséder le privilège `CREATE` sur le schéma de la classe d'opérateur. Ces restrictions assurent que la modification du propriétaire produise le même effet que celui obtenu par la suppression et la re-création de la classe d'opérateur ; néanmoins, un superutilisateur peut modifier le propriétaire de n'importe quelle classe d'opérateur.

Paramètres

`nom`
: Le nom d'une classe d'opérateur.

`méthode_indexage`
: Le nom de la méthode d'indexage à laquelle associer la classe d'opérateur.

`nouveau_nom`
: Le nouveau nom de la classe d'opérateur.

`nouveau_propriétaire`
: Le nouveau propriétaire de la classe d'opérateur.

Compatibilité

Il n'existe pas d'instruction **ALTER OPERATOR CLASS** dans le standard SQL.

Voir aussi

CREATE OPERATOR CLASS(7), DROP OPERATOR CLASS(7), ALTER OPERATOR FAMILY(7)

Nom

ALTER OPERATOR FAMILY — Modifier la définition d'une famille d'opérateur

Synopsis

```
ALTER OPERATOR FAMILY nom USING methode_indexage ADD
  {  OPERATOR numero_strategie nom_operateur ( type_op, type_op )
   | FUNCTION numero_support [ ( type_op [ , type_op ] ) ] nom_fonction (
type_argument [, ...] )
  } [, ... ]
ALTER OPERATOR FAMILY nom USING methode_indexage DROP
  {  OPERATOR numero_strategie ( type_op [ , type_op ] )
   | FUNCTION numero_support ( type_op [ , type_op ] )
  } [, ... ]
ALTER OPERATOR FAMILY nom USING methode_indexage RENAME TO nouveau_nom
ALTER OPERATOR FAMILY nom USING methode_indexage OWNER TO nouveau_proprietaire
```

Description

ALTER OPERATOR FAMILY modifie la définition d'une famille d'opérateur. Vous pouvez ajouter des opérateurs et des fonctions du support à la famille, les supprimer ou modifier le nom et le propriétaire de la famille.

Quand les opérateurs et fonctions de support sont ajoutés à une famille avec la commande **ALTER OPERATOR FAMILY**, ils ne font partie d'aucune classe d'opérateur spécifique à l'intérieur de la famille. Ils sont « lÃ¢ches » dans la famille. Ceci indique que ces opérateurs et fonctions sont compatibles avec la sémantique de la famille but qu'ils ne sont pas requis pour un fonctionnement correct d'un index spécifique. (Les opérateurs et fonctions qui sont ainsi nécessaires doivent être déclarés comme faisant partie d'une classe d'opérateur ; voir CREATE OPERATOR CLASS(7).) PostgreSQL™ la suppression des membres lÃ¢ches d'une famille à tout moment, mais les membres d'une classe d'opérateur ne peuvent pas être supprimés sans supprimer toute la classe et les index qui en dépendent. Typiquement, les opérateurs et fonctions sur un seul type de données font partie des classes d'opérateurs car ils ont besoin de supporter un index sur ce type de données spécifique alors que les opérateurs et familles intertypes sont fait de membres lÃ¢ches de la famille.

Vous devez être superutilisateur pour utiliser **ALTER OPERATOR FAMILY**. (Cette restriction est faite parce qu'une définition erronée d'une famille d'opérateur pourrait gêner voire même arrêter brutalement le serveur.)

ALTER OPERATOR FAMILY ne vérifie pas encore si la définition de l'opérateur de famille inclut tous les opérateurs et fonctions requis par la méthode d'indexage, ni si les opérateurs et les fonctions forment un ensemble cohérent et suffisant. C'est de la responsabilité de l'utilisateur de définir une famille d'opérateur valide.

Voir Section 35.14, « Interfacer des extensions d'index » pour plus d'informations.

Paramètres

nom
: Le nom d'une famille d'opérateur (pouvant être qualifié du schéma).

methode_indexage
: Le nom de la méthode d'indexage.

numero_strategie
: Le numéro de stratégie de la méthode d'indexage pour un opérateur associé avec la famille.

nom_operateur
: Le nom d'un opérateur (pouvant être qualifié du schéma) associé avec la famille d'opérateur.

type_op
: Dans une clause `OPERATOR`, les types de données en opérande de l'opérateur, ou `NONE` pour signifier un opérateur unaire. Contrairement à la syntaxe comparable de **CREATE OPERATOR CLASS**, les types de données en opérande doivent toujours être précisés.

  Dans une clause `ADD FUNCTION`, les types de données des opérandes que la fonction est sensée supporter, si différent des types de données en entrée de la fonction. Pour les index B-tree et hash, il n'est pas strictement nécessaire de spécifier *op_type* car les types de données en entrée de la fonction sont toujours les bons à utiliser. Pour les index GIN et GiST, il est nécessaire de spécifier le type de données en entrée qui sera utilisé par la fonction.

Dans une clause `DROP FUNCTION`, les types de données en opérande que la fonction est sensée supportée doivent être précisés.

`numero_support`
Le numéro de la procédure de support de la méthode d'indexage associé avec la famille d'opérateur.

`nom_fonction`
Le nom (pouvant être qualifié du schéma) d'une fonction qui est une procédure de support de la méthode d'indexage pour la famille d'opérateur.

`argument_types`
Les types de données pour les arguments de la fonction.

`nouveau_nom`
Le nouveau nom de la famille d'opérateur

`nouveau_proprietaire`
Le nouveau propriétaire de la famille d'opérateur

Les clauses `OPERATOR` et `FUNCTION` peuvent apparaÃ®tre dans n'importe quel ordre.

Notes

Notez que la syntaxe `DROP` spécifie uniquement le « slot » dans la famille d'opérateur, par stratégie ou numéro de support et types de données en entrée. Le nom de l'opérateur ou de la fonction occupant le slot n'est pas mentionné. De plus, pour `DROP FUNCTION`, les types à spécifier sont les types de données en entrée que la fonction doit supporter ; pour les index GIN et GiST, ceci pourrait ne rien avoir à faire avec les types d'argument en entrée de la fonction.

Comme le processus des index ne vérifie pas les droits sur les fonctions avant de les utiliser, inclure une fonction ou un opérateur dans une famille d'opérateur est équivalent à donner le droit d'exécution à public. Ceci n'est généralement pas un problème pour les tris de fonction qui sont utiles à une famille d'opérateur.

Les opérateurs ne doivent pas être définis par des fonctions SQL. Une fonction SQL risque d'être remplacée dans la requête appelante, ce qui empêchera l'optimiseur de savoir si la requête peut utiliser un index.

Avant PostgreSQL™ 8.4, la clause `OPERATOR` pouvait inclure une option `RECHECK`. Ce n'est plus supporté parce que le fait qu'un opérateur d'index soit « à perte » est maintenant déterminé à l'exécution. Cela permet une gestion plus efficace des cas où un opérateur pourrait ou non être à perte.

Exemples

La commande exemple suivant ajoute des opérateurs inter-type de données et ajoute les fonctions de support pour une famille d'opérateur qui contient déjà les classes d'opérateur B_tree pour les types de données int4 et int2.

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

Pour supprimer de nouveau ces entrées :

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
```

```
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

Compatibilité

Il n'existe pas d'instruction **ALTER OPERATOR FAMILY** dans le standard SQL.

Voir aussi

CREATE OPERATOR FAMILY(7), DROP OPERATOR FAMILY(7), CREATE OPERATOR CLASS(7), ALTER OPERATOR CLASS(7), DROP OPERATOR CLASS(7)

Nom

ALTER ROLE — Modifier un rôle de base de données

Synopsis

```
ALTER ROLE nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT limiteconnexion
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'motdepasse'
    | VALID UNTIL 'dateheure'

ALTER ROLE nom RENAME TO nouveau_nom

ALTER ROLE nom [ IN DATABASE nom_base ] SET parametre_configuration { TO | = } {
valeur | DEFAULT }
ALTER ROLE nom [ IN DATABASE nom_base ] SET parametre_configuration FROM CURRENT
ALTER ROLE nom [ IN DATABASE nom_base ] RESET parametre_configuration
ALTER ROLE nom [ IN DATABASE nom_base ] RESET ALL
```

Description

ALTER ROLE modifie les attributs d'un rôle PostgreSQL™.

La première variante listée dans le synopsis, permet de modifier la plupart des attributs de rôle spécifiables dans la commande CREATE ROLE(7) (à lire pour plus de détails). (Tous les attributs possibles sont couverts, à l'exception de la gestion des appartenances ; GRANT(7) et REVOKE(7) sont utilisés pour cela.) Les attributs qui ne sont pas mentionnés dans la commande conservent leur paramètrage précédent. Tous ces attributs peuvent âtre modifiés pour tout rôle par les superutilisateurs de base de données. Les rôles qui possédent le privilège `CREATEROLE` peuvent modifier ces paramètres, mais uniquement pour les rôles qui ne sont pas superutilisateur. Les rôles ordinaires ne peuvent modifier que leur mot de passe.

La deuxième variante permet de modifier le nom du rôle. Les superutilisateurs peuvent renommer n'importe quel rôle. Les rôles disposant du droit `CREATEROLE` peuvent renommer tout rôle qui n'est pas superutilisateur. L'utilisateur de la session en cours ne peut pas être renommé. (On se connectera sous un autre utilisateur pour cela.) Comme les mots de passe chiffrés par `MD5` utilisent le nom du rôle comme grain de chiffrement, renommer un rôle efface son mot de passe si ce dernier est chiffré avec `MD5`.

Les autres variantes modifient la valeur par défaut d'une variable de configuration de session pour un rôle, soit pour toutes les bases soit, quand la clause `IN DATABASE` est spécifiée, uniquement pour les sessions dans la base nommée. Quand le rôle lance une nouvelle session après cela, la valeur spécifiée devient la valeur par défaut de la session, surchargeant tout paramètrage présent dans `postgresql.conf` ou provenant de la ligne de commande de **postgres**. Ceci arrive seulement lors de la connexion ; exécuter SET ROLE(7) ou SET SESSION AUTHORIZATION(7) ne cause pas la configuration de nouvelles valeurs pour les paramètres. L'ensemble des paramètres pour toutes les bases est surchargé par les paramètres spécifique à cette base attachés à un rôle Les superutilisateurs peuvent modifier les valeurs de session de n'importe quel utilisateur. Les rôles disposant du droit `CREATEROLE` peuvent modifier les valeurs par défaut pour les rôles ordinaires (non superutilisateurs). Les rôles standards peuvent seulement configurer des valeurs par défaut pour eux-mêmes. Certaines variables ne peuvent être configurées de cette façon ou seulement par un superutilisateur.

Paramètres

nom
: Le nom du rôle dont les attributs sont modifiés.

`SUPERUSER`, `NOSUPERUSER`, `CREATEDB`, `NOCREATEDB`, `CREATEROLE`, `NOCREATEROLE`, `CREATEUSER`, `NOCREATEUSER`, `INHERIT`, `NOINHERIT`, `LOGIN`, `NOLOGIN`, `CONNECTION LIMIT` *limite_connexion*, `PASSWORD` *motdepasse*, `ENCRYPTED`, `UNENCRYPTED`, `VALID UNTIL '`*dateheure*`'`
: Ces clauses modifient les attributs originairement configurés par CREATE ROLE(7). Pour plus d'informations, voir la page de référence **CREATE ROLE**.

nouveau_nom
Le nouveau nom du rôle.

nom_base
Le nom d'une base où se fera la configuration de la variable.

paramètre_configuration, *valeur*
Positionne la valeur de session par défaut à *valeur* pour le paramètre de configuration *paramètre*. Si `DEFAULT` est donné pour *valeur* ou, de façon équivalente, si `RESET` est utilisé, le positionnement spécifique de la variable pour le rôle est supprimé. De cette façon, le rôle hérite de la valeur système par défaut pour les nouvelles sessions. `RESET ALL` est utilisé pour supprimer tous les paramètrages rôle. `SET FROM CURRENT` sauvegarde la valeur de la session de ce paramètre en tant que valeur du rôle. Si `IN DATABASE` est précisé, le paramètre de configuration est intialisé ou supprimé seulement pour le rôle et la base indiqués.

Les paramètres spécifiques au rôle ne prennent effet qu'à la connexion ; SET ROLE(7) et SET SESSION AUTHORIZATION(7) ne traitent pas les paramètres de rôles.

Voir SET(7) et Chapitre 18, Configuration du serveur pour plus d'informations sur les noms et les valeurs autorisés pour les paramètres.

Notes

CREATE ROLE(7) est utilisé pour ajouter de nouveaux rôles et DROP ROLE(7) pour les supprimer.

ALTER ROLE ne peut pas modifier les appartenances à un rôle. GRANT(7) et REVOKE(7) sont conçus pour cela.

Faites attention lorsque vous précisez un mot de passe non chiffré avec cette commande. Le mot de passe sera transmis en clair au serveur. Il pourrait se trouver tracer dans l'historique des commandes du client et dans les traces du serveur. psql(1) contient une commande **\password** qui peut être utilisé pour changer le mot de passe d'un rôle sans exposer le mot de passe en clair.

Il est également possible de lier une valeur de session par défaut à une base de données plutôt qu'à un rôle ; voir ALTER DATABASE(7). S'il y a un conflit, les paramètres spécifiques à la paire base de données/rôle surchargent ceux spécifiques au rôle, qui eux-même surchargent ceux spécifiques à la base de données.

Exemples

Modifier le mot de passe d'un rôle :

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Supprimer le mot de passe d'un rôle :

```
ALTER ROLE davide WITH PASSWORD NULL;
```

Modifier la date d'expiration d'un mot de passe, en spécifiant que le mot de passe doit expirer à midi le 4 mai 2015 fuseau horaire UTC plus 1 heure :

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Créer un mot de passe toujours valide :

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Donner à un rôle la capacité de créer d'autres rôles et de nouvelles bases de données :

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Donner à un rôle une valeur différente de celle par défaut pour le paramètre maintenance_work_mem :

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Donner à un rôle une configuration dufférente, spécifique à une base de données, du paramètre client_min_messages :

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

Compatibilité

L'instruction **ALTER ROLE** est une extension PostgreSQL™.

Voir aussi

CREATE ROLE(7), DROP ROLE(7), SET(7)

Nom

ALTER SCHEMA — Modifier la définition d'un schéma

Synopsis

```
ALTER SCHEMA nom RENAME TO nouveau_nom
ALTER SCHEMA nom OWNER TO nouveau_propriétaire
```

Description

ALTER SCHEMA modifie la définition d'un schéma.

Seul le propriétaire du schéma peut utiliser **ALTER SCHEMA**. Pour renommer le schéma, le droit `CREATE` sur la base est obligatoire. Pour modifier le propriétaire, il faut être membre, direct ou indirect, du nouveau rôle propriétaire, et posséder le droit `CREATE` sur la base (les superutilisateurs ont automatiquement ces droits).

Paramètres

`nom`
: Le nom du schéma.

`nouveau_nom`
: Le nouveau nom du schéma. Il ne peut pas commencer par `pg_`, noms réservés aux schémas système.

`nouveau_propriétaire`
: Le nouveau propriétaire du schéma.

Compatibilité

Il n'existe pas de relation **ALTER SCHEMA** dans le standard SQL.

Voir aussi

CREATE SCHEMA(7), DROP SCHEMA(7)

Nom

ALTER SEQUENCE — Modifier la définition d'un générateur de séquence

Synopsis

```
ALTER SEQUENCE nom [ INCREMENT [ BY ] increment ]
    [ MINVALUE valeurmin | NO MINVALUE ] [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ RESTART [ [ WITH ] nouveau_début ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table.colonne | NONE } ]
ALTER SEQUENCE nom OWNER TO nouveau_propriétaire
ALTER SEQUENCE nom RENAME TO nouveau_nom
ALTER SEQUENCE nom SET SCHEMA nouveau_schema
```

Description

ALTER SEQUENCE modifie les paramètres d'un générateur de séquence. Tout paramètre non précisé dans la commande **ALTER SEQUENCE** conserve sa valeur précédente. Pour modifier le propriétaire, vous devez aussi être un membre direct ou indirect du nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma de la séquence (ces restrictions permettent de s'assurer que modifier le propriétaire ne fait rien de plus que ce que vous pourriez faire en supprimant puis recréant la séquence ; néanmoins un superutilisateur peut déjà modifier le propriétaire de toute séquence).

Seul le propriétaire de la séquence peut utiliser **ALTER SEQUENCE**. Pour modifier le schéma de la séquence, il faut posséder le droit `CREATE` sur le nouveau schéma.

Paramètres

`nom`
: Le nom de la séquence à modifier (éventuellement qualifié du nom du schéma).

`increment`
: La clause `INCREMENT BY` *`increment`* est optionnelle. Une valeur positive crée une séquence croissante, une valeur négative une séquence décroissante. Lorsque cette clause n'est pas spécifiée, la valeur de l'ancien incrément est conservée.

`valeurmin`, `NO MINVALUE`
: La clause optionnelle `MINVALUE` *`valeurmin`*, détermine la valeur minimale de la séquence. Si `NO MINVALUE` est utilisé, les valeurs par défaut, 1 et $-2^{63}-1$ sont utilisées respectivement pour les séquences croissantes et decroissantes. Si aucune option n'est précisée, la valeur minimale courante est conservée.

`valeurmax`, `NO MAXVALUE`
: La clause optionnelle `MAXVALUE` *`valeurmax`* détermine la valeur maximale de la séquence. Si `NO MAXVALUE` est utilisé, les valeurs par défaut $2^{63}-1$ et -1 sont utilisées respectivement pour les séquences croissantes et décroissantes. Si aucune option n'est précisée, la valeur maximale courante est conservée.

`début`
: La clause optionnelle `START WITH` *`début`* modifie la valeur de départ enregistré pour la séquence. Cela n'a pas d'effet sur la valeur *actuelle* de celle-ci ; cela configure la valeur que les prochaines commandes **ALTER SEQUENCE RESTART** utiliseront.

`restart`
: La clause optionnelle `RESTART [ WITH` *`restart`* `]` modifie la valeur actuelle de la séquence. C'est équivalent à l'appel de la fonction `setval` avec `is_called = false` : la valeur spécifiée sera renvoyée par le *prochain* appel à `nextval`. Écrire `RESTART` sans valeur pour *`restart`* est équivalent à fournir la valeur de début enregistrée par **CREATE SEQUENCE** ou par **ALTER SEQUENCE START WITH**.

`cache`
: La clause `CACHE` *`cache`* active la préallocation des numéros de séquences et leur stockage en mémoire pour en accélerer l'accès. 1 est la valeur minimale (une seule valeur est engendrée à la fois, soit pas de cache). Lorsque la clause n'est pas spécifiée, l'ancienne valeur est conservée.

`CYCLE`
: Le mot clé optionnel `CYCLE` est utilisé pour autoriser la séquence à boucler lorsque *`valeurmax`* ou *`valeurmin`* est at-

teint par, respectivement, une séquence croissante ou décroissante. Lorsque la limite est atteinte, le prochain numéro engendré est, respectivement, *valeurmin* ou *valeurmax*.

`NO CYCLE`
Si le mot clé optionnel `NO CYCLE` est spécifié, tout appel à `nextval` alors que la séquence a atteint sa valeur maximale, dans le cas d'une séquence croissante, ou sa valeur minimale dans le cas contraire, retourne une erreur. Lorsque ni `CYCLE` ni `NO CYCLE` ne sont spécifiés, l'ancien comportement est préservé.

`OWNED BY` *table.colonne*, `OWNED BY NONE`
L'option `OWNED BY` permet d'associer la séquence à une colonne spécifique d'une table pour que cette séquence soit supprimée automatiquement si la colonne (ou la table complète) est supprimée. Si cette option est spécifiée, cette association remplacera toute ancienne association de cette séquence. La table indiquée doit avoir le même propriétaire et être dans le même schéma que la séquence. Indiquer `OWNED BY NONE` supprime toute association existante, rendant à la séquence son « autonomie ».

nouveau_propriétaire
Le nom utilisateur du nouveau propriétaire de la séquence.

nouveau_nom
Le nouveau nom de la séquence.

nouveau_schema
Le nouveau schéma de la séquence.

Notes

Pour éviter de bloquer des transactions concurrentes lors de la demande de numéros issus de la même séquence, les effets d'**ALTER SEQUENCE** sur les paramètres de génération de la séquence ne sont jamais annulables. Ces changements prennent effet immédiatement et ne sont pas réversibles. Néanmoins, les clauses `OWNED BY`, `OWNER TO`, `RENAME TO` et `SET SCHEMA` sont des modifications ordinaires du catalogue et, de ce fait, peuvent être annulées.

ALTER SEQUENCE n'affecte pas immédiatement les résultats de `nextval` pour les sessions, à l'exception de la session courante, qui ont préalloué (caché) des valeurs de la séquence. Elles épuisent les valeurs en cache avant de prendre en compte les modifications sur les paramètres de génération de la séquence. La session à l'origine de la commande est, quant à elle, immédiatement affectée.

ALTER SEQUENCE ne modifie pas le statut `currval` d'une séquence (avant PostgreSQL™ 8.3, c'était le cas quelque fois).

Pour des raisons historiques, **ALTER TABLE** peut aussi être utilisé avec les séquences, mais seules les variantes d'**ALTER TABLE** autorisées pour les séquences sont équivalentes aux formes affichées ci-dessus.

Exemples

Redémarrez la séquence `serial` à 105 :

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibilité

ALTER SEQUENCE est conforme au standard SQL, à l'exception des variantes `START WITH`, `OWNED BY`, `OWNER TO`, `RENAME TO` et `SET SCHEMA` qui sont une extension PostgreSQL™.

Voir aussi

CREATE SEQUENCE(7), DROP SEQUENCE(7)

Nom

ALTER SERVER — modifier la définition d'un serveur distant

Synopsis

```
ALTER SERVER nom_serveur [ VERSION 'nouvelle_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] ) ]
ALTER SERVER nom_serveur OWNER TO nouveau_propriétaire
```

Description

ALTER SERVER modifie la définition d'un serveur distant. La première forme modifie la chaîne de version du serveur ou les options génériques du serveur (au moins une clause est nécessaire). La seconde forme modifie le propriétaire du serveur.

Pour modifier le serveur, vous devez être le propriétaire du serveur. De plus, pour modifier le propriétaire, vous devez posséder le serveur ainsi qu'être un membre direct ou indirect du nouveau rôle, et vous devez avoir le droit `USAGE` sur le wrapper de données distantes du serveur. (Notez que les superutilisateurs satisfont à tout ces critères automatiquement.)

Paramètres

`nom_serveur`
: Le nom d'un serveur existant.

`nouvelle_version`
: Nouvelle version du serveur.

`OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )`
: Modifie des options pour le serveur. `ADD`, `SET` et `DROP` spécifient les actions à exécuter. Si aucune opération n'est spécifiée explicitement, l'action est `ADD`. Les noms d'options doivent être uniques ; les noms et valeurs sont aussi validés en utilisant la bibliothèque de wrapper de données distantes.

Exemples

Modifier le serveur `foo` et lui ajouter des options de connexion :

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'dbfoo');
```

Modifier le serveur `foo`, modifier sa version, modifier son option `host` :

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

Compatibilité

ALTER SERVER est conforme à ISO/IEC 9075-9 (SQL/MED).

Voir aussi

CREATE SERVER(7), DROP SERVER(7)

Nom

ALTER TABLE — Modifier la définition d'une table

Synopsis

```
ALTER TABLE [ ONLY ] nom [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] nom [ * ]
    RENAME [ COLUMN ] colonne TO nouvelle_colonne
ALTER TABLE nom
    RENAME TO nouveau_nom
ALTER TABLE nom
    SET SCHEMA nouveau_schema

où action peut être :

    ADD [ COLUMN ] colonne type [ contrainte_colonne [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] colonne [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] colonne [ SET DATA ] TYPE type [ USING expression ]
    ALTER [ COLUMN ] colonne SET DEFAULT expression
    ALTER [ COLUMN ] colonne DROP DEFAULT
    ALTER [ COLUMN ] colonne { SET | DROP } NOT NULL
    ALTER [ COLUMN ] colonne SET STATISTICS entier
    ALTER [ COLUMN ] column SET ( option_attribut = valeur [, ... ] )
    ALTER [ COLUMN ] column RESET ( option_attribut [, ... ] )
    ALTER [ COLUMN ] colonne SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD contrainte_table
    DROP CONSTRAINT [ IF EXISTS ] nom_contrainte [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ nom_declencheur | ALL | USER ]
    ENABLE TRIGGER [ nom_declencheur | ALL | USER ]
    ENABLE REPLICA TRIGGER nom_trigger
    ENABLE ALWAYS TRIGGER nom_trigger
    DISABLE RULE nom_regle_reecriture
    ENABLE RULE nom_regle_reecriture
    ENABLE REPLICA RULE nom_regle_reecriture
    ENABLE ALWAYS RULE nom_regle_reecriture
    CLUSTER ON nom_index
    SET WITHOUT CLUSTER
    SET WITH OIDS
    SET WITHOUT OIDS
    SET ( paramètre_stockage = valeur [, ... ] )
    RESET ( paramètre_stockage [, ... ] )
    INHERIT table_parent
    NO INHERIT table_parent
    OWNER TO nouveau_proprietaire
    SET TABLESPACE nouvel_espacelogique
```

Description

ALTER TABLE modifie la définition d'une table existante. Il existe plusieurs variantes :

`ADD COLUMN`
: Ajoute une nouvelle colonne à la table en utilisant une syntaxe identique à celle de CREATE TABLE(7).

`DROP COLUMN [ IF EXISTS ]`
: Supprime une colonne de la table. Les index et les contraintes de table référençant cette colonne sont automatiquement supprimés. L'option `CASCADE` doit être utilisée lorsque des objets en dehors de la table dépendent de cette colonne, comme par exemple des références de clés étrangères ou des vues. Si `IF EXISTS` est indiqué et que la colonne n'existe pas, aucune erreur n'est renvoyée. Dans ce cas, un message d'avertissement est envoyé à la place.

`SET DATA TYPE`
: Change le type d'une colonne de la table. Les index et les contraintes simples de table qui impliquent la colonne sont automatiquement convertis pour utiliser le nouveau type de la colonne en réanalysant l'expression d'origine. La clause optionnelle `USING` précise comment calculer la nouvelle valeur de la colonne à partir de l'ancienne ; en cas d'omission, la conversion par défaut est identique à une affectation de transtypage de l'ancien type vers le nouveau. Une clause `USING` doit être

fournie s'il n'existe pas de conversion implicite ou d'assignement entre les deux types.

`SET/DROP DEFAULT`
Ajoute ou supprime les valeurs par défaut d'une colonne. Les valeurs par défaut ne s'appliquent qu'aux commandes **INSERT** ultérieures. Elles ne modifient pas les lignes déjà présentes dans la table. Des valeurs par défaut peuvent aussi être créées pour les vues. Dans ce cas, elles sont ajoutées aux commandes **INSERT** de la vue avant que la règle `ON INSERT` de la vue ne soit appliquée.

`SET/DROP NOT NULL`
Modifie l'autorisation de valeurs NULL. `SET NOT NULL` ne peut être utilisé que si la colonne ne contient pas de valeurs NULL.

`SET STATISTICS`
Permet de modifier l'objectif de collecte de statistiques par colonne pour les opérations d'analyse (ANALYZE(7)) ultérieures. L'objectif prend une valeur entre 0 et 10000. il est positionné à -1 pour utiliser l'objectif de statistiques par défaut du système (default_statistics_target). Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes de PostgreSQL™, voir Section 14.2, « Statistiques utilisées par le planificateur ».

`SET ( attribute_option = value [, ... ] ), RESET ( attribute_option [, ... ] )`
Cette syntaxe permet de configurer ou de réinitialiser des propriétés. Actuellement, les seules propriétés acceptées sont `n_distinct` et `n_distinct_alled`, qui surchargent l'estimation du nombre de valeurs distinctes calculée par ANALYZE(7) `n_distinct` affecte les statistiques de la table elle-même alors que `n_distinct_alled` affecte les statistiques récupérées pour la table et les tables en héritant. Si configuré à une valeur positive, **ANALYZE** supposera que la colonne contient exactement le nombre spécifié de valeurs distinctes non NULL. Si configuré à une valeur négative qui doit être supérieur ou égale à -1, **ANALYZE** supposera que le nombre de valeurs distinctes non NULL dans la colonne est linéaire par rapport à la taille de la table ; le nombre total est à calculer en multipliant la taille estimée de la table par la valeur absolue de ce nombre. Par exemple, une valeur de -1 implique que toutes les valeurs dans la colonne sont distinctes alors qu'une valeur de -0,5 implique que chaque valeur apparaît deux fois en moyenne. Ceci peut être utile quand la taille de la table change dans le temps, car la multiplication par le nombre de lignes dans la table n'est pas réalisée avant la planification. Spécifiez une valeur de 0 pour retourner aux estimations standards du nombre de valeurs distinctes. Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes PostgreSQL™, référez-vous à Section 14.2, « Statistiques utilisées par le planificateur ».

`SET STORAGE`

Modifie le mode de stockage pour une colonne. Cela permet de contrôler si cette colonne est conservée en ligne ou dans une deuxième table, appelée table TOAST, et si les données sont ou non compressées. `PLAIN`, en ligne, non compressé, est utilisé pour les valeurs de longueur fixe, comme les integer. `MAIN` convient pour les données en ligne, compressibles. `EXTERNAL` est fait pour les données externes non compressées, `EXTENDED` pour les données externes compressées. `EXTENDED` est la valeur par défaut pour la plupart des types qui supportent les stockages différents de `PLAIN`. L'utilisation d'`EXTERNAL` permet d'accélérer les opérations d'extraction de sous-chaînes sur les très grosses valeurs de types text et bytea mais utilise plus d'espace de stockage. `SET STORAGE` ne modifie rien dans la table, il configure la stratégie à poursuivre lors des mises à jour de tables suivantes. Voir Section 54.2, « TOAST » pour plus d'informations.

`ADD contrainte_table`
Ajoute une nouvelle contrainte à une table en utilisant une syntaxe identique à CREATE TABLE(7).

`DROP CONSTRAINT [ IF EXISTS ]`
Supprime la contrainte de table précisée. Si `IF EXISTS` est précisé et que la contrainte n'existe pas, aucune erreur n'est renvoyée. Par contre, un message d'avertissement est lancé.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`
Configure l'exécution des déclencheurs définis sur la table. Un déclencheur désactivé est toujours connu par le système mais n'est plus exécuté lorsque l'événement déclencheur survient. Pour un déclencheur retardé, le statut d'activité est vérifié au moment où survient l'événement, et non quand la fonction du déclencheur est réellement exécutée. Il est possible de désactiver ou d'activer un déclencheur spécifique (précisé par son nom), tous les déclencheurs d'une table ou seulement les déclencheurs utilisateur de cette table (cette option exclut les déclencheurs générés en interne pour gérer les contraintes comme ceux utilisés pour implanter les contraintes de clés étrangères ou les contraintes déferrés uniques ou d'exclusion). Désactiver ou activer les déclencheurs implicites de contraintes requiert des droits de superutilisateur ; cela doit se faire avec précaution car l'intégrité de la contrainte ne peut pas être garantie si les déclencheurs ne sont pas exécutés. Le mécanisme de déclenchement des triggers est aussi affecté par la variable de configuration session_replication_role. Les triggers activés (`ENABLE`) se déclencheront quand le rôle de réplication est « origin » (la valeur par défaut) ou « local ». Les triggers configurés `ENABLE REPLICA` se déclencheront seulement si la session est en mode « replica » et les triggers `ENABLE ALWAYS` se déclencheront à chaque fois, quelque soit le mode de réplication.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] RULE`
Ces formes configurent le déclenchement des règles de ré-écriture appartenant à la table. Une règle désactivée est toujours

connue par le système mais non appliquée lors de la ré-écriture de la requête. La sémantique est identique celles des triggers activés/désactivés. Cette configuration est ignorée pour les règles `ON SELECT` qui sont toujours appliqués pour conserver le bon fonctionnement des vues même si la session actuelle n'est pas dans le rôle de réplication par défaut.

`CLUSTER`
Sélectionne l'index par défaut pour les prochaines opérations CLUSTER(7). La table n'est pas réorganisée.

`SET WITHOUT CLUSTER`
Supprime de la table la spécification d'index CLUSTER(7) la plus récemment utilisée. Cela agit sur les opérations de réorganisation suivantes qui ne spécifient pas d'index.

`SET WITH OIDS`
Cette forme ajoute une colonne système `oid` à la table (voir Section 5.4, « Colonnes système »). Elle ne fait rien si la table a déjà des OID.

Ce n'est pas équivalent à `ADD COLUMN oid oid`. Cette dernière ajouterait une colonne normale nommée `oid`, qui n'est pas une colonne système.

`SET WITHOUT OIDS`
Supprime la colonne système `oid` de la table. Cela est strictement équivalent à `DROP COLUMN oid RESTRICT`, à ceci près qu'aucun avertissement n'est émis si la colonne `oid` n'existe plus.

`SET ( paramètre_stockage = valeur [, ... ] )`
Cette forme modifie un ou plusieurs paramètres de stockage pour la table. Voir la section intitulée « Paramètres de stockage » pour les détails sur les paramètres disponibles. Le contenu de la table ne sera pas modifié immédiatement par cette commande ; en fonction du paramètre, il pourra s'avérer nécessaire de réécrire la table pour obtenir les effets désirés. Ceci peut se faire avec CLUSTER(7) ou une des formes d'**ALTER TABLE** qui force une réécriture de la table.

Note

Bien que **CREATE TABLE** autorise la spécification de `OIDS` avec la syntaxe `WITH (paramètre_stockage)`, **ALTER TABLE** ne traite pas les `OIDS` comme un paramètre de stockage. À la place, utiliser les formes `SET WITH OIDS` et `SET WITHOUT OIDS` pour changer le statut des OID sur la table.

`RESET ( paramètre_stockage [, ... ] )`
Cette forme réinitialise un ou plusieurs paramètres de stockage à leur valeurs par défaut. Comme avec `SET`, une réécriture de table pourrait être nécessaire pour mettre à jour entièrement la table.

`INHERIT table_parent`
Cette forme ajoute la table cible comme nouvel enfant à la table parent indiquée. En conséquence, les requêtes concernant le parent ajouteront les enregistrements de la table cible. Pour être ajoutée en tant qu'enfant, la table cible doit déjà contenir toutes les colonnes de la table parent (elle peut avoir des colonnes supplémentaires). Les colonnes doivent avoir des types qui correspondent, et s'il y a des contraintes `NOT NULL` défini pour le parent, alors elles doivent aussi avoir les contraintes `NOT NULL` pour l'enfant.

Il doit y avoir aussi une correspondance des contraintes de tables enfants pour toutes les contraintes `CHECK`. Actuellement, les contraintes `UNIQUE`, `PRIMARY KEY` et `FOREIGN KEY` ne sont pas prises en compte mais ceci pourrait changer dans le futur.

`NO INHERIT table_parent`
Cette forme supprime une table cible de la liste des enfants de la table parent indiquée. Les requêtes envers la table parent n'incluront plus les enregistrements de la table cible.

`OWNER`
Change le propriétaire d'une table, d'une séquence ou d'une vue. Le nouveau propriétaire est celui passé en paramètre.

`SET TABLESPACE`
Remplace le tablespace de la table par le tablespace spécifié et déplace le(s) fichier(s) de données associé(s) à la table vers le nouveau tablespace. Les index de la table, s'il y en a, ne sont pas déplacés ; mais ils peuvent l'être séparément à l'aide de commandes `SET TABLESPACE` supplémentaires. Voir aussi CREATE TABLESPACE(7).

`RENAME`
Change le nom d'une table (d'un index, d'une séquence ou d'une vue) ou le nom d'une colonne individuelle de la table. Cela n'a aucun effet sur la donnée stockée.

`SET SCHEMA`
Déplace la table dans un autre schéma. Les index, les contraintes et les séquences utilisées dans les colonnes de table sont également déplacés.

Toutes les actions à l'exception de `RENAME` et `SET SCHEMA` peuvent être combinées dans une liste d'altérations à appliquer en parallèle. Par exemple, il est possible d'ajouter plusieurs colonnes et/ou de modifier le type de plusieurs colonnes en une seule commande. Ceci est particulièrement utile avec les grosses tables car une seule passe sur la table est alors nécessaire.

Il faut être propriétaire de la table pour utiliser **ALTER TABLE**. Pour modifier le schéma d'une table, le droit `CREATE` sur le nouveau schéma est requis. Pour ajouter la table en tant que nouvel enfant d'une table parent, vous devez aussi être propriétaire de la table parent. Pour modifier le propriétaire, il est nécessaire d'être un membre direct ou indirect du nouveau rôle et ce dernier doit avoir le droit `CREATE` sur le schéma de la table. (Ces restrictions assurent que la modification du propriétaire ne diffère en rien de ce qu'il est possible de faire par la suppression et le re-création de la table. Néanmoins, un superutilisateur peut modifier le propriétaire de n'importe quelle table.)

Paramètres

`nom`
: Le nom (éventuellement qualifié du nom du schéma) de la table à modifier. Si `ONLY` est indiqué avant le nom de la table, seule cette table est modifiée. Dans le cas contraire, la table et toutes ses tables filles (s'il y en a) sont modifiées. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.

`colonne`
: Le nom d'une colonne, existante ou nouvelle.

`nouvelle_colonne`
: Le nouveau nom d'une colonne existante.

`nouveau_nom`
: Le nouveau nom de la table.

`type`
: Le type de données de la nouvelle colonne, ou le nouveau type de données d'une colonne existante.

`contraintedetable`
: Une nouvelle contrainte de table pour la table.

`nomdecontrainte`
: Le nom d'une contrainte existante à supprimer.

`CASCADE`
: Les objets qui dépendent de la colonne ou de la contrainte supprimée sont automatiquement supprimés (par exemple, les vues référençant la colonne).

`RESTRICT`
: La colonne ou la contrainte n'est pas supprimée si des objets en dépendent. C'est le comportement par défaut.

`nom_declencheur`
: Le nom d'un déclencheur isolé à désactiver ou activer.

`ALL`
: Désactiver ou activer tous les déclencheurs appartenant à la table. (Les droits de superutilisateur sont nécessaires si l'un des déclencheurs est un déclencheur interne pour la gestion d'une constrainte comme ceux utilisés pour implanter les contraintes de type clés étrangères ou les contraintes déferrables comme les contraintes uniques et d'exclusion.)

`USER`
: Désactiver ou activer tous les déclencheurs appartenant à la table sauf les déclencheurs systèmes permettant de gérer en interne certaines contraintes, comme celles utilisées pour implanter les contraintes de type clés étrangères ou les contraintes déferrables comme les contraintes uniques et d'exclusion.)

`nomindex`
: Le nom de l'index sur lequel la table doit être réorganisée.

`paramètre_stockage`
: Le nom d'un paramètre de stockage de la table.

`valeur`
: La nouvelle valeur d'un paramètre de stockage de la table. Cela peut être un nombre ou un mot suivant le paramètre.

`table_parent`
: Une table parent à associer ou dissocier de cette table.

`nouveau_propriétaire`
: Le nom du nouveau propriétaire de la table.

nouvel_espacelogique
Le nom du tablespace où déplacer la table.

nouveau_schema
Le nom du schéma où déplacer la table.

Notes

Le mot clé `COLUMN` n'est pas nécessaire. Il peut être omis.

Quand une colonne est ajoutée avec `ADD COLUMN`, toutes les lignes existantes de cette table sont initialisées avec la valeur par défaut de la colonne (NULL si aucune clause `DEFAULT` n'a été définie).

Ajouter une colonne avec une valeur par défaut différente de NULL ou modifier le type d'une colonne existante requiert que la table entière et les index soient réécrits. Cela peut prendre un temps considérable pour une grande table ; et cela demande temporairement le double d'espace disque. Ajouter ou supprimer une colonne système `oid` requiert de la même façon de réécrire la table entière.

Ajouter une contrainte `CHECK` ou `NOT NULL` requiert de parcourir la table pour vérifier que les lignes existantes respectent cette contrainte.

La raison principale de la possibilité de spécifier des changements multiples à l'aide d'une seule commande **ALTER TABLE** est la combinaison en une seule passe sur la table de plusieurs parcours et réécritures.

La forme `DROP COLUMN` ne supprime pas physiquement la colonne, mais la rend simplement invisible aux opérations SQL. Par la suite, les ordres d'insertion et de mise à jour sur cette table stockent une valeur NULL pour la colonne. Ainsi, supprimer une colonne ne réduit pas immédiatement la taille de la table sur disque car l'espace occupé par la colonne n'est pas récupéré. Cet espace est récupéré au fur et à mesure des mises à jour des lignes de la table. (Ceci n'est pas vrai quand on supprime la colonne système `oid` ; ceci est fait avec une réécriture immédiate de la table.)

Le fait qu'`SET DATA TYPE` requiert la réécriture de toute la table est parfois un avantage car le processus de réécriture élimine tout espace mort dans la table. Par exemple, pour réclamer immédiatement la place occupée par une colonne supprimée, la façon la plus rapide est :

```
ALTER TABLE table ALTER COLUMN toutecolonne TYPE touttype;
```

où `toutecolonne` est une colonne conservée de la table et `touttype` est le type courant de la colonne. Sémantiquement, aucune modification n'est visible, mais la commande force la réécriture, ce qui supprime toute donnée devenue inutile.

L'option `USING` de `SET DATA TYPE` peut en fait utiliser une expression qui implique d'anciennes valeurs de la ligne ; c'est-à-dire qu'il peut être fait référence aussi bien aux autres colonnes qu'à celle en cours de conversion. Cela permet d'effectuer des conversions très générales à l'aide de la syntaxe `SET DATA TYPE`. À cause de cette flexibilité, l'expression `USING` n'est pas appliquée à la valeur par défaut de la colonne (s'il y en a une) : le résultat pourrait ne pas être une expression constante requise pour une valeur par défaut. Lorsqu'il n'existe pas de transtypage, implicite ou d'affectation, entre les deux types, `SET DATA TYPE` peut échouer à convertir la valeur par défaut alors même que la clause `USING` est spécifiée. Dans de ce cas, il convient de supprimer valeur par défaut avec `DROP DEFAULT`, d'exécuter `ALTER TYPE` et enfin d'utiliser `SET DEFAULT` pour ajouter une valeur par défaut appropriée. Des considérations similaires s'appliquent aux index et contraintes qui impliquent la colonne.

Si une table est héritée, il n'est pas possible d'ajouter, de renommer ou de modifier le type d'une colonne dans la table parent sans le faire aussi pour ses descendantes. De ce fait, la commande **ALTER TABLE ONLY** est rejetée. Cela assure que les colonnes des tables descendantes correspondent toujours à celles de la table parent.

Un appel récursif à `DROP COLUMN` supprime la colonne d'une table descendante si et seulement si cette table n'hérite pas cette colonne d'une autre table et que la colonne n'y a pas été définie indépendamment de tout héritage. Une suppression non récursive de colonne (**ALTER TABLE ONLY ... DROP COLUMN**) ne supprime jamais les colonnes descendantes ; elles sont marquées comme définies de manière indépendante, plutôt qu'héritées.

Les actions `TRIGGER`, `CLUSTER`, `OWNER`, et `TABLESPACE` ne sont jamais propagées aux tables descendantes ; c'est-à-dire qu'elles agissent comme si `ONLY` est spécifié. Seuls les ajouts de contraintes `CHECK` sont propagés, et c'est d'ailleurs obligatoire pour ces contraintes.

Tout changement sur une table du catalogue système est interdit.

Voir la commande CREATE TABLE(7) pour avoir une description plus complète des paramètres valides. Chapitre 5, Définition des données fournit de plus amples informations sur l'héritage.

Exemples

Ajouter une colonne de type varchar à une table :

```
ALTER TABLE distributeurs ADD COLUMN adresse varchar(30);
```

Supprimer une colonne de table :

```
ALTER TABLE distributeurs DROP COLUMN adresse RESTRICT;
```

Changer les types de deux colonnes en une seule opération :

```
ALTER TABLE distributeurs
    ALTER COLUMN adresse TYPE varchar(80),
    ALTER COLUMN nom TYPE varchar(100);
```

Convertir une colonne de type integer (entier) contenant une estampille temporelle UNIX en timestamp with time zone à l'aide d'une clause `USING` :

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

La même, quand la colonne a une expression par défaut qui ne sera pas convertie automatiquement vers le nouveau type de données :

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

Renommer une colonne existante :

```
ALTER TABLE distributeurs RENAME COLUMN adresse TO city;
```

Renommer une table existante :

```
ALTER TABLE distributeurs RENAME TO fournisseurs;
```

Ajouter une contrainte NOT NULL à une colonne :

```
ALTER TABLE distributeurs ALTER COLUMN rue SET NOT NULL;
```

Supprimer la contrainte NOT NULL d'une colonne :

```
ALTER TABLE distributeurs ALTER COLUMN rue DROP NOT NULL;
```

Ajouter une contrainte de vérification sur une table et tous ses enfants :

```
ALTER TABLE distributeurs ADD CONSTRAINT verif_cp CHECK (char_length(code_postal) = 5);
```

Supprimer une contrainte de vérification d'une table et de toutes ses tables filles :

```
ALTER TABLE distributeurs DROP CONSTRAINT verif_cp;
```

Pour enlever une contrainte check d'une table seule (pas sur ses enfants)

```
ALTER TABLE ONLY distributeurs DROP CONSTRAINT verif_cp;
```

(La contrainte check reste en place pour toutes les tables filles).

Ajouter une contrainte de clé étrangère à une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT dist_fk FOREIGN KEY (adresse) REFERENCES
adresses (adresse) MATCH FULL;
```

Ajouter une contrainte unique (multicolonnes) à une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT dist_id_codepostal_key UNIQUE (dist_id,
code_postal);
```

Ajouter une clé primaire nommée automatiquement à une table. Une table ne peut jamais avoir qu'une seule clé primaire.

```
ALTER TABLE distributeurs ADD PRIMARY KEY (dist_id);
```

Déplacer une table dans un tablespace différent :

```
ALTER TABLE distributeurs SET TABLESPACE tablespacerapide;
```

Déplacer une table dans un schéma différent :

```
ALTER TABLE mon_schema.distributeurs SET SCHEMA votre_schema;
```

Compatibilité

Les formes `ADD`, `DROP`, `SET DEFAULT` et `SET DATA TYPE` (sans `USING`) se conforment au standard SQL. Les autres formes sont des extensions PostgreSQL™, tout comme la possibilité de spécifier plusieurs manipulations en une seule commande **ALTER TABLE**.

ALTER TABLE DROP COLUMN peut être utilisé pour supprimer la seule colonne d'une table, laissant une table dépourvue de colonne. C'est une extension au SQL, qui n'autorise pas les tables sans colonne.

Nom

ALTER TABLESPACE — Modifier la définition d'un tablespace

Synopsis

```
ALTER TABLESPACE nom RENAME TO nouveau_nom
ALTER TABLESPACE nom OWNER TO nouveau_propriétaire
ALTER TABLESPACE name SET ( option_tablespace = valeur [, ... ] )
ALTER TABLESPACE name RESET ( option_tablespace [, ... ] )
```

Description

ALTER TABLESPACE modifie la définition d'un tablespace.

Seul le propriétaire du tablespace peut utiliser **ALTER TABLESPACE**. Pour modifier le propriétaire, il est nécessaire d'être un membre direct ou indirect du nouveau rôle propriétaire (les superutilisateurs ont automatiquement tous ces droits).

Paramètres

nom
: Le nom du tablespace.

nouveau_nom
: Le nouveau nom du tablespace. Le nouveau nom ne peut pas débuter par `pg_` car ces noms sont réservés aux espaces logiques système.

nouveau_propriétaire
: Le nouveau propriétaire du tablespace.

paramètre_tablespace
: Un paramètre du tablespace à configurer ou réinitialiser. Actuellement, les seuls paramètres disponibles sont `seq_page_cost` et `random_page_cost`. Configurer une valeur pour un tablespace particulier surchargera l'estimation habituelle du planificateur pour le coût de lecture de pages pour les tables du tablespace, comme indiqué par les paramètres de configuration du même nom (voir seq_page_cost, random_page_cost). Ceci peut être utile si un tablespace se trouve sur un disque qui est plus rapide ou plus lent du reste du système d'entrées/sorties.

Exemples

Renommer le tablespace `espace_index` en `raid_rapide` :

```
ALTER TABLESPACE espace_index RENAME TO raid_rapide;
```

Modifier le propriétaire du tablespace `espace_index` :

```
ALTER TABLESPACE espace_index OWNER TO mary;
```

Compatibilité

Il n'existe pas d'instruction **ALTER TABLESPACE** dans le standard SQL.

Voir aussi

CREATE TABLESPACE(7), DROP TABLESPACE(7)

Nom

ALTER TEXT SEARCH CONFIGURATION — modifier la définition d'une configuration de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH CONFIGURATION nom
    ADD MAPPING FOR type_jeton [, ... ] WITH nom_dictionnaire [, ... ]
ALTER TEXT SEARCH CONFIGURATION nom
    ALTER MAPPING FOR type_jeton [, ... ] WITH nom_dictionnaire [, ... ]
ALTER TEXT SEARCH CONFIGURATION nom
    ALTER MAPPING REPLACE vieux_dictionnaire WITH nouveau_dictionnaire
ALTER TEXT SEARCH CONFIGURATION nom
    ALTER MAPPING FOR type_jeton [, ... ] REPLACE vieux_dictionnaire WITH
nouveau_dictionnaire
ALTER TEXT SEARCH CONFIGURATION nom
    DROP MAPPING [ IF EXISTS ] FOR type_jeton [, ... ]
ALTER TEXT SEARCH CONFIGURATION nom RENAME TO nouveau_nom
ALTER TEXT SEARCH CONFIGURATION nom OWNER TO nouveau_proprietaire
```

Description

ALTER TEXT SEARCH CONFIGURATION modifie la définition d'une configuration de recherche plein texte. Vous pouvez modifier les correspondances à partir des types de jeton vers des dictionnaires, ou modifier le nom ou le propriétaire de la configuration.

Vous devez être le propriétaire de la configuration pour utiliser **ALTER TEXT SEARCH CONFIGURATION**.

Paramètres

nom
: Le nom de la configuration de recherche plein texte (pouvant être qualifié du schéma).

type_jeton
: Le nom d'un type de jeton qui est émis par l'analyseur de configuration.

nom_dictionnaire
: Le nom d'un dictionnaire de recherche plein texte à consulter pour le type de jeton spécifié. Si plusieurs dictionnaires sont listés, ils sont consultés dans l'ordre d'apparence.

ancien_dictionnaire
: Le nom d'un dictionnaire de recherche plein texte à remplacer dans la correspondance.

nouveau_dictionnaire
: Le nom d'un dictionnaire de recherche plein texte à substituer à *ancien_dictionnaire*.

nouveau_nom
: Le nouveau nom de la configuration de recherche plein texte.

newowner
: Le nouveau propriétaire de la configuration de recherche plein texte.

La forme `ADD MAPPING FOR` installe une liste de dictionnaires à consulter pour les types de jeton indiqués ; il y a une erreur s'il y a déjà une correspondance pour un des types de jeton. La forme `ALTER MAPPING FOR` fait de même mais en commençant par supprimer toute correspondance existante avec ces types de jeton. Les formes `ALTER MAPPING REPLACE` substituent *nouveau_dictionnaire* par *ancien_dictionnaire* partout où ce dernier apparaît. Ceci se fait pour les seuls types de jeton indiqués quand `FOR` apparaît ou pour toutes les correspondances de la configuration dans le cas contraire. La forme `DROP MAPPING` supprime tous les dictionnaire pour les types de jeton spécifiés, faisant en sorte que les jetons de ces types soient ignorés par la configuration de recherche plein texte. Il y a une erreur s'il n'y a pas de correspondance pour les types de jeton sauf si `IF EXISTS` a été ajouté.

Exemples

L'exemple suivant remplace le dictionnaire `english` avec le dictionnaire `swedish` partout où `english` est utilisé dans `ma_config`.

```
ALTER TEXT SEARCH CONFIGURATION ma_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Compatibilité

Il n'existe pas d'instructions **ALTER TEXT SEARCH CONFIGURATION** dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH CONFIGURATION(7), DROP TEXT SEARCH CONFIGURATION(7)

Nom

ALTER TEXT SEARCH DICTIONARY — modifier la définition d'un dictionnaire de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH DICTIONARY nom (
    option [ = valeur ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY nom RENAME TO nouveau_nom
ALTER TEXT SEARCH DICTIONARY nom OWNER TO nouveau_proprietaire
```

Description

ALTER TEXT SEARCH DICTIONARY modifie la définition d'un dictionnaire de recherche plein texte. Vous pouvez modifier les options spécifiques au modèle d'un dictionnaire. Vous pouvez aussi modifier le nom du dictionnaire et son propriétaire.

Vous devez être superutilisateur pour utiliser **ALTER TEXT SEARCH DICTIONARY**.

Paramètres

nom
: Le nom du dictionnaire de recherche plein texte (pouvant être qualifié du schéma).

option
: Le nom d'une option, spécifique au modèle, à configurer pour ce dictionnaire.

valeur
: La nouvelle valeur à utiliser pour une option spécifique au modèle. Si le signe égale et la valeur sont omises, alors toute valeur précédente de cette option est supprimée du dictionnaire, permettant ainsi à l'utilisation de la valeur par défaut.

nouveau_nom
: Le nouveau nom du dictionnaire de recherche plein texte.

nouveau_proprietaire
: Le nouveau propriétaire du dictionnaire de recherche plein texte.

Les options spécifiques au modèle peuvent apparaître dans n'importe quel ordre.

Exemples

La commande exemple suivant modifie la liste des mots d'arrêt par un dictionnaire basé sur Snowball. Les autres paramètres restent inchangés.

```
ALTER TEXT SEARCH DICTIONARY mon_dico ( StopWords = nouveaurusse );
```

La commande exemple suivante modifie la langue par le hollandais et supprime complètement l'option des mots d'arrêt.

```
ALTER TEXT SEARCH DICTIONARY mon_dico ( language = dutch, StopWords );
```

La commande exemple suivante « met à jour » la définition du dictionnaire sans rien modifier.

```
ALTER TEXT SEARCH DICTIONARY mon_dico ( dummy );
```

(Ceci fonctionne parce que le code de suppression de l'option ne se plaint pas s'il n'y a pas d'options.) Cette astuce est utile lors de la modification des fichiers de configuration pour le dictionnaire : la commande **ALTER** forcera les sessions existantes à relire les fichiers de configuration, ce qu'elles ne feraient jamais si elles les avaient déjà lus.

Compatibilité

Il n'existe pas d'instruction **ALTER TEXT SEARCH DICTIONARY** dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH DICTIONARY(7), DROP TEXT SEARCH DICTIONARY(7)

Nom

ALTER TEXT SEARCH PARSER — modifier la définition d'un analyseur de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH PARSER nom RENAME TO nouveau_nom
```

Description

ALTER TEXT SEARCH PARSER modifie la définition d'un analyseur de recherche plein texte. Actuellement, la seule fonctionnalité supportée est la modification du nom de l'analyseur.

Vous devez être superutilisateur pour utiliser **ALTER TEXT SEARCH PARSER**.

Paramètres

nom
: Le nom de l'analyseur de recherche plein texte (pouvant être qualifié du schéma).

nouveau_nom
: Le nouveau nom de l'analyseur de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction **ALTER TEXT SEARCH PARSER** dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH PARSER(7), DROP TEXT SEARCH PARSER(7)

Nom

ALTER TEXT SEARCH TEMPLATE — modifier la définition d'un modèle de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH TEMPLATE nom RENAME TO nouveau_nom
```

Description

ALTER TEXT SEARCH TEMPLATE modifie la définition d'un modèle de recherche plein texte. Actuellement, la seule fonctionnalité supportée est la modification du nom du modèle.

Vous devez être superutilisateur pour utiliser **ALTER TEXT SEARCH TEMPLATE**.

Paramètres

nom
: Le nom du modèle de recherche plein texte (pouvant être qualifié du schéma).

nouveau_nom
: Le nouveau nom du modèle de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction **ALTER TEXT SEARCH TEMPLATE** dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH TEMPLATE(7), DROP TEXT SEARCH TEMPLATE(7)

Nom

ALTER TRIGGER — Modifier la définition d'un déclencheur

Synopsis

```
ALTER TRIGGER nom ON table RENAME TO nouveau_nom
```

Description

ALTER TRIGGER modifie les propriétés d'un déclencheur. La clause `RENAME` renomme le déclencheur sans en changer la définition.

Seul le propriétaire de la table sur laquelle le déclencheur agit peut modifier ses propriétés.

Paramètres

nom
: Le nom du déclencheur à modifier.

table
: La table sur laquelle le déclencheur agit.

nouveau_nom
: Le nouveau nom du déclencheur.

Notes

La possibilité d'activer ou de désactiver temporairement un déclencheur est offerte par ALTER TABLE(7), et non par **ALTER TRIGGER** qui ne permet pas d'agir sur tous les déclencheurs d'une table en une seule opération.

Exemples

Renommer un déclencheur :

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Compatibilité

ALTER TRIGGER est une extension PostgreSQL™ au standard SQL.

Voir aussi

ALTER TABLE(7)

Nom

ALTER TYPE — Modifier la définition d'un type

Synopsis

```
ALTER TYPE nom RENAME TO nouveau_nom
ALTER TYPE nom OWNER TO nouveau_propriétaire
ALTER TYPE nom SET SCHEMA nouveau_schéma
```

Description

ALTER TYPE modifie la définition d'un type existant.

Seul le propriétaire du type peut utiliser **ALTER TYPE**. Pour modifier le schéma d'un type, le droit `CREATE` sur le nouveau schéma est requis. Pour modifier le propriétaire, il faut être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit `CREATE` sur le schéma du type (ces restrictions assurent que la modification du propriétaire ne va pas au-delà de ce qui est possible par la suppression et la recréation du type ; toutefois, un superutilisateur peut modifier le propriétaire de n'importe quel type).

Paramètres

`nom`
: Le nom du type à modifier (éventuellement qualifié du nom du schéma).

`nouveau_nom`
: Le nouveau nom du type.

`nouveau_propriétaire`
: Le nom du nouveau propriétaire du type.

`nouveau_schema`
: Le nouveau schéma du type.

Exemples

Pour renommer un type de données :

```
ALTER TYPE courrier_electronique RENAME TO courriel;
```

Donner la propriété du type `courriel` à `joe` :

```
ALTER TYPE courriel OWNER TO joe;
```

Changer le schéma du type `courriel` en `clients` :

```
ALTER TYPE courriel SET SCHEMA clients;
```

Compatibilité

Il n'existe aucune instruction **ALTER TYPE** dans le standard SQL.

Nom

ALTER USER — Modifier un rôle de la base de données

Synopsis

```
ALTER USER nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'motdepasse'
    | VALID UNTIL 'dateheure'

ALTER USER nom RENAME TO nouveau_nom

ALTER USER nom SET parametre_configuration { TO | = } { valeur | DEFAULT }
ALTER USER nom SET parametre_configuration FROM CURRENT
ALTER USER nom RESET parametre_configuration
ALTER USER nom RESET ALL
```

Description

ALTER USER est désormais un alias de ALTER ROLE(7).

Compatibilité

La commande **ALTER USER** est une extension PostgreSQL™. En effet, le standard SQL laisse le choix de la définition des utilisateurs au SGBD.

Voir aussi

ALTER ROLE(7)

Nom

ALTER USER MAPPING — change la définition d'une correspondance d'utilisateurs (user mapping)

Synopsis

```
ALTER USER MAPPING FOR { nom_utilisateur | USER | CURRENT_USER | PUBLIC }
    SERVER nom_serveur
    OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )
```

Description

ALTER USER MAPPING change la définition d'une correspondance d'utilisateur (user mapping).

Le propriétaire d'un serveur distant peut aussi altérer les correspondances d'utilisateurs pour ce serveur pour tout utilisateur. Par ailleurs, un utilisateur peut modifier une correspondance d'utilisateur pour son propre nom d'utilisateur s'il a reçu le droit `USAGE` sur le serveur distant.

Paramètres

`nom_utilisateur`
: Nom d'utilisateur de la correspondance. `CURRENT_USER` et `USER` correspondent au nom de l'utilisateur courant. `PUBLIC` est utilisé pour correspondre à tous les noms d'utilisateurs présents et futurs du système.

`nom_serveur`
: Nom du serveur de la correspondance d'utilisateur.

`OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )`
: Modifie l'option pour la correspondance d'utilisateur. La nouvelle option écrase toute option précédemment spécifiée. `ADD`, `SET` et `DROP` spécifient l'action à exécuter. Si aucune action n'est spécifiée, l'action est `ADD`. Les noms d'options doivent être uniques ; les options sont aussi validées par le wrapper de données distantes du serveur.

Exemples

Modifier le mot de passe pour la correspondance d'utilisateur `bob`, et le serveur `foo` :

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'public');
```

Compatibilité

ALTER USER MAPPING est conforme à la norme ISO/IEC 9075-9 (SQL/MED). Il y a un problème de syntaxe subtil : le standard omet le mot clé `FOR`. Puisque `CREATE USER MAPPING` et `DROP USER MAPPING` utilisent tous les deux `FOR` à un endroit analogue et que DB2 d'IBM (l'autre implémentation majeure de SQL/MED) l'impose aussi pour `ALTER USER MAPPING`, PostgreSQL diverge du standard pour des raisons de cohérence et de compatibilité.

Voir aussi

CREATE USER MAPPING(7), DROP USER MAPPING(7)

Nom

ALTER VIEW — modifier la définition d'une vue

Synopsis

```
ALTER VIEW nom ALTER [ COLUMN ] colonne SET DEFAULT expression
ALTER VIEW nom ALTER [ COLUMN ] colonne DROP DEFAULT
ALTER VIEW nom OWNER TO nouveau_propriétaire
ALTER VIEW nom RENAME TO nouveau_nom
ALTER VIEW nom SET SCHEMA nouveau_schéma
```

Description

ALTER VIEW modifie différentes propriétés d'une vue. Si vous voulez modifier la requête définissant la vue, utilisez **CREATE OR REPLACE VIEW**.)

Vous devez être le propriétaire de la vue pour utiliser **ALTER VIEW**. Pour modifier le schéma d'une vue, vous devez aussi avoir le droit `CREATE` sur le nouveau schéma. Pour modifier le propriétaire, vous devez aussi être un membre direct ou indirect de nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma de la vue. Ces restrictions permettent de s'assurer que le changement de propriétaire ne fera pas plus que ce que vous pourriez faire en supprimant et en recréant la vue. Néanmoins, un superutilisateur peut changer le propriétaire de n'importe quelle vue.

Paramètres

nom
: Le nom de la vue (pouvant être qualifié du schéma).

`SET/DROP DEFAULT`
: Ces formes ajoutent ou suppriment la valeur par défaut pour une colonne. Une valeur par défaut associée à la colonne d'une vue est insérée avec des instructions **INSERT** sur la vue avant que la règle `ON INSERT` ne soit appliquée, si **INSERT** n'indique pas de valeur pour la colonne.

nouveau_propriétaire
: Nom utilisateur du nouveau propriétaire de la vue.

nouveau_nom
: Nouveau nom de la vue.

nouveau_schéma
: Nouveau schéma de la vue.

Notes

Pour des raisons historiques, **ALTER TABLE** peut aussi être utilisé avec des vues ; mais seules les variantes de **ALTER TABLE** qui sont acceptées avec les vues sont équivalentes à celles affichées ci-dessus.

Exemples

Pour renommer la vue `foo` en `bar` :

```
ALTER VIEW foo RENAME TO bar;
```

Compatibilité

ALTER VIEW est une extensions PostgreSQL™ du standard SQL.

Voir aussi

CREATE VIEW(7), DROP VIEW(7)

Nom

ANALYZE — Collecter les statistiques d'une base de données

Synopsis

```
ANALYZE [ VERBOSE ] [ table [ (colonne [, ...] ) ] ]
```

Description

ANALYZE collecte des statistiques sur le contenu des tables de la base de données et stocke les résultats dans le catalogue système pg_statistic. L'optimiseur de requêtes les utilise pour déterminer les plans d'exécution les plus efficaces.

Sans paramètre, **ANALYZE** examine chaque table de la base de données courante. Avec un paramètre, **ANALYZE** examine seulement la table concernée. Il est possible de donner une liste de noms de colonnes, auquel cas seules les statistiques concernant ces colonnes sont collectées.

Paramètres

`VERBOSE`
: L'affichage de messages de progression est activé.

`table`
: Le nom (éventuellement qualifié du nom du schéma) de la table à analyser. Par défaut, toutes les tables de la base de données courante sont analysées.

`column`
: Le nom d'une colonne à analyser. Par défaut, toutes les colonnes le sont.

Sorties

Quand `VERBOSE` est spécifié, **ANALYZE** affiche des messages de progression pour indiquer la table en cours de traitement. Diverses statistiques sur les tables sont aussi affichées.

Notes

Dans la configuration par défaut de PostgreSQL™, le démon autovacumm (voir Section 23.1.5, « Le démon auto-vacuum ») l'analyse automatique des tables quand elle est remplie de données sont la première fois, puis à chaque fois qu'elles sont modifiées via les opérations habituelles. Quand l'autovacuum est désactivé, il est intéressant de lancer **ANALYZE** périodiquement ou juste après avoir effectué de grosses modifications sur le contenu d'une table. Des statistiques à jour aident l'optimiseur à choisir le plan de requête le plus approprié et améliorent ainsi la vitesse du traitement des requêtes. Une stratégie habituelle consiste à lancer VACUUM(7) et **ANALYZE** une fois par jour, au moment où le serveur est le moins sollicité.

ANALYZE ne requiert qu'un verrou en lecture sur la table cible. Il peut donc être lancé en parallèle à d'autres activités sur la table.

Les statistiques récupérées par **ANALYZE** incluent habituellement une liste des quelques valeurs les plus communes dans chaque colonne et un histogramme affichant une distribution approximative des données dans chaque colonne. L'un ou les deux peuvent être omis si **ANALYZE** les juge inintéressants (par exemple, dans une colonne à clé unique, il n'y a pas de valeurs communes) ou si le type de données de la colonne ne supporte pas les opérateurs appropriés. Il y a plus d'informations sur les statistiques dans le Chapitre 23, Planifier les tâches de maintenance.

Pour les grosses tables, **ANALYZE** prend aléatoirement plusieurs lignes de la table, au hasard, plutôt que d'examiner chaque ligne. Ceci permet à des tables très larges d'être examinées rapidement. Néanmoins, les statistiques ne sont qu'approximatives et changent légèrement à chaque fois qu'**ANALYZE** est lancé, même si le contenu réel de la table n'a pas changé. Cela peut résulter en de petites modifications dans les coûts estimés par l'optimiseur affichés par EXPLAIN(7). Dans de rares situations, ce non-déterminisme entraîne le choix par l'optimiseur d'un plan de requête différent entre deux lancements d'**ANALYZE**. Afin d'éviter cela, le nombre de statistiques récupérées par **ANALYZE** peut être augmenté, comme cela est décrit ci-dessous.

L'étendue de l'analyse est contrôlée par l'ajustement de la variable de configuration default_statistics_target ou colonne par colonne en initialisant la cible des statistiques par colonne avec **ALTER TABLE ... ALTER COLUMN ... SET STATISTICS** (voir ALTER TABLE(7)). Cette valeur cible initialise le nombre maximum d'entrées dans la liste des valeurs les plus communes et le nombre maximum de points dans l'histogramme. La valeur cible par défaut est fixée à 100 mais elle peut être ajustée vers le haut ou vers le bas afin d'obtenir un bon compromis entre la précision des estimations de l'optimiseur, le temps pris par **ANA-**

LYZE et l'espace total occupé dans `pg_statistic`. En particulier, initialiser la cible des statistiques à zéro désactive la collecte de statistiques pour cette colonne. Cela peut s'avérer utile pour les colonnes qui ne sont jamais utilisées dans les clauses `WHERE`, `GROUP BY` ou `ORDER BY` des requêtes puisque l'optimiseur ne fait aucune utilisation des statistiques de ces colonnes.

La plus grande cible de statistiques parmi les colonnes en cours d'analyse détermine le nombre de lignes testées pour préparer les statistiques de la table. Augmenter cette cible implique une augmentation proportionnelle du temps et de l'espace nécessaires à l'exécution d'**ANALYZE**.

Une des valeurs estimées par **ANALYZE** est le nombre de valeurs distinctes qui apparaissent dans chaque colonne. Comme seul un sous-ensemble des lignes est examiné, cette estimation peut parfoir être assez inexacte, même avec la cible statistique la plus large possible. Si cette inexactitude amène de mauvais plans de requêtes, une valeur plus précise peut être déterminée manuellement, puis configurée avec **ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)** (voir ALTER TABLE(7) pour plus de détails).

Si la table en cours d'analyse a un ou plusieurs enfants, **ANALYZE** récupérera deux fois les statistiques : une fois sur les lignes de la table parent seulement et une deuxième fois sur les lignes de la table parent et de tous ses enfants. Néanmoins, le démon autovacuum ne considérera que les insertions et mises à jour sur la table parent pour décider du lancement automatique d'un ANALYZE. Si des lignes sont rarement insérées ou mises à jour dans cette table, les statistiques d'héritage ne seront à jour que si vous lancez manuellement un **ANALYZE**.

Compatibilité

Il n'existe pas d'instruction **ANALYZE** dans le standard SQL.

Voir aussi

VACUUM(7), vacuumdb(1), Section 18.4.3, « Report du VACUUM en fonction de son coût », Section 23.1.5, « Le démon autovacuum »

Nom

BEGIN — Débuter un bloc de transaction

Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ mode_transaction [, ...] ]

où mode_transaction peut être :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
```

Description

BEGIN initie un bloc de transaction, c'est-à-dire que toutes les instructions apparaissant après la commande **BEGIN** sont exécutées dans une seule transaction jusqu'à ce qu'un COMMIT(7) ou ROLLBACK(7) explicite soit exécuté. Par défaut (sans **BEGIN**), PostgreSQL™ exécute les transactions en mode « autocommit », c'est-à-dire que chaque instruction est exécutée dans sa propre transaction et une validation (commit) est traitée implicitement à la fin de l'instruction (si l'exécution a réussi, sinon une annulation est exécutée).

Les instructions sont exécutées plus rapidement dans un bloc de transaction parce que la séquence début/validation de transaction demande une activité significative du CPU et du disque. L'exécution de plusieurs instructions dans une transaction est aussi utile pour s'assurer d'une cohérence lors de la réalisation de certains modifications liées : les autres sessions ne voient pas les états intermédiaires tant que toutes les mises à jour ne sont pas réalisées.

Si le niveau d'isolation ou le mode lecture/écriture est spécifié, la nouvelle transaction possède ces caractéristiques, comme si SET TRANSACTION(7) était exécutée.

Paramètres

`WORK`, `TRANSACTION`
: Mots clés optionnels. Ils n'ont pas d'effet.

SET TRANSACTION(7) présente la signification des autres paramètres de cette instruction.

Notes

START TRANSACTION(7) a la même fonctionnalité que **BEGIN**.

COMMIT(7) ou ROLLBACK(7) sont utilisés pour terminer un bloc de transaction.

Lancer **BEGIN** en étant déjà dans un bloc de transaction provoque l'apparition d'un message d'avertissement, mais l'état de la transaction n'en est pas affecté. Pour intégrer des transactions à l'intérieur d'un bloc de transaction, les points de sauvegarde sont utilisés (voir SAVEPOINT(7)).

Pour des raisons de compatibilité descendante, les virgules entre chaque *`mode_transaction`* peuvent être omises.

Exemples

Commencer un bloc de transaction :

```
BEGIN;
```

Compatibilité

BEGIN, qui est une extension PostgreSQL™, est équivalent à la commande START TRANSACTION(7) du standard SQL. La page de référence de cette commande contient des informations de compatibilité supplémentaires.

Le mot clé `BEGIN` est utilisé dans un but différent en SQL embarqué. La sémantique de la transaction doit être étudiée avec précaution lors du portage d'applications.

Voir aussi

COMMIT(7), ROLLBACK(7), START TRANSACTION(7), SAVEPOINT(7)

Nom

CHECKPOINT — Forcer un point de vérification dans le journal des transactions

Synopsis

```
CHECKPOINT
```

Description

Les WAL (*Write-Ahead Log*, journaux de transactions) placent un point de vérification dans le journal des transactions à intervalle régulier. (Pour ajuster cet intervalle, voir les options de configuration à l'exécution checkpoint_segments et checkpoint_timeout.) La commande **CHECKPOINT** force un point de vérification immédiat, sans attendre le point de vérification planifié.

Un point de vérification est un point dans la séquence du journal des transactions pour lequel tous les fichiers de données ont été mis à jour pour refléter l'information des journaux. Tous les fichiers de données sont écrits sur le disque. Il convient de se référer à Chapitre 29, Fiabilité et journaux de transaction pour plus d'informations sur le système WAL.

S'il est exécuté durant une restauration, la commande **CHECKPOINT** forcera un point de redémarrage plutôt que l'écriture d'un nouveau point de vérification.

Seuls les superutilisateurs peuvent appeler **CHECKPOINT**. Cette commande ne doit pas être utilisée en fonctionnement normal.

Compatibilité

La commande **CHECKPOINT** est une extension PostgreSQL™.

Nom

CLOSE — Fermer un curseur

Synopsis

```
CLOSE { nom | ALL }
```

Description

CLOSE libère les ressources associées à un curseur ouvert. Une fois le curseur fermé, aucune opération n'est autorisée sur celui-ci. Un curseur doit être fermé lorsqu'il n'est plus nécessaire.

Tout curseur volatil ouvert (NDT : On parle en anglais de `non-holdable cursor`, soit un curseur qui ne perdure pas au-delà de la transaction qui l'a créé) est fermé implicitement lorsqu'une transaction est terminée avec **COMMIT** ou **ROLLBACK**. Un curseur persistant (NDT : `holdable cursor` en anglais, ou curseur qui perdure au-delà de la transaction initiale) est implicitement fermé si la transaction qui l'a créé est annulée via **ROLLBACK**. Si cette transaction est validée (avec succès), ce curseur reste ouvert jusqu'à ce qu'une commande **CLOSE** explicite soit lancée ou jusqu'à la déconnexion du client.

Paramètres

name
: Le nom du curseur ouvert à fermer.

`ALL`
: Ferme tous les curseurs ouverts.

Notes

PostgreSQL™ ne possède pas d'instruction explicite d'ouverture (**OPEN**) de curseur ; un curseur est considéré ouvert à sa déclaration. Un curseur est déclaré à l'aide de l'instruction DECLARE(7).

Vous pouvez voir tous les curseurs disponibles en exécutant une requête sur la vue système pg_cursors.

Si un curseur est fermé après un point de sauvegarde qui est annulé par la suite, la commande **CLOSE** n'est pas annulée ; autrement dit, le curseur reste fermé.

Exemples

Fermer le curseur `liahona` :

```
CLOSE liahona;
```

Compatibilité

CLOSE est totalement conforme au standard SQL. **CLOSE ALL** est une extension PostgreSQL™.

Voir aussi

DECLARE(7), FETCH(7), MOVE(7)

Nom

CLUSTER — Réorganiser une table en fonction d'un index

Synopsis

```
CLUSTER [VERBOSE] nom_table [ USING nom_index ]
CLUSTER [VERBOSE]
```

Description

CLUSTER réorganise (groupe) la table *nom_table* en fonction de l'index *nom_index*. L'index doit avoir été préalablement défini sur *nom_table*.

Une table reorganisée est physiquement réordonnée en fonction des informations de l'index. Ce regroupement est une opération ponctuelle : les actualisations ultérieures ne sont pas réorganisées. C'est-à-dire qu'aucune tentative n'est réalisée pour stocker les lignes nouvelles ou actualisées d'après l'ordre de l'index. (Une réorganisation périodique peut être obtenue en relançant la commande aussi souvent que souhaité. De plus, configurer le paramètre `FILLFACTOR` à moins de 100% peut aider à préserver l'ordre du cluster lors des mises à jour car les lignes mises à jour sont conservées dans la même page si suffisamment d'espace est disponible ici.)

Quand une table est réorganisée, PostgreSQL™ enregistre l'index utilisé à cet effet. La forme **CLUSTER *nom_table*** réorganise la table en utilisant le même index qu'auparavant. Vous pouvez aussi utiliser les formes `CLUSTER` ou `SET WITHOUT CLUSTER` de ALTER TABLE(7) pour initialiser l'index de façon à ce qu'il soit intégré aux prochaines opérations cluster our pour supprimer tout précédent paramètre.

CLUSTER, sans paramètre, réorganise toutes les tables de la base de données courante qui ont déjà été réorganisées et dont l'utilisateur est propriétaire, ou toutes les tables s'il s'agit d'un superutilisateur. Cette forme de **CLUSTER** ne peut pas être exécutée à l'intérieur d'une transaction.

Quand une table est en cours de réorganisation, un verrou `ACCESS EXCLUSIVE` est acquis. Cela empêche toute opération sur la table (à la fois en lecture et en écriture) pendant l'exécution de **CLUSTER**.

Paramètres

nom_table
: Le nom d'une table (éventuellement qualifié du nom du schéma).

nom_index
: Le nom d'un index.

`VERBOSE`
: Affiche la progression pour chaque table traitée.

Notes

Lorsque les lignes d'une table sont accédées aléatoirement et unitairement, l'ordre réel des données dans la table n'a que peu d'importance. Toutefois, si certaines données sont plus accédées que d'autres, et qu'un index les regroupe, l'utilisation de **CLUSTER** peut s'avérer bénéfique. Si une requête porte sur un ensemble de valeurs indexées ou sur une seule valeur pour laquelle plusieurs lignes de la table correspondent, **CLUSTER** est utile. En effet, lorsque l'index identifie la page de la table pour la première ligne correspondante, toutes les autres lignes correspondantes sont déjà probablement sur la même page de table, ce qui diminue les accès disque et accélère la requête.

Lors de l'opération de réorganisation, une copie temporaire de la table est créée qui contient les données de la table dans l'ordre de l'index. Des copies temporaires de chaque index de la table sont également créées. De ce fait, il est nécessaire de disposer d'un espace libre sur le disque au moins égal à la somme de la taille de la table et celles des index.

Puisque **CLUSTER** enregistre les informations de réorganisation, il est possible de réorganiser manuellement les tables souhaitées la première fois et de planifier une réorganisation, à la manière de **VACUUM**, pour que les tables soient régulièrement réorganisées.

Puisque le planificateur enregistre les statistiques d'ordonnancement des tables, il est conseillé de lancer ANALYZE(7) sur la table nouvellement réorganisée. Dans le cas contraire, les plans de requêtes peuvent être mal choisis par le planificateur.

Il existe une autre façon de réorganiser les données. La commande **CLUSTER** réorganise la table originale en la parcourant en

fonction de l'ordre de l'index indiqué ; cela peut être lent pour les tables volumineuses parce que les lignes sont extraites de la table dans l'ordre de l'index et, si la table n'est pas ordonnée, les entrées sont disséminées sur des pages aléatoires. Une page disque est alors lue pour chaque ligne déplacée. (PostgreSQL™ dispose d'un cache mais une grande table n'y tient généralement pas dans sa totalité.) L'autre moyen de réorganiser une table est alors d'utiliser :

```
CREATE TABLE nouvelletable AS
    SELECT * FROM table ORDER BY listecolonnes;
```

qui utilise le code de tri de PostgreSQL™ pour aboutir à l'ordre désiré ; pour des données non triées, cela est généralement bien plus rapide qu'un parcours d'index. L'ancienne table peut alors être supprimée et `nouvelletable` renommée en `table` à l'aide de **ALTER TABLE ... RENAME**. Il ne reste plus qu'à recréer les index de la table. Le gros inconvénient de cette approche est qu'elle ne préserve pas les OID, les contraintes, les relations de clés étrangères, les droits et autres propriétés de la table -- tous ces éléments doivent être recréés manuellement. Un autre inconvénient est la nécessité d'un fichier temporaire de tri de la même taille que la table elle-même. Le pic d'utilisation du disque est alors d'environ trois fois la taille de la table au lieu de deux fois.

Exemples

Réorganiser la table `employes` sur la base de son index `employes_ind` :

```
CLUSTER employes ON employes_ind;
```

Réorganiser la relation `employes` en utilisant le même index que précédemment :

```
CLUSTER employes;
```

Réorganiser toutes les tables de la base de données qui ont déjà été préalablement réorganisées :

```
CLUSTER;
```

Compatibilité

Il n'existe pas d'instruction **CLUSTER** dans le standard SQL.

La syntaxe

```
CLUSTER nom_index ON nom_table
```

est aussi supportée pour la compatibilité avec les versions de PostgreSQL™ antérieures à la 8.3.

Voir aussi

clusterdb(1)

Nom

COMMENT — Définir ou modifier le commentaire associé à un objet

Synopsis

```
COMMENT ON
{
  TABLE nom_objet |
  AGGREGATE nom_agrégat (type_agrégat [, ...] ) |
  CAST (type_source AS type_cible) |
  COLUMN nom_relation.nom_colonne |
  CONSTRAINT nom_contrainte ON nom_table |
  CONVERSION nom_objet |
  DATABASE nom_objet |
  DOMAIN nom_objet |
  FUNCTION nom_fonction ( [ [ modearg ] [ nomarg ] typearg [, ...] ] ) |
  INDEX nom_objet |
  LARGE OBJECT oid_large_objet |
  OPERATOR op (type_operande1, type_operande2) |
  OPERATOR CLASS nom_objet USING méthode_indexage |
  OPERATOR FAMILY nom_objet USING methode_index |
  ROLE nom_objet |
  RULE nom_rêgle ON nom_table |
  SCHEMA nom_objet |
  SEQUENCE nom_objet |
  TABLESPACE nom_objet |
  TEXT SEARCH CONFIGURATION nom_objet |
  TEXT SEARCH DICTIONARY nom_objet |
  TEXT SEARCH PARSER nom_objet |
  TEXT SEARCH TEMPLATE nom_objet |
  TRIGGER nom_déclencheur ON nom_table |
  TYPE nom_objet |
  VIEW nom_objet
} IS 'texte'
```

Description

COMMENT stocke un commentaire sur un objet de la base de données.

Pour modifier un commentaire, il suffit de lancer une nouvelle commande **COMMENT** portant sur le même objet. Une seule chaîne de commentaire est stockée pour chaque objet. Pour supprimer un commentaire, `NULL` est écrit à la place de la chaîne de texte. Les commentaires sont automatiquement supprimés avec l'objet.

Les commentaires sont visibles avec la famille de commandes **\d**, de psql. D'autres interfaces utilisateur de récupération des commentaires peuvent être construites au-dessus des fonctions intégrées qu'utilise psql, à savoir `obj_description`, `col_description` et `shobj_description`. (Voir Tableau 9.51, « Fonctions d'informations sur les commentaires ».)

Paramètres

`nom_objet`, *`nom_relation.nom_colonne`*, *`nom_agrégat`*, *`nom_contrainte`*, *`nom_fonction`*, *`op`*, *`nom_rêgle`*, *`nom_déclencheur`*

Le nom de l'objet à commenter. Les noms des tables, agrégats, domaines, fonctions, index, opérateurs, classes d'opérateurs, familles d'opérateur, séquences, objets de la recherche textuelle, types et vues peuvent être qualifiés du nom du schéma. Lors de l'ajout d'un commentaire sur une colonne, *`nom_relation`* doit faire référence à une table, à une vue ou à un type composite.

`type_agrégat`

Un type de données en entrée sur lequel l'agrégat opère. Pour référencer une fonction d'agrégat sans argument, utilisez `*` à la place de la liste des types de données en entrée.

`type_source`

Le nom du type de donnée source du transtypage.

`type_cible`

Le nom du type de données cible du transtypage.

modearg
Le mode d'un argument de la fonction : `IN`, `OUT`, `INOUT` ou `VARIADIC`. En cas d'omission, la valeur par défaut est `IN`. **COMMENT ON FUNCTION** ne tient pas compte, à l'heure actuelle, des arguments `OUT` car seuls ceux en entrée sont nécessaires pour déterminer l'identité de la fonction. Lister les arguments `IN`, `INOUT` et `VARIADIC` est ainsi suffisant.

nomarg
Le nom d'un argument de la fonction. **COMMENT ON FUNCTION** ne tient pas compte, à l'heure actuelle, des noms des arguments, seuls les types de données des arguments étant nécessaires pour déterminer l'identité de la fonction.

typearg
Les types de données des arguments de la fonction (éventuellement qualifiés du nom du schéma).

oid_objet_large
L'OID de l'objet large.

`PROCEDURAL`
Inutilisé.

texte
Le nouveau commentaire, rédigé sous la forme d'une chaîne littérale ; ou `NULL` pour supprimer le commentaire.

Notes

Il n'existe pas de mécanisme de sécurité pour les commentaires : tout utilisateur connecté à une base de données peut voir les commentaires de tous les objets de la base (mais seuls les superutilisateurs peuvent modifier les commentaires des objets qu'ils ne possèdent pas). Pour les objets partagés comme les bases, les rôles et les tablespaces, les commentaires sont stockées globalement et tout utilisateur connecté à une base peut voir tous les commentaires pour les objets partagés. Du coup, ne placez pas d'informations critiques pour la sécurité dans vos commentaires.

Exemples

Attacher un commentaire à la table `matable` :

```
COMMENT ON TABLE matable IS 'Ceci est ma table.';
```

Suppression du commentaire précédent :

```
COMMENT ON TABLE matable IS NULL;
```

Quelques exemples supplémentaires :

```
COMMENT ON AGGREGATE mon_agregat (double precision) IS 'Calcul d'une variance type';
COMMENT ON CAST (text AS int4) IS 'Transtypage de text en int4';
COMMENT ON COLUMN ma_table.ma_colonne IS 'Numéro employé';
COMMENT ON CONVERSION ma_conv IS 'Conversion vers UTF8';
COMMENT ON DATABASE ma_base IS 'Base de données de développement';
COMMENT ON DOMAIN mon_domaine IS 'Domaine des adresses de courriel';
COMMENT ON FUNCTION ma_fonction (timestamp) IS 'Retourner des chiffres romains';
COMMENT ON INDEX mon_index IS 'S'assurer de l'unicité de l'ID de l'employé';
COMMENT ON LANGUAGE plpython IS 'Support de Python pour les procedures stockées';
COMMENT ON LARGE OBJECT 346344 IS 'Document de planification';
COMMENT ON OPERATOR ^ (text, text) IS 'L\'intersection de deux textes';
COMMENT ON OPERATOR - (NONE, text) IS 'Opérateur de préfixe sur un texte';
COMMENT ON OPERATOR CLASS int4ops USING btree IS 'Opérateurs d'entiers sur quatre
octets pour les index btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'Tous les opérateurs entiers pour
les index btree';
COMMENT ON ROLE mon_role IS 'Groupe d'administration pour les tables finance';
COMMENT ON RULE ma_regle ON my_table IS 'Tracer les mises à jour des enregistrements
d\'employé';
COMMENT ON SCHEMA mon_schema IS 'Données du département';
COMMENT ON SEQUENCE ma_sequence IS 'Utilisé pour engendrer des clés primaires';
COMMENT ON TABLE mon_schema.ma_table IS 'Informations sur les employés';
COMMENT ON TABLESPACE mon_tablespace IS 'Tablespace pour les index';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Filtre des mots spéciaux';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Stemmer Snowball pour le suédois';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Divise le texte en mot';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Stemmer Snowball';
COMMENT ON TRIGGER mon_declencheur ON my_table IS 'Utilisé pour RI';
```

```
COMMENT ON TYPE complex IS 'Type de données pour les nombres complexes';
COMMENT ON VIEW ma_vue IS 'Vue des coûts départementaux';
```

Compatibilité

Il n'existe pas de commande **COMMENT** dans le standard SQL.

Nom

COMMIT — Valider la transaction en cours

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

Description

COMMIT valide la transaction en cours. Tout le monde peut désormais voir les modifications réalisées au cours de la transaction. De plus, leur persistance est garantie en cas d'arrêt brutal du serveur.

Paramètres

`WORK`, `TRANSACTION`
: Mots clés optionnels et sans effet.

Notes

ROLLBACK(7) est utilisé pour annuler une transaction.

Lancer **COMMIT** à l'extérieur d'une transaction n'a aucune conséquence mais provoque l'affichage d'un message d'avertissement.

Exemples

Valider la transaction courante et rendre toutes les modifications persistantes :

```
COMMIT;
```

Compatibilité

Le standard SQL ne spécifie que les deux formes `COMMIT` et `COMMIT WORK`. Pour le reste, cette commande est totalement conforme.

Voir aussi

BEGIN(7), ROLLBACK(7)

Nom

COMMIT PREPARED — Valider une transaction préalablement préparée en vue d'une validation en deux phases

Synopsis

```
COMMIT PREPARED id_transaction
```

Description

COMMIT PREPARED valide une transaction préparée.

Paramètres

id_transaction
: L'identifiant de la transaction à valider.

Notes

Seul l'utilisateur à l'origine de la transaction ou un superutilisateur peut valider une transaction préparée. Il n'est cependant pas nécessaire d'être dans la session qui a initié la transaction.

Cette commande ne peut pas être exécutée à l'intérieur d'un bloc de transaction. La transaction préparée est validée immédiatement.

Toutes les transactions préparées disponibles sont listées dans la vue système pg_prepared_xacts.

Exemples

Valider la transaction identifiée par `foobar` :

```
COMMIT PREPARED 'foobar';
```

Voir aussi

PREPARE TRANSACTION(7), ROLLBACK PREPARED(7)

Nom

COPY — Copier des données depuis/vers un fichier vers/depuis une table

Synopsis

```
COPY nom_table [ ( colonne [, ...] ) ]
    FROM { 'nom_fichier' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]

COPY { nom_table [ ( colonne [, ...] ) ] | ( requête ) }
    TO { 'nom_fichier' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

où option fait partie
de :

    FORMAT nom_format
    OIDS [ oids ]
    DELIMITER 'caractère_délimiteur'
    NULL 'chaîne_null'
    HEADER [ booléean ]
    QUOTE 'caractère_guillemet'
    ESCAPE 'caractère_échappement'
    FORCE_QUOTE { ( colonne [, ...] ) | * }
    FORCE_NOT_NULL ( colonne [, ...] )
```

Description

COPY transfère des données entre les tables de PostgreSQL™ et les fichiers du système de fichiers standard. **COPY TO** copie le contenu d'une table *vers* un fichier tandis que **COPY FROM** copie des données *depuis* un fichier vers une table (ajoutant les données à celles déjà dans la table). **COPY TO** peut aussi copier le résultat d'une requête **SELECT**.

Si une liste de colonnes est précisée, **COPY** ne copie que les données des colonnes spécifiées vers ou depuis le fichier. **COPY FROM** insère les valeurs par défaut des colonnes qui ne sont pas précisées dans la liste.

Si un nom de fichier est précisé, **COPY** lit ou écrit directement dans le fichier. Ce fichier doit être accessible par le serveur et son nom doit être spécifié du point de vue du serveur. Si `STDIN` ou `STDOUT` est indiqué, les données sont transmises au travers de la connexion entre le client et le serveur.

Paramètres

`nom_table`
: Le nom de la table (éventuellement qualifié du nom du schéma).

`colonne`
: Une liste optionnelle de colonnes à copier. Sans précision, toutes les colonnes de la table seront copiées.

`requête`
: Une commande SELECT(7) ou VALUES(7) dont les résultats doivent être copiés. Notez que les parenthèses sont requises autour de la requête.

`nom_fichier`
: Le chemin absolu du fichier en entrée ou en sortie. Les utilisateurs sous Windows peuvent avoir besoin d'utiliser une chaîne `E''` et de doubler tous les antislashs utilisés comme séparateurs de chemin.

`STDIN`
: Les données en entrée proviennent de l'application cliente.

`STDOUT`
: Les données en sortie vont sur l'application cliente.

`boolean`
: Spécifie si l'option sélectionnée doit être activée ou non. Vous pouvez écrire `TRUE`, `ON` ou `1` pour activer l'option, et `FALSE`, `OFF` ou `0` pour la désactiver. La valeur *`boolean`* peut aussi être omise, auquel cas la valeur `TRUE` est prise en compte.

`FORMAT`
Sélectionne le format des données pour la lecture ou l'écriture : `text`, `csv` (valeurs séparées par des virgules), ou `binary`. la valeur par défaut est `text`.

`OIDS`
Copie l'OID de chaque ligne. Une erreur est rapportée si `OIDS` est utilisé pour une table qui ne possède pas d'OID, ou dans le cas de la copie du résultat d'une *`requête`*.

`DELIMITER`
Spécifie le caractère qui sépare les colonnes sur chaque ligne du fichier. La valeur par défaut est une tabulation dans le format texte et une virgule dans le format `CSV`. Il doit être un seul caractère sur un seul octet. Cette option n'est pas autorisée lors de l'utilisation du format `binary`.

`NULL`
Spécifie la chaîne qui représente une valeur NULL. La valeur par défaut est `\N` (antislash-N) dans le format texte et une chaîne vide sans guillemets dans le format `CSV`. Vous pouvez préférer une chaîne vide même dans le format texte pour les cas où vous ne voulez pas distinguer les valeurs NULL des chaînes vides. Cette option n'est pas autorisée lors de l'utilisation du format `binary`.

Note

Lors de l'utilisation de **COPY FROM**, tout élément de données qui correspond à cette chaîne est stocké comme valeur NULL. Il est donc utile de s'assurer que c'est la même chaîne que celle précisée pour le **COPY TO** qui est utilisée.

`HEADER`
Le fichier contient une ligne d'en-tête avec les noms de chaque colonne. En sortie, la première ligne contient les noms de colonne de la table. En entrée, elle est ignorée. Cette option n'est autorisée que lors de l'utilisation du format `CSV`.

`QUOTE`
Spécifie le caractère guillemet à utiliser lorsqu'une valeur doit être entre guillemets. Par défaut, il s'agit du guillemet double. Cela doit de toute façon être un seul caractère sur un seul octet. Cette option n'est autorisée que lors de l'utilisation du format `CSV`.

`ESCAPE`
Spécifie le caractère qui doit apparaître avant un caractère de données qui correspond à la valeur `QUOTE`. La valeur par défaut est la même que la valeur `QUOTE` (du coup, le caractère guillemet est doublé s'il apparaît dans les données). Cela doit être un seul caractère codé en un seul octet. Cette option n'est autorisée que lors de l'utilisation du format `CSV`.

`FORCE_QUOTE`
Force l'utilisation des guillemets pour toutes les valeurs non `NULL` dans chaque colonne spécifiée. La sortie `NULL` n'est jamais entre guillemets. Si `*` est indiqué, les valeurs non `NULL` seront entre guillemets pour toutes les colonnes. Cette option est seulement autorisée avec **COPY TO** et seulement quand le format `CSV` est utilisé.

`FORCE_NOT_NULL`
Ne fait pas correspondre les valeurs des colonnes spécifiées avec la chaîne nulle. Dans le cas par défaut où la chaîne nulle est vide, cela signifie que les valeurs vides seront lues comme des chaînes de longueur nulle plutôt que comme des NULL, même si elles ne sont pas entre guillemets. Cette option est seulement autorisée avec **COPY FROM** et seulement quand le format `CSV` est utilisé.

Affichage

En cas de succès, une commande **COPY** renvoie une balise de la forme

```
COPY nombre
```

Le *`nombre`* correspond au nombre de lignes copiées.

Notes

COPY ne peut être utilisé qu'avec des tables réelles, pas avec des vues. Néanmoins, vous pouvez écrire `COPY (SELECT * FROM nom_vue) TO ...`.

COPY gère seulement la table nommée ; cette commande ne copie pas les données provenant ou vers des tables filles. Donc, par exemple, `COPY table TO` affiche les mêmes données que `SELECT * FROM ONLY table`. Mais `COPY (SELECT *`

`FROM table) TO ...` peut être utilisé pour sauvegarder toutes les données d'un héritage.

Le droit SELECT est requis sur la table dont les valeurs sont lues par **COPY TO** et le droit INSERT sur la table dont les valeurs sont insérées par **COPY FROM**. Il est suffisant d'avoir des droits sur les colonnes listées dans la commande.

Les fichiers nommés dans une commande **COPY** sont lus ou écrits directement par le serveur, non par l'application cliente. De ce fait, la machine hébergeant le serveur de bases de données doit les héberger ou pouvoir y accéder. L'utilisateur PostgreSQL™ (l'identifiant de l'utilisateur qui exécute le serveur), non le client, doit pouvoir y accéder et les lire ou les modifier. L'utilisation de **COPY** avec un fichier n'est autorisé qu'aux superutilisateurs de la base de données car **COPY** autorise la lecture et l'écriture de tout fichier accessible au serveur.

Il ne faut pas confondre **COPY** et l'instruction **\copy** de psql. **\copy** appelle **COPY FROM STDIN** ou **COPY TO STDOUT**, puis lit/stocke les données dans un fichier accessible au client psql. L'accès au fichier et les droits d'accès dépendent alors du client et non du serveur.

Il est recommandé que le chemin absolu du fichier utilisé dans **COPY** soit toujours précisé. Ceci est assuré par le serveur dans le cas d'un **COPY TO** mais, pour les **COPY FROM**, il est possible de lire un fichier spécifié par un chemin relatif. Le chemin est interprété relativement au répertoire de travail du processus serveur (habituellement dans le répertoire des données), pas par rapport au répertoire de travail du client.

COPY FROM appelle tous les déclencheurs et contraintes de vérification sur la table de destination, mais pas les règles.

L'entrée et la sortie de **COPY** sont sensibles à `datestyle`. Pour assurer la portabilité vers d'autres installations de PostgreSQL™ qui éventuellement utilisent des paramétrages `datestyle` différents de ceux par défaut, il est préférable de configurer `datestyle` en `ISO` avant d'utiliser **COPY TO**. Éviter d'exporter les données avec le `IntervalStyle` configuré à `sql_standard` est aussi une bonne idée car les valeurs négatives d'intervalles pourraient être mal interprétées par un serveur qui a une autre configuration pour `IntervalStyle`.

Les données en entrée sont interprétées suivant l'encodage actuel du client. Les données en sortie sont codées suivant l'encodage actuel du client. Ceci est valable même si les données ne passent pas par le client, c'est-à-dire si elles sont lues et écrites directement sur un fichier du serveur.

COPY stoppe l'opération à la première erreur. Si cela ne porte pas à conséquence dans le cas d'un **COPY TO**, il en va différemment dans le cas d'un **COPY FROM**. Dans ce cas, la table cible a déjà reçu les lignes précédentes. Ces lignes ne sont ni visibles, ni accessibles, mais occupent de l'espace disque. Il peut en résulter une perte importante d'espace disque si l'échec se produit lors d'une copie volumineuse. L'espace perdu peut alors être récupéré avec la commande **VACUUM**.

Les données en entrée sont interprétées suivant l'encodage actuel du client et les données en sortie sont encodées suivant l'encodage client même si les données ne passent pas par le client mais sont lues à partir d'un fichier ou écrites dans un fichier.

Formats de fichiers

Format texte

Quand le format `text` est utilisé, les données sont lues ou écrites dans un fichier texte, chaque ligne correspondant à une ligne de la table. Les colonnes sont séparées, dans une ligne, par le caractère de délimitation. Les valeurs des colonnes sont des chaînes, engendrées par la fonction de sortie ou utilisables par celle d'entrée, correspondant au type de données des attributs. La chaîne de spécification des valeurs NULL est utilisée en lieu et place des valeurs nulles. **COPY FROM** lève une erreur si une ligne du fichier ne contient pas le nombre de colonnes attendues. Si `OIDS` est précisé, l'OID est lu ou écrit dans la première colonne, avant celles des données utilisateur.

La fin des données peut être représentée par une ligne ne contenant qu'un antislash et un point (`\.`). Ce marqueur de fin de données n'est pas nécessaire lors de la lecture d'un fichier, la fin du fichier tenant ce rôle. Il n'est réellement nécessaire que lors d'une copie de données vers ou depuis une application cliente qui utilise un protocole client antérieur au 3.0.

Les caractères antislash (`\`) peuvent être utilisés dans les données de **COPY** pour échapper les caractères qui, sans cela, seraient considérés comme des délimiteurs de ligne ou de colonne. Les caractères suivants, en particulier, *doivent* être précédés d'un antislash s'ils apparaissent dans la valeur d'une colonne : l'antislash lui-même, le saut de ligne, le retour chariot et le délimiteur courant.

La chaîne NULL spécifiée est envoyée par **COPY TO** sans ajout d'antislash ; au contraire, **COPY FROM** teste l'entrée au regard de la chaîne NULL avant la suppression des antislash. Ainsi, une chaîne NULL telle que `\N` ne peut pas être confondue avec la valeur de donnée réelle `\N` (représentée dans ce cas par `\\N`).

Les séquences spéciales suivantes sont reconnues par **COPY FROM** :

Séquence	Représente
`\b`	Retour arrière (*backspace*) (ASCII 8)

Séquence	Représente
`\f`	Retour chariot (ASCII 12)
`\n`	Nouvelle ligne (ASCII 10)
`\r`	Retour chariot (ASCII 13)
`\t`	Tabulation (ASCII 9)
`\v`	Tabulation verticale (ASCII 11)
`\`*`chiffres`*	Antislash suivi d'un à trois chiffres en octal représente le caractère qui possède ce code numérique
`\x`*`digits`*	Antislash `x` suivi d'un ou deux chiffres hexadécimaux représente le caractère qui possède ce code numérique

Actuellement, **COPY TO** n'émet pas de séquence octale ou hexadécimale mais utilise les autres séquences listées ci-dessus pour les caractères de contrôle.

Tout autre caractère précédé d'un antislash se représente lui-même. Cependant, il faut faire attention à ne pas ajouter d'antislash qui ne soit pas absolument nécessaire afin d'éviter le risque d'obtenir accidentellement une correspondance avec le marqueur de fin de données (`\.`) ou la chaîne NULL (`\N` par défaut) ; ces chaînes sont reconnues avant tout traitement des antislashs.

Il est fortement recommandé que les applications qui engendrent des données **COPY** convertissent les données de nouvelle ligne et de retour chariot par les séquences respectives `\n` et `\r`. A l'heure actuelle, il est possible de représenter un retour chariot par un antislash et un retour chariot, et une nouvelle ligne par un antislash et une nouvelle ligne. Cependant, il n'est pas certain que ces représentations soient encore acceptées dans les prochaines versions. Celles-ci sont, de plus, extrêmement sensibles à la corruption si le fichier de **COPY** est transféré sur d'autres plateformes (d'un Unix vers un Windows ou inversement, par exemple).

COPY TO termine chaque ligne par une nouvelle ligne de style Unix (« `\n` »). Les serveurs fonctionnant sous Microsoft Windows engendrent un retour chariot/nouvelle ligne (« `\r\n` »), mais uniquement lorsque les données engendrées par **COPY** sont envoyées dans un fichier sur le serveur. Pour des raisons de cohérence entre les plateformes, **COPY TO STDOUT** envoie toujours « `\n` » quelque soit la plateforme du serveur. **COPY FROM** sait gérer les lignes terminant par une nouvelle ligne, un retour chariot ou un retour chariot suivi d'une nouvelle ligne. Afin de réduire les risques d'erreurs engendrées par des nouvelles lignes ou des retours chariot non précédés d'antislash, considéré de fait comme des données, **COPY FROM** émet un avertissement si les fins de lignes ne sont pas toutes identiques.

Format CSV

Ce format est utilisé pour importer et exporter des données au format de fichier `CSV` (acronyme de *Comma Separated Value*, littéralement valeurs séparées par des virgules). Ce format est utilisé par un grand nombre de programmes, tels les tableurs. À la place des règles d'échappement utilisées par le format texte standard de PostgreSQL™, il produit et reconnaît le mécanisme d'échappement habituel de CSV.

Les valeurs de chaque enregistrement sont séparées par le caractère `DELIMITER`. Si la valeur contient ce caractère, le caractère `QUOTE`, la chaîne `NULL`, un retour chariot ou un saut de ligne, la valeur complète est préfixée et suffixée par le caractère `QUOTE`. De plus, toute occurrence du caractère `QUOTE` ou du caractère `ESCAPE` est précédée du caractère d'échappement. `FORCE QUOTE` peut également être utilisé pour forcer les guillemets lors de l'affichage de valeur non-NULL dans des colonnes spécifiques.

Le format `CSV` n'a pas de façon standard de distinguer une valeur `NULL` d'une chaîne vide. La commande **COPY** de PostgreSQL™ gère cela avec les guillemets. Un `NULL` est affiché suivant le paramètre `NULL` et n'est pas entre guillemets, alors qu'une valeur non `NULL` correspondant au paramètre `NULL` est entre guillemets. Par exemple, avec la configuration par défaut, un `NULL` est écrit avec la chaîne vide sans guillemets alors qu'une chaîne vide est écrit avec des guillemets doubles (`""`). La lecture des valeurs suit des règles similaires. Vous pouvez utiliser `FORCE NOT NULL` pour empêcher les comparaisons d'entrée `NULL` pour des colonnes spécifiques.

L'antislash n'est pas un caractère spécial dans le format `CSV`. De ce fait, le marqueur de fin de données, `\.`, peut apparaître dans les donnée. Afin d'éviter toute mauvaise interprétation, une valeur `\.` qui apparaît seule sur une ligne est automatiquement placée entre guillemets en sortie. En entrée, si elle est entre guillemets, elle n'est pas interprétée comme un marqueur de fin de données. Lors du chargement d'un fichier qui ne contient qu'une colonne, dont les valeurs ne sont pas placées entre guillemets, créé par une autre application, qui contient une valeur `\.`, il est nécessaire de placer cette valeur entre guillemets.

Note

Dans le format `CSV`, tous les caractères sont significatifs. Une valeur entre guillemets entourée d'espaces ou de tout autre caractère différent de `DELIMITER` inclut ces caractères. Cela peut être source d'erreurs en cas d'import de données à partir d'un système qui complète les lignes `CSV` avec des espaces fines pour atteindre une longueur fixée.

Dans ce cas, il est nécessaire de pré-traiter le fichier `CSV` afin de supprimer les espaces de complètement avant d'insérer les données dans PostgreSQL™.

> **Note**
>
> Le format CSV sait reconnaître et produire des fichiers CSV dont les valeurs entre guillemets contiennent des retours chariot et des sauts de ligne. De ce fait, les fichiers ne contiennent pas strictement une ligne par ligne de table comme les fichiers du format texte.

> **Note**
>
> Beaucoup de programmes produisent des fichiers CSV étranges et parfois pervers ; le format de fichier est donc plus une convention qu'un standard. Il est alors possible de rencontrer des fichiers que ce mécanisme ne sait pas importer. De plus, **COPY** peut produire des fichiers inutilisables par d'autres programmes.

Format binaire

Le format `binary` fait que toutes les données sont stockées/lues au format binaire plutôt que texte. Il est un peu plus rapide que les formats texte et `CSV` mais un fichier au format binaire est moins portable suivant les architectures des machines et les versions de PostgreSQL™. De plus, le format binaire est très spécifique au type des données ; par exemple, un export de données binaires d'une colonne smallint ne pourra pas être importé dans une colonne integer, même si cela aurait fonctionné dans le format texte.

Le format de fichier `binary` consiste en un en-tête de fichier, zéro ou plusieurs lignes contenant les données de la ligne et un bas-de-page du fichier. Les en-têtes et les données sont dans l'ordre réseau des octets.

> **Note**
>
> Les versions de PostgreSQL™ antérieures à la 7.4 utilisaient un format de fichier binaire différent.

Entête du fichier

L'en-tête du fichier est constitutée de 15 octets de champs fixes, suivis par une aire d'extension de l'en-tête de longueur variable. Les champs fixes sont :

Signature
: séquence de 11 octets `PGCOPY\n\377\r\n\0` -- l'octet zéro est une partie obligatoire de la signature. La signature est conçue pour permettre une identification aisée des fichiers qui ont été déteriorés par un transfert non respectueux des huit bits. Cette signature est modifiée par les filtres de traduction de fin de ligne, la suppression des octets zéro, la suppression des bits de poids forts ou la modification de la parité.

Champs de commutateurs
: masque entier de 32 bits décrivant les aspects importants du format de fichier. Les bits sont numérotés de 0 (LSB, ou *Least Significant Bit*, bit de poids faible) à 31 (MSB, ou *Most Significant Bit*, bit de poids fort). Ce champ est stocké dans l'ordre réseau des octets (l'octet le plus significatif en premier), comme le sont tous les champs entier utilisés dans le format de fichier. Les bits 16 à 31 sont réservés aux problèmes critiques de format de fichier ; tout lecteur devrait annuler l'opération s'il trouve un bit inattendu dans cet ensemble. Les bits 0 à 15 sont réservés pour signaler les problèmes de compatibilité de formats ; un lecteur devrait simplement ignorer les bits inattendus dans cet ensemble. Actuellement, seul un bit est défini, le reste doit être à zéro :

  Bit 16
  : si 1, les OID sont inclus dans la donnée ; si 0, non

Longueur de l'aire d'extension de l'en-tête
: entier sur 32 bits, longueur en octets du reste de l'en-tête, octets de stockage de la longueur non-compris. À l'heure actuelle ce champ vaut zéro. La première ligne suit immédiatement. De futures modifications du format pourraient permettre la présence de données supplémentaires dans l'en-tête. Tout lecteur devrait ignorer silencieusement toute donnée de l'extension de l'en-tête qu'il ne sait pas traitée.

L'aire d'extension de l'en-tête est prévue pour contenir une séquence de morceaux s'auto-identifiant. Le champ de commutateurs n'a pas pour but d'indiquer aux lecteurs ce qui se trouve dans l'aire d'extension. La conception spécifique du contenu de l'extension de l'en-tête est pour une prochaine version.

Cette conception permet l'ajout d'en-têtes compatible (ajout de morceaux d'extension d'en-tête, ou initialisation des octets commutateurs de poids faible) et les modifications non compatibles (initialisation des octets commutateurs de poids fort pour signaler de telles modifications, et ajout des données de support dans l'aire d'extension si nécessaire).

Tuples

Chaque tuple débute par un compteur, entier codé sur 16 bits, représentant le nombre de champs du tuple. (Actuellement, tous les tuples d'une table ont le même compteur, mais il est probable que cela ne soit pas toujours le cas.) On trouve ensuite, répété pour chaque champ du tuple, un mot de 32 bits annonçant le nombre d'octets de stockage de la donnée qui suivent. (Ce mot n'inclut pas sa longueur propre et peut donc être nul.) -1, cas spécial, indique une valeur de champ NULL. Dans ce cas, aucun octet de valeur ne suit.

Il n'y a ni complètement d'alignement ni toute autre donnée supplémentaire entre les champs.

Actuellement, toutes les valeurs d'un fichier d'un format binaire sont supposées être dans un format binaire (code de format). Il est probable qu'une extension future ajoute un champ d'en-tête autorisant la spécification de codes de format par colonne.

La consultation du code source de PostgreSQL™, et en particulier les fonctions `*send` et `*recv` associées à chaque type de données de la colonne, permet de déterminer le format binaire approprié à la donnée réelle. Ces fonctions se situent dans le répertoire `src/backend/utils/adt/` des sources.

Lorsque les OID sont inclus dans le fichier, le champ OID suit immédiatement le compteur de champ. C'est un champ normal, à ceci près qu'il n'est pas inclus dans le compteur. En fait, il contient un mot de stockage de la longueur -- ceci permet de faciliter le passage d'OID sur quatre octets aux OID sur huit octets et permet d'afficher les OID comme étant NULL en cas de besoin.

Queue du fichier

La fin du fichier consiste en un entier sur 16 bits contenant -1. Cela permet de le distinguer aisément du compteur de champs d'un tuple.

Il est souhaitable que le lecteur rapporte une erreur si le mot compteur de champ ne vaut ni -1 ni le nombre attendu de colonnes. Cela assure une vérification supplémentaire d'une éventuelle désynchronisation d'avec les données.

Exemples

Copier une table vers le client en utilisant la barre verticale (`|`) comme délimiteur de champ :

```
COPY pays TO STDOUT (DELIMITER '|');
```

Copier des données d'un fichier vers la table `pays` :

```
COPY pays FROM '/usr1/proj/bray/sql/pays_donnees';
```

Pour copier dans un fichier les pays dont le nom commence par 'A' :

```
COPY (SELECT * FROM pays WHERE nom_pays LIKE 'A%') TO
'/usr1/proj/bray/sql/une_liste_de_pays.copy';
```

Exemple de données convenables pour une copie vers une table depuis `STDIN` :

```
AF      AFGHANISTAN
AL      ALBANIE
DZ      ALGERIE
ZM      ZAMBIE
ZW      ZIMBABWE
```

L'espace sur chaque ligne est en fait un caractère de tabulation.

Les mêmes données, extraites au format binaire. Les données sont affichées après filtrage au travers de l'outil Unix **od -c**. La table a trois colonnes ; la première est de type char(2), la deuxième de type text et la troisième de type integer. Toutes les lignes ont une valeur NULL sur la troisième colonne.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   E 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   E 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   E 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
```

```
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

Compatibilité

Il n'existe pas d'instruction **COPY** dans le standard SQL.

La syntaxe suivante était utilisée avant PostgreSQL™ 9.0 et est toujours supportée :

```
COPY nomtable [ ( colonne [, ...] ) ]
    FROM { 'nomfichier' | STDIN }
    [ [ WITH ]
        [ BINARY ]
        [ OIDS ]
        [ DELIMITER [ AS ] 'délimiteur' ]
        [ NULL [ AS ] 'chaîne NULL' ]
        [ CSV [ HEADER ]
              [ QUOTE [ AS ] 'guillemet' ]
              [ ESCAPE [ AS ] 'échappement' ]
              [ FORCE NOT NULL colonne [, ...] ] ] ]

COPY { nomtable [ ( colonne [, ...] ) ] | ( requête ) }
    TO { 'nomfichier' | STDOUT }
    [ [ WITH ]
        [ BINARY ]
        [ OIDS ]
        [ DELIMITER [ AS ] 'délimiteur' ]
        [ NULL [ AS ] 'chaîne NULL' ]
        [ CSV [ HEADER ]
              [ QUOTE [ AS ] 'guillemet' ]
              [ ESCAPE [ AS ] 'échappement' ]
              [ FORCE QUOTE colonne [, ...] | * } ] ] ]
```

Notez que, dans cette syntaxe, `BINARY` et `CSV` sont traités comme des mots-clés indépendants, pas comme des arguments à l'option `FORMAT`.

La syntaxe suivante, utilisée avant PostgreSQL™ version 7.3, est toujours supportée :

```
COPY [ BINARY ] nom_table [ WITH OIDS ]
    FROM { 'nom_fichier' | STDIN }
    [ [USING] DELIMITERS 'caractère_délimiteur' ]
    [ WITH NULL AS 'chaîne NULL' ]

COPY [ BINARY ] nom_table [ WITH OIDS ]
    TO { 'nom_fichier' | STDOUT }
    [ [USING] DELIMITERS 'caractère_délimiteur' ]
    [ WITH NULL AS 'chaîne NULL' ]
```

Nom

CREATE AGGREGATE — Définir une nouvelle fonction d'agrégat

Synopsis

```
CREATE AGGREGATE nom ( type_donnée_entrée [ , ... ] ) (
    SFUNC = sfonc,
    STYPE = type_donnée_état
    [ , FINALFUNC = ffonc ]
    [ , INITCOND = condition_initiale ]
    [ , SORTOP = operateur_tri ]
)

ou l'ancienne syntaxe

CREATE AGGREGATE nom (
    BASETYPE = type_base,
    SFUNC = sfonc,
    STYPE = type_donnée_état
    [ , FINALFUNC = ffonc ]
    [ , INITCOND = condition_initiale ]
    [ , SORTOP = operateur_tri ]
)
```

Description

CREATE AGGREGATE définit une nouvelle fonction d'agrégat. Quelques fonctions d'agrégat basiques et largement utilisées sont fournies dans la distribution standard ; elles sont documentées dans le Section 9.18, « Fonctions d'agrégat ». **CREATE AGGREGATE** est utilisée pour ajouter des fonctionnalités lors de la définition de nouveaux types ou si une fonction d'agrégat n'est pas fournie.

Si un nom de schéma est donné (par exemple, `CREATE AGGREGATE monschema.monagg ...`), alors la fonction d'agrégat est créée dans le schéma précisé. Sinon, elle est créée dans le schéma courant.

Une fonction d'agrégat est identifiée par son nom et son (ou ses) types de données en entrée. Deux agrégats dans le même schéma peuvent avoir le même nom s'ils opèrent sur des types différents en entrée. Le nom et le(s) type(s) de données en entrée d'un agrégat doivent aussi être distincts du nom et du type de données de toutes les fonctions ordinaires du même schéma.

Une fonction d'agrégat est réalisée à partir d'une ou deux fonctions ordinaires : une fonction de transition d'état *sfonc*, et une fonction de traitement final optionnelle *ffonc*. Elles sont utilisées ainsi :

```
sfonc( état-interne, nouvelle-valeur-données ) ---> prochain-état-interne
ffonc( état-interne ) ---> valeur-agrégat
```

PostgreSQL™ crée une variable temporaire de type *stype* pour contenir l'état interne courant de l'agrégat. À chaque ligne en entrée, la valeur de l'argument de l'agrégat est calculéeet la fonction de transition d'état est appelé avec la valeur d'état courante et la valeur du nouvel argument pour calculer une nouvelle valeur d'état interne. Une fois que toutes les lignes sont traitées, la fonction finale est appelée une seule fois pour calculer la valeur de retour de l'agrégat. S'il n'existe pas de fonction finale, alors la valeur d'état final est retournée en l'état.

Une fonction d'agrégat peut fournir une condition initiale, c'est-à-dire une valeur initiale pour la valeur de l'état interne. Elle est spécifiée et stockée en base comme une valeur de type text mais doit être une représentation externe valide d'une constante du type de donnée de la valeur d'état. Si elle n'est pas fournie, la valeur d'état est initialement positionnée à NULL.

Si la fonction de transition d'état est déclarée « strict », alors elle ne peut pas être appelée avec des entrées NULL. Avec une telle fonction de transition, l'exécution d'agrégat se comporte comme suit. Les lignes avec une valeur NULL en entrée sont ignorées (la fonction n'est pas appelé et la valeur de l'état précédent est conservé). Si la valeur de l'état initial est NULL, alors, à la première ligne sans valeur NULL, la première valeur de l'argument remplace la valeur de l'état, et la fonction de transition est appelée pour les lignes suivantes avec toutes les valeurs non NULL en entrée. Cela est pratique pour implémenter des agrégats comme `max`. Ce comportement n'est possible que quand *type_donnée_état* est identique au premier *type_donnée_entrée*. Lorsque ces types sont différents, une condition initiale non NULL doit être fournie, ou une fonction de transition non stricte utilisée.

Si la fonction de transition d'état n'est pas stricte, alors elle sera appelée sans condition pour chaque ligne en entrée et devra gérer les entrées NULL et les valeurs de transition NULL. Cela permet à l'auteur de l'agrégat d'avoir le contrôle complet sur la gestion des valeurs NULL par l'agrégat.

Si la fonction finale est déclarée « strict », alors elle ne sera pas appelée quand la valeur d'état finale est NULL ; à la place, un résultat NULL sera retourné automatiquement. C'est le comportement normal de fonctions strictes. Dans tous les cas, la fonction finale peut retourner une valeur NULL. Par exemple, la fonction finale pour `avg` renvoie NULL lorsqu'elle n'a aucune lignes en entrée.

Les agrégats qui se comportent comme `MIN` ou `MAX` peuvent parfois être optimisés en cherchant un index au lieu de parcourir toutes les lignes en entrée. Si un agrégat peut être optimisé, un *opérateur de tri* est spécifié. Dans ce cas, il est nécessaire que l'agrégat fournisse le premier élément dans l'ordre imposé par l'opérateur ; en d'autres mots :

```
SELECT agg(col) FROM tab;
```

doit être équivalent à :

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

On suppose également que l'agrégat ignore les entrées NULL et qu'il fournit un résultat NULL si et seulement s'il n'y a aucune entrée NULL. D'ordinaire, l'opérateur `<` d'un type de données est le bon opérateur de tri pour `MIN` et `>` celui pour `MAX`. L'optimisation ne prend jamais effet sauf si l'opérateur spécifié est membre de la stratégie « less than » (NdT : plus petit que) ou « greater than » (NdT : plus grand que) d'une classe d'opérateur pour un index B-tree.

Paramètres

`nom`
: Le nom de la fonction d'agrégat à créer (éventuellement qualifié du nom du schéma).

`type_donnée_entrée`
: Un type de donnée en entrée sur lequel opère la fonction d'agrégat. Pöur créer une fonction d'agrégat sans argument, placez `*` à la place de la liste des types de données en entrée. (la fonction `count(*)` en est un bon exemple.)

`type_base`
: Dans l'ancienne syntaxe de **CREATE AGGREGATE**, le type de données en entrée est spécifiée par un paramètre `type_base` plutôt que d'être écrit à la suite du nom de l'agrégat. Notez que cette syntaxe autorise seulement un paramètre en entrée. Pour définir une fonction d'agrégat sans argument, indiquez only one input parameter. To define a zero-argument aggregate function, `"ANY"` (et non pas `*`) pour le `type_base`.

`sfonc`
: Le nom de la fonction de transition de l'état à appeler pour chaque ligne en entrée. Pour une fonction d'agrégat avec *`N`* arguments, *`sfonc`* doit prendre *`N`*+1 arguments, le premier étant de type *`type_données_état`* et le reste devant correspondre aux types de données en entrée déclarés pour l'agrégat. La fonction doit renvoyer une valeur de type *`type_données_état`*. Cette fonction prend la valeur actuelle de l'état et les valeurs actuelles des données en entrée. Elle renvoit la prochaine valeur de l'état.

`type_donnée_état`
: Le type de donnée pour la valeur d'état de l'agrégat.

`ffonc`
: Le nom de la fonction finale à appeler pour traiter le résultat de l'agrégat une fois que toutes les lignes en entrée ont été parcourues. La fonction prend un seul argument de type *`type_donnée_état`*. Le type de retour de l'agrégat de la fonction est défini comme le type de retour de cette fonction. Si *`ffonc`* n'est pas spécifiée, alors la valeur d'état finale est utilisée comme résultat de l'agrégat et le type de retour est *`type_donnée_état`*.

`condition_initiale`
: La configuration initiale pour la valeur de l'état. Elle doit être une constante de type chaîne de caractères dans la forme acceptée par le type de données *`type_donnée_état`*. Si non spécifié, la valeur d'état est initialement positionnée à NULL.

`sort_operator`
: L'opérateur de tri associé pour un agrégat de type `MIN` ou `MAX`. C'est seulement le nom de l'opérateur (éventuellement qualifié du nom du schéma). L'opérateur est supposé avoir les mêmes types de données en entrée que l'agrégat (qui doit être un agrégat à un seul argument).

Les paramètres de **CREATE AGGREGATE** peuvent être écrits dans n'importe quel ordre, pas uniquement dans l'ordre illustré ci-dessus.

Exemples

Voir Section 35.10, « Agrégats utilisateur ».

Compatibilité

CREATE AGGREGATE est une extension PostgreSQL™. Le standard SQL ne fournit pas de fonctions d'agrégat utilisateur.

Voir aussi

ALTER AGGREGATE(7), DROP AGGREGATE(7)

Nom

CREATE CAST — Définir un transtypage

Synopsis

```
CREATE CAST (type_source AS type_cible)
    WITH FUNCTION nom_fonction (type_argument [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (type_source AS type_cible)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (type_source AS type_cible)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Description

CREATE CAST définit un transtypage. Un transtypage spécifie l'opération de conversion entre deux types de données. Par exemple :

```
SELECT CAST(42 AS float8);
```

convertit la constante entière 42 en float8 en appelant une fonction précédemment définie, `float8(int4)` dans le cas présent (si aucun transtypage convenable n'a été défini, la conversion échoue).

Deux types peuvent être *coercibles binairement*, ce qui signifie que le transtypage peut être fait « gratuitement » sans invoquer aucune fonction. Ceci impose que les valeurs correspondantes aient la même représentation interne. Par exemple, les types text et varchar sont coercibles binairement dans les deux sens. La coercibilité binaire n'est pas forcément une relation symétrique. Par exemple, le transtypage du type xml au type text peut être fait gratuitement dans l'implémentation actuelle, mais l'opération inverse nécessite une fonction qui fasse au moins une validation syntaxique. (Deux types qui sont coercibles binairement dans les deux sens sont aussi appelés binairement compatibles.)

Vous pouvez définir un transtypage comme *transtypage I/O* en utilisant la syntaxe `WITH INOUT`. Un transtype I/O est effectué en appelant la fonction de sortie du type de données source, et en passant le résultat à la fonction d'entrée du type de données cible.

Un transtypage peut être appelé explicitement. Par exemple : `CAST(x AS nomtype)` ou `x::nomtype`.

Si le transtypage est marqué `AS ASSIGNMENT` (NDT : à l'affectation), alors son appel peut être implicite lors de l'affectation d'une valeur à une colonne du type de donnée cible. Par exemple, en supposant que `foo.f1` soit une colonne de type text :

```
INSERT INTO foo (f1) VALUES (42);
```

est autorisé si la conversion du type integer vers le type text est indiquée `AS ASSIGNMENT`. Dans le cas contraire, c'est interdit. Le terme de *transtypage d'affectation* est utilisé pour décrire ce type de conversion.

Si la conversion est marquée `AS IMPLICIT`, alors elle peut être appelée implicitement dans tout contexte, soit par une affectation soit en interne dans une expression (nous utilisons généralement le terme *conversion implicite* pour décrire ce type de conversion.) Par exemple, voici une requête :

```
SELECT 2 + 4.0;
```

L'analyseur marque au début les constantes comme étant de type integer et numeric respectivement. Il n'existe pas d'opérateur integer + numeric dans les catalogues systèmes mais il existe un opérateur numeric + numeric. La requête sera un succès si une conversion de integer vers numeric est disponible et marquée `AS IMPLICIT` -- ce qui est le cas. L'analyseur appliquera la conversion implicite et résoudra la requête comme si elle avait été écrite de cette façon :

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

Maintenant, les catalogues fournissent aussi une conversion de numeric vers integer. Si cette conversion était marquée `AS IMPLICIT` -- mais ce n'est pas le cas -- alors l'analyseur devra choisir entre l'interprétation ci-dessus et son alternative (la conversion de la constante numeric en un integer) et appliquer l'opérateur integer + integer. Comme il n'a aucune information qui lui permettrait de choisir le meilleur moyen, il abandonne et déclare la requête comme étant ambigüe. Le fait qu'une seule des

conversions est indiquée comme implicite est le moyen par lequel nous apprenons à l'analyseur de préférer la première solution (c'est-à-dire de transformer une expression numeric-and-integer en numeric) ; il n'y a pas d'autre moyen.

Il est conseillé d'être conservateur sur le marquage du caractère implicite des transtypages. Une surabondance de transtypages implicites peut conduire PostgreSQL™ à interpréter étrangement des commandes, voire à se retrouver dans l'incapacité totale de les résoudre parce que plusieurs interprétations s'avèrent envisageables. Une bonne règle est de ne réaliser des transtypages implicites que pour les transformations entre types de la même catégorie générale et qui préservent l'information. Par exemple, la conversion entre int2 et int4 peut être raisonnablement implicite mais celle entre float8 et int4 est probablement réservée à l'affectation. Les transtypages inter-catégories, tels que de text vers int4, sont préférablement exécutés dans le seul mode explicite.

> **Note**
>
> Il est parfois nécessaire, pour des raisons de convivialité ou de respect des standards, de fournir plusieurs transtypages implicites sur un ensemble de types de données. Ceux-ci peuvent alors entraîner des ambiguités qui ne peuvent être évitées, comme ci-dessus. L'analyseur possède pour ces cas une heuristique de secours s'appuyant sur les *catégories de types* et les *types préférés*, qui peut aider à fournir le comportement attendu dans ce genre de cas. Voir CREATE TYPE(7) pour plus de détails.

Pour créer un transtypage, il faut être propriétaire du type source ou destination. Seul le superutilisateur peut créer un transtypage binairement compatible (une erreur sur un tel transtypage peut aisément engendrer un arrêt brutal du serveur).

Paramètres

`typesource`
Le nom du type de donnée source du transtypage.

`typecible`
Le nom du type de donnée cible du transtypage.

`nom_fonction` (*`type_argument`* [, ...])
La fonction utilisée pour effectuer la conversion. Le nom de la fonction peut être qualifié du nom du schéma. Si ce n'est pas le cas, la fonction est recherchée dans le chemin des schémas. Le type de données résultant de la fonction doit correspondre au type cible du transtypage. Ses arguments sont explicités ci-dessous.

`WITHOUT FUNCTION`
Indication d'une compatibilité binaire entre le type source et le type cible pour qu'aucune fonction ne soit requise pour effectuer la conversion.

`WITH INOUT`
Inique que le transtypage est un transtypage I/O, effectué en appelant la fonction de sortie du type de données source, et en passant le résultat à la fonction d'entrée du type de données cible.

`AS ASSIGNMENT`
Lors d'une affectation, l'invocation du transtypage peut être implicite.

`AS IMPLICIT`
L'invocation du transtypage peut être implicite dans tout contexte.

Les fonctions de transtypage ont un à trois arguments. Le premier argument est du même type que le type source ou doit être compatible avec ce type. Le deuxième argument, si fourni, doit être de type integer. Il stocke le modificateur de type associé au type de destination, ou `-1` en l'absence de modificateur. Le troisième argument, si fourni, doit être de type boolean. Il vaut `true` si la conversion est explicite, `false` dans le cas contraire. Bizarrement, le standard SQL appelle des comportements différents pour les transtypages explicites et implicites dans certains cas. Ce paramètre est fourni pour les fonctions qui implémentent de tel transtypages. Il n'est pas recommandé de concevoir des types de données utilisateur entrant dans ce cas de figure.

Le type de retour d'une fonction de transtypage doit être identique ou coercible binairement avec le type cible du transtypage.

En général, un transtypage correspond à des type source et destination différents. Cependant, il est permis de déclarer un transtypage entre types source et destination identiques si la fonction de transtypage a plus d'un argument. Cette possibilité est utilisée pour représenter dans le catalogue système des fonctions de transtypage agissant sur la longueur d'un type. La fonction nommée est utilisée pour convertir la valeur d'un type à la valeur du modificateur de type fournie par le second argument.

Quand un transtypage concerne des types source et destination différents et que la fonction a plus d'un argument, le transtypage et la conversion de longeur du type destination sont faites en une seule etape. Quand une telle entrée n'est pas disponible, le transtypage vers un type qui utilise un modificateur de type implique deux étapes, une pour convertir les types de données et la seconde pour appliquer le modificateur.

Notes

DROP CAST(7) est utilisé pour supprimer les transtypages utilisateur.

Pour convertir les types dans les deux sens, il est obligatoire de déclarer explicitement les deux sens.

Il est n'est pas nécessaire habituellement de créer des conversions entre des types définis par l'utilisateur et des types de chaîne standards (text, varchar etchar(*n*), pas plus que pour des types définis par l'utilisateur définis comme entrant dans la catégorie des chaînes). PostgreSQL™ fournit un transtypage I/O automatique pour cela. Ce transtypage automatique vers des types chaînes est traité comme des transtypages d'affectation, alors que les transtypages automatiques à partir de types chaîne sont de type explicite seulement. Vous pouvez changer ce comportement en déclarant votre propre conversion pour remplacer une conversion automatique. La seule raison usuelle de le faire est de vouloir rendre l'appel de la conversion plus simple que le paramétrage standard (affectation seulement ou explicite seulement). Une autre raison envisageable est de vouloir que la conversion se comporte différement de la fonction I/O du type ; mais c'est suffisamment déroutant pour que vous y pensiez à deux fois avant de le faire. (Un petit nombre de types internes ont en fait des comportements différents pour les conversions, principalement à cause des besoins du standard SQL.)

Avant PostgreSQL™ 7.3, toute fonction qui portait le même nom qu'un type de données, retournait ce type de données et prenait un argument d'un autre type était automatiquement détectée comme une fonction de conversion. Cette convention a été abandonnée du fait de l'introduction des schémas et pour pouvoir représenter des conversions binairement compatibles dans les catalogues système. Les fonctions de conversion intégrées suivent toujours le même schéma de nommage mais elle doivent également être présentées comme fonctions de transtypage dans le catalogue système pg_cast.

Bien que cela ne soit pas requis, il est recommandé de suivre l'ancienne convention de nommage des fonctions de transtypage en fonction du type de données de destination. Beaucoup d'utilisateurs sont habitués à convertir des types de données à l'aide d'une notation de style fonction, c'est-à-dire *nom_type*(*x*). En fait, cette notation n'est ni plus ni moins qu'un appel à la fonction d'implantation du transtypage ; sa gestion n'est pas spécifique à un transtypage. Le non-respect de cette convention peut surprendre certains utilisateurs. Puisque PostgreSQL™ permet de surcharger un même nom de fonction avec différents types d'argument, il n'y a aucune difficulté à avoir plusieurs fonctions de conversion vers des types différents qui utilisent toutes le même nom de type destination.

Note

En fait, le paragraphe précédent est une sur-simplification : il existe deux cas pour lesquels une construction d'appel de fonction sera traitée comme une demande de conversion sans qu'il y ait correspondance avec une fonction réelle. Si un appel de fonction *nom*(*x*) ne correspond pas exactement à une fonction existante, mais que *nom* est le nom d'un type de données et que pg_cast fournit une conversion compatible binairement vers ce type à partir du type *x*, alors l'appel sera construit à partir de la conversion compatible binairement. Cette exception est faite pour que les conversions compatibles binairement puissent être appelées en utilisant la syntaxe fonctionnelle, même si la fonction manque. De ce fait, s'il n'y pas d'entrée dans pg_cast mais que la conversion serait à partir de ou vers un type chapîne, l'appel sera réalisé avec une conversion I/O. Cette exception autorise l'appel de conversion I/O en utilisant la syntaxe fonctionnelle.

Exemples

Création d'un transtypage d'affectation du type bigint vers le type int4 à l'aide de la fonction `int4(bigint)` :

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

(Ce transtypage est déjà prédéfini dans le système.)

Compatibilité

La commande **CREATE CAST** est conforme à SQL à ceci près que SQL ne mentionne pas les types binairement compatibles et les arguments supplémentaires pour les fonctions d'implantation. `AS IMPLICIT` est aussi une extension PostgreSQL™.

Voir aussi

CREATE FUNCTION(7), CREATE TYPE(7), DROP CAST(7)

Nom

CREATE CONSTRAINT TRIGGER — Définir un nouveau déclencheur sur contrainte

Synopsis

```
CREATE CONSTRAINT TRIGGER nom
    AFTER événement [ OR ... ]
    ON table
    [ FROM table_référencée ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
    FOR EACH ROW
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE nom_fonction ( arguments )
```

Description

CREATE CONSTRAINT TRIGGER crée un *trigger de contrainte*. C'est identique à un trigger standard sauf que le moment de l'exécution du trigger est ajustable en utilisant SET CONSTRAINTS(7). Les triggers de contraintes doivent être des triggers `AFTER ROW`. Ils peuvent être déclenchés soit à la fin de l'instruction l'ayant déclenché soit à la fin de la transaction le contenant ; dans ce dernier cas, ils sont dit déferrés (*deferred*). Le déclenchement d'un tigger déferré en attente peut voir son exécution forcée en utilisant **SET CONSTRAINTS**.

Paramètres

nom
: Le nom du trigger de contrainte. C'est aussi le nom à utiliser pour modifier le comportement du trigger en utilisant **SET CONSTRAINTS**. Le nom ne peut pas être qualifié du schéma -- le trigger hérite du schéma de sa table.

événement
: Une valeur parmi `INSERT`, `UPDATE` ou `DELETE` ; ceci indique l'événement qui sera la cause de l'exécution du déclencheur. Plusieurs événements peuvent être indiqués en utilisant le mot clé `OR`.

table
: Le nom (éventuellement qualifié du nom du schéma) de la table dans laquelle se déroulent les événements déclenchant.

table_référencée
: Le nom d'une autre table (parfois qualifié du nom du schéma) référencée par la contrainte. Cette option est utilisée pour les contraintes de type clé étrangère et n'est pas recommandée pour un usage général.

`DEFERRABLE`, `NOT DEFERRABLE`, `INITIALLY IMMEDIATE`, `INITIALLY DEFERRED`
: Le moment d'exécution par défaut du trigger. Voir la documentation de CREATE TABLE(7) pour les détails sur ces options de contraintes.

condition
: Une expression booléenne déterminant si la fonction trigger sera en fait exécutée. Ceci agit de la même façon que CREATE TRIGGER(7). Notez en paticulier que l'évaluation de la condition `WHEN` n'est pas déferrée, mais survient immédiatement après l'opération de mise à jour de la ligne. Si la condition n'évalue pas à `true`, alors le trigger n'est pas mis en queue pour une exécution déferrée.

nom_fonction
: La fonction à appeler quand le trigger est déclenché. Voir CREATE TRIGGER(7) pour les détails.

arguments
: Chaînes optionnelles d'argument à passer à la fonction trigger. Voir CREATE TRIGGER(7) pour plus de détails.

Compatibilité

CREATE CONSTRAINT TRIGGER est une extension PostgreSQL™ du standard SQL.

Voir aussi

CREATE TRIGGER(7), DROP TRIGGER(7), SET CONSTRAINTS(7)

Nom

CREATE CONVERSION — Définir une nouvelle conversion d'encodage

Synopsis

```
CREATE [ DEFAULT ] CONVERSION nom
    FOR codage_source TO codage_dest FROM nom_fonction
```

Description

CREATE CONVERSION définit une nouvelle conversion entre les encodages de caractères. De plus, les conversions marquées `DEFAULT` peuvent être utilisées pour automatiser une conversion d'encodage entre le client et le serveur. Pour cela, deux conversions, de l'encodage A vers l'encodage B *et* de l'encodage B vers l'encodage A, doivent être définies.

Pour créer une conversion, il est nécessaire de posséder les droits `EXECUTE` sur la fonction et `CREATE` sur le schéma de destination.

Paramètres

`DEFAULT`
: La clause `DEFAULT` indique une conversion par défaut entre l'encodage source et celui de destination. Il ne peut y avoir, dans un schéma, qu'une seule conversion par défaut pour un couple d'encodages.

`nom`
: Le nom de la conversion. Il peut être qualifié du nom du schéma. Dans la cas contraire, la conversion est définie dans le schéma courant. Le nom de la conversion est obligatoirement unique dans un schéma.

`codage_source`
: Le nom de l'encodage source.

`codage_dest`
: Le nom de l'encodage destination.

`nom_fonction`
: La fonction utilisée pour réaliser la conversion. Son nom peut être qualifié du nom du schéma. Dans le cas contraire, la fonction est recherchée dans le chemin.

  La fonction a la signature suivante :

```
conv_proc(
    integer,  -- ID encodage source
    integer,  -- ID encodage destination
    cstring,  -- chaîne source (chaîne C terminée par un caractère nul)
    internal, -- destination (chaîne C terminée par un caractère nul)
    integer   -- longueur de la chaîne source
) RETURNS void;
```

Notes

DROP CONVERSION est utilisé pour supprimer une conversion utilisateur.

Il se peut que les droits requis pour créer une conversion soient modifiées dans une version ultérieure.

Exemples

Création d'une conversion de l'encodage `UTF8` vers l'encodage `LATIN1` en utilisant `mafonc` :

```
CREATE CONVERSION maconv FOR 'UTF8' TO 'LATIN1' FROM mafonc;
```

Compatibilité

CREATE CONVERSION est une extension PostgreSQL™. Il n'existe pas d'instruction **CREATE CONVERSION** dans le standard SQL. Par contre, il existe une instruction **CREATE TRANSLATION** qui est très similaire dans son but et sa syntaxe.

Voir aussi

ALTER CONVERSION(7), CREATE FUNCTION(7), DROP CONVERSION(7)

Nom

CREATE DATABASE — Créer une nouvelle base de données

Synopsis

```
CREATE DATABASE nom
    [ [ WITH ] [ OWNER [=] nom_utilisateur ]
               [ TEMPLATE [=] modèle ]
               [ ENCODING [=] codage ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
               [ TABLESPACE [=] tablespace ]
               [ CONNECTION LIMIT [=] limite_connexion ] ]
```

Description

CREATE DATABASE crée une nouvelle base de données.

Pour créer une base de données, il faut être superutilisateur ou avoir le droit spécial `CREATEDB`. Voir à ce sujet CREATE USER(7).

Par défaut, la nouvelle base de données est créée en clonant la base système standard `template1`. Un modèle différent peut être utilisé en écrivant `TEMPLATE` *`nom`*. En particulier, la clause `TEMPLATE template0` permet de créer une base de données vierge qui ne contient que les objets standards pré-définis dans la version de PostgreSQL™ utilisée. C'est utile pour ne pas copier les objets locaux ajoutés à `template1`.

Paramètres

`nom`
: Le nom de la base de données à créer.

`nom_utilisateur`
: Le nom de l'utilisateur propriétaire de la nouvelle base de données ou `DEFAULT` pour l'option par défaut (c'est-à-dire le nom de l'utilisateur qui exécute la commande). Pour créer une base de données dont le propriétaire est un autre rôle, vous devez être un membre direct ou direct de ce rôle, ou être un superutilisateur.

`modèle`
: Le nom du modèle squelette de la nouvelle base de données ou `DEFAULT` pour le modèle par défaut (`template1`).

`codage`
: Le jeu de caractères de la nouvelle base de données. Peut-être une chaîne (par exemple `'SQL_ASCII'`), un nombre de jeu de caractères de type entier ou `DEFAULT` pour le jeu de caractères par défaut (en fait, celui de la base modèle). Les jeux de caractères supportés par le serveur PostgreSQL™ sont décrits dans Section 22.2.1, « Jeux de caractères supportés ». Voir ci-dessous pour des restrictions supplémentaires.

`lc_collate`
: L'ordre de tri (`LC_COLLATE`) à utiliser dans la nouvelle base. Ceci affecte l'odre de tri appliqué aux chaînes, par exemple dans des requêtes avec ORDER BY, ainsi que l'ordre utilisé dans les index sur les colonnes texte. Le comportement par défaut est de choisir l'ordre de tri de la base de données modèle. Voir ci-dessous pour les restrictions supplémentaires.

`lc_ctype`
: La classification du jeu de caractères (`LC_CTYPE`) à utiliser dans la nouvelle base. Ceci affecte la catégorisation des caractères, par exemple minuscule, majuscule et chiffre. Le comportement par défaut est d'utiliser la classification de la base de données modèle. Voir ci-dessous pour les restrictions supplémentaires.

`tablespace`
: Le nom du tablespace associé à la nouvelle base de données ou `DEFAULT` pour le tablespace de la base de données modèle. Ce tablespace est celui par défaut pour les objets créés dans cette base de données. Voir CREATE TABLESPACE(7) pour plus d'informations.

`limite_connexion`
: Le nombre de connexions concurrentes à la base de données. -1 (valeur par défaut) signifie qu'il n'y a pas de limite.

L'ordre des paramètres optionnels n'a aucune importance.

Notes

La commande **CREATE DATABASE** ne peut pas être exécutée à l'intérieur d'un bloc de transactions.

Les erreurs sur la ligne « ne peut initialiser le répertoire de la base de données » (« could not initialize database directory » dans la version originale) sont le plus souvent dues à des droits insuffisants sur le répertoire de données, à un disque plein ou à un autre problème relatif au système de fichiers.

L'instruction DROP DATABASE(7) est utilisée pour supprimer la base de données.

Le programme createdb(1) est un enrobage de cette commande fourni par commodité.

Bien qu'il soit possible de copier une base de données autre que `template1` en spécifiant son nom comme modèle, cela n'est pas (encore) prévu comme une fonctionnalité « **COPY DATABASE** » d'usage général. La limitation principale est qu'aucune autre session ne peut être connectée à la base modèle pendant sa copie. **CREATE DATABASE** échouera s'il y a une autre connexion au moment de son exécution ; sinon, les nouveaux connexions à la base modèle seront verrouillées jusqu'à la fin de la commande **CREATE DATABASE**. La Section 21.3, « Bases de données modèles » fournit plus d'informations à ce sujet.

L'encodage du jeu de caractère spécifié pour la nouvelle base de données doit être compatible avec les paramètre de locale (`LC_COLLATE` et `LC_CTYPE`). Si la locale est `C` (ou de la même façon `POSIX`), alors tous les encodages sont autorisés. Pour d'autres paramètres de locale, il n'y a qu'un encodage qui fonctionnera correctement. (Néanmoins, sur Windows, l'encodage UTF-8 peut être utilisée avec toute locale.) **CREATE DATABASE** autorisera les superutilisateurs à spécifier l'encodage `SQL_ASCII` quelque soit le paramètre locale mais ce choix devient obsolète et peut occasionner un mauvais comportement des fonctions sur les chaînes si des données dont l'encodage n'est pas compatible avec la locale sont stockées dans la base.

Les paramètres d'encodage et de locale doivent correspondre à ceux de la base modèle, excepté quand la base template0 est utilisée comme modèle. La raison en est que d'autres bases de données pourraient contenir des données qui ne correspondent pas à l'encodage indiqué, ou pourraient contenir des index dont l'ordre de tri est affecté par `LC_COLLATE` et `LC_CTYPE`. Copier ces données peut résulter en une base de données qui est corrompue suivant les nouveaux paramètres. `template0`, par contre, ne contient aucun index pouvant être affecté par ces paramètres.

L'option `CONNECTION LIMIT` n'est qu'approximativement contraignante ; si deux nouvelles sessions commencent sensiblement en même temps alors qu'un seul « connecteur » à la base est disponible, il est possible que les deux échouent. De plus, les superutilisateurs ne sont pas soumis à cette limite.

Exemples

Créer une nouvelle base de données :

```
CREATE DATABASE lusiadas;
```

Créer une base de données `ventes` possédée par l'utilisateur `app_ventes` utilisant le tablespace `espace_ventes` comme espace par défaut :

```
CREATE DATABASE ventes OWNER app_ventes TABLESPACE espace_ventes;
```

Créer une base de données `musique` qui supporte le jeu de caractères ISO-8859-1 :

```
CREATE DATABASE musique ENCODING 'LATIN1' TEMPLATE template0;
```

Dans cet exemple, la clause `TEMPLATE template0` sera uniquement requise si l'encodage de `template1` n'est pas ISO-8859-1. Notez que modifier l'encodage pourrait aussi nécessiter de sélectionner de nouveaux paramètres pour `LC_COLLATE` et `LC_CTYPE`.

Compatibilité

Il n'existe pas d'instruction **CREATE DATABASE** dans le standard SQL. Les bases de données sont équivalentes aux catalogues, dont la création est définie par l'implantation.

Voir aussi

ALTER DATABASE(7), DROP DATABASE(7)

Nom

CREATE DOMAIN — Définir un nouveau domaine

Synopsis

```
CREATE DOMAIN nom [AS] type_donnee
    [ DEFAULT expression ]
    [ contrainte [ ... ] ]

où contrainte est :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL | CHECK (expression) }
```

Description

CREATE DOMAIN crée un nouveau domaine. Un domaine est essentiellement un type de données avec des contraintes optionnelles (restrictions sur l'ensemble de valeurs autorisées). L'utilisateur qui définit un domaine devient son propriétaire.

Si un nom de schéma est donné (par exemple, `CREATE DOMAIN monschema.mondomaine ...`), alors le domaine est créé dans le schéma spécifié. Sinon, il est créé dans le schéma courant. Le nom du domaine doit être unique parmi les types et domaines existant dans son schéma.

Les domaines permettent d'extraire des contraintes communes à plusieurs tables et de les regrouper en un seul emplacement, ce qui en facilite la maintenance. Par exemple, plusieurs tables pourraient contenir des colonnes d'adresses email, toutes nécessitant la même contrainte de vérification (CHECK) permettant de vérifier que le contenu de la colonne est bien une adresse email. Définissez un domaine plutôt que de configurer la contrainte individuellement sur chaque table.

Paramètres

nom
: Le nom du domaine à créer (éventuellement qualifié du nom du schéma).

type_donnees
: Le type de données sous-jacent au domaine. Il peut contenir des spécifications de tableau.

`DEFAULT` *expression*
: La clause `DEFAULT` permet de définir une valeur par défaut pour les colonnes d'un type de données du domaine. La valeur est une expression quelconque sans variable (les sous-requêtes ne sont pas autorisées). Le type de données de l'expression par défaut doit correspondre à celui du domaine. Si la valeur par défaut n'est pas indiquée, alors il s'agit de la valeur NULL.

  L'expression par défaut est utilisée dans toute opération d'insertion qui ne spécifie pas de valeur pour cette colonne. Si une valeur par défaut est définie sur une colonne particulière, elle surcharge toute valeur par défaut du domaine. De même, la valeur par défaut surcharge toute valeur par défaut associée au type de données sous-jacent.

`CONSTRAINT` *nom_contrainte*
: Un nom optionnel pour une contrainte. S'il n'est pas spécifié, le système en engendre un.

`NOT NULL`
: Les valeurs de ce domaine sont habituellement protégées comme les valeurs NULL. Néanmoins, il est toujours possible qu'un domaine avec cette contrainte prenne une valeur NULL s'il se voit affecté un type de domaine qui est devenu NULL, par exemple via une jointure LEFT OUTER JOIN ou une requête du type **INSERT INTO tab (colonne_domaine) VALUES ((SELECT colonne_domaine FROM tab WHERE false))**.

`NULL`
: Les valeurs de ce domaine peuvent être NULL. C'est la valeur par défaut.

  Cette clause a pour seul but la compatibilité avec les bases de données SQL non standard. Son utilisation est découragée dans les applications nouvelles.

`CHECK (`*expression*`)`
: Les clauses `CHECK` spécifient des contraintes d'intégrité ou des tests que les valeurs du domaine doivent satisfaire. Chaque contrainte doit être une expression produisant un résultat booléen. `VALUE` est obligatoirement utilisé pour se référer à la valeur testée.

  Actuellement, les expressions `CHECK` ne peuvent ni contenir de sous-requêtes ni se référer à des variables autres que `VA-`

LUE.

Exemples

Créer le type de données code_postal_us, et l'utiliser dans la définition d'une table. Un test d'expression rationnelle est utilisé pour vérifier que la valeur ressemble à un code postal US valide :

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
   VALUE ~ '^\\d{5}$'
OR VALUE ~ '^\\d{5}-\\d{4}$'
);

CREATE TABLE courrier_us (
  id_adresse SERIAL PRIMARY KEY,
  rue1 TEXT NOT NULL,
  rue2 TEXT,
  rue3 TEXT,
  ville TEXT NOT NULL,
  code_postal code_postal_us NOT NULL
);
```

Compatibilité

La commande **CREATE DOMAIN** est conforme au standard SQL.

Voir aussi

ALTER DOMAIN(7), DROP DOMAIN(7)

Nom

CREATE FOREIGN DATA WRAPPER — définit un nouveau wrapper de données distantes

Synopsis

```
CREATE FOREIGN DATA WRAPPER nom
    [ VALIDATOR fonction_validation | NO VALIDATOR ]
    [ OPTIONS ( option 'valeur' [, ... ] ) ]
```

Description

CREATE FOREIGN DATA WRAPPER crée un nouveau wrapper de données distantes. L'utilisateur qui définit un wrapper de données distantes devient son propriétaire.

Le nom du wrapper de données distantes doit être unique dans la base de données.

Seuls les super-utilisateurs peuvent créer des wrappers de données distantes.

Paramètres

nom
: Le nom du wrapper de données distantes à créer.

`VALIDATOR` *`fonction_validation`*
: *`fonction_validation`* est le nom d'une fonction déjà enregistrée qui sera appelée pour vérifier les options génériques passées au wrapper de données distantes, au serveur distant et aux correspondances d'utilisateurs (*user mappings*) utilisant le wrapper de données distantes. Si aucune fonction de validation n'est spécifiée ou si `NO VALIDATOR` est spécifié, alors les options ne seront pas vérifiées au moment de la création. (Il est possible que les wrappers de données distantes ignorent ou rejettent des spécifications d'options invalides à l'exécution, en fonction de l'implémentation) La fonction de validation doit prendre deux arguments : l'un du type text[], qui contiendra le tableau d'options, tel qu'il est stocké dans les catalogues systèmes, et l'autre de type oid, qui sera l'OID du catalogue système contenant les options. Le type de retour est inconnu ; la fonction devrait indiquer les options invalides grâce à la fonction `ereport()`.

`OPTIONS (` *`option 'valeur'`* `[, ... ] )`
: Cette clause spécifie les options pour le nouveau wrapper de données distantes. Les noms et valeurs d'options autorisés sont spécifiques à chaque wrapper de données distantes. Ils sont validés par la bibliothèque du wrapper de données distantes. Les noms d'options doivent être uniques.

Notes

Pour le moment, la fonctionnalité de wrapper de données distantes est très rudimentaire. Le but des wrappers de données distantes, des serveurs distants et des correspondances d'utilisateurs est de stocker cette information de manière standard, afin qu'elle puisse être interrogée par les applications intéressées. Une application utilisant cette fonctionnalité est dblink ; voir Section F.8, « dblink ». La fonctionnalité permettant d'exécuter une requête sur des données distantes via une bibliothèque de wrapper de données distantes n'existe pas encore.

Actuellement, une fonction de validation de wrapper de données distantes est disponible : `postgresql_fdw_validator`, qui accepte les options correspondant aux paramètres de connexion de libpq.

Exemples

Créer un wrapper de données distantes `bidon` :

```
CREATE FOREIGN DATA WRAPPER bidon;
```

Créer un wrapper de données distantes `postgresql` avec la fonction de validation `postgresql_fdw_validator` :

```
CREATE FOREIGN DATA WRAPPER postgresql VALIDATOR postgresql_fdw_validator;
```

Créer un wrapper de données distantes `monwrapper` avec des options :

```
CREATE FOREIGN DATA WRAPPER monwrapper
    OPTIONS (debug 'true');
```

Compatibilité

CREATE FOREIGN DATA WRAPPER est conforme à la norme ISO/IEC 9075-9 (SQL/MED), à l'exception de la clause `VALIDATOR` qui est une extension, et des clauses `LIBRARY` et `LANGUAGE` qui ne sont pas encore implémentées dans PostgreSQL.

Notez, cependant, que la fonctionnalité SQL/MED n'est pas encore conforme dans son ensemble.

Voir aussi

ALTER FOREIGN DATA WRAPPER(7), DROP FOREIGN DATA WRAPPER(7), CREATE SERVER(7), CREATE USER MAPPING(7)

Nom

CREATE FUNCTION — Définir une nouvelle fonction

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION
    nom ( [ [ modearg ] [ nomarg ] typearg  [ { DEFAULT | = } expression_par_defaut ]
[, ...] ] ) ] )
    [ RETURNS type_ret
      | RETURNS TABLE ( nom_colonne type_colonne [, ...] ) ]
  { LANGUAGE nom_lang
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | COST cout_execution
    | ROWS nb_lignes_resultat
    | SET parametre { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'fichier_obj', 'symbole_lien'
  } ...
    [ WITH ( attribut [, ...] ) ]
```

Description

CREATE FUNCTION définit une nouvelle fonction. **CREATE OR REPLACE FUNCTION** crée une nouvelle fonction ou la remplace si elle existe déjà. Pour pouvoir créer une fonction, l'utilisateur doit avoir le droit `USAGE` sur le langage associé.

Si un nom de schéma est précisé, la fonction est créée dans le schéma indiqué. Sinon, elle est créée dans le schéma courant. Le nom de la nouvelle fonction ne peut pas correspondre à celui d'une fonction existant avec les mêmes types d'arguments en entrée dans le même schéma. Toutefois, les fonctions de types d'arguments différents peuvent partager le même nom (ceci est appelé *surcharge*).

Pour remplacer la définition actuelle d'une fonction existante, **CREATE OR REPLACE FUNCTION** est utilisé. Il n'est pas possible de changer le nom ou les types d'argument d'une fonction de cette façon (cela crée une nouvelle fonction distincte). De même, **CREATE OR REPLACE FUNCTION** ne permet pas de modifier le type retour d'une fonction existante. Pour cela, il est nécessaire de supprimer et de recréer la fonction. (Lors de l'utilisation de paramètres `OUT`, cela signifie que le type d'un paramètre `OUT` ne peut être modifié que par la suppression de la fonction.)

Quand **CREATE OR REPLACE FUNCTION** est utilisé pour remplacer une fonction existante, le propriétaire et les droits de la fonction ne changent pas. Toutes les autres propriétés de la fonction se voient affectées les valeurs spécifiées dans la commande ou implicites pour les autres. Vous devez être le propriétaire de la fonction pour la remplacer ou être un membre du rôle propriétaire de la fonction.

En cas de suppression et de recréaction d'une fonction, la nouvelle fonction n'est pas la même entité que l'ancienne ; il faut supprimer les règles, vues, déclencheurs, etc. qui référencent l'ancienne fonction. **CREATE OR REPLACE FUNCTION** permet de modifier la définition d'une fonction sans casser les objets qui s'y réfèrent. De plus, **ALTER FUNCTION** peut être utilisé pour modifier la plupart des propriétés supplémentaires d'une fonction existante.

L'utilisateur qui crée la fonction en devient le propriétaire.

Paramètres

nom
: Le nom de la fonction à créer (éventuellement qualifié du nom du schéma).

modearg
: Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. En cas d'omission, la valeur par défaut est `IN`. Seuls des arguments `OUT` peuvent suivre un argument `VARIADIC`. Par ailleurs, des arguments `OUT` et `INOUT` ne peuvent pas être utilisés en même temps que la notation `RETURNS TABLE`.

nomarg
: Le nom d'un argument. Quelques langages (incluant PL/pgSQL, mais pas SQL) permettent d'utiliser ce nom dans le corps de la fonction. Pour les autres langages, le nom d'un argument en entrée est purement documentaire en ce qui concerne la fonction elle-même. Mais vous pouvez utiliser les noms d'arguments en entrée lors de l'appel d'une fonction pour améliorer

la lisibilité (voir Section 4.3, « Fonctions appelantes »). Dans tous les cas, le nom d'un argument en sortie a une utilité car il définit le nom de la colonne dans la ligne résultat. (En cas d'omission du nom d'un argument en sortie, le système choisit un nom de colonne par défaut.)

argtype

Le(s) type(s) de données des arguments de la fonction (éventuellement qualifié du nom du schéma), s'il y en a. Les types des arguments peuvent être basiques, composites ou de domaines, ou faire référence au type d'une colonne.

En fonction du langage, il est possible d'indiquer des « pseudotypes », tel que cstring. Les pseudotypes indiquent que le type d'argument réel est soit non complètement spécifié, soit en dehors de l'ensemble des types de données ordinaires du SQL.

Il est fait référence au type d'une colonne par *nom_table.nomcolonne*%TYPE. Cette fonctionnalité peut servir à rendre une fonction indépendante des modifications de la définition d'une table.

expression_par_defaut

Une expression à utiliser en tant que valeur par défaut si le paramètre n'est pas spécifié. L'expression doit pouvoir être coercible dans le type d'argument du paramètre. Seuls les paramètres d'entrée (dont les INOUT) peuvent avoir une valeur par défaut. Tous les paramètres d'entrée suivant un paramètre avec une valeur par défaut doivent aussi avoir une valeur par défaut.

type_ret

Le type de données en retour (éventuellement qualifié du nom du schéma). Le type de retour peut être un type basique, composite ou de domaine, ou faire référence au type d'une colonne existante. En fonction du langage, il est possible d'indiquer un « pseudotype », tel que cstring. Si la fonction ne doit pas renvoyer de valeur, on indique void comme type de retour.

Quand il y a des paramètres OUT ou INOUT, la clause RETURNS peut être omise. Si elle est présente, elle doit correspondre au type de résultat imposé par les paramètres de sortie : RECORD s'il y en a plusieurs, ou le type du seul paramètre en sortie.

Le modificateur SETOF indique que la fonction retourne un ensemble d'éléments plutôt qu'un seul.

Il est fait référence au type d'une colonne par *nom_table.nomcolonne*%TYPE.

nom_colonne

Le nom d'une colonne de sortie dans la syntaxe RETURNS TABLE. C'est une autre façon de déclarer un paramètre OUT nommé, à la différence près que RETURNS TABLE implique aussi RETURNS SETOF.

type_colonne

Le type de données d'une colonne de sortie dans la syntaxe RETURNS TABLE.

nom_lang

Le nom du langage d'écriture de la fonction. Peut être SQL, C, internal ou le nom d'un langage procédural utilisateur. Pour des raisons de compatibilité descendante, le nom peut être écrit entre guillemets simples.

WINDOW

WINDOW indique que la fonction est une *fonction window* plutôt qu'une fonction simple. Ceci n'est à l'heure actuelle utilisable que pour les fonctions écrites en C. L'attribut WINDOW ne peut pas être changé lors du remplacement d'une définition de fonction existante.

IMMUTABLE, STABLE, VOLATILE

Ces attributs informent l'optimiseur de requêtes sur le comportement de la fonction. Un seul choix est possible. En son absence, VOLATILE est utilisé.

IMMUTABLE indique que la fonction ne peut pas modifier la base de données et qu'à arguments constants, la fonction renvoie toujours le même résultat ; c'est-à-dire qu'elle n'effectue pas de recherches dans la base de données, ou alors qu'elle utilise des informations non directement présentes dans la liste d'arguments. Si cette option est précisée, tout appel de la fonction avec des arguments constants peut être immédiatement remplacé par la valeur de la fonction.

STABLE indique que la fonction ne peut pas modifier la base de données et qu'à l'intérieur d'un seul parcours de la table, à arguments constants, la fonction retourne le même résultat, mais celui-ci varie en fonction des instructions SQL. Cette option est appropriée pour les fonctions dont les résultats dépendent des recherches en base, des variables de paramètres (tel que la zone horaire courante), etc. (Ce mode est inapproprié pour les triggers AFTER qui souhaitent voir les lignes modifiées par la commande en cours.) La famille de fonctions current_timestamp est qualifiée de stable car les valeurs de ces fonctions ne changent pas à l'intérieur d'une transaction.

VOLATILE indique que la valeur de la fonction peut changer même au cours d'un seul parcours de table. Aucune optimisation ne peut donc être réalisée. Relativement peu de fonctions de bases de données sont volatiles dans ce sens ; quelques exemples sont random(), currval(), timeofday(). Toute fonction qui a des effets de bord doit être classée volatile, même si son résultat est assez prévisible. Cela afin d'éviter l'optimisation des appels ; setval() en est un exemple.

Pour des détails complémentaires, voir Section 35.6, « Catégories de volatilité des fonctions ».

CALLED ON NULL INPUT, RETURNS NULL ON NULL INPUT, STRICT

`CALLED ON NULL INPUT` (la valeur par défaut) indique que la fonction est appelée normalement si certains de ses arguments sont NULL. C'est alors de la responsabilité de l'auteur de la fonction de gérer les valeurs NULL.

`RETURNS NULL ON NULL INPUT` ou `STRICT` indiquent que la fonction renvoie toujours NULL si l'un de ses arguments est NULL. Lorsque ce paramètre est utilisé et qu'un des arguments est NULL, la fonction n'est pas exécutée, mais un résultat NULL est automatiquement retourné.

`[EXTERNAL] SECURITY INVOKER`, `[EXTERNAL] SECURITY DEFINER`
`SECURITY INVOKER` indique que la fonction est exécutée avec les droits de l'utilisateur qui l'appelle. C'est la valeur par défaut. `SECURITY DEFINER` spécifie que la fonction est exécutée avec les droits de l'utilisateur qui l'a créé.

Le mot clé `EXTERNAL` est autorisé pour la conformité SQL mais il est optionnel car, contrairement à SQL, cette fonctionnalité s'applique à toutes les fonctions, pas seulement celles externes.

`cout_execution`
Un nombre positif donnant le coût estimé pour l'exécution de la fonction en unité de cpu_operator_cost. Si la fonction renvoie plusieurs lignes, il s'agit d'un coût par ligne renvoyée. Si le coût n'est pas spécifié, une unité est supposée pour les fonctions en langage C et les fonctions internes. Ce coût est de 100 unités pour les fonctions dans tout autre langage. Des valeurs plus importantes feront que le planificateur tentera d'éviter l'évaluation de la fonction aussi souvent que possible.

`nb_lignes_resultat`
Un nombre positif donnant le nombre estimé de lignes que la fonction renvoie, information utile au planificateur. Ceci est seulement autorisé pour les fonctions qui renvoient plusieurs lignes (fonctions SRF). La valeur par défaut est de 1000 lignes.

`parametre`, *`valeur`*
La clause `SET` fait que le paramètre de configuration indiquée est initialisée avec la valeur précisée au lancement de la fonction, puis restaurée à sa valeur d'origine lors de la sortie de la fonction. `SET FROM CURRENT` utilise la valeur actuelle de la session comme valeur à appliquer au lancement de la fonction.

Si une clause `SET` est attachée à une fonction, alors les effets de la commande **SET LOCAL** exécutée à l'intérieur de la fonction pour la même variable sont restreints à la fonction : la valeur précédente du paramètre de configuration est de nouveau restaurée en sortie de la fonction. Néanmoins, une commande **SET** ordinaire (c'est-à-dire sans `LOCAL`) surcharge la clause `SET`, comme il le ferait pour une précédente commande **SET LOCAL** : les effets d'une telle commande persisteront après la sortie de la fonction sauf si la transaction en cours est annulée.

Voir SET(7) et Chapitre 18, Configuration du serveur pour plus d'informations sur les paramètres et valeurs autorisés.

`definition`
Une constante de type chaîne définissant la fonction ; la signification dépend du langage. Cela peut être un nom de fonction interne, le chemin vers un fichier objet, une commande SQL ou du texte en langage procédural.

Il est souvent utile d'utiliser les guillemets dollar (voir Section 4.1.2.4, « Constantes de chaînes avec guillemet dollar ») pour écrire le code de la fonction, au lie des la syntaxe habituelle des guillemets. Sans les guillemets dollar, tout guillemet ou antislash dans la définition de la fonction doit être échappé en les doublant.

`fichier_obj, symbole_lien`
Cette forme de clause `AS` est utilisée pour les fonctions en langage C chargeables dynamiquement lorsque le nom de la fonction dans le code source C n'est pas le même que celui de la fonction SQL. La chaîne *`fichier_obj`* est le nom du fichier contenant l'objet chargeable dynamiquement et *`symbole_lien`* est le symbole de lien de la fonction, c'est-à-dire le nom de la fonction dans le code source C. Si ce lien est omis, il est supposé être le même que le nom de la fonction SQL définie.

Lors d'appels répétés à **CREATE FUNCTION** se référant au même fichier objet, il est chargé seulement une fois par session. Pour décharger et recharger le fichier (par exemple lors du développement de la fonction), démarrez une nouvelle session.

`attribut`
Façon historique d'indiquer des informations optionnelles concernant la fonction. Les attributs suivants peuvent apparaître ici :

`isStrict`
Équivalent à `STRICT` ou `RETURNS NULL ON NULL INPUT`.

`isCachable`
`isCachable` est un équivalent obsolète de `IMMUTABLE` ; il est toujours accepté pour des raisons de compatibilité ascendante.
Les noms d'attribut sont insensibles à la casse.

La lecture de Section 35.3, « Fonctions utilisateur » fournit des informations supplémentaires sur l'écriture de fonctions.

Overloading

PostgreSQL™ autorise la *surcharge* des fonctions ; c'est-à-dire que le même nom peut être utilisé pour des fonctions différentes si tant est qu'elles aient des types d'arguments en entrée distincts. Néanmoins, les noms C de toutes les fonctions doivent être différents. Il est donc nécessaire de donner des noms différents aux fonctions C suchargées (on peut, par exemple, utiliser le type des arguments dans le nom de la fonction).

Deux fonctions sont considérées identiques si elles partagent le même nom et les mêmes types d'argument en *entrée*, sans considération des paramètres `OUT`. Les déclarations suivantes sont, de fait, en conflit :

```
CREATE FUNCTION truc(int) ...
CREATE FUNCTION truc(int, out text) ...
```

Des fonctions ayant des listes de types d'arguments différents ne seront pas considérées comme en conflit au moment de leur création, mais si des valeurs par défauts sont fournies, elles peuvent se retrouver en conflit au moment de l'invocation. Considérez par exemple :

```
CREATE FUNCTION truc(int) ...
CREATE FUNCTION truc(int, int default 42) ...
```

Un appel `truc(10)` échouera à cause de l'ambiguïté sur la fonction à appeler.

Notes

La syntaxe SQL complète des types est autorisé pour déclarer les arguments en entrée et la valeur de sortie d'une fonction. Néanmoins, les modificateurs du type de la fonction (par exemple le champ précision pour un numeric) sont ignorés par **CREATE FUNCTION**. Du coup, par exemple, `CREATE FUNCTION foo (varchar(10)) ...` est identique à `CREATE FUNCTION foo (varchar) ....`

Lors du remplacement d'une fonction existante avec **CREATE OR REPLACE FUNCTION**, il existe des restrictions sur le changement des noms de paramètres. Vous ne pouvez pas modifier le nom de paramètre en entrée déjà affecté mais vous pouvez ajouter des noms aux paramètres qui n'en avaient pas. S'il y a plus d'un paramètre en sortie, vous ne pouvez pas changer les noms des paramètres en sortie car cela changera les noms de colonne du type composite anonyme qui décrit le résultat de la fonction. Ces restrictions sont là pour assurer que les appels suivants à la fonction ne s'arrêtent pas de fonctionner lorsqu'elle est remplacée.

Exemples

Quelques exemples triviaux pour bien débuter sont présentés ci-après. Pour plus d'informations et d'exemples, voir Section 35.3, « Fonctions utilisateur ».

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Incrémenter un entier, en utilisant le nom de l'argument, dans PL/pgSQL :

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Renvoyer un enregistrement contenant plusieurs paramètres en sortie :

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

La même chose, en plus verbeux, avec un type composite nommé explicitement :

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Une autre façon de renvoyer plusieurs colonnes est d'utiliser une fonction `TABLE` :

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Toutefois, une fonction `TABLE` est différente des exemples précédents parce qu'elle retourne en fait un *ensemble* d'enregistrements, pas juste un enregistrement.

Écrire des fonctions `SECURITY DEFINER` en toute sécurité

Parce qu'une fonction `SECURITY DEFINER` est exécutée avec les droits de l'utilisateur qui l'a créé, une certaine attention est nécessaire pour s'assurer que la fonction ne peut pas être utilisée de façon maline. Pour des raisons de sécurité, search_path doit être configuré pour exclure les schémas modifiables par des utilisateurs indignes de confiance. Cela empêche des utilisateurs malveillants de créer des objets qui masquent les objets utilisés par la fonction. Dans ce sens, le schéma des tables temporaires est particulièrement important car il est le premier schéma parcouru et qu'il est normalement modifiable par tous les utilisateurs. Une solution consiste à forcer le parcours de ce schéma en dernier lieu. Pour cela, on écrit `pg_temp` comme dernière entrée de `search_path`. La fonction suivante illustre une utilisation sûre :

```
CREATE FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT)
RETURNS BOOLEAN AS $$
DECLARE ok BOOLEAN;
BEGIN
        -- Effectuer le travail sécurisé de la fonction.
        SELECT  (motdepasse = $2) INTO ok
        FROM    motsdepasse
        WHERE   nomutilisateur = $1;

        RETURN ok;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- Configure un search_path sécurisée : les schémas de confiance, puis 'pg_temp'.
    SET search_path = admin, pg_temp;
```

Avant PostgreSQL™ 8.3, l'option `SET` n'était pas disponible, donc les anciennes fonctions pouvaient contenir un code assez complexe pour sauvegarder, initialiser puis restaurer un paramètre comme `search_path`. L'option `SET` est plus simple à utiliser dans ce but.

Un autre point à garder en mémoire est que, par défaut, le droit d'exécution est donné à `PUBLIC` pour les fonctions nouvellement créées (voir GRANT(7) pour plus d'informations). Fréquemment, vous souhaiterez restreindre l'utilisation d'une fonction « security definer » à seulement quelques utilisateurs. Pour cela, vous devez révoquer les droits `PUBLIC` puis donner le droit d'exécution aux utilisateurs sélectionnés. Pour éviter que la nouvelle fonction soit accessible à tous pendant un court moment, créez-la et initialisez les droits dans une même transaction. Par exemple :

```
BEGIN;
CREATE FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT) TO admins;
COMMIT;
```

Compatibilité

Une commande **CREATE FUNCTION** est définie en SQL:1999 et ultérieur. La version PostgreSQL™ est similaire mais pas entièrement compatible. Les attributs ne sont pas portables, pas plus que les différents langages disponibles.

Pour des raisons de compatibilité avec d'autres systèmes de bases de données, *`modearg`* peut être écrit avant ou après *`nomarg`*. Mais seule la première façon est compatible avec le standard.

Le standard SQL ne définit pas de paramètres par défaut. La syntaxe avec le mot clé `DEFAULT` provient d'Oracle, et elle est assez proche de l'esprit du standard : SQL/PSQL l'utilise pour les valeurs par défaut de variables. La syntaxe avec `=` est utilisée dans T-SQL et Firebird.

Voir aussi

ALTER FUNCTION(7), DROP FUNCTION(7), GRANT(7), LOAD(7), REVOKE(7), createlang(1)

Nom

CREATE GROUP — Définir un nouveau rôle de base de données

Synopsis

```
CREATE GROUP nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'motdepasse'
    | VALID UNTIL 'dateheure'
    | IN ROLE nom_role [, ...]
    | IN GROUP nom_role [, ...]
    | ROLE nom_role [, ...]
    | ADMIN nom_role [, ...]
    | USER nom_role [, ...]
    | SYSID uid
```

Description

CREATE GROUP est désormais un alias de CREATE ROLE(7).

Compatibilité

Il n'existe pas d'instruction **CREATE GROUP** dans le standard SQL.

Voir aussi

CREATE ROLE(7)

Nom

CREATE INDEX — Définir un nouvel index

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ nom ] ON table [ USING méthode ]
    ( { colonne | ( expression ) } [ classeop ] [ ASC | DESC ] [ NULLS { FIRST | LAST
} ] [, ...] )
    [ WITH ( parametre_stockage = valeur [, ... ] ) ]
    [ TABLESPACE espacelogique ]
    [ WHERE prédicat ]
```

Description

CREATE INDEX construit un index sur le (ou les) colonne(s) spécifiée(s) de la table spécifiée. Les index sont principalement utilisés pour améliorer les performances de la base de données (bien qu'une utilisation inappropriée puisse produire l'effet inverse).

Les champs clé pour l'index sont spécifiés à l'aide de noms des colonnes ou par des expressions écrites entre parenthèses. Plusieurs champs peuvent être spécifiés si la méthode d'indexation supporte les index multi-colonnes.

Un champ d'index peut être une expression calculée à partir des valeurs d'une ou plusieurs colonnes de la ligne de table. Cette fonctionnalité peut être utilisée pour obtenir un accès rapide à des données obtenues par transformation des données basiques. Par exemple, un index calculé sur `upper(col)` autorise la clause `WHERE upper(col) = 'JIM'` à utiliser un index.

PostgreSQL™ fournit les méthodes d'indexation B-tree (NDT : arbres balancés), hash (NDT : hachage), GiST (NDT : arbres de recherche généralisés) et GIN. Il est possible, bien que compliqué, de définir des méthodes d'indexation utilisateur.

Lorsque la clause `WHERE` est présente, un *index partiel* est créé. Un index partiel est un index ne contenant des entrées que pour une portion d'une table, habituellement la portion sur laquelle l'indexation est la plus utile. Par exemple, si une table contient des ordres facturés et d'autres qui ne le sont pas, et que les ordres non facturés n'occupent qu'une petite fraction du total de la table, qui plus est fréquemment utilisée, les performances sont améliorées par la création d'un index sur cette portion. Une autre application possible est l'utilisation de la clause `WHERE` en combinaison avec `UNIQUE` pour assurer l'unicité sur un sous-ensemble d'une table. Voir Section 11.8, « Index partiels » pour plus de renseignements.

L'expression utilisée dans la clause `WHERE` peut ne faire référence qu'à des colonnes de la table sous-jacente, mais elle peut utiliser toutes les colonnes, pas uniquement celles indexées. Actuellement, les sous-requêtes et les expressions d'agrégats sont aussi interdites dans la clause `WHERE`. Les mêmes restrictions s'appliquent aux champs d'index qui sont des expressions.

Toutes les fonctions et opérateurs utilisés dans la définition d'index doivent être « immutable » (NDT : immuable), c'est-à-dire que leur résultat ne doit dépendre que de leurs arguments et jamais d'une influence externe (telle que le contenu d'une autre table ou l'heure). Cette restriction permet de s'assurer que le comportement de l'index est strictement défini. Pour utiliser une fonction utilisateur dans une expression d'index ou dans une clause `WHERE`, cette fonction doit être marquée immutable lors de sa création.

Paramètres

`UNIQUE`
: Le système vérifie la présence de valeurs dupliquées dans la table à la création de l'index (si des données existent déjà) et à chaque fois qu'une donnée est ajoutée. Les tentatives d'insertion ou de mises à jour qui résultent en des entrées dupliquées engendrent une erreur.

`CONCURRENTLY`
: Quand cette option est utilisée, PostgreSQL™ construira l'index sans prendre de verrous qui bloquent les insertions, mises à jour, suppression en parallèle sur cette table ; la construction d'un index standard verrouille les écritures (mais pas les lectures) sur la table jusqu'à la fin de la construction. Il est nécessaire d'avoir quelques connaissances avant d'utiliser cette option -- voir la section intitulée « Construire des index en parallèle ».

`nom`
: Le nom de l'index à créer. Aucun nom de schéma ne peut être inclus ici ; l'index est toujours créé dans le même schéma que sa table parent. Si le nom est omis, PostgreSQL™ choisit un nom convenable basé sur le nom de la table parent et celui des colonnes indexées.

`table`
: Le nom de la table à indexer (éventuellement qualifié du nom du schéma).

méthode
Le nom de la méthode à utiliser pour l'index. Les choix sont `btree`, `hash`, `gist` et `gin`. La méthode par défaut est `btree`.

colonne
Le nom d'une colonne de la table.

expression
Une expression basée sur une ou plusieurs colonnes de la table. L'expression doit habituellement être écrite entre parenthèses, comme la syntaxe le précise. Néanmoins, les parenthèses peuvent être omises si l'expression a la forme d'un appel de fonction.

classeop
Le nom d'une classe d'opérateur. Voir plus bas pour les détails.

`ASC`
Spécifie un ordre de tri ascendant (valeur par défaut).

`DESC`
Spécifie un ordre de tri descendant.

`NULLS FIRST`
Spécifie que les valeurs NULL sont présentées avant les valeurs non NULL. Ceci est la valeur par défaut quand `DESC` est indiqué.

`NULLS LAST`
Spécifie que les valeurs NULL sont présentées après les valeurs non NULL. Ceci est la valeur par défaut quand `ASC` est indiqué.

paramètre_stockage
Le nom d'un paramètre de stockage spécifique à la méthode d'indexage. Voir la section intitulée « Paramètres de stockage des index » pour les détails.

espacelogique
Le tablespace dans lequel créer l'index. S'il n'est pas précisé, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

prédicat
L'expression de la contrainte pour un index partiel.

Paramètres de stockage des index

La clause `WITH` optionnelle spécifie des *paramètres de stockage* pour l'index. Chaque méthode d'indexage peut avoir son propre ensemble de paramètres de stockage. Les méthodes d'indexage `B-tree`, `hash` et `GiST` acceptent toutes un seul paramètre :

`FILLFACTOR`
Le facteur de remplissage pour un index est un pourcentage qui détermine à quel point les pages d'index seront remplies par la méthode d'indexage. Pour les B-tree, les pages enfants sont remplies jusqu'à ce pourcentage lors de la construction initiale de l'index, et aussi lors de l'extension de l'index sur la droite (ajoutant les valeurs de clé les plus importantes). Si les pages deviennent ensuite totalement remplies, elles seront partagées, amenant une dégradation graduelle de l'efficacité de l'index. Les arbres B-tree utilisent un facteur de remplissage de 90% par défaut mais toute valeur entière comprise entre 10 et 100 peut être choisie. Si la table est statique, alors un facteur de 100 est meilleur pour minimiser la taille physique de l'index. Pour les tables mises à jour régulièrement, un facteur de remplissage plus petit est meilleur pour minimiser le besoin de pages divisées. Les autres méthodes d'indexage utilisent un facteur de remplissage de façon différente mais en gros analogue ; le facteur de remplissage varie suivant les méthodes.

Les index GIN acceptent un paramètre supplémentaire :

`FASTUPDATE`
Ce paramètre régit l'utilisation de la technique de mise à jour rapide décrite dans Section 53.3.1, « Technique GIN de mise à jour rapide ». C'est un paramètre booléen : `ON` active la mise à jour rapide, `OFF` la désactive. (Les autres façons d'écrire `ON` et `OFF` sont autorisées, comme décrit dans Section 18.1, « Paramètres de configuration ».) La valeur par défaut est `ON`.

Note

Désactiver `FASTUPDATE` via **ALTER INDEX** empêche les insertions futures d'aller dans la liste d'entrées d'index à traiter, mais ne nettoie pas les entrées précédentes de cette liste. Vous voudrez peut être ensuite exé-

cuter un **VACUUM** sur la table, afin de garantir que la liste à traiter soit vidée.

Construire des index en parallèle

Créer un index peut interférer avec les opérations normales d'une base de données. Habituellement, PostgreSQL™ verrouille la table à indexer pour la protéger des écritures et construit l'index complet avec un seul parcours de la table. Les autres transactions peuvent toujours lire la table mais s'ils essaient d'insérer, mettre à jour, supprimer des lignes dans la table, elles seront bloquées jusqu'à la fin de la construction de l'index. Ceci peut avoir un effet sérieux si le système est une base en production. Les très grosses tables peuvent demander plusieurs heures pour être indexées. Même pour les petites tables, une construction d'index peut bloquer les processus qui voudraient écrire dans la table pendant des périodes longues sur un système de production.

PostgreSQL™ supporte la construction des index sans verrouillage des écritures. Cette méthode est appelée en précisant l'option `CONCURRENTLY` de **CREATE INDEX**. Quand cette option est utilisée, PostgreSQL™ doit réaliser deux parcours de table et, en plus, il doit attendre que toutes les transactions existantes qui peuvent utiliser cet index se terminent. Du coup, cette méthode requiert plus de temps qu'une construction standard de l'index et est bien plus longue à se terminer. Néanmoins, comme cela autorise la poursuite des opérations pendant la construction de l'index, cette méthode est utile pour ajouter de nouveaux index dans un environnement en production. Bien sûr, la charge CPU et I/O supplémentaire imposée par la création de l'index peut ralentir les autres opérations.

Dans la construction en parallèle d'un index, l'index est enregistré dans le catalogue système dans une transaction, puis les deux parcours de table interviennent dans une deuxième puis une troisième transaction. Toutes les transactions actives au moment du lancement du deuxième parcours de la table, pas seulement celles qui ont déjà utilisées la table, ont le potentiel de bloquer la création en parallèle de l'index jusqu'à leur fin. Lors de la vérification des transactions qui pourraient toujours utiliser l'index original, la création en parallèle de l'index avance au travers des transactions plus anciennes potentiellement dangereuses, les vérifiant une par une, en obtenant des verrous partagés sur leur identifiant virtuel de transaction pour attendre leur fin. Si un problème survient lors du parcours de la table, comme une violation d'unicité dans un index unique, la commande **CREATE INDEX** échouera mais laissera derrière un index « invalide ». Cet index sera ignoré par les requêtes car il pourrait être incomplet ; néanmoins il consommera quand même du temps lors des mises à jour de l'index. La commande **\d** de psql rapportera cet index comme `INVALID` :

```
postgres=# \d tab
Table "public.tab"
Column |  Type   | Modifiers
--------+---------+-----------
col    | integer |
Indexes:
"idx" btree (col) INVALID
```

La méthode de récupération recommandée dans de tels cas est de supprimer l'index et de tenter de nouveau un **CREATE INDEX CONCURRENTLY**. (Une autre possibilité est de reconstruire l'index avec **REINDEX**. Néanmoins, comme **REINDEX** ne supporte pas la construction d'index en parallèle, cette option ne semble pas très attirante.)

Lors de la construction d'un index unique en parallèle, la contrainte d'unicité est déjà placée pour les autres transactions quand le deuxième parcours de table commence. Cela signifie que des violations de contraintes pourraient être rapportées dans les autres requêtes avant que l'index ne soit disponible, voire même dans des cas où la construction de l'index va échouer. De plus, si un échec survient dans le deuxième parcours, l'index « invalide » continue à forcer la contrainte d'unicité.

Les constructions en parallèle d'index avec expression et d'index partiels sont supportées. Les erreurs survenant pendant l'évaluation de ces expressions pourraient causer un comportement similaire à celui décrit ci-dessus pour les violations de contraintes d'unicité.

Les constructions d'index standards permettent d'autres construction d'index en parallèle sur la même table mais seul une construction d'index en parallèle peut survenir sur une table à un même moment. Dans les deux cas, aucun autre type de modification de schéma n'est autorisé sur la table. Une autre différence est qu'une commande **CREATE INDEX** normale peut être réalisée à l'intérieur d'un bloc de transactions mais **CREATE INDEX CONCURRENTLY** ne le peut pas.

Notes

Chapitre 11, Index présente des informations sur le moment où les index peuvent être utilisés, quand ils ne le sont pas et dans quelles situations particulières ils peuvent être utiles.

Actuellement, seules les méthodes d'indexation B-tree, GiST et GIN supportent les index multi-colonnes. Jusqu'à 32 champs peuvent être spécifiés par défaut. (Cette limite peut être modifiée à la compilation de PostgreSQL™.) Seul B-tree supporte actuellement les index uniques.

Une *classe d'opérateur* peut être spécifiée pour chaque colonne d'un index. La classe d'opérateur identifie les opérateurs à utiliser

par l'index pour cette colonne. Par exemple, un index B-tree sur des entiers codés sur quatre octets utilise la classe `int4_ops`, qui contient des fonctions de comparaison pour les entiers sur quatre octets. En pratique, la classe d'opérateur par défaut pour le type de données de la colonne est généralement suffisant. Les classes d'opérateur trouvent leur intérêt principal dans l'existence, pour certains types de données, de plusieurs ordonnancements significatifs.

Soit l'exemple d'un type de données « nombre complexe » qui doit être classé par sa valeur absolue ou par sa partie réelle. Cela peut être réalisé par la définition de deux classes d'opérateur pour le type de données, puis par la sélection de la classe appropriée lors de la création d'un index.

De plus amples informations sur les classes d'opérateurs sont disponibles dans Section 11.9, « Classes et familles d'opérateurs » et dans Section 35.14, « Interfacer des extensions d'index ».

Pour les méthodes d'indexage qui supportent les parcours ordonnés (actuellement seulement pour les B-tree), les clauses optionnelles `ASC`, `DESC`, `NULLS FIRST` et/ou `NULLS LAST` peuvent être spécifiées pour modifier l'ordre de tri normal de l'index. Comme un index ordonné peut être parcouru en avant et en arrière, il n'est habituellement pas utile de créer un index `DESC` sur une colonne -- ce tri est déjà disponible avec un index standard. L'intérêt de ces options se révèle avec les index multi-colonnes. Ils peuvent être créés pour correspondre à un tri particulier demandé par une requête, comme `SELECT ... ORDER BY x ASC, y DESC`. Les options `NULLS` sont utiles si vous avez besoin de supporter le comportement « nulls sort low », plutôt que le « nulls sort high » par défaut, dans les requêtes qui dépendent des index pour éviter l'étape du tri.

Pour la plupart des méthodes d'indexation, la vitesse de création d'un index est dépendante du paramètre maintenance_work_mem. Une plus grande valeur réduit le temps nécessaire à la création d'index, tant qu'elle ne dépasse pas la quantité de mémoire vraiment disponible, afin d'éviter que la machine ne doive paginer. Pour les index de type hash, la valeur de effective_cache_size est aussi importante pour le temps de création de l'index : PostgreSQL™ utilisera une des deux méthodes de création d'index hash disponibles selon que la taille estimée de l'index à créer est supérieure ou inférieure à `effective_cache_size`. Pour obtenir les meilleurs résultats, assurez-vous que ce paramètre est aussi positionné à quelque chose qui reflète la quantité de mémoire disponible, et soyez attentif à ce que la somme de `maintenance_work_mem` et `effective_cache_size` soit inférieure à la quantité de mémoire disponible de la machine pour PostgreSQL™ (en prenant en compte les autres programmes en cours d'exécution sur la machine).

DROP INDEX(7) est utilisé pour supprimer un index.

Les versions précédentes de PostgreSQL™ ont aussi une méthode d'index R-tree. Cette méthode a été supprimée car elle n'a pas d'avantages par rapport à la méthode GiST. Si `USING rtree` est indiqué, **CREATE INDEX** l'interprétera comme `USING gist` pour simplifier la conversions des anciennes bases à GiST.

Exemples

Créer un index B-tree sur la colonne `titre` dans la table `films` :

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Pour créer un index sur l'expression `lower(titre)`, permettant une recherche efficace quelque soit la casse :

```
CREATE INDEX ON films ((lower(titre)));
```

(dans cet exemple, nous avons choisi d'omettre le nom de l'index, donc le système choisira un nom, typiquement `films_lower_idx`.)

> **Attention**
>
> Les opérations sur les index hash ne sont pas enregistrées dans les journaux de transactions. Du coup, les index hash doivent être reconstruit avec **REINDEX** après un arrêt brutal de la base de données si des modifications n'ont pas été écrites. De plus, les modifications dans les index hash ne sont pas répliquées avec la réplication Warm Standby après la sauvegarde de base initiale, donc ces index donneront de mauvaises réponses aux requêtes qui les utilisent. Pour ces raisons, l'utilisation des index hash est actuellement déconseillée.

Pour créer un index avec un ordre de tri des valeurs NULL différent du standard :

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

Pour créer un index avec un facteur de remplissage différent :

```
CREATE UNIQUE INDEX idx_titre ON films (titre) WITH (fillfactor = 70);
```

Pour créer un index GIN avec les mises à jour rapides désactivées :

```
CREATE INDEX gin_idx ON documents_table USING gin (locations) WITH (fastupdate = off);
```

Créer un index sur la colonne `code` de la table `films` et donner à l'index l'emplacement du tablespace `espaceindex` :

```
CREATE INDEX code_idx ON films (code) TABLESPACE espaceindex;
```

Pour créer un index GiST sur un attribut point, de façon à ce que nous puissions utiliser rapidement les opérateurs box sur le résultat de la fonction de conversion :

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

Pour créer un index sans verrouiller les écritures dans la table :

```
CREATE INDEX CONCURRENTLY index_quentite_ventes ON table_ventes (quantité);
```

Compatibilité

CREATE INDEX est une extension du langage PostgreSQL™. Les index n'existent pas dans le standard SQL.

Voir aussi

ALTER INDEX(7), DROP INDEX(7)

Nom

CREATE LANGUAGE — Définir un nouveau langage procédural

Synopsis

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE nom
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE nom
    HANDLER gestionnaire_appel [ VALIDATOR fonction_validation ]
```

Description

CREATE LANGUAGE enregistre un nouveau langage procédural à une base de données PostgreSQL™. En conséquence, les fonctions et les procédures de déclencheurs peuvent être définies dans ce nouveau langage.

CREATE LANGUAGE associe en fait le nom du langage à un ou des fonctions de gestion qui sont responsable de l'exécution des fonctions écrites dans le langage. Chapitre 38, Langages de procédures offre de plus amples informations sur les gestionnaires de fonctions.

La commande **CREATE LANGUAGE** existe sous deux formes. Dans la première, l'utilisateur ne fournit que le nom du langage désiré et le serveur PostgreSQL™ consulte le catalogue système pg_pltemplate pour déterminer les paramètres adéquats. Dans la seconde, l'utilisateur fournit les paramètres du langage avec son nom. Cette forme peut être utilisée pour créer un langage non défini dans pg_pltemplate. Cette approche est cependant obsolète.

Si le serveur trouve une entrée dans le catalogue pg_pltemplate pour le nom donné, il utilise les données du catalogue quand bien même la commande incluerait les paramètres du langage. Ce comportement simplifie le chargement des anciens fichiers de sauvegarde ; ceux-ci présentent le risque de contenir des informations caduques sur les fonctions de support du langage.

Habituellement, l'utilisateur doit être un superutilisateur PostgreSQL™ pour enregistrer un nouveau langage. Néanmoins, le propriétaire d'une base de données peut enregistrer un nouveau langage dans sa base si le langage est listé dans le catalogue pg_pltemplate et est marqué comme autorisé à être créé par les propriétaires de base (*tmpldbacreate* à true). La valeur par défaut est que les langages de confiance peuvent être créés par les propriétaires de base de données, mais cela peut être modifié par les superutilisateurs en ajustant le contenu de pg_pltemplate. Le créateur d'un langage devient son propriétaire et peut ensuite le supprimer, le renommer ou le donner à un autre propriétaire.

CREATE OR REPLACE LANGUAGE créera un nouveau langage ou remplacera une définition existante. Si le langage existe déjà, ces paramètres sont mis à jour suivant les valeurs indiquées ou prises de pg_pltemplate mais le propriétaire et les droits du langage ne sont pas modifiés et toutes fonctions existantes créées dans le langage sont supposées être toujours valides. En plus des droits nécessaires pour créer un langage, un utilisateur doit être superutilisateur ou propriétaire du langage existant. Le cas `REPLACE` a pour but principal d'être utilisé pour s'assurer que le langage existe. Si le langage a une entrée pg_pltemplate alors `REPLACE` ne modifiera rien sur la définition existante, sauf dans le cas inhabituel où l'entrée pg_pltemplate a été modifiée depuis que le langage a été créé.

Paramètres

`TRUSTED`
: `TRUSTED` indique que le langage ne donne pas accès aux données auquel l'utilisateur n'a pas normalement accès. Si ce mot clé est omis à l'enregistrement du langage, seuls les superutilisateurs peuvent utiliser ce langage pour créer de nouvelles fonctions.

`PROCEDURAL`
: Sans objet.

`nom`
: Le nom du nouveau langage procédural, insensible à la casse. Il ne peut y avoir deux langages portant le même nom au sein de la base de données.

  Pour des raisons de compatibilité descendante, le nom doit être entouré de guillemets simples.

`HANDLER` *`gestionnaire_appel`*
: *`gestionnaire_appel`* est le nom d'une fonction précédemment enregistrée. C'est elle qui est appelée pour exécuter les fonctions du langage procédural. Le gestionnaire d'appels d'un langage procédural doit être écrit dans un langage compilé, tel que le C, avec la convention d'appel version 1 et enregistré dans PostgreSQL™ comme une fonction ne prenant aucun argument et retournant le type language_handler, type servant essentiellement à identifier la fonction comme gestionnaire d'appels.

`INLINE` *`gestionnaire_en_ligne`*

`gestionnaire_en_ligne` est le nom d'une fonction déjà enregistrée qui sera appelée pour exécuter un bloc de code anonyme (voir la commande DO(7)) dans ce langage. Si aucune fonction *`gestionnaire_en_ligne`* n'est indiquée, le langage ne supporte pas les blocs de code anonymes. La fonction de gestion doit prendre un argument du type internal, qui sera la représentation interne de la commande **DO**, et il renverra le type void. La valeur de retour du gestionnaire est ignorée.

`VALIDATOR` *`fonction_validation`*

`fonction_validation` est le nom d'une fonction précédemment enregistrée. C'est elle qui est appelée pour valider toute nouvelle fonction écrite dans ce langage. Si aucune fonction de validation n'est spécifiée, alors toute nouvelle fonction n'est pas vérifiée à sa création. La fonction de validation prend obligatoirement un argument de type oid, OID de la fonction à créer, et renvoie par convention void.

Une fonction de validation contrôle généralement le corps de la fonction pour s'assurer de sa justesse syntaxique mais peut également vérifier d'autres propriétés de la fonction (l'incapacité du langage à gérer certains types d'argument, par exemple). Le signalement d'erreur se fait à l'aide de la fonction `ereport()`. La valeur de retour de la fonction est ignorée.

L'option `TRUSTED` et le(s) nom(s) de la fonction de support sont ignorés s'il existe une entrée dans la table pg_pltemplate pour le nom du langage spécifié.

Notes

Le programme createlang(1) est un simple enrobage de la commande **CREATE LANGUAGE**. Il facilite l'installation des langages procéduraux à partir de la ligne de commande du shell.

DROP LANGUAGE(7), ou mieux, le programme droplang(1) sont utilisés pour supprimer des langages procéduraux.

Le catalogue système `pg_language` (voir Section 45.24, « pg_language ») contient des informations sur les langages installés. De plus, **createlang** dispose d'une option pour lister ces langages.

Pour créer des fonctions dans un langage procédural, l'utilisateur doit posséder le droit `USAGE` pour ce langage. Par défaut, `USAGE` est donné à `PUBLIC` (c'est-à-dire tout le monde) pour les langages de confiance. Ce droit peut être révoqué si nécessaire.

Les langages procéduraux sont installées par base. Néanmoins, un langage peut être installé dans la base de données `template1`, ce qui le rend automatiquement disponible dans toutes les bases de données créées par la suite.

Le gestionnaire d'appels, le gestionnaire en ligne (s'il y en a un) et la fonction de validation (s'il y en a une) doivent exister préalablement si le serveur ne possède pas d'entrée pour ce langage dans pg_pltemplate. Dans le cas contraire, les fonctions n'ont pas besoin de pré-exister ; elles sont automatiquement définies si elles ne sont pas présentes dans la base de données. (Cela peut amener **CREATE LANGUAGE** à échouer si la bibliothèque partagée implémentant le langage n'est pas disponible dans l'installation.)

Dans les versions de PostgreSQL™ antérieures à 7.3, il était nécessaire de déclarer des fonctions de gestion renvoyant le type opaque, plutôt que language_handler. Pour accepter le chargement d'anciens fichiers de sauvegarde, **CREATE LANGUAGE** accepte toute fonction retournant le type opaque mais affiche un message d'avertissement et modifie le type de retour de la fonction en language_handler.

Exemples

Tout langage procédural standard sera préférentiellement créé ainsi :

```
CREATE LANGUAGE plperl;
```

Pour un langage inconnu du catalogue pg_pltemplate, une séquence comme celle-ci est nécessaire :

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibilité

CREATE LANGUAGE est un extension de PostgreSQL™.

Voir aussi

ALTER LANGUAGE(7), CREATE FUNCTION(7), DROP LANGUAGE(7), GRANT(7), REVOKE(7), createlang(1), droplang(1)

Nom

CREATE OPERATOR — Définir un nouvel opérateur

Synopsis

```
CREATE OPERATOR nom (
    PROCEDURE = nom_fonction
    [, LEFTARG = type_gauche ]
    [, RIGHTARG = type_droit ]
    [, COMMUTATOR = op_com ]
    [, NEGATOR = op_neg ]
    [, RESTRICT = proc_res ]
    [, JOIN = proc_join ]
    [, HASHES ] [, MERGES ]
)
```

Description

CREATE OPERATOR définit un nouvel opérateur, *nom*. L'utilisateur qui définit un opérateur en devient propriétaire. Si un nom de schéma est donné, l'opérateur est créé dans le schéma spécifié. Sinon, il est créé dans le schéma courant.

Le nom de l'opérateur est une séquence d'au plus NAMEDATALEN-1 (63 par défaut) caractères parmi la liste suivante :

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

Il existe quelques restrictions dans le choix du nom :

- `--` et `/*` ne peuvent pas apparaître dans le nom d'un opérateur car ils sont pris pour le début d'un commentaire.
- Un nom d'opérateur multicaractères ne peut pas finir avec `+` ou `-` sauf si le nom contient l'un, au moins, de ces caractères :

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  Par exemple, `@-` est un nom d'opérateur autorisé mais `*-` n'en est pas un. Cette restriction permet à PostgreSQL™ d'analyser les commandes compatibles SQL sans nécessiter d'espaces entre les lexèmes.
- L'utilisation de `=>` comme nom d'opérateur est déconseillée. Il pourrait être complètement interdit dans une prochaine version.

L'opérateur `!=` est remplacé par `<>` à la saisie, ces deux noms sont donc toujours équivalents.

Au moins un des deux `LEFTARG` et `RIGHTARG` doit être défini. Pour les opérateurs binaires, les deux doivent l'être. Pour les opérateurs unaires droits, seul `LEFTARG` doit l'être, `RIGHTARG` pour les opérateurs unaires gauches.

La procédure *nom_fonction* doit avoir été précédemment définie par **CREATE FUNCTION** et doit accepter le bon nombre d'arguments (un ou deux) des types indiqués.

Les autres clauses spécifient des clauses optionnelles d'optimisation d'opérateur. Leur signification est détaillée dans Section 35.13, « Informations sur l'optimisation d'un opérateur ».

Paramètres

nom
: Le nom de l'opérateur à définir. Voir ci-dessus pour les caractères autorisés. Le nom peut être qualifié du nom du schéma, par exemple `CREATE OPERATOR monschema.+ (...)`. Dans le cas contraire, il est créé dans le schéma courant. Deux opérateurs dans le même schéma peuvent avoir le même nom s'ils opérent sur des types de données différents. On parle alors de *surchargement*.

nom_fonction
: La fonction utilisée pour implanter cet opérateur.

type_gauche
: Le type de données de l'opérande gauche de l'opérateur, s'il existe. Cette option est omise pour un opérateur unaire gauche.

type_droit
: Le type de données de l'opérande droit de l'opérateur, s'il existe. Cette option est omise pour un opérateur unaire droit.

op_com
Le commutateur de cet opérateur.

op_neg
La négation de cet opérateur.

proc_res
La fonction d'estimation de la sélectivité de restriction pour cet opérateur.

proc_join
La fonction d'estimation de la sélectivité de jointure pour cet opérateur.

HASHES
L'opérateur peut supporter une jointure de hachage.

MERGES
L'opérateur peut supporter une jointure de fusion.

La syntaxe `OPERATOR()` est utilisée pour préciser un nom d'opérateur qualifié d'un schéma dans *op_com* ou dans les autres arguments optionnels. Par exemple :

```
COMMUTATOR = OPERATOR(mon_schema.===) ,
```

Notes

Section 35.12, « Opérateurs définis par l'utilisateur » fournit de plus amples informations.

Il n'est pas possible de spécifier la précédence lexicale d'un opérateur dans **CREATE OPERATOR** car le comportement de précédence de l'analyseur n'est pas modifiable. Voir Section 4.1.6, « Précédence lexicale » pour des détails sur la gestion de la précédence.

Les options obsolètes, `SORT1`, `SORT2`, `LTCMP` et `GTCMP` étaient utilisées auparavant pour spécifier les noms des opérateurs de tris associés avec un opérateur joignable par fusion (`mergejoinable`). Ceci n'est plus nécessaire car l'information sur les opérateurs associés est disponible en cherchant les familles d'opérateur B-tree. Si une des ces options est fournie, elle est ignorée mais configure implicitement `MERGES` à true.

DROP OPERATOR(7) est utilisé pour supprimer les opérateurs utilisateur, ALTER OPERATOR(7) pour les modifier.

Exemples

La commande suivante définit un nouvel opérateur, « area-equality », pour le type de données box :

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES, MERGES
);
```

Compatibilité

CREATE OPERATOR est une extension PostgreSQL™. Il n'existe pas d'opérateurs utilisateur dans le standard SQL.

Voir aussi

ALTER OPERATOR(7), CREATE OPERATOR CLASS(7), DROP OPERATOR(7)

Nom

CREATE OPERATOR CLASS — Définir une nouvelle classe d'opérateur

Synopsis

```
CREATE OPERATOR CLASS nom [ DEFAULT ] FOR TYPE type_donnee
  USING methode_indexage [ FAMILY nom_famille ] AS
  {  OPERATOR numero_strategie nom_operateur [ ( type_op, type_op ) ]
  | FUNCTION numero_support [ ( type_op [ , type_op ] ) ] nom_fonction (
type_argument [, ...] )
   | STORAGE type_stockage
  } [, ... ]
```

Description

CREATE OPERATOR CLASS crée une nouvelle classe d'opérateur. Une classe d'opérateur définit la façon dont un type de données particulier peut être utilisé avec un index. La classe d'opérateur spécifie le rôle particulier ou la « stratégie » que jouent certains opérateurs pour ce type de données et cette méthode d'indexation. La classe d'opérateur spécifie aussi les procédures de support à utiliser par la méthode d'indexation quand la classe d'opérateur est sélectionnée pour une colonne d'index. Tous les opérateurs et fonctions utilisés par une classe d'opérateur doivent être définis avant la création de la classe d'opérateur.

Si un nom de schéma est donné, la classe d'opérateur est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Deux classes d'opérateur ne peuvent avoir le même nom que s'ils concernent des méthodes d'indexation différentes.

L'utilisateur qui définit une classe d'opérateur en devient propriétaire. Actuellement, le créateur doit être superutilisateur. Cette restriction existe parce qu'une définition erronée d'une classe d'opérateur peut gêner le serveur, voire causer un arrêt brutal de celui-ci.

Actuellement, **CREATE OPERATOR CLASS** ne vérifie pas si la définition de la classe d'opérateur inclut tous les opérateurs et fonctions requis par la méthode d'indexation. Il ne verifie pas non plus si les opérateurs et les fonctions forment un ensemble cohérent. Il est de la responsabilité de l'utilisateur de définir une classe d'opérateur valide.

Les classes d'opérateur en relation peuvent être groupées dans des *familles d'opérateurs*. Pour ajouter une nouvelle classe d'opérateur à une famille existante, indiquez l'option `FAMILY` dans **CREATE OPERATOR CLASS**. Sans cette option, la nouvelle classe est placée dans une famille de même nom (créant la famille si elle n'existe pas).

Section 35.14, « Interfacer des extensions d'index » fournit de plus amples informations.

Paramètres

`nom`
: Le nom (éventuellement qualifié du nom du schém) de la classe d'opérateur à créer.

`DEFAULT`
: La classe d'opérateur est celle par défaut pour son type de données. Il ne peut y avoir qu'une classe d'opérateur par défaut pour un type de données et une méthode d'indexation particuliers.

`type_données`
: Le type de données de la colonne auquel s'applique cette classe d'opérateur.

`méthode_index`
: Le nom de la méthode d'indexation à laquelle s'applique la classe d'opérateur.

`nom_famille`
: Le nom d'une famille d'opérateur existante pour lui ajouter cette classe d'opérateur. Si non spécifié, une famille du même nom que l'opérateur est utilisée (la créant si elle n'existe pas déjà).

`numéro_stratégie`
: Le numéro de stratégie de la méthode d'indexation pour un opérateur associé à la classe d'opérateur.

`nom_opérateur`
: Le nom (éventuellement qualifié du nom du schéma) d'un opérateur associé à la classe d'opérateur.

`op_type`
: Dans une clause `OPERATOR`, le(s) type(s) de données de l'opérande d'un opérateur ou `NONE` pour signifier un opérateur unaire (droite ou gauche). Les types de données de l'opérande peuvent être omis dans le cas où ils sont identiques au type de

données de la classe d'opérateur.

Dans une clause `FUNCTION`, le (ou les) types de données en opérande, supporté par la fonction, si différent du type de données en entrée de la fonction (pour les index B-tree et hash) ou le type de données de la classe (pour les index GIN et GiST). Ces valeurs par défaut sont toujours correctes, donc il n'est pas nécessaire de préciser *`type_op`* dans une clause `FUNCTION` de la commande **CREATE OPERATOR CLASS**, mais l'option est fournie pour des raisons de cohérence avec la syntaxe de **ALTER OPERATOR FAMILY**.

`numéro_support`
: Le numéro de procédure support de la méthode d'indexation pour une fonction associée à la classe d'opérateur.

`nom_fonction`
: Le nom (éventuellement qualifié du nom du schéma) d'une fonction procédure support pour la méthode d'indexation de la classe d'opérateur.

`types_argument`
: Le(s) type(s) de données des paramètres de la fonction.

`type_stockage`
: Le type de données réellement stocké dans l'index. C'est normalement le même que le type de données de la colonne mais certaines méthodes d'indexage (GIN et GiST actuellement) autorisent un type différent. La clause `STORAGE` doit être omise sauf si la méthode d'indexation autorise un type différent.

L'ordre des clauses `OPERATOR`, `FUNCTION` et `STORAGE` n'a aucune importance.

Notes

Comme toute la partie d'indexage ne vérifie pas les droits d'accès aux fonctions avant de les utiliser, inclure une fonction ou un opérateur dans une classe d'opérateur the index machinery does not check access permissions on functions before using them, including a function or operator in an operator class is équivalent à donner les droits d'exécution à PUBLIC sur celle-ci. Ce n'est pas un problème habituellement pour les types de fonctions utiles dans une classe d'opérateur.

Les opérateurs ne doivent pas être définis par des fonctions SQL. Une fonction SQL peut être intégrée dans la requête appelante, ce qui empêche l'optimiseur de faire la correspondance avec un index.

Avant PostgreSQL™ 8.4, la clause `OPERATOR` pouvait inclure l'option `RECHECK`. Cela n'est plus supporté car le fait qu'un index soit « à perte » est maintenant déterminé à l'exécution. Ceci permet une gestion plus efficace des cas où l'opérateur pourrait ou non être à perte.

Exemples

La commande issue de l'exemple suivant définit une classe d'opérateur d'indexation GiST pour le type de données `_int4` (tableau de int4). Voir `contrib/intarray/` pour l'exemple complet.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, int, oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

Compatibilité

CREATE OPERATOR CLASS est une extension PostgreSQL™. Il n'existe pas d'instruction **CREATE OPERATOR CLASS** dans le standard SQL.

Voir aussi

ALTER OPERATOR CLASS(7), DROP OPERATOR CLASS(7), CREATE OPERATOR FAMILY(7), ALTER OPERATOR FAMILY(7)

Nom

CREATE OPERATOR FAMILY — définir une nouvelle famille d'opérateur

Synopsis

```
CREATE OPERATOR FAMILY nom USING methode_indexage
```

Description

CREATE OPERATOR FAMILY crée une nouvelle famille d'opérateurs. Une famille d'opérateurs définit une collection de classes d'opérateur en relation et peut-être quelques opérateurs et fonctions de support supplémentaires compatibles avec ces classes d'opérateurs mais non essentiels au bon fonctionnement des index individuels. (Les opérateurs et fonctions essentiels aux index doivent être groupés avec la classe d'opérateur adéquate, plutôt qu'être des membres « lâches » dans la famille d'opérateur. Typiquement, les opérateurs sur un seul type de données peuvent être lâches dans une famille d'opérateur contenant des classes d'opérateur pour les deux types de données.)

La nouvelle famille d'opérateur est initialement vide. Elle sera remplie en exécutant par la suite des commandes **CREATE OPERATOR CLASS** pour ajouter les classes d'opérateurs contenues et, en option, des commandes **ALTER OPERATOR FAMILY** pour ajouter des opérateurs et leur fonctions de support correspondantes en tant que membres « lâches ».

Si un nom de schéma est précisée, la famille d'opérateur est créée dans le schéma en question. Sinon elle est créée dans le schéma en cours. Deux familles d'opérateurs du même schéma ne peuvent avoir le même nom que s'ils sont des méthodes d'indexage différentes.

L'utilisateur qui définit une famille d'opérateur devient son propriétaire. Actuellement, l'utilisateur qui crée doit être un superutilisateur. (Cette restriction est nécessaire car une définition erronée d'une famille d'opérateur pourrait gêner le serveur, voire même l'arrêter brutalement.)

Voir Section 35.14, « Interfacer des extensions d'index » pour plus d'informations.

Paramètres

nom
: Le nom de la famille d'opérateur (pouvant être qualifié du schéma).

methode_indexage
: Le nom de la méthode d'indexage utilisée par cette famille d'opérateur.

Compatibilité

CREATE OPERATOR FAMILY est un extension PostgreSQL™. Il n'existe pas d'instruction **CREATE OPERATOR FAMILY** dans le standard SQL.

Voir aussi

ALTER OPERATOR FAMILY(7), DROP OPERATOR FAMILY(7), CREATE OPERATOR CLASS(7), ALTER OPERATOR CLASS(7), DROP OPERATOR CLASS(7)

Nom

CREATE ROLE — Définir un nouveau rôle de base de données

Synopsis

```
CREATE ROLE nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'motdepasse'
    | VALID UNTIL 'heuredate'
    | IN ROLE nom_role [, ...]
    | IN GROUP nom_role [, ...]
    | ROLE nom_role [, ...]
    | ADMIN nom_role [, ...]
    | USER nom_role [, ...]
    | SYSID uid
```

Description

CREATE ROLE ajoute un nouveau rôle dans une grappe (cluster) de bases de données PostgreSQL™. Un rôle est une entité qui peut posséder des objets de la base de données et avoir des droits sur la base. Il peut être considéré comme un « utilisateur », un « groupe » ou les deux suivant la façon dont il est utilisé. Chapitre 20, Rôles et droits de la base de données et Chapitre 19, Authentification du client donnent de plus amples informations sur la gestion des utilisateurs et l'authentification. Il est nécessaire de posséder le droit `CREATEROLE` ou d'être superutilisateur pour utiliser cette commande.

Les rôles sont définis au niveau de la grappe de bases de données, et sont donc valides dans toutes les bases de la grappe.

Paramètres

nom
: Le nom du nouveau rôle.

`SUPERUSER`, `NOSUPERUSER`
: Ces clauses définissent si le nouveau rôle est un « superutilisateur » et peut ainsi outrepasser les droits d'accès à la base de données. Le statut de superutilisateur est dangereux et ne doit être utilisé que lorsque cela est réellement nécessaire. Seul un superutilisateur peut créer un superutilisateur. `NOSUPERUSER` est la valeur par défaut.

`CREATEDB`, `NOCREATEDB`
: Ces clauses précisent le droit de création de bases de données. Si `CREATEDB` est spécifié, l'autorisation est donnée au rôle, `NOCREATEDB` produit l'effet inverse. `NOCREATEDB` est la valeur par défaut.

`CREATEROLE`, `NOCREATEROLE`
: Ces clauses précisent le droit de création de nouveaux rôles (c'est-à-dire d'exécuter **CREATE ROLE**). Un rôle qui possède le droit `CREATEROLE` peut aussi modifier ou supprimer d'autres rôles. `NOCREATEROLE` est la valeur par défaut.

`CREATEUSER`, `NOCREATEUSER`
: Ces clauses, obsolètes mais toujours acceptées, sont équivalentes à `SUPERUSER` et `NOSUPERUSER`. Elles *ne* sont *pas* équivalentes à `CREATEROLE`.

`INHERIT`, `NOINHERIT`
: Ces clauses précisent si un rôle « hérite » des droits d'un rôle dont il est membre. Un rôle qui possède l'attribut `INHERIT` peut automatiquement utiliser tout privilège détenu par un rôle dont il est membre direct ou indirect. Sans `INHERIT`, l'appartenance à un autre rôle lui confère uniquement la possibilité d'utiliser **SET ROLE** pour acquérir les droits de l'autre rôle ils ne sont disponibles qu'après cela. `INHERIT` est la valeur par défaut.

`LOGIN`, `NOLOGIN`
: Ces clauses précisent si un rôle est autorisé à se connecter, c'est-à-dire si le rôle peut être donné comme nom pour

l'autorisation initiale de session à la connexion du client. Un rôle ayant l'attribut `LOGIN` peut être vu comme un utilisateur. Les rôles qui ne disposent pas de cet attribut sont utiles pour gérer les privilèges de la base de données mais ne sont pas des utilisateurs au sens habituel du mot. `NOLOGIN` est la valeur par défaut, sauf lorsque **CREATE ROLE** est appelé à travers la commande **CREATE USER**.

`CONNECTION LIMIT` *`limiteconnexion`*
Le nombre maximum de connexions concurrentes possibles pour le rôle, s'il possède le droit de connexion. -1 (valeur par défaut) signifie qu'il n'y a pas de limite.

`PASSWORD` *`motdepasse`*
Le mot de passe du rôle. Il n'est utile que pour les rôles ayant l'attribut `LOGIN`, mais il est possible d'en définir un pour les rôles qui ne l'ont pas. Cette option peut être omise si l'authentification par mot de passe n'est pas envisagée. Si aucun mot de passe n'est spécifié, le mot de passe sera NULL et l'authentification par mot de passe échouera toujours pour cet utilisateur. Un mot de passe NULL peut aussi être indiqué explicitement avec `PASSWORD NULL`.

`ENCRYPTED`, `UNENCRYPTED`
Ces mots clés contrôlent le chiffrement du mot de passe stocké dans les catalogues système. En l'absence de précision, le comportement par défaut est déterminé par le paramètre de configuration password_encryption. Si le mot de passe présenté est déjà une chaîne chiffrée avec MD5, il est stocké ainsi, quelque soit le mot-clé spécifié, `ENCRYPTED` ou `UNENCRYPTED` (le système ne peut pas déchiffrer la chaîne déjà chiffrée). Cela permet de recharger des mots de passe chiffrés lors d'opérations de sauvegarde/restauration.

D'anciens clients peuvent ne pas disposer du support pour le mécanisme d'authentification MD5, nécessaire pour travailler avec les mots de passe stockés chiffrés.

`VALID UNTIL '`*`dateheure`*`'`
Cette clause configure la date et l'heure de fin de validité du mot de passe. Sans précision, le mot de passe est indéfiniment valide.

`IN ROLE` *`nom_role`*
Cette clause liste les rôles dont le nouveau rôle est membre. Il n'existe pas d'option pour ajouter le nouveau rôle en tant qu'administrateur ; cela se fait à l'aide d'une commande **GRANT** séparée.

`IN GROUP` *`nom_role`*
`IN GROUP` est un équivalent obsolète de `IN ROLE`.

`ROLE` *`nom_role`*
Cette clause liste les rôles membres du nouveau rôle. Le nouveau rôle devient ainsi un « groupe ».

`ADMIN` *`nom_role`*
Cette clause est équivalente à la clause `ROLE`, à la différence que les rôles nommés sont ajoutés au nouveau rôle avec l'option `WITH ADMIN OPTION`. Cela leur confère le droit de promouvoir à d'autres rôles l'appartenance à celui-ci.

`USER` *`nom_role`*
`USER` est un équivalent osbolète de `ROLE`.

`SYSID` *`uid`*
La clause `SYSID` est ignorée, mais toujours acceptée pour des raisons de compatibilité.

Notes

ALTER ROLE(7) est utilisé pour modifier les attributs d'un rôle, et DROP ROLE(7) pour supprimer un rôle. Tous les attributs positionnés par **CREATE ROLE** peuvent être modifiés par la suite à l'aide de commandes **ALTER ROLE**.

Il est préférable d'utiliser GRANT(7) et REVOKE(7) pour ajouter et supprimer des membres de rôles utilisés comme groupes.

La clause `VALID UNTIL` définit les date et heure d'expiration du mot de passe uniquement, pas du rôle. En particulier, les date et heure d'expiration ne sont pas vérifiées lors de connexions à l'aide de méthodes d'authentification qui n'utilisent pas les mots de passe.

L'attribut `INHERIT` gouverne l'héritage des droits conférables (c'est-à-dire les droits d'accès aux objets de la base de données et les appartenances aux rôles). Il ne s'applique pas aux attributs de rôle spéciaux configurés par **CREATE ROLE** et **ALTER ROLE**. Par exemple, être membre d'un rôle disposant du droit `CREATEDB` ne confère pas automatiquement le droit de création de bases de données, même avec `INHERIT` positionné ; il est nécessaire d'acquérir ce rôle via SET ROLE(7) avant de créer une base de données.

L'attribut `INHERIT` est la valeur par défaut pour des raisons de compatibilité descendante : dans les précédentes versions de PostgreSQL™, les utilisateurs avaient toujours accès à tous les droits des groupes dont ils étaient membres. Toutefois, `NOINHERIT` est plus respectueux de la sémantique spécifiée dans le standard SQL.

Le privilège `CREATEROLE` impose quelques précautions. Il n'y a pas de concept d'héritage des droits pour un tel rôle. Cela signifie qu'un rôle qui ne possède pas un droit spécifique, mais est autorisé à créer d'autres rôles, peut aisément créer un rôle possédant des droits différents des siens (sauf en ce qui concerne la création des rôles superutilisateur). Par exemple, si le rôle « user » a le droit `CREATEROLE` mais pas le droit `CREATEDB`, il peut toujours créer un rôle possédant le droit `CREATEDB`. Il est de ce fait important de considérer les rôles possédant le privilège `CREATEROLE` comme des superutilisateurs en puissance.

PostgreSQL™ inclut un programme, createuser(1) qui possède les mêmes fonctionnalités que **CREATE ROLE** (en fait, il appelle cette commande) et peut être lancé à partir du shell.

L'option `CONNECTION LIMIT` n'est vérifiée qu'approximativement. Si deux nouvelles sessions sont lancées à peu près simultanément alors qu'il ne reste qu'un seul « emplacement » de connexion disponible pour le rôle, il est possible que les deux échouent. De plus, la limite n'est jamais vérifiée pour les superutilisateurs.

Faites attention lorsque vous donnez un mot de passe non chiffré avec cette commande. Le mot de passe sera transmis en clair au serveur. Ce dernier pourrait être tracer dans l'historique des commandes du client ou dans les traces du serveur. Néanmoins, la commande createuser(1) transmet le mot de passe chiffré. De plus, psql(1) contient une commande **\password** que vous pouvez utiliser pour modifier en toute sécurité votre mot de passe.

Exemples

Créer un rôle qui peut se connecter mais sans lui donner de mot de passe :

```
CREATE ROLE jonathan LOGIN;
```

Créer un rôle avec un mot de passe :

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(**CREATE USER** est identique à **CREATE ROLE** mais implique `LOGIN`.)

Créer un rôle avec un mot de passe valide jusqu'à fin 2006. Une seconde après le passage à 2007, le mot de passe n'est plus valide.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2007-01-01';
```

Créer un rôle qui peut créer des bases de données et gérer des rôles :

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Compatibilité

L'instruction **CREATE ROLE** est définie dans le standard SQL. Ce dernier n'impose que la syntaxe

```
CREATE ROLE nom [ WITH ADMIN nom_role ]
```

La possibilité d'avoir plusieurs administrateurs initiaux et toutes les autres options de **CREATE ROLE** sont des extensions PostgreSQL™.

Le standard SQL définit les concepts d'utilisateurs et de rôles mais les considère comme des concepts distincts et laisse la spécification des commandes de définition des utilisateurs à l'implantation de chaque base de données. PostgreSQL™ a pris le parti d'unifier les utilisateurs et les rôles au sein d'une même entité. Ainsi, les rôles ont plus d'attributs optionnels que dans le standard.

Le comportement spécifié par le standard SQL peut être approché en donnant aux utilisateurs l'attribut `NOINHERIT` et aux rôles l'attribut `INHERIT`.

Voir aussi

SET ROLE(7), ALTER ROLE(7), DROP ROLE(7), GRANT(7), REVOKE(7), createuser(1)

Nom

CREATE RULE — Définir une nouvelle règle de réécriture

Synopsis

```
CREATE [ OR REPLACE ] RULE nom AS ON événement
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | commande | ( commande ; commande ... ) }
```

Description

CREATE RULE définit une nouvelle règle sur une table ou une vue. **CREATE OR REPLACE RULE** crée une nouvelle règle ou remplace la règle si elle existe déjà.

Le système de règles de PostgreSQL™ autorise la définition d'actions alternatives sur les insertions, mises à jour ou suppressions dans les tables. Pour résumer, une règle impose des commandes supplémentaires lors de l'exécution d'une instruction sur une table donnée. Une règle `INSTEAD`, au contraire, permet de remplacer une commande par une autre, voire d'empêcher sa réalisation. Ce sont également les règles qui sont utilisées pour implanter les vues.

Une règle est un mécanisme de transformation de commandes, une « macro ». La transformation intervient avant l'exécution de la commande. Pour obtenir une opération qui s'exécute indépendamment pour chaque ligne physique, il faut utiliser des déclencheurs. On trouvera plus d'informations sur le système des règles dans Chapitre 37, Système de règles.

À l'heure actuelle, les règles `ON SELECT` doivent être des règles `INSTEAD` inconditionnelles. Chacune de leurs actions ne peut être constituée que d'une simple commande **SELECT**. Ainsi, une règle `ON SELECT` a pour résultat la transformation effective d'une table en une vue dont le contenu visible est composé des lignes retournées par la commande **SELECT** de la règle ; ce ne sont pas les lignes stockées dans la table (s'il y en a) qui sont retournées. Le création d'une vue à l'aide de la commande **CREATE VIEW** est toujours préférable à la création d'une table réelle associée à une règle `ON SELECT`.

On peut donner l'illusion d'une vue actualisable (« updatable view ») par la définition de règles `ON INSERT`, `ON UPDATE` et `ON DELETE` (ou tout sous-ensemble de celles-ci) pour remplacer les actions de mises à jour de la vue par des mises à jours des tables adéquates. Si vous voulez supporter **INSERT RETURNING**, alors assurez-vous de placer une clause `RETURNING` adéquate à chacune de ces règles.

Il y a quelques chausse-trappes à éviter lors de l'utilisation de règles conditionnelles pour la mise à jour de vues : à chaque action autorisée sur la vue *doit* correspondre une règle `INSTEAD` inconditionnelle. Si la règle est conditionnelle ou n'est pas une règle `INSTEAD`, alors le système rejette toute tentative de mise à jour, ceci afin d'éviter toute action sur la table virtuelle de la vue. Pour gérer tous les cas utiles à l'aide de règles conditionnelles, il convient d'ajouter une règle inconditionnelle `DO INSTEAD NOTHING` afin de préciser au système qu'il ne recevra jamais de demande de mise à jour d'une table virtuelle. La clause `INSTEAD` des règles conditionnelles peut alors être supprimée ;dans les cas où ces règles s'appliquent, l'action `INSTEAD NOTHING` est utilisée. (Néanmoins, cette méthode ne fonctionne pas actuellement avec les requêtes `RETURNING`.)

Paramètres

nom
: Le nom de la règle à créer. Elle doit être distincte du nom de toute autre règle sur la même table. Les règles multiples sur la même table et le même type d'événement sont appliquées dans l'ordre alphabétique des noms.

événement
: `SELECT`, `INSERT`, `UPDATE` ou `DELETE`.

table
: Le nom (éventuellement qualifié du nom du schéma) de la table ou de la vue sur laquelle s'applique la règle.

condition
: Toute expression SQL conditionnelle (renvoyant un type boolean). L'expression de la condition ne peut pas faire référence à une table autre que `NEW` ou `OLD` ni contenir de fonction d'agrégat.

`INSTEAD`
: Les commandes sont exécutées *à la place de* la commande originale.

`ALSO`
: Les commandes sont exécutées *en plus de* la commande originale.

  En l'absence de `ALSO` et de `INSTEAD`, `ALSO` est utilisé par défaut.

commande
Commande(s) réalisant l'action de la règle. Les commandes valides sont **SELECT**, **INSERT**, **UPDATE**, **DELETE** ou **NOTIFY**.

À l'intérieur d'une *condition* ou d'une *commande*, les noms des tables spéciales `NEW` et `OLD` peuvent être utilisés pour faire référence aux valeurs de la table référencée. `NEW` peut être utilisé dans les règles `ON INSERT` et `ON UPDATE` pour faire référence à la nouvelle ligne lors d'une insertion ou à la nouvelle valeur de la ligne lors d'une mise à jour. `OLD` est utilisé dans les règles `ON UPDATE` et `ON DELETE` pour référencer la ligne existant avant modification ou suppression.

Notes

Vous devez être le propriétaire de la table à créer ou sur laquelle vous ajoutez des règles.

Dans une règle pour l'action `INSERT`, `UPDATE` ou `DELETE` sur une vue, vous pouvez ajouter une clause `RETURNING` qui émet les colonnes de la vue. Cette clause sera utilisée pour calculer les sorties si la règle est déclenchée respectivement par une commande **INSERT RETURNING**, **UPDATE RETURNING**ou **DELETE RETURNING**. Quand la règle est déclenchée par une commande sans clause `RETURNING`, la clause `RETURNING` de la règle est ignorée. L'implémentation actuelle autorise seulement des règles `INSTEAD` sans condition pour contenir `RETURNING` ; de plus, il peut y avoir au plus une clause `RETURNING` parmi toutes les règles pour le même événement. (Ceci nous assure qu'il y a seulement une clause `RETURNING` candidate utilisée pour calculer les résultats.) Les requêtes `RETURNING` sur la vue seront rejetées s'il n'existe pas de clause `RETURNING` dans une des règles disponibles.

Une attention particulière doit être portée aux règles circulaires. Ainsi dans l'exemple suivant, bien que chacune des deux définitions de règles soit acceptée par PostgreSQL™, la commande **SELECT** produira une erreur à cause de l'expansion récursive de la règle :

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

Actuellement, si l'action d'une règle contient une commande **NOTIFY**, cette commande est exécutée sans condition, c'est-à-dire que **NOTIFY** est déclenché même si la règle ne s'applique à aucune ligne. Par exemple, dans :

```
CREATE RULE notify_me AS ON UPDATE TO matable DO ALSO NOTIFY matable;

UPDATE matable SET name = 'foo' WHERE id = 42;
```

un événement **NOTIFY** est lancé durant un **UPDATE**, qu'il y ait ou non des lignes satisfaisant la condition `id = 42`. Cette restriction pourrait être corrigée dans les prochaines versions.

Compatibilité

CREATE RULE est une extension PostgreSQL™, tout comme l'est le système complet de réécriture de requêtes.

Nom

CREATE SCHEMA — Définir un nouveau schéma

Synopsis

```
CREATE SCHEMA nom_schéma [ AUTHORIZATION nom_utilisateur ] [ élément_schéma [ ... ] ]
CREATE SCHEMA AUTHORIZATION nom_utilisateur [ élément_schéma [ ... ] ]
```

Description

CREATE SCHEMA crée un nouveau schéma dans la base de données. Le nom du schéma doit être unique au sein de la base de données.

Un schéma est essentiellement un espace de noms : il contient des objets nommés (tables, types de données, fonctions et opérateurs) dont les noms peuvent être identiques à ceux d'objets d'autres schémas. Les objets nommés sont accessibles en préfixant leur nom de celui du schéma (on dit alors que le nom est « qualifié » du nom du schéma), ou par la configuration d'un chemin de recherche incluant le(s) schéma(s) désiré(s). Une commande **CREATE** qui spécifie un objet non qualifié crée l'objet dans le schéma courant (le premier dans le chemin de recherche, obtenu par la fonction `current_schema`).

CREATE SCHEMA peut éventuellement inclure des sous-commandes de création d'objets dans le nouveau schéma. Les sous-commandes sont traitées à la façon de commandes séparées lancées après la création du schéma. La différence réside dans l'utilisation de la clause `AUTHORIZATION`. Dans ce cas, l'utilisateur est propriétaire de tous les objets créés.

Paramètres

`nom_schéma`
: Le nom du schéma à créer. S'il est oublié, le paramètre *`nomutilisateur`* est utilisé comme nom de schéma. Le nom ne peut pas débuter par `pg_`, ces noms étant réservés aux schémas du système.

`nom_utilisateur`
: Le nom de l'utilisateur à qui appartient le schéma. Par défaut, il s'agit de l'utilisateur qui exécute la commande. Pour créer un schéma dont le propriétaire est un autre rôle, vous devez être un membre direct ou indirect de ce rôle, ou être un superutilisateur.

`élément_schéma`
: Une instruction SQL qui définit un objet à créer dans le schéma. À ce jour, seules **CREATE TABLE**, **CREATE VIEW**, **CREATE SEQUENCE**, **CREATE TRIGGER** et **GRANT** peuvent être utilisées dans la commande **CREATE SCHEMA**. Les autres types d'objets sont créés dans des commandes séparées après la création du schéma.

Notes

Pour créer un schéma, l'utilisateur doit avoir le droit `CREATE` sur la base de données. (Les superutilisateurs contournent cette vérification.)

Exemples

Créer un schéma :

```
CREATE SCHEMA mon_schema;
```

Créer un schéma pour l'utilisateur `joe`, schéma nommé `joe` :

```
CREATE SCHEMA AUTHORIZATION joe;
```

Créer un schéma et lui ajouter une table et une vue :

```
CREATE SCHEMA hollywood
    CREATE TABLE films (titre text, sortie date, recompenses text[])
    CREATE VIEW gagnants AS
        SELECT titre, sortie FROM films WHERE recompenses IS NOT NULL;
```

Les sous-commandes ne sont pas terminées par un point-virgule.

La même chose, autre écriture :

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (titre text, sortie date, recompenses text[]);
CREATE VIEW hollywood.gagnants AS
    SELECT titre, sortie FROM hollywood.films WHERE recompenses IS NOT NULL;
```

Compatibilité

Le standard SQL autorise une clause `DEFAULT CHARACTER SET` dans **CREATE SCHEMA**, et des types de sous-commandes en plus grand nombre que ceux supportés actuellement par PostgreSQL™.

Le standard SQL n'impose pas d'ordre d'apparition des sous-commandes dans **CREATE SCHEMA**. L'implantation actuelle de PostgreSQL™ ne gère pas tous les cas de références futures dans les sous-commandes. Il peut s'avérer nécessaire de réordonner les sous-commandes pour éviter ces références.

Dans le standard SQL, le propriétaire d'un schéma est également propriétaire de tous les objets qui s'y trouvent. PostgreSQL™ permet à un schéma de contenir des objets qui n'appartiennent pas à son propriétaire. Cela n'est possible que si le propriétaire du schéma transmet le privilège `CREATE` sur son schéma ou si un superutilisateur choisir d'y créer des objets.

Voir aussi

ALTER SCHEMA(7), DROP SCHEMA(7)

Nom

CREATE SEQUENCE — Définir un nouveau générateur de séquence

Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE nom [ INCREMENT [ BY ] incrément ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
    [ OWNED BY { table.colonne | NONE } ]
```

Description

CREATE SEQUENCE crée un nouveau générateur de séquence de nombres. Cela implique la création et l'initialisation d'une nouvelle table à une seule ligne nommée *nom*. Le générateur appartient à l'utilisateur qui exécute la commande.

Si un nom de schéma est donné, la séquence est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Les séquences temporaires existent dans un schéma spécial, il n'est donc pas utile de préciser un nom de schéma lors de la création d'une séquence temporaire. Le nom de la séquence doit être distinct du nom de toute autre séquence, table, index ou vue du schéma.

Après la création d'une séquence, les fonctions `nextval`, `currval` et `setval` sont utilisées pour agir sur la séquence. Ces fonctions sont documentées dans Section 9.15, « Fonctions de manipulation de séquences ».

Bien qu'il ne soit pas possible de mettre à jour une séquence en accédant directement à la table, une requête telle que :

```
SELECT * FROM nom;
```

peut être utilisée pour examiner les paramètres et l'état courant d'une séquence. En particulier, le champ `last_value` affiche la dernière valeur allouée par une session. (Cette valeur peut être rendue obsolète à l'affichage par des appels effectifs de `nextval` dans des sessions concurrentes.)

Paramètres

`TEMPORARY` ou `TEMP`
: Si ce paramètre est spécifié, l'objet séquence n'est créé que pour la session en cours et est automatiquement supprimé lors de la sortie de session. Les séquences permanentes portant le même nom ne sont pas visibles (dans cette session) tant que la séquence temporaire existe, sauf à être référencées par les noms qualifiés du schéma.

nom
: Le nom (éventuellement qualifié du nom du schéma) de la séquence à créer.

incrément
: La clause optionnelle `INCREMENT BY` *incrément* précise la valeur à ajouter à la valeur courante de la séquence pour créer une nouvelle valeur. Une valeur positive crée une séquence ascendante, une valeur négative une séquence descendante. 1 est la valeur par défaut.

valeurmin, `NO MINVALUE`
: La clause optionnelle `MINVALUE` *valeurmin* détermine la valeur minimale de la séquence. Si cette clause n'est pas fournie ou si `NO MINVALUE` est spécifié, alors les valeurs par défaut sont utilisées. Ces valeurs sont, respectivement, 1 et -2^{63}-1 pour les séquences ascendantes et descendantes.

valeurmax, `NO MAXVALUE`
: La clause optionnelle `MAXVALUE` *valeurmax* détermine la valeur maximale de la séquence. Si cette clause n'est pas fournie ou si `NO MAXVALUE` est specifié, alors les valeurs par défaut sont utilisées. Ces valeurs sont, respectivement, 2^{63}-1 et -1 pour les séquences ascendantes et descendantes.

début
: La clause optionnelle `START WITH` *début* permet à la séquence de démarrer n'importe où. La valeur de début par défaut est *valeurmin* pour les séquences ascendantes et *valeurmax* pour les séquences descendantes.

cache
: La clause optionnelle `CACHE` *cache* spécifie le nombre de numéros de séquence à préallouer et stocker en mémoire pour un accès plus rapide. 1 est la valeur minimale (une seule valeur est engendrée à la fois, soit pas de cache) et la valeur par dé-

faut.

`CYCLE`, `NO CYCLE`
: L'option `CYCLE` autorise la séquence à recommencer au début lorsque *`valeurmax`* ou *`valeurmin`* sont atteintes, respectivement, par une séquence ascendante ou descendante. Si la limite est atteinte, le prochain nombre engendré est respectivement *`valeurmin`* ou *`valeurmax`*.

  Si `NO CYCLE` est spécifié, tout appel à `nextval` alors que la séquence a atteint la valeur maximale (dans le cas d'une séquence ascendante) ou la valeur minimale (dans l'autre cas) retourne une erreur. En l'absence de précision, `NO CYCLE` est la valeur par défaut.

`OWNED BY` *`table.colonne`*, `OWNED BY NONE`
: L'option `OWNED BY` permet d'associer la séquence à une colonne de table spécifique. De cette façon, la séquence sera automatiquement supprimée si la colonne (ou la table entière) est supprimée. La table indiquée doit avoir le même propriétaire et être dans le même schéma que la séquence. `OWNED BY NONE`, valeur par défaut, indique qu'il n'y a pas d'association.

Notes

DROP SEQUENCE est utilisé pour supprimer une séquence.

Les séquences sont fondées sur l'arithmétique bigint, leur échelle ne peut donc pas excéder l'échelle d'un entier sur huit octets (-9223372036854775808 à 9223372036854775807). Sur certaines vieilles plateformes, il peut ne pas y avoir de support compilateur pour les entiers codés sur huit octets. Dans ce cas les séquences utilisent l'arithmétique des entiers traditionnels (type integer) (de -2147483648 à +2147483647).

Des résultats inattendus peuvent être obtenus dans le cas d'un paramétrage de *`cache`* supérieur à un pour une séquence utilisée concurrentiellement par plusieurs sessions. Chaque session alloue et cache des valeurs de séquences successives lors d'un accès à la séquence et augmente en conséquence la valeur de `last_value`. Les *`cache`*-1 appels suivants de `nextval` au cours de la session session retourne simplement les valeurs préallouées sans toucher à la séquence. De ce fait, tout nombre alloué mais non utilisé au cours d'une session est perdu à la fin de la session, créant ainsi des « trous » dans la séquence.

De plus, bien qu'il soit garanti que des sessions différentes engendrent des valeurs de séquence distinctes, si l'on considère toutes les sessions, les valeurs peuvent ne pas être engendrées séquentiellement. Par exemple, avec un paramétrage du *`cache`* à 10, la session A peut réserver les valeurs 1..10 et récupérer `nextval`=1 ; la session B peut alors réserver les valeurs 11..20 et récupérer `nextval`=11 avant que la session A n'ait engendré `nextval`=2. De ce fait, un paramétrage de *`cache`* à un permet d'assumer que les valeurs retournées par `nextval` sont engendrées séquentiellement ; avec un *`cache`* supérieur, on ne peut qu'assumer que les valeurs retournées par `nextval` sont tous distinctes, non qu'elles sont réellement engendrées séquentiellement. De plus, `last_value` reflète la dernière valeur réservée pour toutes les sessions, que `nextval` ait ou non retourné cette valeur.

D'autre part, `setval` exécuté sur une telle séquence n'est pas pris en compte par les autres sessions avant qu'elle n'aient utilisé toutes les valeurs préallouées et cachées.

Exemples

Créer une séquence ascendante appelée `serie`, démarrant à 101 :

```
CREATE SEQUENCE serie START 101;
```

Sélectionner le prochain numéro de cette séquence :

```
SELECT nextval('serie');

 nextval
---------
     101
```

Récupérer le prochain numéro d'une séquence :

```
SELECT nextval('serial');

 nextval
---------
     102
```

Utiliser cette séquence dans une commande **INSERT** :

```
INSERT INTO distributors VALUES (nextval('serie'), 'nothing');
```

Mettre à jour la valeur de la séquence après un **COPY FROM** :

```
BEGIN;
COPY distributeurs FROM 'fichier_entrees';
SELECT setval('serie', max(id)) FROM distributeurs;
END;
```

Compatibilité

CREATE SEQUENCE est conforme au standard SQL, exception faites des remarques suivantes :

- L'expression standard `AS <type donnée>` n'est pas supportée.
- Obtenir la prochaine valeur se fait en utilisant la fonction `nextval()` au lieu de l'expression standard **NEXT VALUE FOR**.
- La clause `OWNED BY` est une extension PostgreSQL™.

Voir aussi

ALTER SEQUENCE(7), DROP SEQUENCE(7)

Nom

CREATE SERVER — Définir un nouveau serveur distant

Synopsis

```
CREATE SERVER nom_serveur [ TYPE 'type_serveur' ] [ VERSION 'version_serveur' ]
    FOREIGN DATA WRAPPER nom_fdw
    [ OPTIONS ( option 'valeur' [, ... ] ) ]
```

Description

CREATE SERVER définit un nouveau serveur de données distantes. L'utilisateur qui définit le serveur devient son propriétaire.

Un serveur distant englobe typiquement des informations de connexion qu'un wrapper de données distantes utilise pour accéder à une ressource externe de données. Des informations de connexions supplémentaires spécifiques à l'utilisateur pourraient être fournies par l'intermédiaire des correspondances d'utilisateur.

Le nom du serveur doit être unique dans la base de données.

La création d'un serveur nécessite d'avoir le droit `USAGE` sur le wrapper de données distant qui est utilisé.

Paramètres

`nom_serveur`
: Nom du serveur de données distant qui sera créé.

`type_serveur`
: Type de serveur (optionnel).

`version_serveur`
: Version du serveur (optionnel).

`nom_fdw`
: Nom du wrapper de données distantes qui gère le serveur.

`OPTIONS ( option 'valeur' [, ... ] )`
: Cette clause spécifie les options pour le serveur. Typiquement, les options définissent les détails de connexion au serveur, mais les noms et valeurs réelles dépendent du wrapper de données distantes du serveur.

Notes

Lors de l'utilisation du module dblink (voir Section F.8, « dblink »), le nom du serveur distant peut être utilisé comme argument de la fonction dblink_connect(3) pour indiquer les paramètres de connexion. Voir aussi ici pour plus d'exemples. Il est nécessaire de disposer du droit `USAGE` sur le serveur distant pour être capable de l'utiliser de cette façon.

Exemples

Créer un serveur `truc` qui utilise le wrapper de données distantes inclus `default` :

```
CREATE SERVER truc FOREIGN DATA WRAPPER "default";
```

Créer un serveur `monserveur` qui utilise le wrapper de données distantes `pgsql` :

```
CREATE SERVER monserveur FOREIGN DATA WRAPPER pgsql OPTIONS (host 'truc', dbname
'trucdb', port '5432');
```

Compatibilité

CREATE SERVER est conforme à ISO/IEC 9075-9 (SQL/MED).

Voir aussi

ALTER SERVER(7), DROP SERVER(7), CREATE FOREIGN DATA WRAPPER(7), CREATE USER MAPPING(7)

Nom

CREATE TABLE — Définir une nouvelle table

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nom_table ( [
  { nom_colonne type_donnees [ DEFAULT default_expr ] [ contrainte_colonne [ ... ] ]
    | contrainte_table
    | LIKE table_parent [ option_like ... ] }
    [, ... ]
] )
[ INHERITS ( table_parent [, ... ] ) ]
[ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nom_table
    OF nom_type [ (
  { nom_colonne WITH OPTIONS [ DEFAULT default_expr ] [ contrainte_colonne [ ... ] ]
    | contrainte_table }
    [, ... ]
) ]
[ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]

où contrainte_colonne
peut être :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL |
  CHECK ( expression ) |
  UNIQUE parametres_index |
  PRIMARY KEY parametres_index |
  EXCLUDE [ USING methode_index ] ( élément_exclude WITH opérateur [, ... ] )
paramètres_index [ WHERE ( prédicat ) ] |
  REFERENCES table_reference [ ( colonne_reference ) ] [ MATCH FULL
| MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

et option_like peut
valoir :

{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |
ALL }

et contrainte_table :

[ CONSTRAINT nom_contrainte ]
{ UNIQUE ( nom_colonne [, ... ] ) parametres_index |
  PRIMARY KEY ( nom_colonne [, ... ] ) parametres_index |
  CHECK ( expression ) |
  FOREIGN KEY ( nom_colonne [, ...
] ) REFERENCES table_reference [ (
colonne_reference [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE
action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

Les paramètres_index dans les
contraintes UNIQUE, PRIMARY KEY et
EXCLUDE sont :

[ WITH ( paramètre_stockage [= valeur] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace ]
```

```
exclude_element dans une
contrainte EXCLUDE peut valoir :

{ column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

Description

CREATE TABLE crée une nouvelle table initialement vide dans la base de données courante. La table appartient à l'utilisateur qui exécute cette commande.

Si un nom de schéma est donné (par exemple, `CREATE TABLE monschema.matable ...`), alors la table est créée dans le schéma spécifié. Dans le cas contraire, elle est créée dans le schéma courant. Les tables temporaires existent dans un schéma spécial, il n'est donc pas nécessaire de fournir un nom de schéma lors de la création d'une table temporaire. Le nom de la table doit être distinct du nom des autres tables, séquences, index ou vues dans le même schéma.

CREATE TABLE crée aussi automatiquement un type de données qui représente le type composé correspondant à une ligne de la table. Ainsi, les tables doivent avoir un nom distinct de tout type de données du même schéma.

Les clauses de contrainte optionnelles précisent les contraintes (ou tests) que les nouvelles lignes ou les lignes mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Une contrainte est un objet SQL qui aide à définir l'ensemble des valeurs valides de différentes façons.

Il existe deux façons de définir des contraintes : celles de table et celles de colonnes. Une contrainte de colonne fait partie de la définition de la colonne. Une définition de contrainte de tables n'est pas liée à une colonne particulière et peut englober plusieurs colonnes. Chaque contrainte de colonne peut être écrite comme une contrainte de table ; une contrainte de colonne n'est qu'un outil de notation utilisé lorsque la contrainte n'affecte qu'une colonne.

Paramètres

`TEMPORARY` ou `TEMP`

La table est temporaire. Les tables temporaires sont automatiquement supprimées à la fin d'une session ou, optionnellement, à la fin de la transaction en cours (voir **ON COMMIT** ci-dessous). Les tables permanentes qui portent le même nom ne sont pas visibles dans la session courante tant que la table temporaire existe sauf s'il y est fait référence par leur nom qualifié du schéma. Tous les index créés sur une table temporaire sont automatiquement temporaires.

Le démon autovacuum ne peut pas accéder et, du coup, ne peut pas exécuter un VACUUM ou un ANALYZE sur les tables temporaires. Pour cette raison, les opérations VACUUM et ANALYZE doivent être traitées via des commandes SQL de session. Par exemple, si une table temporaire doit être utilisée dans des requêtes complexes, il est raisonnable d'exécuter **ANALYZE** sur la table temporaire après qu'elle ait été peuplée.

On peut éventuellement écrire `GLOBAL` ou `LOCAL` avant `TEMPORARY` ou `TEMP`. Cela ne fait pas de différence dans PostgreSQL™ (cf. la section intitulée « Compatibilité »).

`nom_table`

Le nom (éventuellement qualifié du nom du schéma) de la table à créer.

`OF` *`nom_type`*

Crée une *table typée*, qui prend sa structure à partir du type composite spécifié (son nom peut être qualifié du schéma). Une table typée est liée à son type ; par exemple, la table sera supprimée si le type est supprimé (avec `DROP TYPE ... CASCADE`).

Quand une table typée est créée, les types de données des colonnes sont déterminés par le type composite sous-jacent et ne sont pas indiqués par la commande `CREATE TABLE`. Mais la commande `CREATE TABLE` peut ajouter des valeurs par défaut et des contraintes à la table. Elle peut aussi indiquer des paramètres de stockage.

`nom_colonne`

Le nom d'une colonne de la nouvelle table.

`type_données`

Le type de données de la colonne. Cela peut inclure des spécificateurs de tableaux. Pour plus d'informations sur les types de données supportés par PostgreSQL™, on se référera à Chapitre 8, Types de données.

`DEFAULT` *`default_expr`*

La clause `DEFAULT`, apparaissant dans la définition d'une colonne, permet de lui affecter une valeur par défaut. La valeur est une expression libre de variable (les sous-requêtes et références croisées aux autres colonnes de la table courante ne sont pas autorisées). Le type de données de l'expression par défaut doit correspondre au type de données de la colonne.

L'expression par défaut est utilisée dans les opérations d'insertion qui ne spécifient pas de valeur pour la colonne. S'il n'y a pas

de valeur par défaut pour une colonne, elle est NULL.

`INHERITS ( table_parent [, ... ])`

La clause optionnelle `INHERITS` indique une liste de tables dont les colonnes sont automatiquement héritées par la nouvelle table.

L'utilisation d'`INHERITS` crée une relation persistante entre la nouvelle table enfant et sa table parent. Les modifications de schéma du(des) parent(s) se propagent normalement aux enfants et, par défaut, les données de la table enfant sont incluses dans les parcours de(s) parent(s).

Si un même nom de colonne existe dans plusieurs tables parentes, une erreur est rapportée, à moins que les types de données des colonnes ne correspondent dans toutes les tables parentes. S'il n'y a pas de conflit, alors les colonnes dupliquées sont assemblées pour former une seule colonne dans la nouvelle table. Si la liste des noms de colonnes de la nouvelle table contient un nom de colonne hérité, le type de données doit correspondre à celui des colonnes héritées et les définitions des colonnes sont fusionnées. Si la nouvelle table spécifie explicitement une valeur par défaut pour la colonne, cette valeur surcharge toute valeur par défaut héritée. Dans le cas contraire, les parents qui spécifient une valeur par défaut doivent tous spécifier la même, sans quoi une erreur est rapportée.

Les contraintes `CHECK` sont fusionnées, dans les grandes lignes, de la même façon que les colonnes : si des tables parentes multiples et/ou la nouvelle définition de table contient des contraintes `CHECK` de même nom, ces contraintes doivent toutes avoir la même expression de vérification, ou une erreur sera retournée. Les contraintes qui ont le même nom et la même expression seront fusionnées en une seule. Notez qu'une contrainte `CHECK` non nommée dans la nouvelle table ne sera jamais fusionnée puisqu'un nom unique lui sera toujours affecté.

Les paramètres `STORAGE` de la colonne sont aussi copiés des tables parents.

`LIKE table_parent [ option_like ... ]`

La clause `LIKE` spécifie une table à partir de laquelle la nouvelle table copie automatiquement tous les noms de colonnes, leur types de données et les contraintes non NULL.

Contrairement à `INHERITS`, la nouvelle table et la table originale sont complètement découplées à la fin de la création. Les modifications sur la table originale ne sont pas appliquées à la nouvelle table et les données de la nouvelle table sont pas prises en compte lors du parcours de l'ancienne table.

Les expressions par défaut des définitions de colonnes ne seront copiées que si `INCLUDING DEFAULTS` est spécifié. Le comportement par défaut les exclut, ce qui conduit à des valeurs par défaut NULL pour les colonnes copiées de la nouvelle table.

Les contraintes NOT NULL sont toujours copiées sur la nouvelle table. Les contraintes `CHECK` seront seulement copiées si `INCLUDING CONSTRAINTS` est indiqué ; les autres types de contraintes ne sont jamais copiées. De plus, aucune distinction n'est faite entre les contraintes de colonnes et les contraintes de tables -- quand les contraintes sont demandées, toutes les contraintes de vérification sont copiées.

Tout index sur la table originale ne sera pas créé sur la nouvelle table sauf si la clause `INCLUDING INDEXES` est précisée.

Des paramètres `STORAGE` pour les définitions de la colonne copiée seront seulement copiés si `INCLUDING STORAGE` est spécifié. Le comportement par défaut est d'exclure des paramètres `STORAGE`, résultant dans les colonnes copiées dans la nouvelle table ayant des paramètres par défaut spécifiques par type. Pour plus d'informations sur `STORAGE`, voir Section 54.2, « TOAST ».

Les commentaires pour les colonnes, contraintes et index copiés seront seulement copiés si `INCLUDING COMMENTS` est spécifié. Le comportement par défaut est d'exclure les commentaires, ce qui résulte dans des colonnes et contraintes copiées dans la nouvelle table mais sans commentaire.

`INCLUDING ALL` est une forme abrégée de `INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS`.

Contrairement à `INHERITS`, les colonnes et les contraintes copiées par `LIKE` ne sont pas assemblées avec des colonnes et des contraintes nommées de façon similaire. Si le même nom est indiqué explicitement ou dans une autre clause `LIKE`, une erreur est rapportée.

`CONSTRAINT nom_contrainte`

Le nom optionnel d'une contrainte de colonne ou de table. Si la contrainte est violée, le nom de la contrainte est présente dans les messages d'erreur. Donc les noms de contraintes comme `col doit être positive` peut être utilisés pour communiquer des informations utiles aux applications clients. (Des doubles guillemets sont nécessaires pour indiquer les noms des contraintes qui contiennent des espaces.) Si un nom de contrainte n'est pas donné, le système en crée un.

`NOT NULL`

Interdiction des valeurs NULL dans la colonne.

`NULL`

Les valeurs NULL sont autorisées pour la colonne. Comportement par défaut.

Cette clause n'est fournie que pour des raisons de compatibilité avec les bases de données SQL non standard. Son utilisation n'est pas encouragée dans les nouvelles applications.

`CHECK ( expression )`

La clause `CHECK` spécifie une expression de résultat booléen que les nouvelles lignes ou celles mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Les expressions de résultat TRUE ou UNKNOWN réussissent. Si une des lignes de l'opération d'insertion ou de mise à jour produit un résultat FALSE, une exception est levée et la base de données n'est pas modifiée. Une contrainte de vérification sur une colonne ne fait référence qu'à la valeur de la colonne tandis qu'une contrainte sur la table fait référence à plusieurs colonnes.

Actuellement, les expressions `CHECK` ne peuvent ni contenir des sous-requêtes ni faire référence à des variables autres que les colonnes de la ligne courante.

`UNIQUE` (contrainte de colonne), `UNIQUE ( nom_colonne [, ... ] )` (contrainte de table)

La contrainte `UNIQUE` indique qu'un groupe d'une ou plusieurs colonnes d'une table ne peut contenir que des valeurs uniques. Le comportement de la contrainte de table unique est le même que celui des contraintes de colonnes avec la capacité supplémentaire de traiter plusieurs colonnes.

Pour une contrainte unique, les valeurs NULL ne sont pas considérées comme égales.

Chaque contrainte unique de table doit nommer un ensemble de colonnes qui est différent de l'ensemble de colonnes nommées par toute autre contrainte unique ou de clé primaire définie pour la table. (Sinon elle ne ferait que lister la même contrainte deux fois.)

`PRIMARY KEY` (contrainte de colonne), `PRIMARY KEY ( nom_colonne [, ... ] )` (contrainte de table)

La contrainte de clé primaire spécifie qu'une ou plusieurs colonnes d'une table ne peut contenir que des valeurs uniques (non dupliquées) et non NULL. Techniquement, une contrainte `PRIMARY KEY` est tout simplement une combinaison d'une contrainte `UNIQUE` et d'une contrainte `NOT NULL`, mais l'identification d'un ensemble de colonnes en tant que clé primaire fournit aussi une métainformation sur le schéma car la clé primaire indique que les autres tables peuvent se baser sur cet ensemble de colonnes comme d'un identifiant unique des lignes.

Seule une clé primaire peut être spécifiée pour une table, soit en tant que contrainte de colonne ou que contrainte de table.

La contrainte de clé primaire devrait nommer un ensemble de colonnes qui est différent des autres ensembles de colonnes nommées par une contrainte unique définie pour la même table.

`EXCLUDE [ USING méthode_index ] ( élément_exclusion WITH opérateur [, ... ] ) paramètres_index [ WHERE ( prédicat ) ]`

La clause `EXCLUDE` définit une contrainte d'exclusion qui garantit que si deux lignes sont comparées sur la ou les colonnes spécifiées ou des expressions utilisant le ou les opérateurs spécifiés, seulement certaines de ces comparaisons, mais pas toutes, renverront `TRUE`. Si tous les opérateurs spécifiés testent une égalité, ceci est équivalent à une contrainte `UNIQUE` bien qu'une contrainte unique ordinaire sera plus rapide. Néanmoins, ces contraintes d'exclusion peuvent spécifier des contraintes qui sont plus générales qu'une simple égalité. Par exemple, vous pouvez spécifier qu'il n'y a pas deux lignes dans la table contenant des cercles de surcharge (voir Section 8.8, « Types géométriques ») en utilisant l'opérateur `&&`.

Des contraintes d'exclusion sont implantées en utilisant un index, donc chaque opérateur précisé doit être associé avec une classe d'opérateurs appropriée (voir Section 11.9, « Classes et familles d'opérateurs ») pour la méthode d'accès par index, nommée *`méthode_index`*. Les opérateurs doivent être commutatifs. Chaque *`élément_exclusion`* peut spécifier en option une classe d'opérateur et/ou des options de tri ; ils sont décrits complètement sous CREATE INDEX(7).

La méthode d'accès doit supporter `amgettuple` (voir Chapitre 51, Définition de l'interface des méthodes d'accès aux index) ; dès à présent, cela signifie que GIN ne peut pas être utilisé. Bien que cela soit autorisé, il existe peu de raison pour utiliser des index B-tree ou hash avec une contrainte d'exclusion parce que cela ne fait rien de mieux que ce que peut faire une contrainte unique ordinaire. Donc, en pratique, la méthode d'accès sera toujours GiST.

Le *`prédicat`* vous permet de spécifier une contrainte d'exclusion sur un sous-ensemble de la table ; en interne, un index partiel est créé. Notez que ces parenthèses sont requis autour du prédicat.

`REFERENCES table_reference [ ( colonne_reference ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (contrainte de colonne), `FOREIGN KEY ( colonne [, ... ] ) REFERENCES table_reference [ ( colonne_reference [, ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ]` (contrainte de colonne)

Ces clauses spécifient une contrainte de clé étrangère. Cela signifie qu'un groupe de colonnes de la nouvelle table ne peut contenir que des valeurs correspondant à celles des colonnes de référence de la table de référence. Si *`colonne_reference`* est omis, la clé primaire de la *`table_reference`* est utilisée. Les colonnes référencées doivent être celles d'une contrainte d'unicité ou de clé primaire, non déferrable, dans la table référencée. Les contraintes de type clé étran-

gère ne peuvent pas être définies entre des tables temporaires et des tables permanentes.

Une valeur insérée dans les colonnes de la nouvelle table est comparée aux valeurs des colonnes de référence dans la table de référence à l'aide du type de concordance fourni. Il existe trois types de correspondance : `MATCH FULL` (NDT : correspondance totale), `MATCH PARTIAL` (NDT : correspondance partielle) et `MATCH SIMPLE` (NDT : correspondance simple), qui est aussi la valeur par défaut. `MATCH FULL` n'autorise une colonne d'une clé étrangère composite à être NULL que si l'ensemble des colonnes de la clé étrangère sont NULL. `MATCH SIMPLE` autorise une colonne de clé étrangère à être NULL même si les autres parties de la clé étrangère ne sont pas nulles. `MATCH PARTIAL` n'est pas encore implanté.

Lorsque les données des colonnes référencées sont modifiées, des actions sont réalisées sur les données de la table référençant. La clause `ON DELETE` spécifie l'action à réaliser lorsqu'une ligne référencée de la table de référence est supprimée. De la même façon, la clause `ON UPDATE` spécifie l'action à réaliser lorsqu'une colonne référencée est mise à jour. Si la ligne est mise à jour sans que la valeur de la colonne référencée ne soit modifiée, aucune action n'est réalisée. Les actions référentielles autres que la vérification `NO ACTION` ne peuvent pas être différées même si la contrainte est déclarée retardable. Les actions suivantes sont possibles pour chaque clause :

`NO ACTION`
Une erreur est produite pour indiquer que la suppression ou la mise à jour entraîne une violation de la contrainte de clé étrangère. Si la contrainte est différée, cette erreur est produite au moment de la vérification, si toutefois il existe encore des lignes de référence. C'est le comportement par défaut.

`RESTRICT`
Une erreur est produite pour indiquer que la suppression ou la mise à jour entraîne une violation de la contrainte de clé étrangère. Ce comportement est identique à `NO ACTION`, si ce n'est que la vérification n'est pas décalable dans le temps.

`CASCADE`
La mise à jour ou la suppression de la ligne de référence est propagée à l'ensemble des lignes qui la référencent, qui sont, respectivement, mises à jour ou supprimées.

`SET NULL`
La valeur de la colonne qui référence est positionnée à NULL.

`SET DEFAULT`
La valeur de la colonne qui référence est positionnée à celle par défaut.

Si les colonnes référencées sont modifiées fréquemment, il est conseillé d'ajouter un index sur la colonne de clé étrangère de façon à accélérer les actions référentielles associées à la colonne de clé étrangère.

`DEFERRABLE`, `NOT DEFERRABLE`
Ces clauses contrôlent la possibilité de différer la contrainte. Une contrainte qui n'est pas décalable dans le temps est vérifiée immédiatement après chaque commande. La vérification des contraintes décalables est repoussée à la fin de la transaction (à l'aide de la commande SET CONSTRAINTS(7)). `NOT DEFERRABLE` est la valeur par défaut. Actuellement, seules les contraintes `UNIQUE`, `PRIMARY KEY`, `EXCLUDE` et `REFERENCES` (clé étrangère) acceptent cette clause. Les contraintes`NOT NULL` et `CHECK` ne sont pas déferrables.

`INITIALLY IMMEDIATE`, `INITIALLY DEFERRED`
Si une contrainte est décalable dans le temps, cette clause précise le moment de la vérification. Si la contrainte est `INITIALLY IMMEDIATE`, elle est vérifiée après chaque instruction. Si la contrainte est `INITIALLY DEFERRED`, elle n'est vérifiée qu'à la fin de la transaction. Le moment de vérification de la contrainte peut être modifié avec la commande SET CONSTRAINTS(7).

`WITH ( paramètre_stockage [= valeur] [, ... ] )`
Cette clause spécifie les paramètres de stockage optionnels pour une table ou un index ; voir la section intitulée « Paramètres de stockage » pour plus d'informations. La clause `WITH` peut aussi inclure pour une table `OIDS=TRUE` (ou simplement `OIDS`) pour indiquer que les lignes de la nouvelle table doivent se voir affecter des OID (identifiants d'objets) ou `OIDS=FALSE` pour indiquer que les lignes ne doivent pas avoir d'OID. Si `OIDS` n'est pas indiqué, la valeur par défaut dépend du paramètre de configuration default_with_oids. (Si la nouvelle table hérite d'une table qui a des OID, alors`OIDS=TRUE` est forcé même si la commande précise `OIDS=FALSE`.)

Si `OIDS=FALSE` est indiqué ou implicite, la nouvelle table ne stocke pas les OID et aucun OID n'est affecté pour une ligne insérée dans cette table. Ceci est généralement bien considéré car cela réduit la consommation des OID et retarde du coup le retour à zéro du compteur sur 32 bits. Une fois que le compteur est revenu à zéro, les OID ne sont plus considérés uniques ce qui les rend beaucoup moins utiles. De plus, exclure les OID d'une table réduit l'espace requis pour stocker la table sur le disque de quatre octets par ligne (la plupart des machines), améliorant légèrement les performances.

Pour supprimer les OID d'une table une fois qu'elle est créée, utilisez ALTER TABLE(7).

`WITH OIDS`, `WITHOUT OIDS`
Ce sont les syntaxes obsolètes mais équivalentes, respectivement de `WITH (OIDS)` et `WITH (OIDS=FALSE)`. Si vous

souhaitez indiquer à la fois l'option `OIDS` et les paramètres de stockage, vous devez utiliser la syntaxe `WITH ( ... )` ; voir ci-dessus.

`ON COMMIT`
Le comportement des tables temporaires à la fin d'un bloc de transactions est contrôlé à l'aide de la clause `ON COMMIT`. Les trois options sont :

`PRESERVE ROWS`
Aucune action n'est entreprise à la fin des transactions. Comportement par défaut.

`DELETE ROWS`
Toutes les lignes de la table temporaire sont détruites à la fin de chaque bloc de transactions. En fait, un TRUNCATE(7) automatique est réalisé à chaque validation.

`DROP`
La table temporaire est supprimée à la fin du bloc de transactions.

`TABLESPACE` *`tablespace`*
`tablespace` est le nom du tablespace dans lequel est créée la nouvelle table. S'il n'est pas spécifié, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

`USING INDEX TABLESPACE` *`tablespace`*
Les index associés à une contrainte `UNIQUE`, `PRIMARY KEY`, ou `EXCLUDE` sont créés dans le tablespace nommé *`tablespace`*. S'il n'est pas précisé, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

Paramètres de stockage

La clause `WITH` spécifie des *paramètres de stockage* pour les tables ainsi que pour les index associés avec une contrainte `UNIQUE`, `PRIMARY KEY`, ou `EXCLUDE`. Les paramètres de stockage des index sont documentés dans CREATE INDEX(7). Les paramètres de stockage actuellement disponibles pour les tables sont listés ci-dessous. Pour chaque paramètre, sauf contre-indication, il y a un paramètre additionnel, de même nom mais préfixé par `toast.`, qui peut être utilisé pour contrôler le le comportement de la table TOAST (stockage supplémentaire), si elle existe (voir Section 54.2, « TOAST » pour plus d'informations sur TOAST). La table TOAST hérite ses valeurs `autovacuum` de sa table parente s'il n'y a pas de paramètre `toast.autovacuum_*` positionné.

`fillfactor` (integer)
Le facteur de remplissage d'une table est un pourcentage entre 10 et 100. 100 (paquet complet) est la valeur par défaut. Quand un facteur de remplissage plus petit est indiqué, les opérations **INSERT** remplissent les pages de table d'au maximum ce pourcentage ; l'espace restant sur chaque page est réservé à la mise à jour des lignes sur cette page. Cela donne à **UPDATE** une chance de placer la copie d'une ligne mise à jour sur la même page que l'original, ce qui est plus efficace que de la placer sur une page différente. Pour une table dont les entrées ne sont jamais mises à jour, la valeur par défaut est le meilleur choix, mais pour des tables mises à jour fréquemment, des facteurs de remplissage plus petits sont mieux appropriés. Ce paramètre n'est pas disponible pour la table TOAST.

`autovacuum_enabled`, `toast.autovacuum_enabled` (boolean)
Active ou désactive le processus d'autovacuum sur une table particulière. Si à true, le processus d'autovacuum démarrera une opération **VACUUM** sur une table particulière quand le nombre d'enregistrements mis à jour ou supprimés dépassera `autovacuum_vacuum_threshold` plus `autovacuum_vacuum_scale_factor` multiplié par le nombre d'enregistrements estimés actifs dans la relation. De façon similaire, il démarrera une opération **ANALYZE** quand le nombre d'enregistrements insérés, mis à jour ou supprimés dépassera `autovacuum_analyze_threshold` plus `autovacuum_analyze_scale_factor` multiplié par le nombre d'enregistrements estimés actifs dans la relation. Si à false, la table ne sera pas traitée par autovacuum, sauf pour prévenir le bouclage des identifiants de transaction. Voir Section 23.1.4, « Éviter les cycles des identifiants de transactions » pour plus d'information sur la prévention de ce bouclage. Notez que cette variable hérite sa valeur du paramètre autovacuum.

`autovacuum_vacuum_threshold`, `toast.autovacuum_vacuum_threshold` (integer)
Nombre minimum d'enregistrements mis à jour ou supprimés avant de démarrer une opération **VACUUM** sur une table particulière.

`autovacuum_vacuum_scale_factor`, `toast.autovacuum_vacuum_scale_factor` (float4)
Coefficient multiplicateur pour *`reltuples`* (nombre estimé d'enregistrements d'une relation) à ajouter à `autovacuum_vacuum_threshold`.

`autovacuum_analyze_threshold` (integer)
Nombre minimum d'enregistrements insérés, mis à jour ou supprimés avant de démarrer une opération **ANALYZE** sur une table particulière.

`autovacuum_analyze_scale_factor` (float4)
: Coefficient multiplicateur pour *`reltuples`* (nombre estimé d'enregistrements d'une relation) à ajouter à `autovacuum_analyze_threshold`.

`autovacuum_vacuum_cost_delay`, `toast.autovacuum_vacuum_cost_delay` (integer)
: Paramètre autovacuum_vacuum_cost_delay personnalisé.

`autovacuum_vacuum_cost_limit`, `toast.autovacuum_vacuum_cost_limit` (integer)
: Paramètre autovacuum_vacuum_cost_limit personnalisé.

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (integer)
: Paramètre vacuum_freeze_min_age personnalisé. Notez que autovacuum rejettera les tentatives de positionner un `autovacuum_freeze_min_age` plus grand que le paramètre autovacuum_freeze_max_age à la moitié de la plage système d'identifiants.

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (integer)
: Paramètre autovacuum_freeze_max_age personnalisé. L'autovacuum rejettera les tentatives de positionner un `autovacuum_freeze_max_age` plus grand que le paramètre système (il ne peut être que plus petit). Même si `autovacuum_freeze_max_age` peut être positionné à de très petites valeurs, voire zéro, c'est habituellement déconseillé parce que cela forcera des opérations VACUUM très fréquentes.

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (integer)
: Paramètre vacuum_freeze_table_age personnalisé.

Notes

Utiliser les OID dans les nouvelles applications n'est pas recommandé : dans la mesure du possible, un type `SERIAL` ou un autre générateur de séquence sera utilisé comme clé primaire de la table. Néanmoins, si l'application utilise les OID pour identifier des lignes spécifiques d'une table, il est recommandé de créer une contrainte unique sur la colonne *`oid`* de cette table afin de s'assurer que les OID de la table identifient les lignes de façon réellement unique même si le compteur est réinitialisé. Il n'est pas garanti que les OID soient uniques sur l'ensemble des tables. Dans le cas où un identifiant unique sur l'ensemble de la base de données est nécessaire, on utilise préférentiellement une combinaison de *`tableoid`* et de l'OID de la ligne.

Astuce

L'utilisation de `OIDS=FALSE` est déconseillée pour les tables dépourvues de clé primaire. En effet, sans OID ou clé de données unique, il est difficile d'identifier des lignes spécifiques.

PostgreSQL™ crée automatiquement un index pour chaque contrainte d'unicité ou clé primaire afin d'assurer l'unicité. Il n'est donc pas nécessaire de créer un index spécifiqueme pour les colonnes de clés primaires. Voir CREATE INDEX(7) pour plus d'informations.

Les contraintes d'unicité et les clés primaires ne sont pas héritées dans l'implantation actuelle. Cela diminue la fonctionnalité des combinaisons d'héritage et de contraintes d'unicité.

Une table ne peut pas avoir plus de 1600 colonnes (en pratique, la limite réelle est habituellement plus basse du fait de contraintes sur la longueur des lignes).

Exemples

Créer une table films et une table distributeurs :

```
CREATE TABLE films (
    code        char(5) CONSTRAINT premierecle PRIMARY KEY,
    titre       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute
);
```

```
CREATE TABLE distributeurs (
     did    integer PRIMARY KEY DEFAULT nextval('serial'),
     nom    varchar(40) NOT NULL CHECK (nom <> '')
);
```

Créer une table contenant un tableau à deux dimensions :

```
CREATE TABLE array_int (
    vecteur  int[][]
);
```

Définir une contrainte d'unicité pour la table `films`. Les contraintes d'unicité de table peuvent être définies sur une ou plusieurs colonnes de la table :

```
CREATE TABLE films (
    code        char(5),
    titre       varchar(40),
    did         integer,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Définir une contrainte de vérification sur une colonne :

```
CREATE TABLE distributeurs (
    did     integer CHECK (did > 100),
    nom    varchar(40)
);
```

Définir une contrainte de vérification sur la table :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40)
    CONSTRAINT con1 CHECK (did > 100 AND nom <> '')
);
```

Définir une contrainte de clé primaire sur la table films.

```
CREATE TABLE films (
    code        char(5),
    titre       varchar(40),
    did         integer,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute,
    CONSTRAINT code_titre PRIMARY KEY(code,titre)
);
```

Définir une contrainte de clé primaire pour la table distributeurs. Les deux exemples suivants sont équivalents, le premier utilise la syntaxe de contrainte de table, le second la syntaxe de contrainte de colonne :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    PRIMARY KEY(did)
);
```

```
CREATE TABLE distributeurs (
    did     integer PRIMARY KEY,
    nom     varchar(40)
);
```

Affecter une valeur par défaut à la colonne `nom`, une valeur par défaut à la colonne `did`, engendrée à l'aide d'une séquence, et une valeur par défaut à la colonne `modtime`, équivalente au moment où la ligne est insérée :

```
CREATE TABLE distributeurs (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributeurs_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Définir deux contraintes de colonnes `NOT NULL` sur la table `distributeurs`, dont l'une est explicitement nommée :

```
CREATE TABLE distributeurs (
    did     integer CONSTRAINT no_null NOT NULL,
    nom     varchar(40) NOT NULL
);
```

Définir une contrainte d'unicité sur la colonne `nom` :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40) UNIQUE
);
```

La même chose en utilisant une contrainte de table :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    UNIQUE(nom)
);
```

Créer la même table en spécifiant un facteur de remplissage de 70% pour la table et les index uniques :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    UNIQUE(nom) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Créer une table cercles avec une contrainte d'exclusion qui empêche le croisement de deux cercles :

```
CREATE TABLE cercles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Créer une table cinemas dans le tablespace diskvol1 :

```
CREATE TABLE cinemas (
    id serial,
    nom text,
    emplacement text
) TABLESPACE diskvol1;
```

Créer un type composite et une table typée :

```
CREATE TYPE type_employe AS (nom text, salaire numeric);

CREATE TABLE employes OF type_employe (
    PRIMARY KEY (nom),
    salaire WITH OPTIONS DEFAULT 1000
);
```

Compatibilité

La commande **CREATE TABLE** est conforme au standard SQL, aux exceptions indiquées ci-dessous.

Tables temporaires

Bien que la syntaxe de `CREATE TEMPORARY TABLE` ressemble à celle du SQL standard, l'effet n'est pas le même. Dans le standard, les tables temporaires sont définies une seule fois et existent automatiquement (vide de tout contenu au démarrage) dans toute session les utilisant. PostgreSQL™, au contraire, impose à chaque session de lancer une commande `CREATE TEMPORARY TABLE` pour chaque table temporaire utilisée. Cela permet à des sessions différentes d'utiliser le même nom de table temporaire dans des buts différents (le standard contraint toutes les instances d'une table temporaire donnée à pointer sur la même structure de

table).

Le comportement des tables temporaires tel que défini par le standard est largement ignorée. Le comportement de PostgreSQL™ sur ce point est similaire à celui de nombreuses autres bases de données SQL.

PostgreSQL™ ne respecte pas la distinction imposée par le standard entre tables temporaires globales et locales. En effet, cette distinction repose sur le concept de modules que PostgreSQL™ ne gère pas. Pour des raisons de compatibilité, PostgreSQL™ accepte néanmoins les mots-clés `GLOBAL` et `LOCAL` dans la définition d'une table temporaire, mais ils n'ont aucun effet.

La clause `ON COMMIT` sur les tables temporaires diffère quelque peu du standard SQL. Si la clause `ON COMMIT` est omise, SQL spécifie `ON COMMIT DELETE ROWS` comme comportemant par défaut. PostgreSQL™ utilise `ON COMMIT PRESERVE ROWS` par défaut. De plus, l'option `ON COMMIT DROP` n'existe pas en SQL.

Contraintes d'unicité non déferrées

Quand une contrainte `UNIQUE` ou `PRIMARY KEY` est non déferrable, PostgreSQL™ vérifie l'unicité immédiatement après qu'une ligne soit insérée ou modifiée. Le standard SQL indique que l'unicité doit être forcée seulement à la fin de l'instruction ; ceci fait une différence quand, par exemple, une seule commande met à jour plusieurs valeurs de clés. Pour obtenir un comportement compatible au standard, déclarez la contrainte comme `DEFERRABLE` mais non déferrée (c'est-à-dire que `INITIALLY IMMEDIATE`). Faites attention que cela peut être beaucoup plus lent qu'une vérification d'unicité immédiate.

Contraintes de vérification de colonnes

Dans le standard, les contraintes de vérification `CHECK` de colonne ne peuvent faire référence qu'à la colonne à laquelle elles s'appliquent ; seules les contraintes `CHECK` de table peuvent faire référence à plusieurs colonnes. PostgreSQL™ n'impose pas cette restriction ; les contraintes de vérifications de colonnes et de table ont un traitement identique.

`EXCLUDE` Constraint

Le type de contrainte `EXCLUDE` est une extension PostgreSQL™.

Contrainte `NULL`

La « contrainte » `NULL` (en fait, une non-contrainte) est une extension PostgreSQL™ au standard SQL, incluse pour des raisons de compatibilité avec d'autres systèmes de bases de données (et par symétrie avec la contrainte `NOT NULL`). Comme c'est la valeur par défaut de toute colonne, sa présence est un simple bruit.

Héritage

L'héritage multiple via la clause `INHERITS` est une extension du langage PostgreSQL™. SQL:1999 et les versions ultérieures définissent un héritage simple en utilisant une syntaxe et des sémantiques différentes. L'héritage style SQL:1999 n'est pas encore supporté par PostgreSQL™.

Tables sans colonne

PostgreSQL™ autorise la création de tables sans colonne (par exemple, `CREATE TABLE foo();`). C'est une extension du standard SQL, qui ne le permet pas. Les tables sans colonne ne sont pas très utiles mais les interdire conduit à un comportement étrange de **ALTER TABLE DROP COLUMN**. Il est donc plus sage d'ignorer simplement cette restriction.

`WITH` clause

La clause `WITH` est une extension PostgreSQL™ ; ni les paramètres de stockage ni les OID ne sont dans le standard.

Tablespaces

Le concept PostgreSQL™ de tablespace n'est pas celui du standard. De ce fait, les clauses `TABLESPACE` et `USING INDEX TABLESPACE` sont des extensions.

Tables typées

Les tables typées implémentent un sous-ensemble du standard SQL. Suivant le standard, une table typée a des colonnes correspondant au type composite ainsi qu'une autre colonne qui est la « colonne auto-référente ». PostgreSQL ne supporte pas ces colonnes auto-référentes explicitement mais le même effet est disponible en utilisant la fonctionnalité OID.

Voir aussi

ALTER TABLE(7), DROP TABLE(7), CREATE TABLESPACE(7)

Nom

CREATE TABLE AS — Définir une nouvelle table à partir des résultats d'une requête

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nom_table
    [ (nom_colonne [, ...] ) ]
    [ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE espace_logique ]
    AS requête
    [ WITH [ NO ] DATA ]
```

Description

CREATE TABLE AS crée une table et y insère les données récupérées par une commande **SELECT**. Les colonnes de la table ont les noms et les types de données associés aux colonnes en sortie du **SELECT** (les noms des colonnes peuvent toutefois être surchargés).

CREATE TABLE AS semble possèder des similitudes avec la création d'une vue mais est, en fait, assez différente : elle crée une nouvelle table et n'évalue la requête qu'une seule fois, pour le chargement initial de la nouvelle table. Les modifications ultérieures de la table source ne sont pas prises en compte. Au contraire, une vue réévalue l'instruction **SELECT** de définition à chaque appel.

Paramètres

`GLOBAL` ou `LOCAL`
: Ignoré. Conservé pour la compatibilité (cf. CREATE TABLE(7)).

`TEMPORARY` ou `TEMP`
: Si spécifié, la table est temporaire (cf. CREATE TABLE(7)).

`nom_table`
: Le nom de la table à créer (éventuellement qualifié du nom du schéma).

`nom_colonne`
: Le nom d'une colonne dans la nouvelle table. Si les noms de colonnes ne sont pas précisés, ils sont issus des noms des colonnes en sortie de la requête. Les noms des colonnes ne peuvent pas être précisés lorsque la table est créée à partir d'une commande **EXECUTE**.

`WITH ( paramètre_stockage [= valeur] [, ... ] )`
: Cette clause indique les paramètres de stockage optionnels pour la nouvelle table ; voir la section intitulée « Paramètres de stockage » pour plus d'informations. La clause `WITH` peut aussi inclure `OIDS=TRUE` (ou simplement `OIDS`) pour indiquer que les lignes de la nouvelle table doivent avoir des OID (identifiants d'objets) ou `OIDS=FALSE` pour indiquer le contraire. Voir CREATE TABLE(7) pour plus d'informations.

`WITH OIDS`, `WITHOUT OIDS`
: Ce sont les syntaxes obsolètes mais équivalentes, respectivement de `WITH (OIDS)` et `WITH (OIDS=FALSE)`. Si vous souhaitez indiquer à la fois l'option `OIDS` et les paramètres de stockage, vous devez utiliser la syntaxe `WITH ( ... )` ; voir ci-dessus.

`ON COMMIT`
: Le comportement des tables temporaires à la fin d'un bloc de transaction est contrôlable en utilisant `ON COMMIT`. Voici les trois options :

  `PRESERVE ROWS`
  : Aucune action spéciale n'est effectuée à la fin de la transaction. C'est le comportement par défaut.

  `DELETE ROWS`
  : Toutes les lignes de la table temporaire seront supprimées à la fin de chaque bloc de transaction. Habituellement, un TRUNCATE(7) automatique est effectué à chaque COMMIT.

  `DROP`
  : La table temporaire sera supprimée à la fin du bloc de transaction en cours.

`TABLESPACE espace_logique`

L'*espace_logique* est le nom du tablespace dans lequel est créée la nouvelle table. S'il n'est pas indiqué, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

requête
Une commande SELECT(7), TABLE ou VALUES(7), voire une commande EXECUTE(7) qui exécute un **SELECT** préparé, **TABLE** ou une requête **VALUES**.

`WITH [ NO ] DATA`
Cette clause indique si les données produites par la requêtes doivent être copiées dans la nouvelle table. Si non, seule la structure de la table est copiée. La valeur par défaut est de copier les données.

Notes

Cette commande est fonctionnellement équivalente à SELECT INTO(7). Elle lui est cependant préférée car elle présente moins de risques de confusion avec les autres utilisations de la syntaxe **SELECT INTO**. De plus, **CREATE TABLE AS** offre plus de fonctionnalités que **SELECT INTO**.

Avant PostgreSQL™ 8.0, **CREATE TABLE AS** incluait toujours les OIDs dans la table créée. À partir de PostgresSQL™ 8.0, la commande **CREATE TABLE AS** autorise l'utilisateur à spécifier explicitement la présence des OID. En l'absence de précision, la variable de configuration default_with_oids est utilisée. À partir de PostgreSQL™ 8.1, la valeur par défaut de cette variable est « faux » ; le comportement par défaut n'est donc pas identique à celui des versions précédant la 8.0. Il est préférable que les applications qui nécessitent des OID dans la table créée par **CREATE TABLE AS** indiquent explicitement `WITH (OIDS)` pour s'assurer du comportement souhaité.

Exemples

Créer une table `films_recent` contenant les entrées récentes de la table `films` :

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2006-01-01';
```

Pour copier une table complètement, la forme courte utilisant la clause `TABLE` peut aussi être utilisée :

```
CREATE TABLE films2 AS
  TABLE films;
```

Créer une nouvelle table temporaire `films_recents` consistant des seules entrées récentes provenant de la table `films` en utilisant une instruction préparée. La nouvelle table a des OID et sera supprimée à la validation (COMMIT) :

```
PREPARE films_recents(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recents WITH (OIDS) ON COMMIT DROP AS
  EXECUTE films_recents('2002-01-01');
```

Compatibilité

CREATE TABLE AS est conforme au standard SQL. The following are nonstandard extensions :

- Le standard requiert des parenthèses autour de la clause de la sous-requête ; elles sont optionnelles dans PostgreSQL™.
- Dans le standard, la clause `WITH [ NO ] DATA` est requise alors que PostgreSQL la rend optionnelle.
- PostgreSQL™ gère les tables temporaires d'une façon bien différente de celle du standard ; voir CREATE TABLE(7) pour les détails.
- La clause `WITH` est une extension PostgreSQL™ ; ni les paramètres de stockage ni les OID ne sont dans le standard.
- Le concept PostgreSQL™ des tablespaces ne fait pas partie du standard. Du coup, la clause `TABLESPACE` est une extension.

Voir aussi

CREATE TABLE(7), EXECUTE(7), SELECT(7), SELECT INTO(7), VALUES(7)

Nom

CREATE TABLESPACE — Définir un nouvel tablespace

Synopsis

```
CREATE TABLESPACE nom_tablespace
[ OWNER nom_utilisateur ]
LOCATION 'répertoire'
```

Description

CREATE TABLESPACE enregistre un nouveau tablespace pour la grappe de bases de données. Le nom du tablespace doit être distinct du nom de tout autre tablespace de la grappe.

Un tablespace permet aux superutilisateurs de définir un nouvel emplacement sur le système de fichiers pour le stockage des fichiers de données contenant des objets de la base (comme les tables et les index).

Un utilisateur disposant des droits appropriés peut passer *nom_tablespace* comme paramètre de **CREATE DATABASE**, **CREATE TABLE**, **CREATE INDEX** ou **ADD CONSTRAINT** pour que les fichiers de données de ces objets soient stockés à l'intérieur du tablespace spécifié.

Paramètres

nom_tablespace
: Le nom du tablespace à créer. Le nom ne peut pas commencer par `pg_`, de tels noms sont réservés pour les tablespaces système.

nom_utilisateur
: Le nom de l'utilisateur, propriétaire du tablespace. En cas d'omission, il s'agit de l'utilisateur ayant exécuté la commande. Seuls les superutilisateurs peuvent créer des tablespaces mais ils peuvent en donner la propriété à des utilisateurs standard.

répertoire
: Le répertoire qui sera utilisé pour le tablespace. Le répertoire doit être vide et doit appartenir à l'utilisateur système PostgreSQL™. Le répertoire doit être spécifié par un chemin absolu.

Notes

Les tablespaces ne sont supportés que sur les systèmes gérant les liens symboliques.

CREATE TABLESPACE ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Exemples

Créer un tablespace `espace_base` sur `/data/dbs` :

```
CREATE TABLESPACE espace_base LOCATION '/data/dbs';
```

Créer un tablespace `espace_index` sur `/data/indexes` et en donner la propriété à l'utilisatrice `genevieve` :

```
CREATE TABLESPACE espace_index OWNER genevieve LOCATION '/data/indexes';
```

Compatibilité

CREATE TABLESPACE est une extension PostgreSQL™.

Voir aussi

CREATE DATABASE(7), CREATE TABLE(7), CREATE INDEX(7), DROP TABLESPACE(7), ALTER TABLESPACE(7)

Nom

CREATE TEXT SEARCH CONFIGURATION — définir une nouvelle configuration de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH CONFIGURATION nom (
    PARSER = nom_analyseur |
    COPY = config_source
)
```

Description

CREATE TEXT SEARCH CONFIGURATION crée une nouvelle configuration de recherche plein texte. Une configuration indique l'analyseur qui peut diviser une chaîne en jetons, ainsi que les dictionnaires pouvant être utilisés pour déterminer les jetons intéressants à rechercher.

Si seul l'analyseur est indiqué, la nouvelle configuration de recherche plein texte n'a initialement aucune relation entre les types de jeton et les dictionnaires et, du coup, ignorera tous les mots. De nouveaux appels aux commandes **ALTER TEXT SEARCH CONFIGURATION** doivent être utilisés pour créer les correspondances et rendre la configuration réellement utile. Autrement, une configuration de recherche plein texte peut être copiée.

Si un nom de schéma est précisé, alors le modèle de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

L'utilisateur qui définit une configuration de recherche plein texte en devient son propriétaire.

Voir Chapitre 12, Recherche plein texte pour plus d'informations.

Paramètres

nom
: Le nom de la configuration de recherche plein texte (pouvant être qualifié du schéma).

parser_name
: Le nom de l'analyseur de recherche plein texte à utiliser pour cette configuration.

source_config
: Le nom d'une configuration existante de recherche plein texte à copier.

Notes

Les options `PARSER` et `COPY` sont mutuellement exclusives car, quand une configuration existante est copiée, sa sélection de son analyseur est aussi copiée.

Compatibilité

Il n'existe pas d'instruction **CREATE TEXT SEARCH CONFIGURATION** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH CONFIGURATION(7), DROP TEXT SEARCH CONFIGURATION(7)

Nom

CREATE TEXT SEARCH DICTIONARY — définir un dictionnaire de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH DICTIONARY nom (
    TEMPLATE = modele
    [, option = valeur [, ... ]]
)
```

Description

CREATE TEXT SEARCH DICTIONARY crée un nouveau dictionnaire de recherche plein texte. Un dictionnaire de recherche plein texte indique une façon de distinguer les mots intéressants à rechercher des mots inintéressants. Un dictionnaire dépend d'un modèle de recherche plein texte qui spécifie les fonctions qui font réellement le travail. Typiquement, le dictionnaire fournit quelques options qui contrôlent le comportement détaillé des fonctions du modèle.

Si un nom de schéma est précisé, alors le dictionnaire de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

L'utilisateur qui définit un dictionnaire de recherche plein texte en devient son propriétaire.

Voir Chapitre 12, Recherche plein texte pour plus d'informations.

Paramètres

nom
: Le nom du dictionnaire de recherche plein texte (pouvant être qualifié du schéma).

modele
: Le nom du modèle de recherche plein texte qui définira le comportement basique de ce dictionnaire.

option
: Le nom d'une option, spécifique au modèle, à configurer pour ce dictionnaire.

valeur
: La valeur à utiliser pour une option spécifique au modèle. Si la valeur n'est pas un simple identifiant ou un nombre, elle doit être entre guillemets simples (mais vous pouvez toujours le faire si vous le souhaitez).

Les options peuvent apparaître dans n'importe quel ordre.

Exemples

La commande exemple suivante crée un dictionnaire basé sur Snowball avec une liste spécifique de mots d'arrêt.

```
CREATE TEXT SEARCH DICTIONARY mon_dico_russe (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

Compatibilité

Il n'existe pas d'instructions **CREATE TEXT SEARCH DICTIONARY** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH DICTIONARY(7), DROP TEXT SEARCH DICTIONARY(7)

Nom

CREATE TEXT SEARCH PARSER — définir un nouvel analyseur de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH PARSER nom (
    START = fonction_debut ,
    GETTOKEN = function_gettoken ,
    END = fonction_fin ,
    LEXTYPES = fonction_lextypes
    [, HEADLINE = fonction_headline ]
)
```

Description

CREATE TEXT SEARCH PARSER crée un nouvel analyseur de recherche plein texte. Un analyseur de recherche plein texte définit une méthode pour diviser une chaîne en plusieurs jetons et pour assigner des types (catégories) aux jetons. Un analyseur n'est pas particulièrement utile en lui-même mais doit être limité dans une configuration de recherche plein texte avec certains dictionnaires de recherche plein texte à utiliser pour la recherche.

Si un nom de schéma est précisé, alors le dictionnaire de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

Vous devez être un superutilisateur pour utiliser **CREATE TEXT SEARCH PARSER**. (Cette restriction est faite parce que la définition d'un analyseur de recherche plein texte peut gêner, voire arrêter brutalement, le serveur.)

Voir Chapitre 12, Recherche plein texte pour plus d'informations.

Paramètres

name
: Le nom d'un analyseur de recherche plein texte (pouvant être qualifié du schéma).

fonction_debut
: Le nom d'une fonction de démarrage pour l'analyseur.

fonction_gettoken
: Le nom d'une fonction pour l'obtention du prochain jeton (get-next-token) pour l'analyseur.

fonction_fin
: Le nom de la fonction d'arrêt de l'analyseur.

fonction_lextypes
: Le nom de la fonction lextypes pour l'analyseur (une fonction qui renvoie de l'information sur l'ensemble de types de jeton qu'il produit).

fonction_headline
: Le nom de la fonction headline pour l'analyseur (une fonction qui résume un ensemble de jetons).

Les noms des fonctions peuvent se voir qualifier du nom du schéma si nécessaire. Le type des arguments n'est pas indiqué car la liste d'argument pour chaque type de fonction est prédéterminé. Toutes les fonctions sont obligatoires sauf headline.

Les options peuvent apparaître dans n'importe quel ordre, pas seulement celui indiqué ci-dessus.

Compatibilité

Il n'existe pas d'instruction **CREATE TEXT SEARCH PARSER** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH PARSER(7), DROP TEXT SEARCH PARSER(7)

Nom

CREATE TEXT SEARCH TEMPLATE — définir un nouveau modèle de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH TEMPLATE nom (
    [ INIT = fonction_init , ]
    LEXIZE = fonction_lexize
)
```

Description

CREATE TEXT SEARCH TEMPLATE crée un nouveau modèle de recherche plein texte. Les modèles de recherche plein texte définissent les fonctions qui implémentent les dictionnaires de recherche plein texte. Un modèle n'est pas utile en lui-même mais doit être instancié par un dictionnaire pour être utilisé. Le dictionnaire spécifie typiquement les paramètres à donner aux fonctions modèle.

Si un nom de schéma est précisé, alors le modèle de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

Vous devez être un superutilisateur pour utiliser **CREATE TEXT SEARCH TEMPLATE**. Cette restriction est faite parce que la définition d'un modèle de recherche plein texte peut gêner, voire arrêter brutalement le serveur. La raison de la séparation des modèles et des dictionnaires est qu'un modèle encapsule les aspects « non sûrs » de la définition d'un dictionnaire. Les paramètres qui peuvent être définis lors de la mise en place d'un dictionnaire sont suffisamment sûrs pour être utilisé par des utilisateurs sans droits. Du coup, la création d'un dictionnaire ne demande pas de droits particuliers.

Voir Chapitre 12, Recherche plein texte pour plus d'informations.

Paramètres

nom
: Le nom du modèle de recherche plein texte (pouvant être qualifié du schéma).

fonction_init
: Le nom de la fonction d'initialisation du modèle.

fonction_lexize
: Le nom de la fonction lexize du modèle.

Les noms des fonctions peuvent se voir qualifier du nom du schéma si nécessaire. Le type des arguments n'est pas indiqué car la liste d'argument pour chaque type de fonction est prédéterminé. La fonction lexize est obligatoire mais la fonction init est optionnelle.

Les arguments peuvent apparaître dans n'importe quel ordre, pas seulement dans celui indiqué ci-dessus.

Compatibilité

Il n'existe pas d'instruction **CREATE TEXT SEARCH TEMPLATE** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH TEMPLATE(7), DROP TEXT SEARCH TEMPLATE(7)

Nom

CREATE TRIGGER — Définir un nouveau déclencheur

Synopsis

```
CREATE TRIGGER nom { BEFORE | AFTER } { evenement [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE nom_fonction ( arguments )
```

Description

CREATE TRIGGER crée un nouveau déclencheur. Le déclencheur est associé à la table spécifiée et exécute la fonction *nom_fonction* lorsque certains événements surviennent.

L'appel du déclencheur peut avoir lieu avant que l'opération ne soit tentée sur une ligne (avant la vérification des contraintes et la tentative d'**INSERT**, **UPDATE** ou **DELETE**) ou une fois que l'opération est terminée (après la vérification des contraintes et la fin de la commande **INSERT**, **UPDATE** ou **DELETE**). Si le déclencheur est lancé avant l'événement, le déclencheur peut ignorer l'opération sur la ligne courante ou modifier la ligne en cours d'insertion (uniquement pour les opérations **INSERT** et **UPDATE**). Si le déclencheur est activé après l'événement, toute modification, dont les autres triggers, est « visible » par le déclencheur.

Un déclencheur marqué `FOR EACH ROW` est appelé pour chaque ligne que l'opération modifie. Par exemple, un **DELETE** affectant dix lignes entraîne dix appels distincts de tout déclencheur `ON DELETE` sur la relation cible, une fois par ligne supprimée. Au contraire, un déclencheur marqué `FOR EACH STATEMENT` ne s'exécute qu'une fois pour une opération donnée, quelque soit le nombre de lignes modifiées (en particulier, une opération qui ne modifie aucune ligne résulte toujours en l'exécution des déclencheurs `FOR EACH STATEMENT` applicables).

De plus, les triggers peuvent être définis pour être déclenchés suite à l'exécution d'un **TRUNCATE**, mais seulement dans le cas d'un trigger `FOR EACH STATEMENT`.

De plus, la définition d'un trigger peut spécifier une condition `WHEN` qui sera testée pour voir si le trigger doit réellement être déclenché. Dans les triggers au niveau ligne, la condition `WHEN` peut examiner l'ancienne et/ou la nouvelle valeurs des colonnes de la ligne. Les triggers au niveau instruction peuvent aussi avoir des conditions `WHEN`, bien que la fonctionnalité n'est pas aussi utile pour elles car la condition ne peut pas faire référence aux valeurs de la table.

Si plusieurs déclencheurs du même genre sont définis pour le même événement, ils sont déclenchés suivant l'ordre alphabétique de leur nom.

SELECT ne modifie aucune ligne ; la création de déclencheurs sur **SELECT** n'est donc pas possible. Les règles et vues sont plus appropriées dans ce cas.

Chapitre 36, Déclencheurs (triggers) présente de plus amples informations sur les déclencheurs.

Paramètres

nom
: Le nom du nouveau déclencheur. Il doit être distinct du nom de tout autre déclencheur sur la table.

`BEFORE`, `AFTER`
: Détermine si la fonction est appelée avant ou après l'événement.

événement
: Peut-être **INSERT**, **UPDATE** ou **DELETE** ou **TRUNCATE** ; précise l'événement qui active le déclencheur. Plusieurs événements peuvent être précisés en les séparant par `OR`.

  Pour les triggers se déclenchant suite à un **UPDATE**, il est possible de spécifier une liste de colonnes utilisant cette syntaxe :

  ```
  UPDATE OF nom_colonne_1 [, nom_colonne_2 ... ]
  ```

  Le trigger se déclenchera seulement si au moins une des colonnes listées est mentionnée comme cible de la mise à jour.

table
: Le nom (éventuellement qualifié du nom du schéma) de la table à laquelle est rattaché le déclencheur.

`FOR EACH ROW`, `FOR EACH STATEMENT`
: Précise si la procédure du déclencheur doit être lancée pour chaque ligne affectée par l'événement ou simplement pour chaque instruction SQL. `FOR EACH STATEMENT` est la valeur par défaut.

`condition`
: Une expression booléenne qui détermine si la fonction trigger sera réellement exécutée. Si `WHEN` est indiqué, la fonction sera seulement appelée si la *`condition`* renvoie `true`. Pour les triggers `FOR EACH ROW`, la condition `WHEN` peut faire référence aux valeurs des colonnes des ancienne et nouvelle lignes en utilisant la notation `OLD.`*`nom_colonne`* ou `NEW.`*`nom_colonne`*, respectivement. Bien sûr, le triggers sur `INSERT` ne peuvent pas faire référence à `OLD` et ceux sur `DELETE` ne peuvent pas faire référence à `NEW`.

  Actuellement, les expressions `WHEN` ne peuvent pas contenir de sous-requêtes.

`nom_fonction`
: Une fonction utilisateur, déclarée sans argument et renvoyant le type `trigger`, exécutée à l'activation du déclencheur.

`arguments`
: Une liste optionnelle d'arguments séparés par des virgules à fournir à la fonction lors de l'activation du déclencheur. Les arguments sont des chaînes littérales constantes. Il est possible d'écrire ici de simples noms et des constantes numériques mais ils sont tous convertis en chaîne. L'accès aux arguments du trigger depuis la fonction peut différer de l'accès aux arguments d'une fonction standard ; la consultation des caractéristiques d'implantation du langage de la fonction peut alors s'avérer utile.

Notes

Pour créer un déclencheur sur une table, l'utilisateur doit posséder le droit `TRIGGER` sur la table. L'utilisateur doit aussi avoir + le droit `EXECUTE` sur la fonction trigger.

Utiliser DROP TRIGGER(7) pour supprimer un déclencheur.

Un trigger sur colonne spécifique (`FOR UPDATE OF` *`nom_colonne`*) se déclenchera quand une des colonnes indiquées est listée comme cible de la liste `SET` pour la commande **UPDATE**. Il est possible qu'une valeur de colonne change même si le trigger n'est pas déclenché parceque les modifications au contenu de la ligne par les triggers `BEFORE UPDATE` ne sont pas pris en compte. De même, une commande comme `UPDATE ... SET x = x ...` déclenchera le trigger sur la colonne `x`, bien que la valeur de cette colonne ne changera pas.

Dans un trigger `BEFORE`, la condition `WHEN` est évaluée juste avant l'exécution de la fonction, donc utiliser `WHEN` n'est pas matériellement différent de tester la même condition au début de la fonction trigger. Notez en particulier que la ligne `NEW` vu par la condition est sa valeur courante et possiblement modifiée par des triggers précédents. De plus, la condition `WHEN` d'un trigger `BEFORE` n'est pas autorisé à examiner les colonnes système de la ligne `NEW` (comme l'`oid`), car elles n'auront pas encore été initialisées.

Dans un trigger `AFTER`, la condition `WHEN` est évaluée juste après la mise à jour de la ligne et elle détermine si un événement doit déclencher le trigger à la fin de l'instruction. Donc, quand la condition `WHEN` d'un trigger `AFTER` ne renvoie pas true, il n'est pas nécessaire de préparer un événement ou de relire la ligne à la fin de l'instruction. Cela peut apporter une amélioration significative des performances dans les instructions qui modifient de nombreuses lignes, si le trigger a besoin d'être déclencher pour quelques lignes.

Dans les versions de PostgreSQL™ antérieures à la 7.3, il était nécessaire de déclarer un type opaque de retour pour les fonctions déclencheur, plutôt que trigger. Pour pouvoir charger d'anciens fichiers de sauvegarde, **CREATE TRIGGER** accepte qu'une fonction déclare une valeur de retour de type opaque, mais il affiche un message d'avertissement et change le type de retour déclaré en trigger.

Exemples

Exécutez la fonction `check_account_update` quand une ligne de la table `accounts` est sur le point d'être mise à jour :

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

Idem, mais avec une exécution de la fonction seulement si la colonne `balance` est spécifiée comme cible de la commande **UPDATE** :

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
```

```
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

Cette forme exécute la fonction seulement si la colonne `balance` a réellement changé de valeur :

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

Appelle une fonction pour tracer les mises à jour de la table `accounts`, mais seulement si quelque chose a changé :

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

Section 36.4, « Un exemple complet de trigger » contient un exemple complet d'une fonction trigger écrit en C.

Compatibilité

L'instruction **CREATE TRIGGER** de PostgreSQL™ implante un sous-ensemble du standard SQL. Les fonctionnalités manquantes sont :

- SQL permet de définir des alias pour les lignes « old » et « new » ou pour les tables utilisée dans la définition des actions déclenchées (c'est-à-dire `CREATE TRIGGER ... ON nomtable REFERENCING OLD ROW AS unnom NEW ROW AS unautrenom...`). PostgreSQL™ autorise l'écriture de procédures de déclencheurs dans tout langage l'utilisateur. De ce fait, l'accès aux données est géré spécifiquement pour chaque langage.
- PostgreSQL™ n'autorise comme action déclenchée que l'exécution d'une fonction utilisateur. Le standard SQL, en revanche, autorise l'exécution d'autres commandes SQL, telles que **CREATE TABLE**. Cette limitation de PostgreSQL™ peut être facilement contournée par la création d'une fonction utilisateur qui exécute les commandes désirées.

Le standard SQL définit l'ordre de création comme ordre de lancement des déclencheurs multiples. PostgreSQL™ utilise l'ordre alphabétique de leur nom, jugé plus pratique.

Le standard SQL précise que les déclencheurs `BEFORE DELETE` sur des suppressions en cascade se déclenchent *après* la fin du `DELETE` en cascade. PostgreSQL™ définit que `BEFORE DELETE` se déclenche toujours avant l'action de suppression, même lors d'une action en cascade. Cela semble plus cohérent. Il existe aussi un comportement non prévisible quand les triggers `BEFORE` modifient les lignes ou empêchent les mises à jour causées par une action référente. Ceci peut amener à des violations de contraintes ou au stockage de données qui n'honorent pas la contrainte référentielle.

La capacité à préciser plusieurs actions pour un seul déclencheur avec `OR` est une extension PostgreSQL™.

La possibilité d'exécuter un trigger suite à une commande **TRUNCATE** est une extension PostgreSQL™ du standard SQL.

Voir aussi

CREATE FUNCTION(7), ALTER TRIGGER(7), DROP TRIGGER(7)

Nom

CREATE TYPE — Définir un nouveau type de données

Synopsis

```
CREATE TYPE nom AS
    ( nom_attribut type_donnée [, ... ] )

CREATE TYPE nom AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE nom (
    INPUT = fonction_entrée,
    OUTPUT = fonction_sortie
    [ , RECEIVE = fonction_réception ]
    [ , SEND = fonction_envoi ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = fonction_analyse ]
    [ , INTERNALLENGTH = { longueurinterne | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignement ]
    [ , STORAGE = stockage ]
    [ , LIKE = type_like ]
    [ , CATEGORY = catégorie ]
    [ , PREFERRED = préféré ]
    [ , DEFAULT = défaut ]
    [ , ELEMENT = élément ]
    [ , DELIMITER = délimiteur ]
)

CREATE TYPE nom
```

Description

CREATE TYPE enregistre un nouveau type de données utilisable dans la base courante. L'utilisateur qui définit un type en devient le propriétaire.

Si un nom de schéma est précisé, le type est créé dans ce schéma. Sinon, il est créé dans le schéma courant. Le nom du type doit être distinct du nom de tout type ou domaine existant dans le même schéma. Les tables possèdent des types de données associés. Il est donc nécessaire que le nom du type soit également distinct du nom de toute table existant dans le même schéma.

Types composites

La première forme de **CREATE TYPE** crée un type composite. Le type composite est défini par une liste de noms d'attributs et de types de données. Pour l'essentiel, il est équivalent au type ligne (NDT : *row type* en anglais) d'une table, mais l'utilisation de **CREATE TYPE** permet d'éviter la création d'une table réelle quand seule la définition d'un type est voulue. Un type composite autonome est utile comme type d'argument ou de retour d'une fonction.

Types énumérés

La seconde forme de **CREATE TYPE** crée un type énuméré (enum), comme décrit dans Section 8.7, « Types énumération ». Les types enum prennent une liste de un à plusieurs labels entre guillemets, chacun devant faire moins de NAMEDATALEN octets (64 dans une installation PostgreSQL™ standard).

Types de base

La troisième forme de **CREATE TYPE** crée un nouveau type de base (type scalaire). Pour créer un nouveau type de base, il faut être superutilisateur. (Cette restriction est imposée parce qu'une définition de type erronée pourrait embrouiller voire arrêter brutalement le serveur.)

L'ordre des paramètres, dont la plupart sont optionnels, n'a aucune d'importance. Avant de définir le type, il est nécessaire de définir au moins deux fonctions (à l'aide de la commande **CREATE FUNCTION**). Les fonctions de support `fonction_entrée` et `fonction_sortie` sont obligatoires. Les fonctions `fonction_réception`, `fonction_envoi`, `type_modifier_input_function`, `type_modifier_output_function` et `fonction_analyse` sont option-

nelles. Généralement, ces fonctions sont codées en C ou dans un autre langage de bas niveau.

La *fonction_entrée* convertit la représentation textuelle externe du type en représentation interne utilisée par les opérateurs et fonctions définis pour le type. La *fonction_sortie* réalise la transformation inverse. La fonction entrée peut être déclarée avec un argument de type cstring ou trois arguments de types cstring, oid, integer. Le premier argument est le texte en entrée sous la forme d'une chaîne C, le second argument est l'OID du type (sauf dans le cas des types tableau où il s'agit de l'OID du type de l'élément) et le troisième est le `typmod` de la colonne destination, s'il est connu (-1 sinon). La fonction entrée doit renvoyer une valeur du nouveau type de données. Habituellement, une fonction d'entrée devrait être déclarée comme STRICT si ce n'est pas le cas, elle sera appelée avec un premier paramètre NULL à la lecture d'une valeur NULL en entrée. La fonction doit toujours envoyer NULL dans ce cas, sauf si une erreur est rapportée. (Ce cas a pour but de supporter les fonctions d'entrée des domaines qui ont besoin de rejeter les entrées NULL.) La fonction sortie doit prendre un argument du nouveau type de données, et retourner le type cstring. Les fonctions sortie ne sont pas appelées pour des valeurs NULL.

La *fonction_réception*, optionnelle, convertit la représentation binaire externe du type en représentation interne. Si cette fonction n'est pas fournie, le type n'accepte pas d'entrée binaire. La représentation binaire est choisie de telle sorte que sa conversion en forme interne soit peu coûteuse, tout en restant portable. (Par exemple, les types de données standard entiers utilisent l'ordre réseau des octets comme représentation binaire externe alors que la représentation interne est dans l'ordre natif des octets de la machine.) La fonction de réception réalise les vérifications adéquates pour s'assurer que la valeur est valide. Elle peut être déclarée avec un argument de type internal ou trois arguments de types internal, integer et oid. Le premier argument est un pointeur vers un tampon StringInfo qui contient la chaîne d'octets reçue ; les arguments optionnels sont les mêmes que pour la fonction entrée de type texte. La fonction de réception retourne une valeur du type de données. Habituellement, une fonction de réception devrait être déclarée comme STRICT si ce n'est pas le cas, elle sera appelée avec un premier paramètre NULL à la lecture d'une valeur NULL en entrée. La fonction doit toujours envoyer NULL dans ce cas, sauf si une erreur est rapportée. (Ce cas a pour but de supporter les fonctions de réception des domaines qui ont besoin de rejeter les entrées NULL.) De façon similaire, la *fonction_envoi*, optionnelle, convertit la représentation interne en représentation binaire externe. Si cette fonction n'est pas fournie, le type n'accepte pas de sortie binaire. La fonction d'envoi doit être déclarée avec un argument du nouveau type de données et retourner le type bytea. Les fonctions réception ne sont pas appelées pour des valeurs NULL.

À ce moment-là, vous pouvez vous demander comment les fonctions d'entrée et de sortie peuvent être déclarées avoir un résultat ou un argument du nouveau type alors qu'elles sont à créer avant que le nouveau type ne soit créé. La réponse est que le type sera tout d'abord défini en tant que *type squelette* (*shell type*), une ébauche de type sans propriété à part un nom et un propriétaire. Ceci se fait en exécutant la commande `CREATE TYPE` *nom* sans paramètres supplémentaires. Ensuite, les fonctions d'entrée/sortie peuvent être définies en référençant le squelette. Enfin, le **CREATE TYPE** avec une définition complète remplace le squelette avec une définition complète et valide du type, après quoi le nouveau type peut être utilisé normalement.

Les fonctions optionnelles *type_modifier_input_function* et *type_modifier_output_function* sont nécessaires si le type supporte des modificateurs, c'est-à-dire des contraintes optionnelles attachées à une déclaration de type comme `char(5)` ou `numeric(30,2)`. PostgreSQL™ autorise les types définis par l'utilisateur à prendre une ou plusieurs constantes ou identifiants comme modifieurs ; néanmoins, cette information doit être capable d'être englobée dans une seule valeur entière positive pour son stockage dans les catalogues système. *type_modifier_input_function* se voit fourni le modifieur déclaré de la forme d'un tableau de cstring. Il doit vérifier la validité des valeurs et renvoyer une erreur si elles sont invalides. Dans le cas contraire, il renvoie une valeur entière postivequi sera stockée dans la colonne « typmod ». Les modifieurs de type seront rejetés si le type n'a pas de *type_modifier_input_function*. *type_modifier_output_function* convertit la valeur typmod integer en une forme correcte pour l'affichage. Il doit renvoyer une valeur de type cstring qui est la chaîne exacte à ajouter au nom du type ; par exemple la fonction de numeric pourrait renvoyer `(30,2)`. Il est permis d'omettre le *type_modifier_output_function*, auquel cas le format d'affichage par défaut est simplement la valeur typmod stockée entre parenthèses.

La *fonction_analyse*, optionnelle, calcule des statistiques spécifiques au type de données pour les colonnes de ce type. Par défaut, **ANALYZE** tente de récupérer des statistiques à l'aide des opérateurs d'« égalité » et d'« infériorité » du type, s'il existe une classe d'opérateur B-tree par défaut pour le type. Ce comportement est inadapté aux types non-scalaires ; il peut être surchargé à l'aide d'une fonction d'analyse personnalisée. La fonction d'analyse doit être déclarée avec un seul argument de type internal et un résultat de type boolean. L'API détaillée des fonctions d'analyses est présentée dans `src/include/commands/vacuum.h`.

Alors que les détails de la représentation interne du nouveau type ne sont connus que des fonctions d'entrées/sorties et des fonctions utilisateurs d'interaction avec le type, plusieurs propriétés de la représentation interne doivent être déclarées à PostgreSQL™. La première est *longueurinterne*. Les types de données basiques peuvent être de longueur fixe (dans ce cas, *longueurinterne* est un entier positif) ou de longueur variable (indiquée par le positionnement de *longueurinterne* à `VARIABLE` ; en interne, cela est représenté en initialisant `typlen` à -1). La représentation interne de tous les types de longueur variable doit commencer par un entier de quatre octets indiquant la longueur totale de cette valeur.

Le drapeau optionnel `PASSEDBYVALUE` indique que les valeurs de ce type de données sont passées par valeur plutôt que par référence. Les types dont la représentation interne est plus grande que la taille du type Datum (quatre octets sur la plupart des machines, huit sur quelques-unes) ne doivent pas être passés par valeur.

Le paramètre *alignement* spécifie l'alignement de stockage requis pour le type de données. Les valeurs permises sont des ali-

gnements sur 1, 2, 4 ou 8 octets. Les types de longueurs variables ont un alignement d'au moins quatre octets car leur premier composant est nécessairement un int4.

Le paramètre *stockage* permet de choisir une stratégie de stockage pour les types de données de longueur variable. (Seul `plain` est autorisé pour les types de longueur fixe.) `plain` indique des données stockées en ligne et non compressées. Dans le cas d'`extended` le système essaie tout d'abord de compresser une valeur longue et déplace la valeur hors de la ligne de la table principale si elle est toujours trop longue. `external` permet à la valeur d'être déplacée hors de la table principale mais le système ne tente pas de la compresser. `main` autorise la compression mais ne déplace la valeur hors de la table principale qu'en dernier recours. (Ils seront déplacés s'il n'est pas possible de placer la ligne dans la table principale, mais sont préférentiellement conservés dans la table principale, contrairement aux éléments `extended` et `external`.)

Le paramètre *type_like* fournit une méthode alternative pour spécifier les propriétés de représentation de base d'un type de données : les copier depuis un type existant. Les valeurs de *longueurinterne*, *passedbyvalue*, *alignement* et *stockage* sont copiées du type indiqué. (C'est possible, mais habituellement non souhaité, d'écraser certaines de ces valeurs en les spécifiant en même temps que la clause `LIKE`.) Spécifier la représentation de cette façon est particulièrement pratique quand l'implémentation de bas niveau du nouveau type emprunte celle d'un type existant d'une façon ou d'une autre.

Les paramètres *catégorie* et *préféré* peuvent être utilisés pour aider à contrôler la conversion implicite appliquée en cas d'ambiguïté. Chaque type de données appartient à une catégorie identifiée par un seul caractère ASCII, et chaque type est « préféré » ou pas de sa catégorie. L'analyseur préfèrera convertir vers des types préférés (mais seulement à partir d'autres types dans la même catégorie) quand cette règle peut servir à résoudre des fonctions ou opérateurs surchargés. Pour plus de détails, voir Chapitre 10, Conversion de types. Pour les types qui n'ont pas de conversion implicite de ou vers d'autres types, on peut se contenter de laisser ces paramètres aux valeurs par défaut. Par contre, pour un groupe de types liés entre eux qui ont des conversions implicites, il est souvent pratique de les marquer tous comme faisant partie d'une même catégorie, et de choisir un ou deux des types les « plus généraux » comme étant les types préférés de la catégorie. Le paramètre *catégorie* est particulièrement utile quand on ajoute un type défini par l'utilisateur à un type interne, comme un type numérique ou chaîne. Toutefois, c'est aussi tout à fait possible de créer des catégories de types entièrement nouvelles. Choisissez un caractère ASCII autre qu'une lettre en majuscule pour donner un nom à une catégorie de ce genre.

Une valeur par défaut peut être spécifiée dans le cas où l'utilisateur souhaite que cette valeur soit différente de NULL pour les colonnes de ce type. La valeur par défaut est précisée à l'aide du mot clé `DEFAULT`. (Une telle valeur par défaut peut être surchargée par une clause `DEFAULT` explicite attachée à une colonne particulière.)

Pour indiquer qu'un type est un tableau, le type des éléments du tableau est précisé par le mot clé `ELEMENT`. Par exemple, pour définir un tableau d'entiers de quatre octets (int4), `ELEMENT = int4` est utilisé. Plus de détails sur les types tableau apparaissent ci-dessous.

Pour préciser le délimiteur de valeurs utilisé dans la représentation externe des tableaux de ce type, *délimiteur* peut être positionné à un caractère particulier. Le délimiteur par défaut est la virgule (`,`). Le délimiteur est associé avec le type élément de tableau, pas avec le type tableau.

Types tableau

À chaque fois qu'un type défini par un utilisateur est créé, PostgreSQL™ crée automatiquement un type tableau associé dont le nom est composé à partir du type de base préfixé d'un tiret bas et tronqué si nécessaire pour que le nom généré fasse moins de NAMEDATALEN octets. (Si le nom généré est en conflit avec un autre nom, le traitement est répété jusqu'à ce qu'un nom sans conflit soit trouvé.) Ce type tableau créé implicitement est de longueur variable et utilise les fonctions d'entrée et sortie `array_in` et `array_out`. Le type tableau trace tout changement dans du type de base pour le propriétaire et le schéma. Il est aussi supprimé quand le type de base l'est.

Pourquoi existe-t-il une option `ELEMENT` si le système fabrique automatiquement le bon type tableau ? La seule utilité d'`ELEMENT` est la création d'un type de longueur fixe représenté en interne par un tableau d'éléments identiques auxquels on souhaite accéder directement par leurs indices (en plus de toute autre opération effectuée sur le type dans sa globalité). Par exemple, le type point est représenté par deux nombres à virgule flottante, qui sont accessibles par `point[0]` et `point[1]`. Cette fonctionnalité n'est possible qu'avec les types de longueur fixe dont la forme interne est strictement une séquence de champs de longueur fixée. Un type de longueur variable est accessible par ses indices si sa représentation interne généralisée est celle utilisée par `array_in` et `array_out`. Pour des raisons historiques (c'est-à-dire pour de mauvaises raisons, mais il est trop tard pour changer) les indices des tableaux de types de longueur fixe commencent à zéro et non à un comme c'est le cas pour les tableaux de longueur variable.

Paramètres

nom
: Le nom (éventuellement qualifié du nom du schéma) du type à créer.

nom_attribut
: Le nom d'un attribut (colonne) du type composite.

type_données
Le nom d'un type de données existant utilisé comme colonne du type composite.

label
Une chaîne représentant le label associé à une valeur du type enum.

fonction_entrée
Le nom d'une fonction de conversion des données de la forme textuelle externe du type en forme interne.

fonction_sortie
Le nom d'une fonction de conversion des données de la forme interne du type en forme textuelle externe.

fonction_réception
Le nom d'une fonction de conversion des données de la forme binaire externe du type en forme interne.

fonction_envoi
Le nom d'une fonction de conversion des données de la forme interne du type en forme binaire externe.

type_modifier_input_function
Le nom d'une fonction qui convertit un tableau de modifieurs pour le type vers sa forme interne.

type_modifier_output_function
Le nom d'une fonction qui convertit la forme interne des modifieurs du type vers leur forme textuelle externe.

analyze_function
Le nom d'une fonction d'analyses statistiques pour le type de données.

longueurinterne
Une constante numérique qui précise la longueur en octets de la représentation interne du nouveau type. Supposée variable par défaut.

alignement
La spécification d'alignement du stockage du type de données. Peut être `char`, `int2`, `int4` ou `double` ; `int4` par défaut.

stockage
La stratégie de stockage du type de données. Peut être `plain`, `external`, `extended` ou `main` ; `plain` par défaut.

type_like
Le nom d'un type de données existant dont le nouveau type partagera la représentation. Les valeurs de *longueurinterne*, *passedbyvalue*, *alignement* et *stockage* sont recopiées à partir de ce type, sauf si elles sont écrasées explicitement ailleurs dans la même commande **CREATE TYPE**.

catégorie
Le code de catégorie (un unique caractère ASCII) pour ce type. La valeur par défaut est `U` pour « user-defined type » (type défini par l'utilisateur). Les autres codes standard de catégorie peuvent être trouvés dans Tableau 45.45, « Codes *typcategory* ». Vous pouvez aussi choisir d'autres caractères ASCII pour créer vos propres catégories personnalisées.

préféré
True si ce type est un type préféré dans sa catégorie de types, sinon false. La valeur par défaut est false. Faites très attention en créant un nouveau type préféré à l'intérieur d'une catégorie existante car cela pourrait créer des modifications surprenantes de comportement.

défaut
La valeur par défaut du type de données. Omise, elle est NULL.

élément
Type des éléments du type tableau créé.

délimiteur
Le caractère délimiteur des valeurs des tableaux de ce type.

Notes

Comme il n'y a pas de restrictions à l'utilisation d'un type de données une fois qu'il a été créé, créer un type de base est équivalent à donner les droits d'exécution ssur les fonctions mentionnées dans la définition du type. Ce n'est pas un problème habituellement pour le genre de fonctions utiles dans la définition d'un type mais réfléchissez bien avant de concevoir un type d'une façon qui nécessiterait que des informations « secrètes » soient utilisées lors de sa convertion vers ou à partir d'une forme externe.

Avant PostgreSQL™ version 8.3, le nom d'un type tableau généré était toujours exactement le nom du type élément avec un caractère tiret bas (_) en préfixe. (Les noms des types étaient du coup limités en longueur à un caractère de moins que les autres

noms.) Bien que cela soit toujours le cas, le nom d'un type tableau peut varier entre ceci dans le cas des noms de taille maximum et les collisions avec des noms de type utilisateur qui commencent avec un tiret bas. Écrire du code qui dépend de cette convention est du coup obsolète. À la place, utilisez pg_type.*typarray* pour situer le type tableau associé avec un type donné.

Il est conseillé d'éviter d'utiliser des noms de table et de type qui commencent avec un tiret bas. Alors que le serveur changera les noms des types tableau générés pour éviter les collisions avec les noms donnés par un utilisateur, il reste toujours un risque de confusion, particulièrement avec les anciens logiciels clients qui pourraient supposer que les noms de type commençant avec un tiret bas représentent toujours des tableaux.

Avant PostgreSQL™ version 8.2, la syntaxe `CREATE TYPE` *nom* n'existait pas. La façon de créer un nouveau type de base était de créer en premier les fonctions paramètres. Dans cette optique, PostgreSQL™ verra tout d'abord le nom d'un nouveau type de données comme type de retour de la fonction en entrée. Le type shell est créé implicitement dans ce cas et il est ensuite référencé dans le reste des fonctions d'entrée/sortie. Cette approche fonctionne toujours mais est obsolète et pourrait être interdite dans une version future. De plus, pour éviter de faire grossir les catalogues de façon accidentelle avec des squelettes de type erronés, un squelette sera seulement créé quand la fonction en entrée est écrit en C.

Dans les versions de PostgreSQL™ antérieures à la 7.3, la création d'un type coquille était habituellement évitée en remplaçant les références des fonctions au nom du type par le pseudotype opaque. Les arguments cstring et les résultats étaient également déclarés opaque. Pour supporter le chargement d'anciens fichiers de sauvegarde, **CREATE TYPE** accepte les fonctions d'entrées/sorties déclarées avec le pseudotype opaque mais un message d'avertissement est affiché. La déclaration de la fonction est également modifiée pour utiliser les bons types.

Exemples

Créer un type composite utilisé dans la définition d'une fonction :

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

Cet exemple crée un type énuméré et l'utilise dans la création d'une table :

```
CREATE TYPE statut_bogue AS ENUM ('nouveau', 'ouvert', 'fermé');

CREATE TABLE bogue (
    id serial,
    description text,
    status statut_bogue
);
```

Créer le type de données basique box utilisé dans la définition d'une table :

```
CREATE TYPE box;

CREATE FUNCTION ma_fonction_entree_box(cstring) RETURNS box AS ... ;
CREATE FUNCTION ma_fonction_sortie_box(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = ma_fonction_entree_box,
    OUTPUT = ma_fonction_sortie_box
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

Si la structure interne de box est un tableau de quatre éléments float4, on peut écrire :

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = ma_fonction_entree_box,
    OUTPUT = ma_fonction_sortie_box,
    ELEMENT = float4
);
```

ce qui permet d'accéder aux nombres composant la valeur d'une boîte par les indices. Le comportement du type n'est pas modifié.

Créer un objet large utilisé dans la définition d'une table :

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

D'autres exemples, intégrant des fonctions utiles d'entrée et de sortie, peuvent être consultés dans Section 35.11, « Types utilisateur ».

Compatibilité

La commande **CREATE TYPE** présentée ici est une extension PostgreSQL™. L'instruction **CREATE TYPE** du standard SQL est, dans le détail, assez différente.

Voir aussi

CREATE FUNCTION(7), DROP TYPE(7), ALTER TYPE(7), CREATE DOMAIN(7)

Nom

CREATE USER — Définir un nouveau rôle de base de données

Synopsis

```
CREATE USER nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'motdepasse'
    | VALID UNTIL 'dateheure'
    | IN ROLE nom_role [, ...]
    | IN GROUP nom_role [, ...]
    | ROLE nom_role [, ...]
    | ADMIN nom_role [, ...]
    | USER nom_role [, ...]
    | SYSID uid
```

Description

CREATE USER est dorénavant un alias de CREATE ROLE(7). Il y a toutefois une petite différence entre les deux commandes. Lorsque la commande **CREATE USER** est éxécutée, `LOGIN` est le comportement par défaut. Au contraire, quand **CREATE ROLE** est exécutée, `NOLOGIN` est utilisé.

Compatibilité

L'instruction **CREATE USER** est une extension PostgreSQL™. Le standard SQL laisse la définition des utilisateurs à l'implantation.

Voir aussi

CREATE ROLE(7)

Nom

CREATE USER MAPPING — Définir une nouvelle correspondance d'utilisateur (*user mapping*) pour un serveur distant

Synopsis

```
CREATE USER MAPPING FOR { nom_utilisateur | USER | CURRENT_USER | PUBLIC }
    SERVER nom_serveur
    [ OPTIONS ( option 'valeur' [ , ... ] ) ]
```

Description

CREATE USER MAPPING définit une nouvelle correspondance d'utilisateur (*user mapping*) pour un serveur distant. Une correspondance d'utilisateur englobe typiquement les informations de connexion qu'un wrapper de données distantes utilise avec l'information d'un serveur distant pour accéder à des ressources externes de données.

Le propriétaire d'un serveur distant peut créer des correspondances d'utilisateur pour ce serveur pour n'importe quel utilisateur. Par ailleurs, un utilisateur peut créer une correspondance d'utilisateur pour son propre nom d'utilisateur si le droit `USAGE` a été donné sur le serveur à son utilisateur.

Paramètres

`nom_utilisateur`
: Le nom d'un utilisateur existant qui est mis en correspondance sur un serveur distant. `CURRENT_USER` et `USER` correspondent au nom de l'utilisateur courant. Quand `PUBLIC` est ajoutée, une correspondance appelée publique est créée pour être utilisée quand aucune correspondance d'utilisateur spécifique n'est applicable.

`nom_serveur`
: Le nom d'un serveur existant pour lequel la correspondance d'utilisateur sera créée.

`OPTIONS ( option 'valeur' [, ... ] )`
: Cette clause définit les options pour la correspondance d'utilisateurs. Les options définissent typiquement le nom et le mot de passe réels de la correspondance. Les nom d'options doivent être uniques. Les noms et valeurs d'options autorisés sont propres au wrapper de données étrangère du serveur.

Exemples

Créer une correspondance d'utilisateur pour l'utilisateur `bob`, sur le serveur `truc` :

```
CREATE USER MAPPING FOR bob SERVER truc OPTIONS (user 'bob', password 'secret');
```

Compatibilité

CREATE USER MAPPING est conforme à la norme ISO/IEC 9075-9 (SQL/MED).

Voir aussi

ALTER USER MAPPING(7), DROP USER MAPPING(7), CREATE FOREIGN DATA WRAPPER(7), CREATE SERVER(7)

Nom

CREATE VIEW — Définir une vue

Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW nom [ ( nom_colonne [, ...] ) ]
    AS requête
```

Description

CREATE VIEW définit une vue d'après une requête. La vue n'est pas matérialisée physiquement. Au lieu de cela, la requête est lancée chaque fois qu'une vue est utilisée dans une requête.

CREATE OR REPLACE VIEW a la même finalité, mais si une vue du même nom existe déjà, elle est remplacée. La nouvelle requête doit générer les mêmes colonnes que celles de l'ancienne requête (c-est-à-dire les mêmes noms de colonnes dans le même ordre avec les mêmes types de données). Par contre, elle peut ajouter des colonnes supplémentaires en fin de liste. Les traitements qui donnent les colonnes en sortie pourraient être complètement différents.

Si un nom de schéma est donné (par exemple `CREATE VIEW monschema.mavue ...`), alors la vue est créée dans ce schéma. Dans le cas contraire, elle est créée dans le schéma courant. Les vues temporaires existent dans un schéma spécial. Il n'est donc pas nécessaire de fournir de schéma pour les vues temporaires. Le nom de la vue doit être différent du nom de toute autre vue, table, séquence ou index du même schéma.

Paramètres

`TEMPORARY` ou `TEMP`
: La vue est temporaire. Les vues temporaires sont automatiquement supprimées en fin de session. Les relations permanentes qui portent le même nom ne sont plus visibles pour la session tant que la vue temporaire existe, sauf s'il y est fait référence avec le nom du schéma.

  Si l'une des tables référencées par la vue est temporaire, la vue est alors elle-aussi temporaire (que `TEMPORARY` soit spécifié ou non).

`nom`
: Le nom de la vue à créer (éventuellement qualifié du nom du schéma).

`nom de colonne`
: Une liste optionnelle de noms à utiliser pour les colonnes de la vue. Si elle n'est pas donnée, le nom des colonnes est déduit de la requête.

`requête`
: Une commande SELECT(7) ou VALUES(7) qui fournira les colonnes et lignes de la vue.

Notes

Actuellement, les vues sont en lecture seule : le système n'autorise pas une insertion, une mise à jour ou une suppression sur une vue. Les effets d'une vue actualisable peuvent être reproduits par la création de règles de réécriture des insertions, suppressions, mises à jour, etc. sur la vue en actions appropriées sur les autres tables. CREATE RULE(7) présente des informations utiles à ce propos.

L'instruction DROP VIEW(7) est utilisée pour supprimer les vues.

Il est important de s'assurer que le nom et le type des colonnes de la vue correspondent à ce qui est souhaité. Ainsi :

```
CREATE VIEW vista AS SELECT 'Hello World';
```

présente deux défauts majeurs : le nom de la colonne prend la valeur implicite `?column?` et son type de données le type implicite unknown. Pour obtenir une chaîne de caractères dans le résultat de la vue, on peut écrire :

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

L'accès aux tables référencées dans la vue est déterminé par les droits du propriétaire de la vue. Dans certains cas, cela peut être utilisé pour fournir un accès sécurisé. Cependant, toutes les vues ne sont pas sécurisables ; voir Section 37.4, « Règles et droits » pour des détails. Les fonctions appelées dans la vue sont traitées de la même façon que si elles avaient été appelées directement dans la requête utilisant la vue. Du coup, l'utilisateur d'une vue doit avoir les droits pour appeler toutes les fonctions utilisées par

la vue.

Quand **CREATE OR REPLACE VIEW** est utilisé sur une vue existante, seule la règle SELECT définissant la vue est modifiée. Les autres propriétés, comme les droits, le propriétaire et les règles autres que le SELECT, ne sont pas modifiées. Vous devez être le propriétaire de la vue pour la remplacer (ceci incluant aussi les membres du rôle propriétaire).

Exemples

Créer une vue composée des comédies :

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE genre = 'Comédie';
```

Compatibilité

Le standard SQL spécifie quelques possibilités supplémentaires pour l'instruction **CREATE VIEW** :

```
CREATE VIEW nom [ ( nom_colonne [, ...] ) ]
    AS requête
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Les clauses optionnelles de la commande SQL complète sont :

`CHECK OPTION`
: Cette option concerne les vues actualisables. Toutes les commandes **INSERT** et **UPDATE** appliquées à la vue sont contrôlées pour s'assurer que les données satisfont les conditions de définition de la vue (les nouvelles données sont visibles au travers de la vue). Si ce n'est pas le cas, la mise à jour est rejetée.

`LOCAL`
: Contrôle d'intégrité de la vue.

`CASCADED`
: Contrôle d'intégrité de la vue et de toutes les vues dépendantes. `CASCADED` est implicite si ni `CASCADED` ni `LOCAL` ne sont précisés.

CREATE OR REPLACE VIEW est une extension PostgreSQL™, tout comme le concept de vue temporaire.

Voir aussi

ALTER VIEW(7), DROP VIEW(7)

Nom

DEALLOCATE — Désaffecter (libérer) une instruction préparée

Synopsis

```
DEALLOCATE [ PREPARE ] { nom | ALL }
```

Description

DEALLOCATE est utilisé pour désaffecter une instruction SQL préparée précédemment. Une instruction préparée qui n'est pas explicitement libérée l'est automatiquement en fin de session.

Pour plus d'informations sur les instructions préparées, voir PREPARE(7).

Paramètres

`PREPARE`
: Mot clé ignoré.

`nom`
: Le nom de l'instruction préparée à désaffecter.

`ALL`
: Désaffecte toutes les instructions préparées.

Compatibilité

Le standard SQL inclut une instruction **DEALLOCATE** qui n'est utilisée que pour le SQL imbriqué.

Voir aussi

EXECUTE(7), PREPARE(7)

Nom

DECLARE — Définir un curseur

Synopsis

```
DECLARE nom [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR requête
```

Description

DECLARE permet à un utilisateur de créer des curseurs. Ils peuvent être utilisés pour récupérer un petit nombre de lignes à la fois à partir d'une requête plus importante. Après la création du curseur, les lignes sont récupérées en utilisant FETCH(7).

> **Note**
>
> Cette page décrit l'utilisation des curseurs au niveau de la commande SQL. Si vous voulez utiliser des curseurs dans une fonction PL/pgSQL, les règles sont différentes -- voir Section 39.7, « Curseurs ».

Paramètres

nom
: Le nom du curseur à créer.

`BINARY`
: Le curseur retourne les données au format binaire.

`INSENSITIVE`
: Les données récupérées à partir du curseur ne doivent pas être affectées par les mises à jour des tables concernées par le curseur qui surviennent une fois que ce dernier est créé. Dans PostgreSQL™, c'est le comportement par défaut ; ce mot-clé n'a aucun effet. Il est seulement accepté pour des raisons de compatibilité avec le standard SQL.

`SCROLL`, `NO SCROLL`
: `SCROLL` indique une utilisation possible du curseur pour récupérer des lignes de façon non séquentielle (c'est-à-dire en remontant la liste). En fonction de la complexité du plan d'exécution de la requête, `SCROLL` peut induire des pertes de performance sur le temps d'exécution de la requête. `NO SCROLL` indique que le curseur ne peut pas être utilisé pour récupérer des lignes de façon non séquentielle. La valeur par défaut autorise la non-séquentialité du curseur dans certains cas ; ce n'est pas la même chose que de spécifier `SCROLL`. Voir la section intitulée « Notes » pour les détails.

`WITH HOLD`, `WITHOUT HOLD`
: `WITH HOLD` (NDT : persistant) indique une utilisation possible du curseur après la validation de la transaction qui l'a créé. `WITHOUT HOLD` (NDT : volatil) interdit l'utilisation du curseur en dehors de la transaction qui l'a créé. `WITHOUT HOLD` est la valeur par défaut.

requête
: Une commande SELECT(7) ou VALUES(7) qui fournira les lignes à renvoyer par le curseur.

Les mots clés `BINARY`, `INSENSITIVE` et `SCROLL` peuvent apparaître dans n'importe quel ordre.

Notes

Les curseurs normaux renvoient les données au format texte, le même que produirait un **SELECT**. L'option `BINARY` spécifie que le curseur doit renvoyer les données au format binaire. Ceci réduit les efforts de conversion pour le serveur et le client, au coût d'un effort particulier de développement pour la gestion des formats de données binaires dépendants des plateformes. Comme exemple, si une requête renvoie une valeur de un dans une colonne de type integer, vous obtiendrez une chaîne `1` avec un curseur par défaut. Avec un curseur binaire, vous obtiendrez un champ sur quatre octet contenant la représentation interne de la valeur (dans l'ordre big-endian).

Les curseurs binaires doivent être utilisés en faisant très attention. Beaucoup d'applications, incluant psql, ne sont pas préparées à gérer des curseurs binaires et s'attendent à ce que les données reviennent dans le format texte.

> **Note**

> Quand l'application cliente utilise le protocole des « requêtes étendues » pour exécuter la commande **FETCH**, le message Bind du protocole spécifie si les données sont à récupérer au format texte ou binaire. Ce choix surcharge la façon dont le curseur est défini. Le concept de curseur binaire est donc obsolète lors de l'utilisation du protocole des requêtes étendues -- tout curseur peut être traité soit en texte soit en binaire.

Si la clause `WITH HOLD` n'est pas précisée, le curseur créé par cette commande ne peut être utilisé qu'à l'intérieur d'une transaction. Ainsi, **DECLARE** sans `WITH HOLD` est inutile à l'extérieur d'un bloc de transaction : le curseur survivrait seulement jusqu'à la fin de l'instruction. PostgreSQL™ rapporte donc une erreur si cette commande est utilisée en dehors d'un bloc de transactions. On utilise BEGIN(7) et COMMIT(7) (ou ROLLBACK(7)) pour définir un bloc de transaction.

Si la clause `WITH HOLD` est précisée, et que la transaction qui a créé le curseur est validée, ce dernier reste accessible par les transactions ultérieures de la session. Au contraire, si la transaction initiale est annulée, le curseur est supprimé. Un curseur créé avec la clause `WITH HOLD` est fermé soit par un appel explicite à la commande **CLOSE**, soit par la fin de la session. Dans l'implantation actuelle, les lignes représentées par un curseur persistant (`WITH HOLD`) sont copiées dans un fichier temporaire ou en mémoire afin de garantir leur disponibilité pour les transactions suivantes.

`WITH HOLD` n'est pas utilisable quand la requête contient déjà `FOR UPDATE` ou `FOR SHARE`.

L'option `SCROLL` est nécessaire à la définition de curseurs utilisés en récupération remontante (retour dans la liste des résultats, backward fetch), comme précisé par le standard SQL. Néanmoins, pour des raisons de compatibilité avec les versions antérieures, PostgreSQL™ autorise les récupérations remontantes sans que l'option `SCROLL` ne soit précisé, sous réserve que le plan d'exécution du curseur soit suffisamment simple pour être géré sans surcharge. Toutefois, il est fortement conseillé aux développeurs d'application ne pas utiliser les récupérations remontantes avec des curseurs qui n'ont pas été créés avec l'option `SCROLL`. Si `NO SCROLL` est spécifié, les récupérations remontantes sont toujours dévalidées.

Les parcours inverses sont aussi interdits lorsque la requête inclut les clauses `FOR UPDATE` et `FOR SHARE` ; donc `SCROLL` peut ne pas être indiqué dans ce cas.

> **Attention**
>
> Les curseurs scrollables et avec l'option `WITH HOLD` pourraient donner des résultats inattendues s'ils font appel à des fonctions volatiles (voir Section 35.6, « Catégories de volatilité des fonctions »). Quand une ligne précédemment récupérée est de nouveau récupérée, la fonction pourrait être ré-exécutée, amenant peut-être des résultats différentes de la première exécution. Un contournement est de déclarer le curseur `WITH HOLD` et de valider la transaction avant de lire toute ligne de ce curseur. Cela forcera la sortie entière du cuseur à être matérialisée dans un stockage temporaire, pour que les fonctions volatiles soient exécutées exactement une fois pour chaque ligne.

Si la requête du curseur inclut les clauses `FOR UPDATE` ou `FOR SHARE`, alors les lignes renvoyées sont verrouillées au moment où elles sont récupérées, de la même façon qu'une commande SELECT(7) standard avec ces options. De plus, les lignes renvoyées seront les versions les plus à jour ; du coup, ces options fournissent l'équivalent de ce que le standard SQL appelle un « curseur sensible ». (Indiquer `INSENSITIVE` avec soit `FOR UPDATE` soit `FOR SHARE` est une erreur.)

> **Attention**
>
> Il est généralement recommandé d'utiliser `FOR UPDATE` si le curseur doit être utilisé avec **UPDATE ... WHERE CURRENT OF** ou **DELETE ... WHERE CURRENT OF**. Utiliser `FOR UPDATE` empêche les autres sessions de modifier les lignes entre le moment où elles sont récupérées et celui où elles sont modifiées. Sans `FOR UPDATE`, une commande `WHERE CURRENT OF` suivante n'aura pas d'effet si la ligne a été modifiée depuis la création du curseur.
>
> Une autre raison d'utiliser `FOR UPDATE` est que, sans ce dernier, un appel suivant à `WHERE CURRENT OF` pourrait échouer si la requête curseur ne répond pas aux règles du standard SQL d'être « mise à jour simplement » (en particulier, le curseur doit référencer une seule table et ne pas utiliser de regroupement ou de tri comme `ORDER BY`). Les curseurs qui ne peuvent pas être mis à jour pourraient fonctionner, ou pas, suivant les détails du plan choisi ; dans le pire des cas, une application pourrait fonctionner lors des tests puis échouer en production.
>
> La principale raison de ne pas utiliser `FOR UPDATE` avec `WHERE CURRENT OF` est si vous avez besoin que le curseur soit déplaçable ou qu'il soit insensible aux mises à jour suivantes (c'est-à-dire qu'il continue à afficher les anciennes données). Si c'est un prérequis, faites très attention aux problèmes expliqués ci-dessus.

Le standard SQL ne mentionne les curseurs que pour le SQL embarqué. PostgreSQL™ n'implante pas l'instruction **OPEN** pour les curseurs ; un curseur est considéré ouvert à sa déclaration. Néanmoins, ECPG, le préprocesseur de SQL embarqué pour PostgreSQL™, supporte les conventions du standard SQL relatives aux curseurs, dont celles utilisant les instructions **DECLARE** et **OPEN**.

Vous pouvez voir tous les curseurs disponibles en exécutant une requête sur la vue système pg_cursors.

Exemples

Déclarer un curseur :

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

Voir FETCH(7) pour plus d'exemples sur l'utilisation des curseurs.

Compatibilité

Le standard SQL indique que la sensibilité des curseurs aux mises à jour en parallèle des données récupérées est dépendante de l'implantation par défaut. Dans PostgreSQL™, les curseurs n'ont pas ce comportement par défaut, mais peuvent le devenir en ajoutant `FOR UPDATE`. D'autres produits peuvent gérer cela différement.

Le standard SQL n'autorise les curseurs que dans le SQL embarqué et dans les modules. PostgreSQL™ permet une utilisation interactive des curseurs.

Le standard SQL autorise les curseurs à mettre à jour les données d'une table. Tous les curseurs PostgreSQL™ sont en lecture seule.

Les curseurs binaires sont une extension PostgreSQL™.

Voir aussi

CLOSE(7), FETCH(7), MOVE(7)

Nom

DELETE — Supprimer des lignes d'une table

Synopsis

```
DELETE FROM [ ONLY ] table [ * ] [ [ AS ] alias ]
    [ USING liste_using ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
    [ RETURNING * | expression_sortie [ [ AS ] output_name ] [, ...] ]
```

Description

DELETE supprime de la table spécifiée les lignes qui satisfont la clause `WHERE`. Si la clause `WHERE` est absente, toutes les lignes de la table sont supprimées. Le résultat est une table valide, mais vide.

> **Astuce**
>
> TRUNCATE(7) est une extension PostgreSQL™ qui fournit un mécanisme plus rapide de suppression de l'ensemble des lignes d'une table.

Il existe deux façons de supprimer des lignes d'une table en utilisant les informations d'autres tables de la base de données : les sous-sélections ou la spécification de tables supplémentaires dans la clause `USING`. La technique la plus appropriée dépend des circonstances.

La clause `RETURNING` optionnelle fait que **DELETE** calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours de suppression. Toute expression utilisant les colonnes de la table et/ou les colonnes de toutes les tables mentionnées dans `USING` peut être calculée. La syntaxe de la liste `RETURNING` est identique à celle de la commande **SELECT**.

Il est nécessaire de posséder le droit `DELETE` sur la table pour en supprimer des lignes, et le droit `SELECT` sur toute table de la clause `USING` et sur toute table dont les valeurs sont lues dans la *`condition`*.

Paramètres

`table`
: Le nom (éventuellement qualifié du nom du schéma) de la table dans laquelle il faut supprimer des lignes. Si `ONLY` est indiqué avant le nom de la table, les lignes supprimées ne concernent que la table nommée. Si `ONLY` n'est pas indiquée, les lignes supprimées font partie de la table nommée et de ses tables filles. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être incluses.

`alias`
: Un nom de substitution pour la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table. Par exemple, avec `DELETE FROM foo AS f`, le reste de l'instruction **DELETE** doit référencer la table avec `f` et non plus `foo`.

`liste_using`
: Une liste d'expressions de table, qui permet de faire apparaître des colonnes d'autres tables dans la condition `WHERE`. C'est semblable à la liste des tables utilisées dans la clause la section intitulée « Clause `FROM` » d'une instruction **SELECT** ; un alias du nom d'une table peut ainsi être utilisé. La table cible ne doit pas être précisée dans *`liste_using`*, sauf si une auto-jointure est envisagée.

`condition`
: Une expression retournant une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie `true` seront supprimées.

`nom_curseur`
: Le nom du curseur à utiliser dans une condition `WHERE CURRENT OF`. La ligne à supprimer est la dernière ligne récupérée avec ce curseur. Le curseur doit être une requête sans regroupement sur la table cible du **DELETE**. Notez que `WHERE CURRENT OF` ne peut pas se voir ajouter de condition booléenne. Voir DECLARE(7) pour plus d'informations sur l'utilisation des curseurs avec `WHERE CURRENT OF`.

`expression_sortie`
: Une expression à calculer et renvoyée par la commande **DELETE** après chaque suppression de ligne. L'expression peut utiliser tout nom de colonne de la *`table`* ou des tables listées dans la clause `USING`. Indiquez `*` pour que toutes les colonnes

soient renvoyées.

nom_sortie
Un nom à utiliser pour une colonne renvoyée.

Sorties

En cas de succès, une commande **DELETE** renvoie une information de la forme

```
DELETE nombre
```

Le *nombre* correspond au nombre de lignes supprimées. Si *nombre* vaut 0, c'est qu'aucune ligne ne correspond à *condition* (ce qui n'est pas considéré comme une erreur).

Si la commande **DELETE** contient une clause `RETURNING`, le résultat sera similaire à celui d'une instruction **SELECT** contenant les colonnes et les valeurs définies dans la liste `RETURNING`, à partir de la liste des lignes supprimées par la commande.

Notes

PostgreSQL™ autorise les références à des colonnes d'autres tables dans la condition `WHERE` par la spécification des autres tables dans la clause `USING`. Par exemple, pour supprimer tous les films produits par un producteur donné :

```
DELETE FROM films USING producteurs
  WHERE id_producteur = producteurs.id AND producteurs.nom = 'foo';
```

Pour l'essentiel, une jointure est établie entre films et producteurs avec toutes les lignes jointes marquées pour suppression. Cette syntaxe n'est pas standard. Une façon plus standard de procéder consiste à utiliser une sous-selection :

```
DELETE FROM films
  WHERE id_producteur IN (SELECT id FROM producteur WHERE nom = 'foo');
```

Dans certains cas, la jointure est plus facile à écrire ou plus rapide à exécuter que la sous-sélection.

Exemples

Supprimer tous les films qui ne sont pas des films musicaux :

```
DELETE FROM films WHERE genre <> 'Comédie musicale';
```

Effacer toutes les lignes de la table `films` :

```
DELETE FROM films;
```

Supprimer les tâches terminées tout en renvoyant le détail complet des lignes supprimées :

```
DELETE FROM taches WHERE statut = 'DONE' RETURNING *;
```

Supprimer la ligne de taches sur lequel est positionné le curseur `c_taches` :

```

DELETE FROM taches WHERE CURRENT OF c_taches;
```

Compatibilité

Cette commande est conforme au standard SQL, à l'exception des clauses `USING` et `RETURNING`, qui sont des extensions de PostgreSQL™.

Nom

DISCARD — Annuler l'état de la session

Synopsis

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

Description

DISCARD libère les ressources internes associées avec une session de la base de données. Ces ressources sont normalementlibérer à la fin de la session.

DISCARD TEMP supprime toutes les tables temporaires créées pendant cette session. **DISCARD PLANS** libère tous les plans internes de requête mis en cache. **DISCARD ALL** réinitialise une session à son état d'origine, supprimant ainsi les ressources temporaires et réinitialisant les modifications locales de configuration de la session.

Paramètres

`TEMPORARY` or `TEMP`
: Supprime toutes les tables temporaires créées pendant cette session.

`PLANS`
: Libère tous les plans de requête mis en cache.

`ALL`
: Libère les ressources temporaires associées à cette session et réinitialise une session à son état d'origine. Actuellement, ceci a le même effet que la séquence d'instructions suivantes :

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
```

Notes

DISCARD ALL ne peut pas être utilisé dans un bloc de transaction.

Compatibilité

DISCARD est une extension PostgreSQL™.

Nom

DO — exécute un bloc de code anonyme

Synopsis

```
DO [ LANGUAGE nom_langage ] code
```

Description

DO exécute un bloc de code anonyme, autrement dit une fonction temporaire dans le langage de procédure indiqué.

Le bloc de code est traité comme le corps d'une fonction sans paramètre et renvoyant void. Il est analysé et exécuté une seule fois.

La clause `LANGUAGE` optionnelle est utilisable avant ou après le bloc de code.

Paramètres

`code`
: Le code à exécuter. Il doit être spécifié comme une chaîne litérale, tout comme une fonction **CREATE FUNCTION**. L'utilisation de la syntaxe des guillemets dollar est recommandée.

`nom_langage`
: Le nom du langage utilisé par le code. Par défaut à `plpgsql`.

Notes

Le langage de procédure utilisé doit déjà être installé dans la base de données avec l'instruction **CREATE LANGUAGE**. `plpgsql` est installé par défaut contrairement aux autres langages.

L'utilisateur doit avoir le droit `USAGE` sur le langage de procédures ou être un superutilisateur s'il ne s'agit pas d'un langage de confiance. Il s'agit des mêmes prérequis que pour la création d'une fonction dans ce langage.

Exemples

Donner les droits sur toutes les vues du schéma `public` au rôle `webuser` :

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

Compatibilité

Il n'existe pas d'instruction **DO** dans le standard SQL.

Voir aussi

CREATE LANGUAGE(7)

Nom

DROP AGGREGATE — Supprimer une fonction d'agrégat

Synopsis

```
DROP AGGREGATE [ IF EXISTS ] nom ( type ) [ CASCADE | RESTRICT ]
```

Description

DROP AGGREGATE supprime une fonction d'agrégat. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire de la fonction.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom (éventuellement qualifié du nom de schéma) d'une fonction d'agrégat.

`type`
: Un type de données en entrée avec lequel la fonction d'agrégat opère. input data type on which the aggregate function operates. Pour référencer une fonction d'agrégat sans arguments, écrivez `*` à la place de la liste des types.

`CASCADE`
: Les objets qui dépendent de la fonction d'agrégat sont automatiquement supprimés.

`RESTRICT`
: La fonction d'agrégat n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la fonction d'agrégat `mamoyenne` pour le type integer :

```
DROP AGGREGATE mamoyenne(integer);
```

Compatibilité

Il n'existe pas d'instruction **DROP AGGREGATE** dans le standard SQL.

Voir aussi

ALTER AGGREGATE(7), CREATE AGGREGATE(7)

Nom

DROP CAST — Supprimer un transtypage

Synopsis

```
DROP CAST [ IF EXISTS ] (type_source AS type_cible) [ CASCADE | RESTRICT ]
```

Description

DROP CAST supprime un transtypage (conversion entre deux types de données) précédemment défini.

Seul le propriétaire du type de données source ou cible peut supprimer un transtypage. Les mêmes droits sont requis que pour la création d'un transtypage.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`type_source`
: Le nom du type de données source du transtypage.

`type_cible`
: Le nom du type de données cible du transtypage.

`CASCADE`, `RESTRICT`
: Ces mots clés n'ont pas d'effet car il n'y aucune dépendance dans les transtypages.

Exemples

Supprimer le transtypage du type text en type int :

```
DROP CAST (text AS int);
```

Compatibilité

La commande **DROP CAST** est conforme au standard SQL.

Voir aussi

CREATE CAST(7)

Nom

DROP CONVERSION — Supprimer une conversion

Synopsis

```
DROP CONVERSION [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP CONVERSION supprime une conversion précédemment définie. Seul son propriétaire peut supprimer une conversion.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom
: Le nom de la conversion (éventuellement qualifié du nom de schéma).

`CASCADE`, `RESTRICT`
: Ces mots clés n'ont pas d'effet car il n'existe pas de dépendances sur les conversions.

Exemples

Supprimer la conversion nommée `mon_nom` :

```
DROP CONVERSION mon_nom;
```

Compatibilité

Il n'existe pas d'instruction **DROP CONVERSION** dans le standard SQL. Par contre, une instruction **DROP TRANSLATION** est disponible. Elle va de paire avec l'instruction **CREATE TRANSLATION** qui est similaire à l'instruction **CREATE CONVERSION** de PostgreSQL.

Voir aussi

ALTER CONVERSION(7), CREATE CONVERSION(7)

Nom

DROP DATABASE — Supprimer une base de données

Synopsis

```
DROP DATABASE [ IF EXISTS ] nom
```

Description

La commande **DROP DATABASE** détruit une base de données. Elle supprime les entrées du catalogue pour la base et le répertoire contenant les données. Elle ne peut être exécutée que par le propriétaire de la base de données ou le superutilisateur. De plus, elle ne peut être exécutée si quelqu'un est connecté sur la base de données cible, y compris l'utilisateur effectuant la demande de suppression. (On peut se connecter à `postgres` ou à toute autre base de données pour lancer cette commande.)

DROP DATABASE ne peut pas être annulée. Il convient donc de l'utiliser avec précaution !

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`name`
: Le nom de la base de données à supprimer.

Notes

DROP DATABASE ne peut pas être exécutée à l'intérieur d'un bloc de transactions.

Cette commande ne peut pas être exécutée en cas de connexion à la base de données cible. Il peut paraître plus facile d'utiliser le programme dropdb(1) à la place, qui est un enrobage de cette commande.

Compatibilité

Il n'existe pas d'instruction **DROP DATABASE** dans le standard SQL.

Voir aussi

CREATE DATABASE(7), Variables d'environnement (Section 31.13, « Variables d'environnement »)

Nom

DROP DOMAIN — Supprimer un domaine

Synopsis

```
DROP DOMAIN [ IF EXISTS ] nom [, ...]  [ CASCADE | RESTRICT ]
```

Description

DROP DOMAIN supprime un domaine. Seul le propriétaire d'un domaine peut le supprimer.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom
: Le nom (éventuellement qualifié du nom du schéma) d'un domaine.

`CASCADE`
: Les objets qui dépendent du domaine (les colonnes de table, par exemple) sont automatiquement supprimés.

`RESTRICT`
: Le domaine n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer le domaine boite :

```
DROP DOMAIN boite;
```

Compatibilité

Cette commande est conforme au standard SQL, à l'exception de l'option `IF EXISTS` qui est une extension PostgreSQL™.

Voir aussi

CREATE DOMAIN(7), ALTER DOMAIN(7)

Nom

DROP FOREIGN DATA WRAPPER — Supprimer un wrapper de données distantes

Synopsis

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP FOREIGN DATA WRAPPER supprime un wrapper de données distantes existant. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire du wrapper de données distantes.

Paramètres

`IF EXISTS`
: Ne génère pas d'erreur si le wrapper de données distantes n'existe pas. Un avertissement est émis dans ce cas.

`nom`
: Le nom d'un wrapper de données distantes existant.

`CASCADE`
: Supprime automatiquement les objets dépendant du wrapper de données distantes (tels que les serveurs).

`RESTRICT`
: Refuse de supprimer le wrapper de données distantes si un objet dépend de celui-ci. C'est le cas par défaut.

Exemples

Supprimer le wrapper de données distantes `dbi` :

```
DROP FOREIGN DATA WRAPPER dbi;
```

Compatibilité

DROP FOREIGN DATA WRAPPER est conforme à la norme ISO/IEC 9075-9 (SQL/MED). La clause `IF EXISTS` est une extension PostgreSQL™ .

Voir aussi

CREATE FOREIGN DATA WRAPPER(7), ALTER FOREIGN DATA WRAPPER(7)

Nom

DROP FUNCTION — Supprimer une fonction

Synopsis

```
DROP FUNCTION [ IF EXISTS ] nom ( [ [ modearg ] [ nomarg ] typearg [, ...] ] )
    [ CASCADE | RESTRICT ]
```

Description

DROP FUNCTION supprime la définition d'une fonction. Seul le propriétaire de la fonction peut exécuter cette commande. Les types d'argument de la fonction doivent être précisés car plusieurs fonctions peuvent exister avec le même nom et des listes différentes d'arguments.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom (éventuellement qualifié du nom du schéma) de la fonction.

`modearg`
: Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. Sans précision, la valeur par défaut est `IN`. **DROP FUNCTION** ne s'intéresse pas aux arguments `OUT` car seuls ceux en entrée déterminent l'identité de la fonction. Il est ainsi suffisant de lister les arguments `IN`, `INOUT` et `VARIADIC`.

`nomarg`
: Le nom d'un argument. **DROP FUNCTION** ne tient pas compte des noms des arguments car seuls les types de données sont nécessaires pour déterminer l'identité de la fonction.

`typearg`
: Le(s) type(s) de données des arguments de la fonction (éventuellement qualifié(s) du nom du schéma).

`CASCADE`
: Les objets qui dépendent de la fonction (opérateurs ou déclencheurs) sont automatiquement supprimés.

`RESTRICT`
: La fonction n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la fonction de calcul d'une racine carrée :

```
DROP FUNCTION sqrt(integer);
```

Compatibilité

Une instruction **DROP FUNCTION** est définie dans le standard SQL mais elle n'est pas compatible avec celle décrite ici.

Voir aussi

CREATE FUNCTION(7), ALTER FUNCTION(7)

Nom

DROP GROUP — Supprimer un rôle de base de données

Synopsis

```
DROP GROUP [ IF EXISTS ] nom [, ... ]
```

Description

DROP GROUP est désormais un alias de DROP ROLE(7).

Compatibilité

Il n'existe pas d'instruction **DROP GROUP** dans le standard SQL.

Voir aussi

DROP ROLE(7)

Nom

DROP INDEX — Supprimer un index

Synopsis

```
DROP INDEX [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP INDEX supprime un index. Seul le propriétaire de l'index peut exécuter cette commande.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom (éventuellement qualifié du nom du schéma) de l'index à supprimer.

`CASCADE`
: Les objets qui dépendent de l'index sont automatiquement supprimés.

`RESTRICT`
: L'index n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer l'index `title_idx` :

```
DROP INDEX title_idx;
```

Compatibilité

DROP INDEX est une extension PostgreSQL™. Il n'est pas fait mention des index dans le standard SQL.

Voir aussi

CREATE INDEX(7)

Nom

DROP LANGUAGE — Supprimer un langage procédural

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP LANGUAGE supprime la définition d'un langage procédural enregistré précédemment. Vous devez être un superutilisateur ou le propriétaire du langage pour utiliser **DROP LANGUAGE**.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si le langage n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom du langage procédural à supprimer. Pour une compatibilité ascendante, le nom peut être entouré de guillemets simples.

`CASCADE`
: Les objets qui dépendent du langage (fonctions, par exemple) sont automatiquement supprimés.

`RESTRICT`
: Le langage n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer le langage procédural `plexemple` :

```
DROP LANGUAGE plexemple;
```

Compatibilité

Il n'existe pas d'instruction **DROP LANGUAGE** dans le standard SQL.

Voir aussi

ALTER LANGUAGE(7), CREATE LANGUAGE(7), droplang(1)

Nom

DROP OPERATOR — Supprimer un opérateur

Synopsis

```
DROP OPERATOR [ IF EXISTS ] nom ( { type_gauche | NONE }, { type_droit | NONE } )
    [ CASCADE | RESTRICT ]
```

Description

DROP OPERATOR supprime un opérateur. Seul le propriétaire de l'opérateur peut exécuter cette commande.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom de l'opérateur (éventuellement qualifié du nom du schéma) à supprimer.

`type_gauche`
: Le type de données de l'opérande gauche de l'opérateur ; `NONE` est utilisé si l'opérateur n'en a pas.

`type_droit`
: Le type de données de l'opérande droit de l'opérateur ; `NONE` est utilisé si l'opérateur n'en a pas.

`CASCADE`
: Les objets qui dépendent de l'opérateur sont automatiquement supprimés.

`RESTRICT`
: L'opérateur n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer l'opérateur puissance `a^b` sur le type integer :

```
DROP OPERATOR ^ (integer, integer);
```

Supprimer l'opérateur de complément binaire `~b` sur le type bit :

```
DROP OPERATOR ~ (none, bit);
```

Supprimer l'opérateur unaire factorielle `x!` sur le type bigint :

```
DROP OPERATOR ! (bigint, none);
```

Compatibilité

Il n'existe pas d'instruction **DROP OPERATOR** dans le standard SQL.

Voir aussi

CREATE OPERATOR(7), ALTER OPERATOR(7)

Nom

DROP OPERATOR CLASS — Supprimer une classe d'opérateur

Synopsis

```
DROP OPERATOR CLASS [ IF EXISTS ] nom USING méthode_index [ CASCADE | RESTRICT ]
```

Description

DROP OPERATOR CLASS supprime une classe d'opérateur. Seul le propriétaire de la classe peut la supprimer.

DROP OPERATOR CLASS ne supprime aucun des opérateurs et aucune des fonctions référencés par la classe. Si un index dépend de la classe d'opérateur, vous devez indiquer `CASCADE` pour que la suppression se fasse réellement.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom (éventuellement qualifié du nom du schéma) d'une classe d'opérateur.

`méthode_index`
: Le nom de la méthode d'accès aux index pour laquelle l'opérateur est défini.

`CASCADE`
: Les objets qui dépendent de cette classe sont automatiquement supprimés.

`RESTRICT`
: La classe d'opérateur n'est pas supprimée si un objet en dépend. Comportement par défaut.

Notes

DROP OPERATOR CLASS ne supprimera pas la famille d'opérateur contenant la classe, même si la famille en devient vide (en particulier, dans le cas où la famille a été implicitement créée par **CREATE OPERATOR CLASS**). Avoir une famille d'opérateur vide est sans risque. Pour plus de clareté, il est préférable de supprimer la famille avec **DROP OPERATOR FAMILY** ; ou encore mieux, utilisez **DROP OPERATOR FAMILY** dès le début.

Exemples

Supprimer la classe d'opérateur `widget_ops` des index de type arbre-balancé (B-tree) :

```
DROP OPERATOR CLASS widget_ops USING btree;
```

La commande échoue si un index utilise la classe d'opérateur. `CASCADE` permet de supprimer ces index simultanément.

Compatibilité

Il n'existe pas d'instruction **DROP OPERATOR CLASS** dans le standard SQL.

Voir aussi

ALTER OPERATOR CLASS(7), CREATE OPERATOR CLASS(7), DROP OPERATOR FAMILY(7)

Nom

DROP OPERATOR FAMILY — Supprimer une famille d'opérateur

Synopsis

```
DROP OPERATOR FAMILY [ IF EXISTS ] nom USING methode_indexage [ CASCADE | RESTRICT ]
```

Description

DROP OPERATOR FAMILY supprime une famille d'opérateur existante. Pour exécuter cette commande, vous devez être le propriétaire de la famille d'opérateur.

DROP OPERATOR FAMILY inclut la suppression de toutes classes d'opérateur contenues dans la famille, mais elle ne supprime pas les opérateurs et fonctions référencées par la famille. Si des index dépendent des classes d'opérateur de la famille, vous devez ajouter `CASCADE` pour que la suppression réussisse.

Paramètres

`IF EXISTS`
: Ne renvoie pas une erreur si la famille d'opérateur n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

`nom`
: Le nom de la famille d'opérateur (quelque fois qualifié du schéma).

`methode_indexage`
: Le nom de la méthode d'accès à l'index associée à la famille d'opérateur.

`CASCADE`
: Supprime automatiquement les objets dépendant de cette famille d'opérateur.

`RESTRICT`
: Refuse la suppression de la famille d'opérateur si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer la famille d'opérateur B-tree `float_ops` :

```
DROP OPERATOR FAMILY float_ops USING btree;
```

Cette commande échouera car il existe des index qui utilisent les classes d'opérateur de cette famille. Ajoutez `CASCADE` pour supprimer les index avec la famille d'opérateurs.

Compatibilité

Il n'existe pas d'instruction **DROP OPERATOR FAMILY** dans le standard SQL.

Voir aussi

ALTER OPERATOR FAMILY(7), CREATE OPERATOR FAMILY(7), ALTER OPERATOR CLASS(7), CREATE OPERATOR CLASS(7), DROP OPERATOR CLASS(7)

Nom

DROP OWNED — Supprimer les objets de la base possédés par un rôle

Synopsis

```
DROP OWNED BY nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP OWNED supprime tous les objets de la base qui ont pour propriétaire un des rôles spécifiés. Tout droit donné à un des rôles sur ces objets ainsi qu'aux objets partagés (bases de données, tablespaces) sera aussi supprimé.

Paramètres

nom
: Le nom d'un rôle dont les objets seront supprimés et dont les droits seront révoqués.

`CASCADE`
: Supprime automatiquement les objets qui dépendent des objets affectés.

`RESTRICT`
: Refuse de supprimer les objets possédés par un rôle si un autre objet de la base dépend de ces objets. C'est la valeur par défaut.

Notes

DROP OWNED est souvent utilisé pour préparer la suppression d'un ou plusieurs rôles. Comme **DROP OWNED** affecte seulement les objets de la base en cours, il est généralement nécessaire d'exécuter cette commande dans chaque base contenant des objets appartenant au rôle à supprimer.

Utiliser l'option `CASCADE` pourrait demander la suppression d'objets appartenant à d'autres utilisateurs.

La commande REASSIGN OWNED(7) est une alternative qui ré-affecte la propriété de tous les objets de la base possédés par un ou plusieurs rôles.

Les bases de données et les tablespaces appartenant au(x) rôle(s) ne seront pas supprimés.

Compatibilité

L'instruction **DROP OWNED** est une extension PostgreSQL™.

Voir aussi

REASSIGN OWNED(7), DROP ROLE(7)

Nom

DROP ROLE — Supprimer un rôle de base de données

Synopsis

```
DROP ROLE [ IF EXISTS ] nom [, ...]
```

Description

DROP ROLE supprime le(s) rôle(s) spécifié(s). Seul un superutilisateur peut supprimer un rôle de superutilisateur. Le droit `CREATEROLE` est nécessaire pour supprimer les autres rôles.

Un rôle ne peut pas être supprimé s'il est toujours référencé dans une base de données du groupe. Dans ce cas, toute tentative aboutit à l'affichage d'une erreur. Avant de supprimer un rôle, il est nécessaire de supprimer au préalable tous les objets qu'il possède (ou de modifier leur appartenance) et de supprimer tous les droits définis par ce rôle. Les commandes REASSIGN OWNED(7) et DROP OWNED(7) peuvent être utiles pour cela.

Néanmoins, il n'est pas nécessaire de supprimer toutes les appartenances de rôle impliquant ce rôle ; **DROP ROLE** supprime automatiquement toute appartenance du rôle cible dans les autres rôles et des autres rôles dans le rôle cible. Les autres rôles ne sont pas supprimés ou affectés.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom du rôle à supprimer.

Notes

PostgreSQL™ inclut un programme dropuser(1) qui a la même fonctionnalité que cette commande (en fait, il appelle cette commande) mais qui est lancé à partir du shell.

Exemples

Supprimer un rôle :

```
DROP ROLE jonathan;
```

Compatibilité

Le standard SQL définit **DROP ROLE** mais il ne permet la suppression que d'un seul rôle à la fois et il spécifie d'autres droits obligatoires que ceux utilisés par PostgreSQL™.

Voir aussi

CREATE ROLE(7), ALTER ROLE(7), SET ROLE(7)

Nom

DROP RULE — Supprimer une règle de réécriture

Synopsis

```
DROP RULE [ IF EXISTS ] nom ON table [ CASCADE | RESTRICT ]
```

Description

DROP RULE supprime une règle de réécriture.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom de la règle à supprimer.

`table`
: Le nom (éventuellement qualifié du nom du schéma) de la table ou vue sur laquelle s'applique la règle.

`CASCADE`
: Les objets qui dépendent de la règle sont automatiquement supprimés.

`RESTRICT`
: La règle n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Suppression de la règle de réécriture `nouvellerègle` :

```
DROP RULE nouvellerègle ON matable;
```

Compatibilité

Il n'existe pas d'instruction **DROP RULE** dans le standard SQL.

Voir aussi

CREATE RULE(7)

Nom

DROP SCHEMA — Supprimer un schéma

Synopsis

```
DROP SCHEMA [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP SCHEMA supprime des schémas de la base de données.

Un schéma ne peut être supprimé que par son propriétaire ou par un superutilisateur. Son propriétaire peut supprimer un schéma et tous les objets qu'il contient quand bien même il ne possède pas tous les objets contenus dans ce schéma.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom
: Le nom du schéma.

`CASCADE`
: Les objets (tables, fonctions...) contenus dans le schéma sont automatiquement supprimés.

`RESTRICT`
: Le schéma n'est pas supprimé s'il contient des objets. Comportement par défaut.

Exemples

Supprimer le schéma `mes_affaires` et son contenu :

```
DROP SCHEMA mes_affaires CASCADE;
```

Compatibilité

DROP SCHEMA est totalement compatible avec le standard SQL. Le standard n'autorise cependant pas la suppression de plusieurs schémas en une seule commande. L'option `IF EXISTS` est aussi une extension de PostgreSQL™.

Voir aussi

ALTER SCHEMA(7), CREATE SCHEMA(7)

Nom

DROP SEQUENCE — Supprimer une séquence

Synopsis

```
DROP SEQUENCE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP SEQUENCE permet de supprimer les générateurs de nombres séquentiels. Une séquence peut seulement être supprimée par son propriétaire ou par un superutilisateur.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom de la séquence (éventuellement qualifié du nom du schéma).

`CASCADE`
: Les objets qui dépendent de la séquence sont automatiquement supprimés.

`RESTRICT`
: La séquence n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la séquence `serie` :

```
DROP SEQUENCE serie;
```

Compatibilité

DROP SEQUENCE est conforme au standard SQL. Cependant, le standard n'autorise pas la suppression de plusieurs séquences en une seule commande. De plus, l'option `IF EXISTS` est une extension de PostgreSQL™.

Voir aussi

CREATE SEQUENCE(7), ALTER SEQUENCE(7)

Nom

DROP SERVER — Supprimer un descripteur de serveur distant

Synopsis

```
DROP SERVER [ IF EXISTS ] nom_serveur [ CASCADE | RESTRICT ]
```

Description

DROP SERVER supprime un descripteur de serveur distant existant. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire du serveur.

Paramètres

`IF EXISTS`
: Ne génère pas d'erreur si le serveur n'existe pas. Un avertissement est émis dans ce cas.

`nom_serveur`
: Nom d'un serveur existant.

`CASCADE`
: Supprime automatiquement les objets dépendant du serveur (tels que les correspondances d'utilisateur).

`RESTRICT`
: Refuse de supprimer le serveur si des objets en dépendent. C'est le cas par défaut.

Exemples

Supprimer un serveur `truc` s'il existe :

```
DROP SERVER IF EXISTS truc;
```

Compatibilité

DROP SERVER est conforme à la norme ISO/IEC 9075-9 (SQL/MED). La clause `IF EXISTS` est une extension PostgreSQL™ .

Voir aussi

CREATE SERVER(7), ALTER SERVER(7)

Nom

DROP TABLE — Supprimer une table

Synopsis

```
DROP TABLE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP TABLE supprime des tables de la base de données. Seul son propriétaire peut détruire une table. DELETE(7) et TRUNCATE(7) sont utilisées pour supprimer les lignes d'une table sans détruire la table.

DROP TABLE supprime tout index, règle, déclencheur ou contrainte qui existe sur la table cible. Néanmoins, pour supprimer une table référencée par une vue ou par une contrainte de clé étrangère d'une autre table, `CASCADE` doit être ajouté. (`CASCADE` supprime complètement une vue dépendante mais dans le cas de la clé étrangère, il ne supprime que la contrainte, pas l'autre table.)

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom de la table à supprimer (éventuellement qualifié du nom du schéma).

`CASCADE`
: Les objets qui dépendent de la table (vues, par exemple) sont automatiquement supprimés.

`RESTRICT`
: La table n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer les deux tables `films` et `distributeurs` :

```
DROP TABLE films, distributeurs;
```

Compatibilité

Cette commande est conforme au standard SQL. Cependant, le standard n'autorise pas la suppression de plusieurs tables en une seule commande. De plus, l'option `IF EXISTS` est une extension de PostgreSQL™.

Voir aussi

ALTER TABLE(7), CREATE TABLE(7)

Nom

DROP TABLESPACE — Supprimer un tablespace

Synopsis

```
DROP TABLESPACE [ IF EXISTS ] nom_tablespace
```

Description

DROP TABLESPACE supprime un tablespace du système.

Un tablespace ne peut être supprimé que par son propriétaire ou par un superutilisateur. Le tablespace doit être vide de tout objet de base de données avant sa suppression. Même si le tablespace ne contient plus d'objets de la base de données courante, il est possible que des objets d'autres bases de données l'utilisent. De plus, si le tablespace se trouve parmi les tablespaces du paramètre temp_tablespaces d'une session active, la commande **DROP** pourrait échouer à cause de fichiers temporaires stockés dans le tablespace.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom_tablespace`
: Le nom du tablespace.

Notes

DROP TABLESPACE ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Exemples

Supprimer le tablespace `mes_affaires` :

```
DROP TABLESPACE mes_affaires;
```

Compatibilité

DROP TABLESPACE est une extension PostgreSQL™.

Voir aussi

CREATE TABLESPACE(7), ALTER TABLESPACE(7)

Nom

DROP TEXT SEARCH CONFIGURATION — Supprimer une configuration de recherche plein texte

Synopsis

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH CONFIGURATION supprime une configuration existante de la recherche plein texte. Pour exécuter cette commande, vous devez être le propriétaire de la configuration.

Paramètres

`IF EXISTS`
: Ne renvoie pas une erreur si la configuration de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

`name`
: Le nom de la configuration de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`
: Supprime automatiquement les objets dépendant de cette configuration de recherche plein texte.

`RESTRICT`
: Refuse la suppression de la configuration de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer la configuration de recherche plein texte the text search configuration `my_english`:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

Cette commande échouera s'il existe des index qui référencent la configuration dans des appels `to_tsvector`. Ajoutez `CASCADE` pour supprimer ces index avec la configuration de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction **DROP TEXT SEARCH CONFIGURATION** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH CONFIGURATION(7), CREATE TEXT SEARCH CONFIGURATION(7)

Nom

DROP TEXT SEARCH DICTIONARY — Supprimer un dictionnaire de recherche plein texte

Synopsis

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH DICTIONARY supprime un dictionnaire existant de la recherche plein texte. Pour exécuter cette commande, vous devez être le propriétaire du dictionnaire.

Paramètres

`IF EXISTS`
: Ne renvoie pas une erreur si le dictionnaire de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

`name`
: Le nom du dictionnaire de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`
: Supprime automatiquement les objets dépendant de ce dictionnaire de recherche plein texte.

`RESTRICT`
: Refuse la suppression du dictionnaire de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer le dictionnaire de recherche plein texte `english` :

```
DROP TEXT SEARCH DICTIONARY english;
```

Cette commande échouera s'il existe des configurations qui utilisent ce dictionnaire. Ajoutez `CASCADE` pour supprimer ces configurations avec le dictionnaire de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction **DROP TEXT SEARCH DICTIONARY** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH DICTIONARY(7), CREATE TEXT SEARCH DICTIONARY(7)

Nom

DROP TEXT SEARCH PARSER — Supprimer un analyseur de recherche plein texte

Synopsis

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH PARSER supprime un analyseur existant de la recherche plein texte. Pour exécuter cette commande, vous devez être superutilisateur.

Paramètres

`IF EXISTS`
: Ne renvoie pas une erreur si l'analyseur de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

`name`
: Le nom de l'analyseur de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`
: Supprime automatiquement les objets dépendant de l'analyseur de recherche plein texte.

`RESTRICT`
: Refuse la suppression de l'analyseur de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer l'analyseur de recherche plein texte `mon_analyseur` :

```
DROP TEXT SEARCH PARSER mon_analyseur;
```

Cette commande échouera s'il existe des configurations qui utilisent ce dictionnaire. Ajoutez `CASCADE` pour supprimer ces configurations avec l'analyseur de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction **DROP TEXT SEARCH PARSER** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH PARSER(7), CREATE TEXT SEARCH PARSER(7)

Nom

DROP TEXT SEARCH TEMPLATE — Supprimer un modèle de recherche plein texte

Synopsis

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH TEMPLATE supprime un modèle existant de la recherche plein texte. Pour exécuter cette commande, vous devez superutilisateur.

Paramètres

`IF EXISTS`
: Ne renvoie pas une erreur si le modèle de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

`name`
: Le nom du modèle de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`
: Supprime automatiquement les objets dépendant de ce modèle de recherche plein texte.

`RESTRICT`
: Refuse la suppression du modèle de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer le modèle de recherche plein texte `thesaurus` :

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

Cette commande échouera s'il existe des dictionnaires qui utilisent ce modèles. Ajoutez `CASCADE` pour supprimer ces dictionnaires avec le modèle de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction **DROP TEXT SEARCH TEMPLATE** dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH TEMPLATE(7), CREATE TEXT SEARCH TEMPLATE(7)

Nom

DROP TRIGGER — Supprimer un déclencheur

Synopsis

```
DROP TRIGGER [ IF EXISTS ] nom ON table [ CASCADE | RESTRICT ]
```

Description

DROP TRIGGER supprime la définition d'un déclencheur. Seul le propriétaire de la table sur laquelle le déclencheur est défini peut exécuter cette commande.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`
: Le nom du déclencheur à supprimer.

`table`
: Le nom de la table (éventuellement qualifié du nom du schéma) sur laquelle le déclencheur est défini.

`CASCADE`
: Les objets qui dépendent du déclencheur sont automatiquement supprimés.

`RESTRICT`
: Le déclencheur n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Destruction du déclencheur `si_dist_existe` de la table `films` :

```
DROP TRIGGER si_dist_existe ON films;
```

Compatibilité

L'instruction **DROP TRIGGER** de PostgreSQL™ est incompatible avec le standard SQL. Dans le standard, les noms de déclencheurs ne se définissent pas par rapport aux tables. La commande est donc simplement `DROP TRIGGER` *`nom`*.

Voir aussi

CREATE TRIGGER(7)

Nom

DROP TYPE — Supprimer un type de données

Synopsis

```
DROP TYPE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP TYPE supprime un type de données utilisateur. Seul son propriétaire peut le supprimer.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom
: Le nom du type de données (éventuellement qualifié du nom de schéma) à supprimer.

`CASCADE`
: Les objets qui dépendent du type (colonnes de table, fonctions, opérateurs...) sont automatiquement supprimés.

`RESTRICT`
: Le type n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer le type de données boite :

```
DROP TYPE boite;
```

Compatibilité

Cette commande est similaire à celle du standard SQL en dehors de l'option `IF EXISTS` qui est une extension PostgreSQL™. La commande **CREATE TYPE** et les mécanismes d'extension de type de données de PostgreSQL™ diffèrent du standard.

Voir aussi

CREATE TYPE(7), ALTER TYPE(7)

Nom

DROP USER — Supprimer un rôle de base de données

Synopsis

```
DROP USER [ IF EXISTS ] nom [, ...]
```

Description

DROP USER est désormais un alias de DROP ROLE(7).

Compatibilité

L'instruction **DROP USER** est une extension PostgreSQL™. Le standard SQL laisse la définition des utilisateurs à l'implantation.

Voir aussi

DROP ROLE(7)

Nom

DROP USER MAPPING — Supprimer une correspondance d'utilisateur pour un serveur distant

Synopsis

```
DROP USER MAPPING [ IF EXISTS ] FOR { nom_utilisateur | USER | CURRENT_USER | PUBLIC
} SERVER nom_serveur
```

Description

DROP USER MAPPING supprime une correspondance d'utilisateur existant pour un serveur distant.

Le propriétaire d'un serveur distant peut supprimer les correspondances d'utilisateur pour ce serveur pour n'importe quel utilisateur. Par ailleurs, un utilisateur peut supprimer une correspondance d'utilisateur pour son propre nom d'utilisateur s'il a reçu le droit `USAGE` sur le serveur.

Paramètres

`IF EXISTS`
: Ne génère pas d'erreur si la correspondance d'utilisateur n'existe pas. Un avertissement est émis dans ce cas.

`nom_utilisateur`
: Nom d'utilisateur de la correspondance. `CURRENT_USER` et `USER` correspondent au nom de l'utilisateur courant. `PUBLIC` est utilisé pour correspondre à tous les noms d'utilisateurs présents et futurs du système.

`nom_serveur`
: Nom du serveur de la correspondance d'utilisateur.

Exemples

Supprimer une correspondance d'utilisateur `bob`, sur le serveur `truc` si elle existe :

```
DROP USER MAPPING IF EXISTS FOR bob SERVER truc;
```

Compatibilité

DROP USER MAPPING est conforme à la norme ISO/IEC 9075-9 (SQL/MED). La clause `IF EXISTS` est une extension PostgreSQL™.

Voir aussi

CREATE USER MAPPING(7), ALTER USER MAPPING(7)

Nom

DROP VIEW — Supprimer une vue

Synopsis

```
DROP VIEW [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP VIEW supprime une vue existante. Seul le propriétaire de la vue peut exécuter cette commande.

Paramètres

`IF EXISTS`
: Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom
: Le nom de la vue (éventuellement qualifié du nom de schéma) à supprimer.

`CASCADE`
: Les objets qui dépendent de la vue (d'autres vues, par exemple) sont automatiquement supprimés.

`RESTRICT`
: La vue n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la vue `genre` :

```
DROP VIEW genre;
```

Compatibilité

Cette commande est conforme au standard SQL. Cependant, le standard n'autorise pas la suppression de plusieurs vues en une seule commande. De plus, l'option `IF EXISTS` est une extension de PostgreSQL™.

Voir aussi

ALTER VIEW(7), CREATE VIEW(7)

Nom

END — Valider la transaction en cours

Synopsis

```
END [ WORK | TRANSACTION ]
```

Description

END valide la transaction en cours. Toutes les modifications réalisées lors de la transaction deviennent visibles pour les autres utilisateurs et il est garanti que les données ne seront pas perdues si un arrêt brutal survient. Cette commande est une extension PostgreSQL™ équivalente à COMMIT(7).

Paramètres

`WORK`, `TRANSACTION`
: Mots clés optionnels. Ils n'ont pas d'effet.

Notes

ROLLBACK(7) est utilisé pour annuler une transaction.

Lancer **END** à l'extérieur d'une transaction n'a aucun effet mais provoque un message d'avertissement.

Exemples

Valider la transaction en cours et rendre toutes les modifications persistantes :

```
END;
```

Compatibilité

END est une extension PostgreSQL™ fournissant une fonctionnalité équivalente à COMMIT(7), spécifié dans le standard SQL.

Voir aussi

BEGIN(7), COMMIT(7), ROLLBACK(7)

Nom

EXECUTE — Exécuter une instruction préparée

Synopsis

```
EXECUTE nom [ (paramètre [, ...] ) ]
```

Description

EXECUTE est utilisé pour exécuter une instruction préparée au préalable. Comme les instructions préparées existent seulement pour la durée d'une session, l'instruction préparée doit avoir été créée par une instruction **PREPARE** exécutée plus tôt dans la session en cours.

Si l'instruction **PREPARE** qui crée l'instruction est appelée avec des paramètres, un ensemble compatible de paramètres doit être passé à l'instruction **EXECUTE**, sinon une erreur est levée. Contrairement aux fonctions, les instructions préparées ne sont pas surchargées en fonction de leur type ou du nombre de leurs paramètres ; le nom d'une instruction préparée doit être unique au sein d'une session.

Pour plus d'informations sur la création et sur l'utilisation des instructions préparées, voir PREPARE(7).

Paramètres

nom
: Le nom de l'instruction préparée à exécuter.

paramètre
: La valeur réelle du paramètre d'une instruction préparée. Ce paramètre doit être une expression ramenant une valeur dont le type est compatible avec celui spécifié pour ce paramètre positionnel dans la commande **PREPARE** qui a créé l'instruction préparée.

Sorties

La sortie renvoyée par la commande **EXECUTE** est celle de l'instruction préparée, et non celle de la commande `EXECUTE`.

Exemples

Des exemples sont donnés dans la section la section intitulée « Exemples » de la documentation de PREPARE(7).

Compatibilité

Le standard SQL inclut une instruction **EXECUTE** qui n'est utilisée que dans le SQL embarqué. La syntaxe utilisée par cette version de l'instruction **EXECUTE** diffère quelque peu.

Voir aussi

DEALLOCATE(7), PREPARE(7)

Nom

EXPLAIN — Afficher le plan d'exécution d'une instruction

Synopsis

```
EXPLAIN [ ( option [, ...] ) ] instruction
EXPLAIN [ ANALYZE ] [ VERBOSE ] instruction

où option est :

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Description

Cette commande affiche le plan d'exécution que l'optimiseur de PostgreSQL™ engendre pour l'instruction fournie. Le plan d'exécution décrit le parcours de la (des) table(s) utilisée(s) dans la requête -- parcours séquentiel, parcours d'index, etc. -- . Si plusieurs tables sont référencées, il présente également les algorithmes de jointures utilisés pour rassembler les lignes issues des différentes tables.

La partie la plus importante de l'affichage concerne l'affichage des coûts estimés d'exécution. Ils représentent l'estimation faite par le planificateur des temps d'exécution de la requête (mesurés en unités de récupération de pages sur le disque). Deux nombres sont affichés : le temps de démarrage, écoulé avant que la première ligne soit renvoyée, et le temps d'exécution total, nécessaire au renvoi de toutes les lignes. Pour la plupart des requêtes, le temps qui importe est celui d'exécution totale. Mais dans certains cas, tel que pour une sous-requête dans la clause `EXISTS`, le planificateur choisira le temps de démarrage le plus court, et non celui d'exécution totale (car, de toute façon, l'exécuteur s'arrête après la récupération d'une ligne). De même, lors de la limitation des résultats à retourner par une clause `LIMIT`, la planificateur effectue une interpolation entre les deux temps limites pour choisir le plan réellement le moins coûteux.

L'option `ANALYZE` impose l'exécution de la requête en plus de sa planification. Le temps total d'exécution de chaque nœud du plan (en millisecondes) et le nombre total de lignes effectivement retournées sont ajoutés à l'affichage. C'est utile pour vérifier la véracité des informations fournies par le planificateur.

Important

Il ne faut pas oublier que l'instruction est réellement exécutée avec l'option `ANALYZE`. Bien qu'**EXPLAIN** inhibe l'affichage des retours d'une commande **SELECT**, les autres effets de l'instruction sont présents. Si **EXPLAIN ANALYZE** doit être utilisé sur une instruction **INSERT**, **UPDATE**, **DELETE CREATE TABLE AS** ou **EXECUTE** sans que la commande n'affecte les données, l'approche suivante peut être envisagée :

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Seules les options `ANALYZE` et `VERBOSE` peuvent être utilisées et dans cet ordre seulement si la liste d'options entre parenthèses n'est pas utilisé. Avant PostgreSQL™ 9.0, la seule syntaxe supportée était celle sans parenthèses. Les nouvelles options ne seront supportées que par la nouvelle syntaxe, celle avec les parenthèses.

Paramètres

`ANALYZE`
: Exécute la commande et affiche les temps d'exécution réels. Ce paramètre est par défaut à `FALSE`.

`VERBOSE`
: Affiche des informations supplémentaires sur le plan. Cela inclut la liste des colonnes en sortie pour chaque nœud du plan, les noms des tables et fonctions avec le nom du schéma, les labels des variables dans les expressions avec des alias de tables et le nom de chaque trigger pour lesquels les statistiques sont affichées. Ce paramètre est par défaut à `FALSE`.

`COSTS`

Inclut des informations sur le coût estimé au démarrage et au total de chaque nœud du plan, ainsi que le nombre estimé de lignes et la largeur estimée de chaque ligne. Ce paramètre est par défaut à `TRUE`.

`BUFFERS`
: Inclut des informations sur l'utilisation des blocs. Cela inclut le nombre de lecture de blocs partagés en cache, sur le disque ainsi que le nombre de blocs écrits. Cela inclut aussi le nombre de blocs locaux lus dans le cache, sur le disque et le nombre de blocs locaux écrits. Enfin, cela inclut le nombre de blocs temporaires lus et écrits. Les blocs partagés, les blocs locaux et les blocs temporaires contiennent respectivement des tables et des index, des tables temporaires et des index temporaires, et les blocs disques utilisés dans les tris et les plans matérialisés. Le nombre de blocs affichés pour un nœud de haut niveau inclut ceux utilisés par tous ses enfants. Dans le format texte, seuls les valeurs différentes de zéro sont affichés. Ce paramètre pourrait seulement être utilisé avec le paramètre `ANALYZE`. Il vaut par défaut `FALSE`.

`FORMAT`
: Indique le format de sortie. Il peut valoir TEXT, XML, JSON ou YAML. Toutes les sorties contiennent les mêmes informations, mais les programmes pourront plus facilement traiter les sorties autres que TEXT. Ce paramètre est par défaut à `TEXT`.

`boolean`
: Spécifie si l'option sélectionnée doit être activée ou désactivée. Vous pouvez écrire `TRUE`, `ON` ou `1` pour activer l'option, et `FALSE`, `OFF` ou `0` pour la désactiver. La valeur de type *`boolean`* peut aussi être omise, auquel cas la valeur sera `TRUE`.

`instruction`
: Toute instruction **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **VALUES EXECUTE**, **DECLARE** ou **CREATE TABLE AS** dont le plan d'exécution est souhaité.

Notes

La documentation sur l'utilisation faite par l'optimiseur des informations de coût est assez réduite dans PostgreSQL™. On peut se référer à Section 14.1, « Utiliser **EXPLAIN** » pour plus d'informations.

Pour que le planificateur de requêtes de PostgreSQL™ puisse prendre des décisions en connaissance de cause, l'instruction ANALYZE(7) doit avoir été exécutée afin d'enregistrer les statistiques de distribution des données dans la table. Si cela n'a pas été fait, (ou si la distribution statistique des données dans la table a changé de manière significative depuis la dernière exécution de la commande **ANALYZE**) les coûts estimés risquent de ne pas refléter les propriétés réelles de la requête. De ce fait, un plan de requête inférieur risque d'être choisi.

Pour mesurer le coût d'exécution du plan d'exécution, l'implémentation actuelle de **EXPLAIN ANALYZE** peut ajouter un délai considérable à l'exécution de la requête à cause du profilage. De ce fait, exécuter **EXPLAIN ANALYZE** sur une requête peut prendre bien plus de temps que d'exécuter la requête seule. Ce délai dépend de la nature de la requête.

Exemples

Afficher le plan d'une requête simple sur une table d'une seule colonne de type integer et 10000 lignes :

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

Voici le même plan, mais formaté avec JSON :

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                              +
   {                            +
     "Plan": {                  +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                          +
   }                            +
 ]
```

```
(1 row)
```

S'il existe un index et que la requête contient une condition `WHERE` indexable, **EXPLAIN** peut afficher un plan différent :

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Voici le même plan, mais formaté avec YAML :

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-------------------------------
 - Plan:                        +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

L'obtention du format XML est laissé en exercice au lecteur.

Voici le même plan avec les coûts supprimés :

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

Exemple de plan de requête pour une requête utilisant une fonction d'agrégat :

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                             QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Exemple d'utilisation de **EXPLAIN EXECUTE** pour afficher le plan d'exécution d'une requête préparée :

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                       QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7
loops=1)
   ->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311 width=8) (actual
time=0.050..0.395 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Total runtime: 0.851 ms
```

```
(4 rows)
```

Il est évident que les nombres présentés ici dépendent du contenu effectif des tables impliquées. De plus, les nombres, et la stratégie sélectionnée elle-même, peuvent différer en fonction de la version de PostgreSQL™ du fait des améliorations apportées au planificateur. Il faut également savoir que la commande **ANALYZE** calcule les statistiques des données à partir d'extraits aléatoires ; il est de ce fait possible que les coûts estimés soient modifiés après l'exécution de cette commande, alors même la distribution réelle des données dans la table n'a pas changé.

Compatibilité

L'instruction **EXPLAIN** n'est pas définie dans le standard SQL.

Voir aussi

ANALYZE(7)

Nom

FETCH — Récupérer les lignes d'une requête à l'aide d'un curseur

Synopsis

```
FETCH [ direction [ FROM | IN ] ] nom_curseur

où direction peut être vide ou être :

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE nombre
    RELATIVE nombre
    nombre
    ALL
    FORWARD
    FORWARD nombre
    FORWARD ALL
    BACKWARD
    BACKWARD nombre
    BACKWARD ALL
```

Description

FETCH récupère des lignes en utilisant un curseur précédemment ouvert.

À un curseur est associée une position associée utilisée par **FETCH**. Le curseur peut être positionné avant la première ligne du résultat de la requête, sur une ligne particulière du résultat ou après la dernière ligne du résultat. À sa création, le curseur est positionné avant la première ligne. Après récupération de lignes, le curseur est positionné sur la ligne la plus récemment récupérée. Si **FETCH** atteint la fin des lignes disponibles, il est positionné après la dernière ligne ou avant la première ligne dans le cas d'une récupération remontante. **FETCH ALL** ou **FETCH BACKWARD ALL** positionne toujours le curseur après la dernière ligne ou avant la première ligne.

Les formes `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE`, `RELATIVE` récupèrent une seule ligne après déplacement approprié du curseur. Si cette ligne n'existe pas, un résultat vide est renvoyé et le curseur est positionné avant la première ligne ou après la dernière ligne, en fonction du sens de la progression.

Les formes utilisant `FORWARD` et `BACKWARD` récupèrent le nombre de lignes indiqué en se déplaçant en avant ou en arrière, laissant le curseur positionné sur la dernière ligne renvoyée (ou après/avant toutes les lignes si *`nombre`* dépasse le nombre de lignes disponibles).

`RELATIVE 0`, `FORWARD 0` et `BACKWARD 0` récupèrent tous la ligne actuelle sans déplacer le curseur, c'est-à-dire qu'ils effectuent une nouvelle récupération de la ligne dernièrement récupérée. La commande réussit sauf si le curseur est positionné avant la première ligne ou après la dernière ligne ; dans ce cas, aucune ligne n'est renvoyée.

> **Note**
>
> Cette page décrit l'utilisation des curseurs au niveau de la commande SQL. Si vous voulez utiliser des curseurs dans une fonction PL/pgSQL, les règles sont différentes -- voir Section 39.7, « Curseurs ».

Paramètres

`direction`

La direction et le nombre de lignes à récupérer. Ce paramètre peut prendre les valeurs suivantes :

`NEXT`
La ligne suivante est récupérée. C'est le comportement par défaut si *`direction`* est omis.

`PRIOR`
La ligne précédente est récupérée.

`FIRST`

La première ligne de la requête est récupérée. C'est identique à `ABSOLUTE 1`.

`LAST`
La dernière ligne de la requête est récupérée. C'est identique à `ABSOLUTE -1`.

`ABSOLUTE` *`nombre`*
La *`nombre`*-ième ligne de la requête est récupérérée, ou la `abs(`*`nombre`*`)`-ième ligne à partir de la fin si *`nombre`* est négatif. Le curseur est positionné avant la première ligne ou après la dernière si *`nombre`* est en dehors des bornes ; en particulier, `ABSOLUTE 0` le positionne avant la première ligne.

`RELATIVE` *`nombre`*
La *`nombre`*-ième ligne suivante est récupérée, ou la `abs(`*`nombre`*`)`-ième ligne précédente si *`nombre`* est négatif. `RELATIVE 0` récupère de nouveau la ligne courante, si elle existe.

`nombre`
Les *`nombre`* lignes suivantes sont récupérées. C'est identique à `FORWARD` *`nombre`*.

`ALL`
Toutes les lignes restantes sont récupérées. C'est identique à `FORWARD ALL`).

`FORWARD`
La ligne suivante est récupérée. C'est identique à `NEXT`.

`FORWARD` *`nombre`*
Les *`nombre`* lignes suivantes sont récupérées. `FORWARD 0` récupère de nouveau la ligne courante.

`FORWARD ALL`
Toutes les lignes restantes sont récupérées.

`BACKWARD`
La ligne précédente est récupérée. C'est identique à `PRIOR`.

`BACKWARD` *`nombre`*
Les *`nombre`* lignes précédentes sont récupérées (parcours inverse). `BACKWARD 0` récupère de nouveau la ligne courante.

`BACKWARD ALL`
Toutes les lignes précédentes sont récupérées (parcours inverse).

`nombre`
Constante de type entier éventuellement signé, qui précise l'emplacement ou le nombre de lignes à récupérer. Dans le cas de `FORWARD` et `BACKWARD`, préciser une valeur négative pour *`nombre`* est équivalent à modifier le sens de `FORWARD` et `BACKWARD`.

`nom_curseur`
Le nom d'un curseur ouvert.

Sorties

En cas de succès, une commande **FETCH** renvoie une balise de commande de la forme

```
FETCH nombre
```

Le *`nombre`* est le nombre de lignes récupérées (éventuellement zéro). Dans psql, la balise de commande n'est pas réellement affichée car psql affiche à la place les lignes récupérées.

Notes

Le curseur doit être déclaré avec l'option `SCROLL` si les variantes de **FETCH** autres que **FETCH NEXT** ou **FETCH FORWARD** avec un nombre positif sont utilisées. Pour les requêtes simples, PostgreSQL™ autorise les parcours inverses à partir de curseurs non déclarés avec `SCROLL`. il est toutefois préférable de ne pas se fonder sur ce comportement. Si le curseur est déclaré avec `NO SCROLL`, aucun parcours inverse n'est autorisé.

Les récupérations `ABSOLUTE` ne sont pas plus rapides que la navigation vers la ligne désirée par déplacement relatif : de toute façon, l'implantation sous-jacente doit parcourir toutes les lignes intermédiaires. Les récupérations absolues négatives font même pis : la requête doit être lue jusqu'à la fin pour trouver la dernière ligne, puis relue en sens inverse à partir de là. Néanmoins, remonter vers le début de la requête (comme avec `FETCH ABSOLUTE 0`) est rapide.

DECLARE(7) est utilisé pour définir un curseur. MOVE(7) est utilisé pour modifier la position du curseur sans récupérer les données.

Exemples

Parcourir une table à l'aide d'un curseur :

```
BEGIN WORK;

-- Initialiser le curseur :
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Récupèrer les 5 premières lignes du curseur liahona :
FETCH FORWARD 5 FROM liahona;

 code  |          titre          | did | date_prod  |   genre  |  longueur
-------+-------------------------+-----+------------+----------+-----------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- Récupèrer la ligne précédente :
FETCH PRIOR FROM liahona;

 code  |  titre  | did | date_prod  |  genre |  longueur
-------+---------+-----+------------+--------+-----------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- Fermer le curseur et terminer la transaction:
CLOSE liahona;
COMMIT WORK;
```

Compatibilité

Le standard SQL ne définit **FETCH** que pour une utilisation en SQL embarqué. La variante de **FETCH** décrite ici renvoie les données comme s'il s'agissait du résultat d'un **SELECT** plutôt que de le placer dans des variables hôtes. À part cela, **FETCH** est totalement compatible avec le standard SQL.

Les formes de **FETCH** qui impliquent `FORWARD` et `BACKWARD`, ainsi que les formes `FETCH` *`nombre`* et `FETCH ALL`, dans lesquelles `FORWARD` est implicite, sont des extensions PostgreSQL™.

Le standard SQL n'autorise que `FROM` devant le nom du curseur ; la possibilité d'utiliser `IN`, ou de les laisser, est une extension.

Voir aussi

CLOSE(7), DECLARE(7), MOVE(7)

Nom

GRANT — Définir les droits d'accès

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( nom_colonne [, ...] )
    [,...] | ALL [ PRIVILEGES ] ( nom_colonne [, ...] ) }
    ON [ TABLE ] nom_table [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [,...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE nom_séquence [, ...]
         | ALL SEQUENCES IN SCHEMA nom_schéma [, ...] }
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE nom_base [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER nom_fdw [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER nom_serveur [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION nom_fonction ( [ [ mode_arg ] [ nom_arg ] type_arg [, ...] ] ) [,
...]
         | ALL FUNCTIONS IN SCHEMA nom_schéma [, ...] }
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nom_lang [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [,...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nom_schéma [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT nom_rôle [, ...] TO nom_rôle [, ...] [ WITH ADMIN OPTION ]
```

Description

La commande **GRANT** a deux variantes basiques : la première donne des droits sur un objet de la base de données (table, vue, séquence, base de données, wrapper de données distantes, serveur distant, fonction, langage de procédure, schéma ou espace logique), la seconde gère les appartenances à un rôle. Ces variantes sont assez similaires mais somme toute assez différentes pour être décrites séparément.

GRANT sur les objets de la base de données

Cette variante de la commande **GRANT** donne des droits spécifiques sur un objet de la base de données a un ou plusieurs rôles. Ces droits sont ajoutés à ceux déjà possédés, s'il y en a.

Il existe aussi une option pour donner les droits sur tous les objets d'un même type sur un ou plusieurs schémas. Cette fonctionnalité n'est actuellement proposée que pour les tables, séquences et fonctions (mais notez que `ALL TABLES` incluent aussi les vues).

Le mot clé `PUBLIC` indique que les droits sont donnés à tous les rôles, y compris ceux créés ultérieurement. `PUBLIC` peut être vu comme un groupe implicitement défini qui inclut en permanence tous les rôles. Un rôle particulier dispose de la somme des droits qui lui sont acquis en propre, des droits de tout rôle dont il est membre et des droits donnés à `PUBLIC`.

Si `WITH GRANT OPTION` est précisé, celui qui reçoit le droit peut le transmettre à son tour (NDT : par la suite on parlera d'« option de transmission de droit », là où en anglais il est fait mention de « grant options »). Sans l'option GRANT, l'utilisateur ne peut pas le faire. Cette option ne peut pas être donnée à `PUBLIC`.

Il n'est pas nécessaire d'accorder des droits au propriétaire d'un objet (habituellement l'utilisateur qui l'a créé) car, par défaut, le propriétaire possède tous les droits. (Le propriétaire peut toutefois choisir de révoquer certains de ses propres droits.)

Le droit de supprimer un objet ou de modifier sa définition n'est pas configurable avec cette commande. Il est spécifique au propriétaire de l'objet. Ce droit ne peut ni être donné ni supprimé. Néanmoins, il est possible d'avoir le même effet en rendant un utilisateur membre du rôle qui possède cet object ou en le supprimant de ce rôle. Le propriétaire a aussi implicitement les options de transmission de droits pour l'objet.

En fonction du type de l'objet, les privilèges initiaux par défaut peuvent inclure la transmission de certains privilèges à `PUBLIC`. Par défaut, aucun accès public n'est accordé sur les tables, colonnes, schémas et tablespaces ; le droit de création de table `CONNECT` et `TEMP` est accordé sur les bases de données ; le droit `EXECUTE` sur les fonctions ; et le droit `USAGE` sur les langages. Le propriétaire de l'objet peut évidemment choisir de révoquer ces droits. (Pour un maximum de sécurité, **REVOKE** est lancé dans la même transaction que la création de l'objet ; ainsi, il n'y a pas de laps de temps pendant lequel un autre utilisateur peut utiliser l'objet.) De plus, cette configuration des droits par défaut peut être modifiée en utilisant la commande ALTER DEFAULT PRIVILEGES(7).

Les droits possibles sont :

SELECT
: Autorise SELECT(7) sur toutes les colonnes, ou sur les colonnes listées spécifiquement, de la table, vue ou séquence indiquée. Autorise aussi l'utilisation de COPY(7) TO. De plus, ce droit est nécessaire pour référencer des valeurs de colonnes existantes avec UPDATE(7) ou DELETE(7). Pour les séquences, ce droit autorise aussi l'utilisation de la fonction `currval`. Pour les « Large Objects », ce droit permet la lecture de l'objet.

INSERT
: Autorise INSERT(7) d'une nouvelle ligne dans la table indiquée. Si des colonnes spécifiques sont listées, seules ces colonnes peuvent être affectées dans une commande **INSERT**, (les autres colonnes recevront par conséquent des valeurs par défaut). Autorise aussi COPY(7) FROM.

UPDATE
: Autorise UPDATE(7) sur toute colonne de la table spécifiée, ou sur les colonnes spécifiquement listées. (En fait, toute commande **UPDATE** non triviale nécessite aussi le droit `SELECT` car elle doit référencer les colonnes pour déterminer les lignes à mettre à jour et/ou calculer les nouvelles valeurs des colonnes.) `SELECT ... FOR UPDATE` et `SELECT ... FOR SHARE` requièrent également ce droit sur au moins une colonne en plus du droit `SELECT`. Pour les séquences, ce droit autorise l'utilisation des fonctions `nextval` et `setval`. Pour les « Large Objects », ce droit permet l'écriture et le tronquage de l'objet.

DELETE
: Autorise DELETE(7) d'une ligne sur la table indiquée. (En fait, toute commande **DELETE** non triviale nécessite aussi le droit `SELECT` car elle doit référencer les colonnes pour déterminer les lignes à supprimer.)

TRUNCATE
: Autorise TRUNCATE(7) sur la table indiquée.

REFERENCES
: Ce droit est requis sur les colonnes de référence et les colonnes qui référencent pour créer une contrainte de clé étrangère. Le droit peut être accordé pour toutes les colonnes, ou seulement des colonnes spécifiques.

TRIGGER
: Autorise la création d'un déclencheur sur la table indiquée. (Voir l'instruction CREATE TRIGGER(7).)

CREATE
: Pour les bases de données, autorise la création de nouveaux schémas dans la base de données.

  Pour les schémas, autorise la création de nouveaux objets dans le schéma. Pour renommer un objet existant, il est nécessaire

d'en être le propriétaire *et* de posséder ce droit sur le schéma qui le contient.

Pour les tablespaces, autorise la création de tables, d'index et de fichiers temporaires dans le tablespace et autorise la création de bases de données utilisant ce tablespace par défaut. (Révoquer ce privilège ne modifie pas l'emplacement des objets existants.)

CONNECT
: Autorise l'utilisateur à se connecter à la base indiquée. Ce droit est vérifié à la connexion (en plus de la vérification des restrictions imposées par `pg_hba.conf`).

TEMPORARY, TEMP
: Autorise la création de tables temporaires lors de l'utilisation de la base de données spécifiée.

EXECUTE
: Autorise l'utilisation de la fonction indiquée et l'utilisation de tout opérateur défini sur cette fonction. C'est le seul type de droit applicable aux fonctions. (Cette syntaxe fonctionne aussi pour les fonctions d'agrégat.)

USAGE
: Pour les langages procéduraux, autorise l'utilisation du langage indiqué pour la création de fonctions. C'est le seul type de droit applicable aux langages procéduraux.

Pour les schémas, autorise l'accès aux objets contenus dans le schéma indiqué (en supposant que les droits des objets soient respectés). Cela octroie, pour l'essentiel, au bénéficiaire le droit de « consulter » les objets contenus dans ce schéma. Sans ce droit, il est toujours possible de voir les noms des objets en lançant des requêtes sur les tables système. De plus, après avoir révoqué ce droit, les processus serveur existants pourraient recevoir des requêtes qui ont déjà réalisé cette recherche auparavant, donc ce n'est pas un moyen complètement sécurisé d'empêcher l'accès aux objets.

Pour les séquences, ce droit autorise l'utilisation des fonctions `currval` et `nextval`.

Pour des wrappers de données distantes, ce droit autorise son bénéficiaire à créer de nouveaux serveurs utilisant ce wrapper.

Pour les serveurs distants, ce droit autorise son bénéficiaire à créer, modifier et supprimer les correspondances utilisateur de l'utilisateur associé à ce serveur. De plus, il autorise son bénéficiaire à interroger les options du serveur et les correspondances d'utilisateurs associées.

ALL PRIVILEGES
: Octroie tous les droits disponibles en une seule opération. Le mot clé `PRIVILEGES` est optionnel sous PostgreSQL™ mais est requis dans le standard SQL.

Les droits requis par les autres commandes sont listés sur les pages de référence de ces commandes.

GRANT sur les rôles

Cette variante de la commande **GRANT** définit l'appartenance d'un (ou plusieurs) rôle(s) à un autre. L'appartenance à un rôle est importante car elle offre tous les droits accordés à un rôle à l'ensemble de ses membres.

Si `WITH ADMIN OPTION` est spécifié, le membre peut à la fois en octroyer l'appartenance à d'autres rôles, et la révoquer. Sans cette option, les utilisateurs ordinaires ne peuvent pas le faire. Un rôle ne dispose pas de l'option `WITH ADMIN OPTION` lui-même mais il peut donner ou enlever son appartenance à partir d'une session où l'utilisateur correspond au rôle. Les superutilisateurs peuvent donner ou supprimer l'appartenance à tout rôle. Les rôles disposant de l'attribut `CREATEROLE` peuvent donner ou supprimer l'appartenance à tout rôle qui n'est pas un superutilisateur.

Contrairement au cas avec les droits, l'appartenance à un rôle ne peut pas être donné à `PUBLIC`. Notez aussi que ce format de la commande n'autorise pas le mot `GROUP`.

Notes

La commande REVOKE(7) est utilisée pour retirer les droits d'accès.

Depuis PostgreSQL™ 8.1, le concept des utilisateurs et des groupes a été unifié en un seul type d'entité appelé rôle. Il n'est donc plus nécessaire d'utiliser le mot clé `GROUP` pour indiquer si le bénéficiaire est un utilisateur ou un groupe. `GROUP` est toujours autorisé dans cette commande mais est ignoré.

Un utilisateur peut exécuter des **SELECT**, **INSERT**, etc. sur une colonne si il a le privilège soit sur cette colonne spécifique, soit sur la table entière. Donner un privilège de table puis le révoquer pour une colonne ne fera pas ce que vous pourriez espérer : l'autorisation au niveau de la table n'est pas affectée par une opération au niveau de la colonne.

Quand un utilisateur, non propriétaire d'un objet, essaie d'octroyer des droits sur cet objet, la commande échoue si l'utilisateur n'a aucun droit sur l'objet. Tant que des privilèges existent, la commande s'exécute, mais n'octroie que les droits pour lesquels l'utilisateur dispose de l'option de transmission. Les formes **GRANT ALL PRIVILEGES** engendrent un message d'avertissement

si aucune option de transmission de droit n'est détenue, tandis que les autres formes n'engendrent un message que lorsque les options de transmission du privilège concerné par la commande ne sont pas détenues. (Cela s'applique aussi au propriétaire de l'objet, mais comme on considère toujours que ce dernier détient toutes les options de transmission, le problème ne se pose jamais.)

Les superutilisateurs de la base de données peuvent accéder à tous les objets sans tenir compte des droits qui les régissent. Cela est comparable aux droits de `root` sur un système Unix. Comme avec `root`, il est déconseillé d'opérer en tant que superutilisateur, sauf en cas d'impérieuse nécessité.

Si un superutilisateur lance une commande **GRANT** ou **REVOKE**, tout se passe comme si la commande était exécutée par le propriétaire de l'objet concerné. Les droits octroyés par cette commande semblent ainsi l'avoir été par le propriétaire de l'objet. (L'appartenance à rôle, elle, semble être donnée par le rôle conteneur.)

GRANT et **REVOKE** peuvent aussi être exécutées par un rôle qui n'est pas le propriétaire de l'objet considéré, mais est membre du rôle propriétaire de l'objet, ou membre du rôle titulaire du privilège `WITH GRANT OPTION` sur cet objet. Dans ce cas, les droits sont enregistrés comme donnés par le rôle propriétaire de l'objet ou titulaire du privilège `WITH GRANT OPTION`. Par exemple, si la table `t1` appartient au rôle `g1`, dont le rôle `u1` est membre, alors `u1` peut donner les droits sur `t1` à `u2`, mais ces droits apparaissent octroyés directement par `g1`. Tout autre membre du rôle `g1` peut les révoquer par la suite.

Si le rôle qui exécute **GRANT** détient, de manière indirecte, les droits souhaités à travers plus d'un niveau d'appartenance, il est difficile de prévoir le rôle reconnu comme fournisseur du privilège. Dans de tels cas, le meilleur moyen d'utiliser **SET ROLE** est de devenir le rôle qui doit octroyer les droits.

Donner un droit sur une table n'étend pas automatiquement les droits sur les séquences utilisées par cette table, ceci incluant les séquences liées par des colonnes de type SERIAL. Les droits sur les séquences doivent être donnés séparément.

La commande **\dp** de psql(1) permet d'obtenir des informations sur les droits existants pour les tables et colonnes, par exemple :

```
=> \z matable
                            Access privileges
 Schema |  Name   | Type  |   Access privileges   | Column access privileges
--------+---------+-------+-----------------------+--------------------------
 public | mytable | table | miriam=arwdDxt/miriam | col1:
                          : =r/miriam             :   miriam_rw=rw/miriam
                          : admin=arw/miriam
(1 row)
```

Les entrées affichées par **\dp** sont interprétées ainsi :

```
      rolename=xxxx -- privileges granted to a role
              =xxxx -- privileges granted to PUBLIC


                  r -- SELECT ("lecture")
                  w -- UPDATE ("écriture")
                  a -- INSERT ("ajout")
                  d -- DELETE
                  D -- TRUNCATE
                  x -- REFERENCES
                  t -- TRIGGER
                  X -- EXECUTE
                  U -- USAGE
                  C -- CREATE
                  c -- CONNECT
                  T -- TEMPORARY
            arwdDxt -- ALL PRIVILEGES (pour les tables, varie pour les autres objets)
                  * -- option de transmission du privilège qui précède

              /yyyy -- role qui a donné le droit
```

L'exemple ci-dessus présente ce que voit l'utilisatrice `miriam` après la création de la table `matable` et l'exécution de :

```
GRANT SELECT ON matable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON matable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON matable TO miriam_rw;
```

Pour les objects non-tables, il y a d'autres commandes **\d** qui peuvent afficher leurs privilèges.

Si la colonne « Access privileges » est vide pour un objet donné, cela signifie que l'objet possède les droits par défaut (c'est-à-dire que la colonne des droits est NULL). Les droits par défaut incluent toujours les droits complets pour le propriétaire et peuvent inclure quelques droits pour `PUBLIC` en fonction du type d'objet comme cela est expliqué plus haut. Le premier **GRANT** ou **REVOKE** sur un objet instancie les droits par défaut (produisant, par exemple, `{=,miriam=arwdDxt/miriam}`) puis les modi-

fie en fonction de la requête spécifiée. Les entrées sont affichées en « Privilèges d'accès aux colonnes » seulement pour les colonnes qui ont des privilèges différents de ceux par défaut. (Notez que, dans ce but, « default privileges » signifie toujours les droits par défaut inhérents au type de l'objet. Un objet dont les droits ont été modifiés avec la commande **ALTER DEFAULT PRIVILEGES** sera toujours affiché avec une entrée de droit effective qui inclut les effets de la commande **ALTER**.)

Les options de transmission de privilèges implicites du propriétaire ne sont pas indiquées dans l'affichage des droits d'accès. Une `*` apparaît uniquement lorsque les options de transmission ont été explicitement octroyées.

Exemples

Donner le droit d'insertion à tous les utilisateurs sur la table `films` :

```
GRANT INSERT ON films TO PUBLIC;
```

Donner tous les droits possibles à l'utilisateur `manuel` sur la vue `genres` :

```
GRANT ALL PRIVILEGES ON genres TO manuel;
```

Bien que la commande ci-dessus donne tous les droits lorsqu'elle est exécutée par un superutilisateur ou par le propriétaire de `genres`, exécutée par quelqu'un d'autre, elle n'accorde que les droits pour lesquels cet utilisateur possède l'option de transmission.

Rendre `joe` membre de `admins` :

```
GRANT admins TO joe;
```

Compatibilité

Conformément au standard SQL, le mot clé `PRIVILEGES` est requis dans `ALL PRIVILEGES`. Le standard SQL n'autorise pas l'initialisation des droits sur plus d'un objet par commande.

PostgreSQL™ autorise un propriétaire d'objet à révoquer ses propres droits ordinaires : par exemple, le propriétaire d'un objet peut le placer en lecture seule pour lui-même en révoquant ses propres droits `INSERT`, `UPDATE`, `DELETE` et `TRUNCATE`. Le standard SQL ne l'autorise pas. La raison en est que PostgreSQL™ traite les droits du propriétaire comme ayant été donnés par le propriétaire ; il peut, de ce fait, aussi les révoquer. Dans le standard SQL, les droits du propriétaire sont donnés par une entité « _SYSTEM ». N'étant pas « _SYSTEM », le propriétaire ne peut pas révoquer ces droits.

Le standard SQL fournit un droit `USAGE` sur d'autres types d'objet : jeux de caractères, collations, conversions, domaines.

Les droits sur les bases de données, tablespaces, langages, schémas et séquences sont des extensions PostgreSQL™.

Voir aussi

REVOKE(7), ALTER DEFAULT PRIVILEGES(7)

Nom

INSERT — Insérer de nouvelles lignes dans une table

Synopsis

```
INSERT INTO table [ ( colonne [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | requête
}
    [ RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] ]
```

Description

INSERT insère de nouvelles lignes dans une table. Vous pouvez insérer une ou plusieurs lignes spécifiées par les expressions de valeur, ou zéro ou plusieurs lignes provenant d'une requête.

L'ordre des noms des colonnes n'a pas d'importance. Si aucune liste de noms de colonnes n'est donnée, toutes les colonnes de la table sont utilisée dans l'ordre de leur déclaration (les *N* premiers noms de colonnes si seules *N* valeurs de colonnes sont fournies dans la clause `VALUES` ou dans la *requête*). Les valeurs fournies par la clause `VALUES` ou par la *requête* sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Chaque colonne absente de la liste, implicite ou explicite, des colonnes se voit attribuer sa valeur par défaut, s'il y en a une, ou NULL dans le cas contraire.

Un transtypage automatique est entrepris lorsque l'expression d'une colonne ne correspond pas au type de donnée déclaré.

La clause `RETURNING` optionnelle fait que **INSERT** calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours d'insertion. C'est principalement utile pour obtenir les valeurs qui ont été fournies par défaut, comme un numéro de séquence. Néanmoins, toute expression utilisant les colonnes de la table est autorisée. La syntaxe de la liste `RETURNING` est identique à celle de la commande **SELECT**.

Le droit `INSERT` sur une table est requis pour pouvoir y insérer des lignes. Si une liste de colonne est indiquée, vous avez seulement besoin d'avoir le droit `INSERT` sur les colonnes listées. L'utilisation de la clause `RETURNING` nécessite le droit `SELECT` sur tous les colonnes mentionnées dans `RETURNING`. Si la clause *requête* est utilisée pour insérer des lignes, le droit `SELECT` sur toute table ou colonne utilisée dans la requête est également requis.

Paramètres

table
: Le nom de la table (éventuellement qualifié du nom du schéma).

colonne
: Le nom d'une colonne de *table*. Le nom de la colonne peut être qualifié avec un nom de sous-champ ou un indice de tableau, si nécessaire. (N'insérer que certains champs d'une colonne composite laisse les autres champs à NULL.)

`DEFAULT VALUES`
: Toutes les colonnes se voient attribuer leur valeur par défaut.

expression
: Une expression ou une valeur à affecter à la *colonne* correspondante.

`DEFAULT`
: La *colonne* correspondante se voit attribuer sa valeur par défaut.

requête
: Une requête (instruction **SELECT**) dont le résultat fournit les lignes à insérer. La syntaxe complète de la commande est décrite dans la documentation de l'instruction SELECT(7).

expression_sortie
: Une expression à calculer et renvoyée par la commande **INSERT** après chaque insertion de ligne. L'expression peut utiliser tout nom de colonne de la *table*. Indiquez `*` pour que toutes les colonnes soient renvoyées.

nom_sortie
: Un nom à utiliser pour une colonne renvoyée.

Sorties

En cas de succès, la commande **INSERT** renvoie un code de la forme

```
INSERT oid nombre
```

nombre correspond au nombre de lignes insérées. Si *nombre* vaut exactement un et que la table cible contient des OID, alors *oid* est l'OID affecté à la ligne insérée. Sinon, *oid* vaut zéro.

Si la commande **INSERT** contient une clause RETURNING, le résultat sera similaire à celui d'une instruction **SELECT** contenant les colonnes et les valeurs définies dans la liste RETURNING, à partir de la liste des lignes insérées par la commande.

Exemples

Insérer une ligne dans la table films :

```
INSERT INTO films
        VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comédie', '82 minutes');
```

Dans l'exemple suivant, la colonne longueur est omise et prend donc sa valeur par défaut :

```
INSERT INTO films (code, titre, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drame');
```

L'exemple suivant utilise la clause DEFAULT pour les colonnes date plutôt qu'une valeur précise :

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comédie', '82 minutes');
INSERT INTO films (code, titre, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame');
```

Insérer une ligne constituée uniquement de valeurs par défaut :

```
INSERT INTO films DEFAULT VALUES;
```

Pour insérer plusieurs lignes en utilisant la syntaxe multi-lignes **VALUES** :

```
INSERT INTO films (code, titre, did, date_prod, genre) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

Insérer dans la table films des lignes extraites de la table tmp_films (la disposition des colonnes est identique dans les deux tables) :

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

Insérer dans des colonnes de type tableau :

```
-- Créer un jeu de 3 cases sur 3
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- Les indices de l'exemple ci-dessus ne sont pas vraiment nécessaires
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Insérer une ligne simple dans la table distributeurs, en renvoyant le numéro de séquence généré par la clause DEFAULT :

```
INSERT INTO distributeurs (did, dnom) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Compatibilité

INSERT est conforme au standard SQL, sauf la clause RETURNING qui est une extension PostgreSQL™. Le standard n'autorise toutefois pas l'omission de la liste des noms de colonnes alors qu'une valeur n'est pas affectée à chaque colonne, que ce soit à l'aide de la clause VALUES ou à partir de la *requête*.

Les limitations possibles de la clause *requête* sont documentées sous SELECT(7).

Nom

LISTEN — Attendre une notification

Synopsis

```
LISTEN canal
```

Description

LISTEN enregistre la session courante comme listener du canal de notification *canal*. Si la session courante est déjà enregistrée comme listener de ce canal de notification, il ne se passe rien de plus.

À chaque appel de la commande **NOTIFY *canal***, que ce soit par cette session ou par une autre connectée à la même base de données, toutes les sessions attendant sur ce canal en sont avisées et chacune en avise en retour son client. Voir **NOTIFY** pour plus d'informations.

La commande **UNLISTEN** permet d'annuler l'enregistrement d'une session comme listener d'un canal de notification. Les enregistrements d'écoute d'une session sont automatiquement effacés lorsque la session se termine.

La méthode utilisé par un client pour détecter les événements de notification dépend de l'interface de programmation PostgreSQL™ qu'il utilise. Avec la bibliothèque libpq, l'application exécute **LISTEN** comme une commande SQL ordinaire, puis appelle périodiquement la fonction `PQnotifies` pour savoir si un événement de notification est reçu. Les autres interfaces, telle libpgtcl, fournissent des méthodes de plus haut niveau pour gérer les événements de notification ; en fait, avec libpgtcl, le développeur de l'application n'a même pas à lancer **LISTEN** ou **UNLISTEN** directement. Tous les détails se trouvent dans la documentation de l'interface utilisée.

NOTIFY(7) décrit plus en détails l'utilisation de **LISTEN** et **NOTIFY**.

Paramètres

canal
: Le nom d'un canal de notification (tout identifiant).

Notes

LISTEN prend effet à la validation de la transaction. Si **LISTEN** ou **UNLISTEN** est exécuté dans une transaction qui sera ensuite annulée, l'ensemble des canaux de notification écoutés sera inchangé.

Une transaction qui a exécuté **LISTEN** ne peut pas être préparée pour la validation en deux phases.

Exemples

Configurer et exécuter une séquence listen/notify à partir de psql :

```
LISTEN virtual;
NOTIFY virtual;
Notification asynchrone "virtual" reçue en provenance du processus serveur de PID
8448.
```

Compatibilité

Il n'existe pas d'instruction **LISTEN** dans le standard SQL.

Voir aussi

NOTIFY(7), UNLISTEN(7)

Nom

LOAD — Charger une bibliothèque partagée

Synopsis

```
LOAD 'fichier'
```

Description

Cette commande charge une bibliothèque partagée dans l'espace d'adressage de PostgreSQL™. Si le fichier a déjà été chargé, la commande ne fait rien. Les fichiers des bibliothèques partagées contenant des fonctions C sont automatiquement chargés à chaque fois qu'une de leur fonctions est appelée. Du coup, un appel explicite à **LOAD** est habituellement seulement nécessaire pour charger une bibliothèque qui modifie le comportement du serveur via des « points d'accroche » plutôt qu'en fournissant un ensemble de fonctions.

Le nom du fichier est indiqué de la même façon que pour les noms de bibliothèques partagées dans CREATE FUNCTION(7) ; il est, en particulier, possible d'utiliser un chemin de recherche et l'ajout automatique de l'extension de la bibliothèque partagée, suivant les standards système. Voir Section 35.9, « Fonctions en langage C » pour plus d'informations sur ce thème.

Les utilisateurs normaux peuvent seulement utiliser **LOAD** avec des bibliothèques situées dans `$libdir/plugins/` -- le *`nom_fichier`* indiqué doit commencer avec cette chaîne exacte. (Il est de la responsabilité de l'administrateur de bases de données de s'assurer que seules des bibliothèques « sûres » y sont installées.)

Compatibilité

LOAD est une extension PostgreSQL™.

Voir aussi

CREATE FUNCTION(7)

Nom

LOCK — verrouiller une table

Synopsis

```
LOCK [ TABLE ] [ ONLY ] nom [ * ] [, ...] [ IN mode_verrou MODE ] [ NOWAIT ]

où mode_verrou peut être :

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE
ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Description

LOCK TABLE prend un verrou de niveau table, attendant si nécessaire que tout verrou conflictuel soit relâché. Si `NOWAIT` est spécifié, **LOCK TABLE** n'attend pas l'acquisition du verrou désiré : s'il ne peut pas être obtenu immédiatement, la commande est annulée et une erreur est émise. Une fois obtenu, le verrou est conservé jusqu'à la fin de la transaction en cours. (Il n'y a pas de commande **UNLOCK TABLE** ; les verrous sont systématiquement relâchés à la fin de la transaction.)

Lors de l'acquisition automatique de verrous pour les commandes qui référencent des tables, PostgreSQL™ utilise toujours le mode de verrou le moins restrictif possible. **LOCK TABLE** est utilisable lorsqu'il est nécessaire d'obtenir des verrous plus restrictifs.

Soit, par exemple, une application qui exécute une transaction de niveau d'isolation « lecture des validés » (*Read Committed*). Pour s'assurer que les données de la table sont immuables pendant toute la durée de la transaction, un verrou `SHARE` de niveau table peut être obtenu avant d'effectuer la requête. Cela empêche toute modification concurrente des données. Cela assure également que toute lecture intervenant ensuite sur la table accède à la même vue des données validées. En effet, un verrou `SHARE` entre en conflit avec le verrou `ROW EXCLUSIVE` pris par les modificateurs et l'instruction **LOCK TABLE *nom* IN SHARE MODE** attend que tout détenteur concurrent de verrous de mode `ROW EXCLUSIVE` valide ou annule. De ce fait, une fois le verrou obtenu, il ne reste aucune écriture non validée en attente ; de plus, aucune ne peut commencer tant que le verrou acquis n'est pas relâché.

Pour obtenir un effet similaire lors de l'exécution d'une transaction de niveau d'isolation sérialisable, il est nécessaire d'exécuter l'instruction **LOCK TABLE** avant toute instruction **SELECT** ou de modification de données. La vue des données utilisée par une transaction sérialisable est figée au moment où débute la première instruction **SELECT** ou de modification des données. Un **LOCK TABLE** ultérieur empêche encore les écritures concurrentes -- mais il n'assure pas que la transaction lit les dernières données validées.

Si une telle transaction modifie les données de la table, elle doit utiliser le mode de verrou `SHARE ROW EXCLUSIVE` au lieu du mode `SHARE`. Cela assure l'exécution d'une seule transaction de ce type à la fois. Sans cela, une situation de verrou mort est possible : deux transactions peuvent acquérir le mode `SHARE` et être ensuite incapables d'acquérir aussi le mode `ROW EXCLUSIVE` pour réellement effectuer leurs mises à jour. (Les verrous d'une transaction ne sont jamais en conflit. Une transaction peut de ce fait acquérir le mode `ROW EXCLUSIVE` alors qu'elle détient le mode `SHARE` -- mais pas si une autre transaction détient le mode `SHARE`.) Pour éviter les verrous bloquants, il est préférable que toutes les transactions qui acquièrent des verrous sur les mêmes objets le fassent dans le même ordre. De plus si de multiples modes de verrous sont impliqués pour un même objet, le verrou de mode le plus restrictif doit être acquis le premier.

Plus d'informations sur les modes de verrou et les stratégies de verrouillage sont disponibles dans Section 13.3, « Verrouillage explicite ».

Paramètres

nom

Le nom d'une table à verrouiller (éventuellement qualifié du nom du schéma). Si `ONLY` est précisé avant le nom de la table, seule cette table est verrouillée. Dans le cas contraire, la table et toutes ses tables filles (si elle en a) sont verrouillées. En option, `*` peut être placé après le nom de la table pour indiquer explicitement que les tables filles sont inclues.

La commande `LOCK a, b;` est équivalente à `LOCK a; LOCK b;`. Les tables sont verrouillées une par une dans l'ordre précisé par la commande **LOCK TABLE**.

modeverrou

Le mode de verrou précise les verrous avec lesquels ce verrou entre en conflit. Les modes de verrou sont décrits dans Section 13.3, « Verrouillage explicite ».

Si aucun mode de verrou n'est précisé, `ACCESS EXCLUSIVE`, mode le plus restrictif, est utilisé.

`NOWAIT`
LOCK TABLE n'attend pas que les verrous conflictuels soient relâchés : si le verrou indiqué ne peut être acquis immédiatement sans attente, la transaction est annulée.

Notes

`LOCK TABLE ... IN ACCESS SHARE MODE` requiert les droits `SELECT` sur la table cible. Toutes les autres formes de **LOCK** requièrent au moins un des droits `UPDATE`, `DELETE` et `TRUNCATE`.

LOCK TABLE est inutile à l'extérieur d'un bloc de transaction : le verrou est détenu jusqu'à la fin de l'instruction. Du coup, PostgreSQL™ renvoie une erreur si **LOCK** est utilisé en dehors d'un bloc de transaction. Utilisez BEGIN(7) et COMMIT(7) (ou ROLLBACK(7)) pour définir un bloc de transaction.

LOCK TABLE ne concernent que les verrous de niveau table. Les noms de mode contenant `ROW` sont donc tous mal nommés. Ces noms de modes doivent généralement être compris comme indiquant l'intention de l'utilisateur d'acquérir des verrous de niveau ligne à l'intérieur de la table verrouillée. Le mode `ROW EXCLUSIVE` est également un verrou de table partageable. Tous les modes de verrou ont des sémantiques identiques en ce qui concerne **LOCK TABLE** ; ils ne diffèrent que dans les règles de conflit entre les modes. Pour des informations sur la façon d'acquérir un vrai verrou de niveau ligne, voir Section 13.3.2, « Verrous au niveau ligne » et la section intitulée « Clause `FOR UPDATE`/`FOR SHARE` » dans la documentation de référence de **SELECT**.

Exemples

Obtenir un verrou `SHARE` sur une table avec clé primaire avant de réaliser des insertions dans une table disposant de la clé étrangère :

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE nom = 'Star Wars : Episode I - La menace fantôme';
-- Effectuer un ROLLBACK si aucun enregistrement n'est retourné
INSERT INTO commentaires_films VALUES
    (_id_, 'SUPER ! Je l''attendais depuis si longtemps !');
COMMIT WORK;
```

Prendre un verrou `SHARE ROW EXCLUSIVE` sur une table avec clé primaire lors du début des opérations de suppression :

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM commentaires_films WHERE id IN
    (SELECT id FROM films WHERE score < 5);
DELETE FROM films WHERE score < 5;
COMMIT WORK;
```

Compatibilité

LOCK TABLE n'existe pas dans le standard SQL. À la place, il utilise **SET TRANSACTION** pour spécifier les niveaux de concurrence entre transactions. PostgreSQL™ en dispose également ; voir SET TRANSACTION(7) pour les détails.

À l'exception des modes de verrous `ACCESS SHARE`, `ACCESS EXCLUSIVE` et `SHARE UPDATE EXCLUSIVE`, les modes de verrou PostgreSQL™ et la syntaxe **LOCK TABLE** sont compatibles avec ceux présents dans Oracle™.

Nom

MOVE — positionner un curseur

Synopsis

```
MOVE [ direction [ FROM | IN ] ] nom_curseur
```

Description

MOVE repositionne un curseur sans retourner de donnée. **MOVE** fonctionne exactement comme la commande **FETCH** à la différence que **MOVE** ne fait que positionner le curseur et ne retourne aucune ligne.

Les paramètres de la commande **MOVE** sont identiques à ceux de la commande **FETCH**. FETCH(7) contient les détails de syntaxe et d'utilisation.

Sortie

En cas de réussite, une commande **MOVE** retourne une balise de commande de la forme

```
MOVE compteur
```

compteur est le nombre de lignes qu'une commande **FETCH** avec les mêmes paramètres aurait renvoyée (éventuellement zéro).

Exemples

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Saute les 5 premières lignes :
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Récupère la 6ème ligne à partir du curseur liahona :
FETCH 1 FROM liahona;
 code  | titre  | did | date_prod  | genre  |  longueur
-------+--------+-----+------------+--------+-----------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- Ferme le curseur liahona et termine la transaction :
CLOSE liahona;
COMMIT WORK;
```

Compatibilité

Il n'existe pas d'instruction **MOVE** dans le standard SQL.

Voir aussi

CLOSE(7), DECLARE(7), FETCH(7)

Nom

NOTIFY — engendrer une notification

Synopsis

```
NOTIFY canal [ , charge ]
```

Description

La commande **NOTIFY** envoie une notification avec une chaîne de « charge » supplémentaire à chaque application cliente qui a exécuté précédemment la commande **LISTEN** *canal* dans la base de données courante pour le nom du canal indiqué.

NOTIFY fournit un mécanisme simple de communication interprocessus pour tout ensemble de processus accédant à la même base de données PostgreSQL™. Une chaîne de charge peut être envoyée avec la notification, et des mécanismes de plus haut niveau permettant de passer des données structurées peuvent être construits en utilisant les tables de la base de données.

L'information passée au client pour une notification inclut le nom de la notification et le PID du processus serveur de la session le notifiant.

C'est au concepteur de la base de données de définir les noms de notification utilisés dans une base de données précise et la signification de chacun. Habituellement, le nom du canal correspond au nom d'une table dans la base de données. L'événement notify signifie essentiellement « J'ai modifié cette table, jetez-y un œil pour vérifier ce qu'il y a de nouveau ». Mais cette association n'est pas contrôlée par les commandes **NOTIFY** et **LISTEN**. Un concepteur de bases de données peut, par exemple, utiliser plusieurs noms de canal différents pour signaler différentes sortes de modifications au sein d'une même table. Sinon, la chaîne de charge peut être utilisée pour différencier plusieurs cas.

Lorsque **NOTIFY** est utilisé pour signaler des modifications sur une table particulière, une technique de programmation utile est de placer le **NOTIFY** dans une règle déclenchée par les mises à jour de la table. De cette façon, la notification est automatique lors d'une modification de la table et le programmeur de l'application ne peut accidentellement oublier de le faire.

NOTIFY interagit fortement avec les transactions SQL. Primo, si un **NOTIFY** est exécuté à l'intérieur d'une transaction, les événements notify ne sont pas délivrés avant que la transaction ne soit validée, et à cette condition uniquement. En effet, si la transaction est annulée, les commandes qu'elle contient n'ont aucun effet, y compris **NOTIFY**. Cela peut toutefois s'avérer déconcertant pour quiconque s'attend à une délivrance immédiate des notifications.

Secondo, si une session à l'écoute reçoit un signal de notification alors qu'une transaction y est active, la notification n'est pas délivrée au client connecté avant la fin de cette transaction (par validation ou annulation). Là encore, si une notification est délivrée à l'intérieur d'une transaction finalement annulée, on pourrait espérer annuler cette notification par quelque moyen -- mais le serveur ne peut pas « reprendre » une notification déjà envoyée au client. C'est pourquoi les notifications ne sont délivrés qu'entre les transactions. Il est, de ce fait, important que les applications qui utilisent **NOTIFY** pour l'envoi de signaux en temps réel conservent des transactions courtes.

Si le même nom de canal est signalé plusieurs fois à partir de la même transaction avec des chaînes de charge identiques, le serveur de bases de données peut décider de délivrer une seule notification. Par contre, les notifications avec des chaînes de charges distinctes seront toujours délivrées par des notifications distinctes. De façon similaire, les notifications provenant de différentes transactions ne seront jamais regroupées en une seule notification. Sauf pour supprimer des instances ultérieures de notifications dupliquées, la commande **NOTIFY** garantie que les notifications de la même transaction seront délivrées dans l'ordre où elles ont été envoyées. Il est aussi garantie que les messages de transactions différentes seront délivrées dans l'ordre dans lequel les transactions ont été validées.

Il est courant qu'un client qui exécute **NOTIFY** écoute lui-même des notifications de même canal. Dans ce cas, il récupère une notification, comme toutes les autres sessions en écoute. Suivant la logique de l'application, cela peut engendre un travail inutile, par exemple lire une table de la base de données pour trouver les mises à jour que cette session a elle-même écrites. Il est possible d'éviter ce travail supplémentaire en verifiant si le PID du processus serveur de la session notifiante (fourni dans le message d'événement de la notification) est le même que le PID de la session courante (disponible à partir de libpq). S'ils sont identiques, la notification est le retour du travail actuel et peut être ignorée.

Paramètres

canal
: Nom du canal à signaler (identifiant quelconque).

charge
: La chaîne de « charge » à communiquer avec la notification. Elle doit être spécifiée comme une chaîne litérale. Dans la

configuration par défaut, elle doit avoir une taille inférieure à 8000 octets. (Si des données binaires ou de tailles plus importantes doivent être communiquées, il est mieux de les placer dans une table de la base et d'envoyer la clé correspondant à l'enregistrement.)

Notes

Il existe une queue qui récupère les notifications qui ont été envoyées mais pas encore traitées par les sessions en écoute. Si la queue est remplie, les transactions appelant **NOTIFY** échoueront à la validation. La queue est assez large (8 Go dans une installation standard) et devrait être suffisamment bien taillée dans la majorité des cas. Néanmoins, aucun nettoyage ne peut se faire si une session exécute **LISTEN** puis entre en transaction pendant une longue période. Une fois qu'une queue est à moitié pleine, des messages d'avertissements seront envoyés dans les traces indiquant la session qui empêche le nettoyage. Dans ce cas, il faut s'assurer que la session termine sa transaction en cours pour que le nettoyage puisse se faire.

Une transaction qui a exécuté **NOTIFY** ne peut pas être préparée pour une validation en deux phases.

pg_notify

Pour envoyer une notification, vous pouvez aussi utiliser la fonction `pg_notify(text, text)`. La fonction prend en premier argument le nom du canal et en second la charge. La fonction est bien plus simple à utiliser que la commande **NOTIFY** si vous avez besoin de travailler avec des noms de canaux et des charges non constants.

Exemples

Configurer et exécuter une séquence listen/notify à partir de psql :

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from
server process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process
with PID 14728.
```

Compatibilité

Il n'y a pas d'instruction **NOTIFY** dans le standard SQL.

Voir aussi

LISTEN(7), UNLISTEN(7)

Nom

PREPARE — prépare une instruction pour exécution

Synopsis

```
PREPARE nom [ (type_données [, ...] ) ] AS instruction
```

Description

PREPARE crée une instruction préparée. Une instruction préparée est un objet côté serveur qui peut être utilisé pour optimiser les performances. Quand l'instruction **PREPARE** est exécutée, l'instruction spécifiée est analysée, réécrite et planifiée. Quand une commande **EXECUTE** est lancée par la suite, l'instruction préparée a seulement besoin d'être exécutée. Du coup, les étapes d'analyse, de réécriture et de planification sont réalisées une seule fois, à la place de chaque fois que l'instruction est exécutée.

Les instructions préparées peuvent prendre des paramètres : les valeurs sont substituées dans l'instruction lorsqu'elle est exécutée. Lors de la création de l'instruction préparée, faites référence aux paramètres suivant leur position, `$1`, `$2`, etc. Une liste correspondante des types de données des paramètres peut être spécifiée si vous le souhaitez. Quand le type de donnée d'un paramètre n'est pas indiqué ou est déclaré comme inconnu (`unknown`), le type est inféré à partir du contexte dans lequel le paramètre est utilisé (si possible). Lors de l'exécution de l'instruction, indiquez les valeurs réelles de ces paramètres dans l'instruction **EXECUTE**. Référez-vous à EXECUTE(7) pour plus d'informations à ce sujet.

Les instructions préparées sont seulement stockées pour la durée de la session en cours. Lorsque la session se termine, l'instruction préparée est oubliée et, du coup, elle doit être recréée avant d'être utilisée de nouveau. Ceci signifie aussi qu'une seule instruction préparée ne peut pas être utilisée par plusieurs clients de bases de données simultanément ; néanmoins, chaque client peut créer sa propre instruction préparée à utiliser. L'instruction préparée peut être supprimés manuellement en utilisant la commande DEALLOCATE(7).

Les instructions préparées sont principalement intéressantes quand une seule session est utilisée pour exécuter un grand nombre d'instructions similaires. La différence de performances est particulièrement significative si les instructions sont complexes à planifier ou à réécrire, par exemple, si la requête implique une jointure de plusieurs tables ou requiert l'application de différentes règles. Si l'instruction est relativement simple à planifier ou à réécrire mais assez coûteuse à exécuter, l'avantage de performance des instructions préparées est moins net.

Paramètres

`nom`
: Un nom quelconque donné à cette instruction préparée particulière. Il doit être unique dans une session et est utilisé par la suite pour exécuter ou désallouer cette instruction préparée.

`type_données`
: Le type de données d'un paramètre de l'instruction préparée. Si le type de données d'un paramètre particulier n'est pas spécifié ou est spécifié comme étant inconnu (`unknown`), il sera inferré à partir du contexte dans lequel le paramètre est utilisé. Pour référencer les paramètres de l'instruction préparée, utilisez `$1`, `$2`, etc.

`instruction`
: Toute instruction **SELECT**, **INSERT**, **UPDATE**, **DELETE** ou **VALUES**.

Notes

Dans certaines situations, le plan de requête produit par une instruction préparée est inférieur au plan qui aurait été produit si l'instruction avait été soumise et exécutée normalement. C'est parce que, quand l'instruction est planifiée et que le planificateur tente de déterminer le plan de requête optimal, les valeurs réelles de tous les paramètres spécifiés dans l'instruction ne sont pas disponibles. PostgreSQL™ récupère les statistiques de la distribution des données dans la table et peut utiliser les valeurs constantes dans une instruction pour deviner le résultat probable de l'exécution de l'instruction. Comme cette donnée n'est pas disponible lors de la planification d'instructions préparées avec paramètres, le plan choisi pourrait ne pas être optimal. Pour examiner le plan de requête que PostgreSQL™ a choisi pour une instruction préparée, utilisez EXPLAIN(7).

Pour plus d'informations sur la planification de la requête et les statistiques récupérées par PostgreSQL™ dans ce but, voir la documentation de ANALYZE(7).

Vous pouvez voir toutes les instructions préparées disponibles d'une session en exécutant une requête sur la vue système pg_prepared_statements.

Exemples

Crée une instruction préparée pour une instruction **INSERT**, puis l'exécute :

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Crée une instruction préparée pour une instruction **SELECT**, puis l'exécute :

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Note that the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used.

Compatibilité

Le standard SQL inclut une instruction **PREPARE** mais il est seulement utilisé en SQL embarqué. Cette version de l'instruction **PREPARE** utilise aussi une syntaxe quelque peu différente.

Voir aussi

DEALLOCATE(7), EXECUTE(7)

Nom

PREPARE TRANSACTION — prépare la transaction en cours pour une validation en deux phases

Synopsis

```
PREPARE TRANSACTION id_transaction
```

Description

PREPARE TRANSACTION prépare la transaction courante en vue d'une validation en deux phases. À la suite de cette commande, la transaction n'est plus associée à la session courante ; au lieu de cela, son état est entièrement stocké sur disque. La probabilité est donc forte qu'elle puisse être validée avec succès, y compris en cas d'arrêt brutal de la base de données avant la demande de validation.

Une fois préparée, une transaction peut être validée ou annulée ultérieurement par, respectivement, COMMIT PREPARED(7) et ROLLBACK PREPARED(7). Ces commandes peuvent être exécutées à partir d'une session quelconque. Il n'est pas nécessaire de le faire depuis celle qui a exécuté la transaction initiale.

Du point de vue de la session l'initiant, **PREPARE TRANSACTION** diffère peu de la commande **ROLLBACK** : après son exécution, il n'y a plus de transaction active et les effets de la transaction préparée ne sont plus visibles. (Les effets redeviendront visibles si la transaction est validée.)

Si la commande **PREPARE TRANSACTION** échoue, quelqu'en soit la raison, elle devient une commande **ROLLBACK** : la transaction courante est annulée.

Paramètres

`id_transaction`
: Un identifiant arbitraire de la transaction pour les commandes **COMMIT PREPARED** et **ROLLBACK PREPARED**. L'identifiant, obligatoirement de type chaîne littérale, doit être d'une longueur inférieure à 200 octets. Il ne peut être identique à un autre identifiant de transaction préparée.

Notes

PREPARE TRANSACTION n'a pas pour but d'être utilisé dans des applications ou des sessions interactives. Son but est de permettre à un gestionnaire de transactions externe pour réaliser des transactions globales atomiques au travers de plusieurs bases de données ou de ressources transactionnelles. Sauf si vous écrivez un gestionnaire de transactions, vous ne devriez probablement pas utiliser **PREPARE TRANSACTION**.

Cette commande doit être utilisée dans un bloc de transaction, initié par BEGIN(7).

Il n'est actuellement pas possible de préparer (**PREPARE**) une transaction qui a exécuté des opérations impliquant des tables temporaires ou qui a créé des curseurs `WITH HOLD`, ou qui a exécuté **LISTEN** ou **UNLISTEN**. Ces fonctionnalités sont trop intégrées à la session en cours pour avoir la moindre utilité dans une transaction préparée.

Si la transaction a modifié des paramètres en exécution à l'aide de la commande **SET** (sans l'option `LOCAL`), ces effets persistent au-delà du **PREPARE TRANSACTION** et ne seront pas affectés par les commandes **COMMIT PREPARED** et **ROLLBACK PREPARED**. Du coup, dans ce cas, **PREPARE TRANSACTION** agit plus comme **COMMIT** que comme **ROLLBACK**.

Toutes les transactions préparées disponibles sont listées dans la vue système pg_prepared_xacts.

> **Attention**
>
> Il est préférable de ne pas conserver trop longtemps des transactions préparées dans cet état ; cela compromet, par exemple, les possibilités de récupération de l'espace par **VACUUM**, et dans certains cas extrêmes peut causer l'arrêt de la base de données pour empêcher une réutilisation d'identifiants de transactions (voir Section 23.1.4, « Éviter les cycles des identifiants de transactions »). Il ne faut pas oublier non plus qu'une telle transaction maintient les verrous qu'elle a posé. L'usage principal de cette fonctionnalité consiste à valider ou annuler une transaction préparée dès lors qu'un gestionnaire de transactions externe a pu s'assurer que les autres bases de données sont préparées à la validation.
>
> Si vous n'avez pas configuré un gestionnaire de transactions externe pour gérer les transactions préparées et vous

assurer qu'elles sont fermées rapidement, il est préférable de désactiver la fonctionnalité des transactions préparées en configurant max_prepared_transactions à zéro. Ceci empêchera toute création accidentelle de transactions préparées qui pourraient alors être oubliées, ce qui finira par causer des problèmes.

Exemples

Préparer la transaction en cours pour une validation en deux phases en utilisant `foobar` comme identifiant de transaction :

```
PREPARE TRANSACTION 'foobar';
```

Voir aussi

COMMIT PREPARED(7), ROLLBACK PREPARED(7)

Nom

REASSIGN OWNED — Modifier le propriétaire de tous les objets de la base appartenant à un rôle spécifique

Synopsis

```
REASSIGN OWNED BY ancien_rôle [, ...] TO nouveau_rôle
```

Description

REASSIGN OWNED demande au système de changer le propriétaire certains objets de la base. Les objets appartenant à ancien_rôle auront ensuite comme propriétaire nouveau_rôle.

Paramètres

ancien_rôle
: Le nom d'un rôle. Tous les objets de la base appartenant à ce rôle seront la propriété de *nouveau_rôle*.

nouveau_rôle
: Le nom du rôle qui sera le nouveau propriétaire des objets affectés.

Notes

REASSIGN OWNED est souvent utilisé pour préparer à la suppression de un ou plusieurs rôles. Comme **REASSIGN OWNED** touche seulement les objets de la base où l'utilisateur est connecté, il est généralement nécessaire d'exécuter cette commande pour chaque base contenant des objets dont le rôle à supprimer est propriétaire.

REASSIGN OWNED nécessite des droits sur le rôle source et sur le rôle cible.

La commande DROP OWNED(7) est une alternative qui supprime tous les objets de la base possédés par un ou plusieurs rôles. Notez aussi que **DROP OWNED** nécessite seulement des droits sur le rôle source.

La commande **REASSIGN OWNED** ne modifie pas les droits donnés à ancien_rôle pour les objets dont il n'est pas propriétaire. Utilisez **DROP OWNED** pour supprimer ces droits.

La commande **REASSIGN OWNED** ne modifie pas le propriétaire des bases de données, même si le rôle est en propriétaire. Utilisez ALTER DATABASE(7) pour modifier le propriétaire des bases de données.

Compatibilité

L'instruction **REASSIGN OWNED** est une extension PostgreSQL™.

Voir aussi

DROP OWNED(7), DROP ROLE(7), ALTER DATABASE(7)

Nom

REINDEX — reconstruit les index

Synopsis

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } nom [ FORCE ]
```

Description

REINDEX reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. Il y a plusieurs raisons pour utiliser **REINDEX** :

- Un index a été corrompu et ne contient plus de données valides. Bien qu'en théorie, ceci ne devrait jamais arriver, en pratique, les index peuvent se corrompre à cause de bogues dans le logiciel ou d'échecs matériels. **REINDEX** fournit une méthode de récupération.
- L'index en question a « explosé », c'est-à-dire qu'il contient beaucoup de pages d'index mortes ou presque mortes. Ceci peut arriver avec des index B-tree dans PostgreSQL™ sous certains modèles d'accès inhabituels. **REINDEX** fournit un moyen de réduire la consommation d'espace de l'index en écrivant une nouvelle version de l'index sans les pages mortes. Voir Section 23.2, « Ré-indexation régulière » pour plus d'informations.
- Vous avez modifié un paramètre de stockage (par exemple, fillfactor) pour un index et vous souhaitez vous assurer que la modification a été prise en compte.
- La construction d'un index avec l'option `CONCURRENTLY` a échoué, laissant un index « invalide ». De tels index sont inutiles donc il est intéressant d'utiliser **REINDEX** pour les reconstruire. Notez que **REINDEX** n'exécutera pas une construction en parallèle. Pour construire l'index sans interférer avec le système en production, vous devez supprimer l'index et ré-exécuter la commande **CREATE INDEX CONCURRENTLY**.

Paramètres

`INDEX`
: Recrée l'index spécifié.

`TABLE`
: Recrée tous les index de la table spécifiée. Si la table a une seconde table « TOAST », elle est aussi réindexée.

`DATABASE`
: Recrée tous les index de la base de données en cours. Les index sur les catalogues système partagés sont aussi traités. Cette forme de **REINDEX** ne peut pas être exécuté à l'intérieur d'un bloc de transaction.

`SYSTEM`
: Recrée tous les index des catalogues système à l'intérieur de la base de données en cours. Les index sur les catalogues système partagés sont aussi inclus. Les index des tables utilisateur ne sont pas traités. Cette forme de **REINDEX** ne peut pas être exécuté à l'intérieur d'un bloc de transaction.

`nom`
: Le nom de l'index, de la table ou de la base de données spécifique à réindexer. Les noms de table et d'index peuvent être qualifiés du nom du schéma. Actuellement, **REINDEX DATABASE** et **REINDEX SYSTEM** ne peuvent réindexer que la base de données en cours, donc ce paramètre doit correspondre au nom de la base de données en cours.

`FORCE`
: Ceci est une option obsolète ; elle sera ignorée si celle-ci est spécifiée.

Notes

Si vous suspectez la corruption d'un index sur une table utilisateur, vous pouvez simplement reconstruire cet index, ou tous les index de la table, en utilisant **REINDEX INDEX** ou **REINDEX TABLE**.

Les choses sont plus difficiles si vous avez besoin de récupérer la corruption d'un index sur une table système. Dans ce cas, il est important pour le système de ne pas avoir utilisé lui-même un des index suspects. (En fait, dans ce type de scénario, vous pourriez constater que les processus serveur s'arrêtent brutalement au lancement du service, en cause l'utilisation des index corrompus.) Pour récupérer proprement, le serveur doit être lancé avec l'option `-P`, qui inhibe l'utilisation des index pour les recherches

dans les catalogues système.

Une autre façon est d'arrêter le serveur et de relancer le serveur PostgreSQL™ en mode simple utilisateur avec l'option `-P` placée sur la ligne de commande. Ensuite, **REINDEX DATABASE**, **REINDEX SYSTEM**, **REINDEX TABLE** ou **REINDEX INDEX** peuvent être lancés suivant ce que vous souhaitez reconstruire. En cas de doute, utilisez la commande **REINDEX SYSTEM** pour activer la reconstruction de tous les index système de la base de données. Enfin, quittez la session simple utilisateur du serveur et relancez le serveur en mode normal. Voir la page de référence de postgres(1) pour plus d'informations sur l'interaction avec l'interface du serveur en mode simple utilisateur.

Une session standard du serveur peut aussi être lancée avec `-P` dans les options de la ligne de commande. La méthode pour ce faire varie entre les clients mais dans tous les clients basés sur libpq, il est possible de configurer la variable d'environnement `PGOPTIONS` à `-P` avant de lancer le client. Notez que, bien que cette méthode ne verrouille pas les autres clients, il est conseillé d'empêcher les autres utilisateurs de se connecter à la base de données endommagée jusqu'à la fin des réparations.

REINDEX est similaire à une suppression et à une nouvelle création de l'index, dans les fait le contenu de l'index est complètement recréé. Néanmoins, les considérations de verrouillage sont assez différentes. **REINDEX** verrouille les écritures mais pas les lectures de la table mère de l'index. Il positionne également un verrou exclusif sur l'index en cours de traitement, ce qui bloque les lectures qui tentent de l'utiliser. Au contraire, **DROP INDEX** crée temporairement un verrou exclusif sur la table parent, bloquant ainsi écritures et lectures. Le **CREATE INDEX** qui suit verrouille les écritures mais pas les lectures ; comme l'index n'existe pas, aucune lecture ne peut être tentée, signifiant qu'il n'y a aucun blocage et que les lectures sont probablement forcées de réaliser des parcours séquentiels complets.

Ré-indexer un seul index ou une seule table requiert d'être le propriétaire de cet index ou de cette table. Ré-indexer une base de données requiert d'être le propriétaire de la base de données (notez du coup que le propriétaire peut reconstruire les index de tables possédées par d'autres utilisateurs). Bien sûr, les superutilisateurs peuvent toujours tout ré-indexer.

Avant PostgreSQL™ 8.1, **REINDEX DATABASE** traitait seulement les index systèmes, pas tous les index comme on pourrait le supposer d'après le nom. Ceci a été modifié pour réduire le facteur de surprise. L'ancien comportement est disponible en tant que **REINDEX SYSTEM**.

Avant PostgreSQL™ 7.4, **REINDEX TABLE** ne traitait pas automatiquement les tables TOAST et du coup, elles devaient être réindexées par des commandes séparées. C'est toujours possible mais redondant.

Exemples

Reconstruit un index simple :

```
REINDEX INDEX my_index;
```

Recrée les index sur la table `ma_table` :

```
REINDEX TABLE ma_table;
```

Reconstruit tous les index d'une base de données particulière sans faire confiance à la validité des index système :

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Compatibilité

Il n'existe pas de commande **REINDEX** dans le standard SQL.

Nom

RELEASE SAVEPOINT — détruit un point de sauvegarde précédemment défini

Synopsis

```
RELEASE [ SAVEPOINT ] nom_pointsauvegarde
```

Description

RELEASE SAVEPOINT détruit un point de sauvegarde défini précédemment dans la transaction courante.

La destruction d'un point de sauvegarde le rend indisponible comme point de retour. C'est, pour l'utilisateur, le seul comportement visible. Elle ne défait pas les commandes exécutées après l'établissement du point de sauvegarde (pour cela, voir ROLLBACK TO SAVEPOINT(7)). Détruire un point de sauvegarde quand il n'est plus nécessaire peut permettre au système de récupérer certaines ressources sans attendre la fin de la transaction.

RELEASE SAVEPOINT détruit aussi tous les points de sauvegarde créés ultérieurement au point de sauvegarde indiqué.

Paramètres

nom_pointsauvegarde
: Le nom du point de sauvegarde à détruire.

Notes

Spécifier un nom de point de sauvegarde qui n'a pas été défini est une erreur.

Il n'est pas possible de libérer un point de sauvegarde lorsque la transaction est dans un état d'annulation.

Si plusieurs points de transaction ont le même nom, seul le plus récent est libéré.

Exemples

Pour établir puis détruire un point de sauvegarde :

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT mon_pointsauvegarde;
COMMIT;
```

La transaction ci-dessus insère à la fois 3 et 4.

Compatibilité

Cette commande est conforme au standard SQL. Le standard impose le mot clé `SAVEPOINT` mais PostgreSQL™ autorise son omission.

Voir aussi

BEGIN(7), COMMIT(7), ROLLBACK(7), ROLLBACK TO SAVEPOINT(7), SAVEPOINT(7)

Nom

RESET — reinitialise un paramètre d'exécution à sa valeur par défaut

Synopsis

```
RESET paramètre_configuration
RESET ALL
```

Description

RESET réinitialise les paramètres d'exécution à leur valeur par défaut. **RESET** est une alternative à

```
SET paramètre_configuration TO DEFAULT
```

On pourra se référer à SET(7) pour plus de détails.

La valeur par défaut est définie comme la valeur qu'aurait la variable si aucune commande **SET** n'avait modifié sa valeur pour la session en cours. La source effective de cette valeur peut être dans les valeurs par défaut compilées, le fichier de configuration, les options de la ligne de commande ou les paramétrages spécifiques à la base de données ou à l'utilisateur. Ceci est subtilement différent de le définir comme « la valeur qu'a le paramètre au lancement de la session » parce que, si la valeur provenait du fichier de configuration, elle sera annulée par ce qui est spécifié maintenant dans le ficher deconfiguration. Voir Chapitre 18, Configuration du serveur pour les détails.

Le comportement transactionnel de **RESET** est identique à celui de la commande **SET** : son effet sera annulée par une annulation de la transaction.

Paramètres

paramètre_configuration
: Nom d'un paramètre configurable. Les paramètres disponibles sont documentés dans Chapitre 18, Configuration du serveur et sur la page de référence SET(7).

`ALL`
: Réinitialise tous les paramètres configurables à l'exécution.

Exemples

Pour réinitialiser `timezone` :

```
RESET timezone;
```

Compatibilité

RESET est une extension de PostgreSQL™.

Voir aussi

SET(7), SHOW(7)

Nom

REVOKE — supprime les droits d'accès

Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( colonne [, ...] )
    [,...] | ALL [ PRIVILEGES ] ( colonne [, ...] ) }
    ON [ TABLE ] nom_table [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [,...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE nom_séquence [, ...]
         | ALL SEQUENCES IN SCHEMA nom_schéma [, ...] }
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE nom_base [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER nom_fdw [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER nom_serveur [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION nom_fonction ( [ [ mode_arg ] [ nom_arg ] type_arg [, ...] ] ) [,
...]
         | ALL FUNCTIONS IN SCHEMA nom_schéma [, ...] }
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nom_lang [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [,...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
```

```
    { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nom_schéma [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE nom_tablespace [, ...]
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    nom_rôle [, ...] FROM nom_rôle [, ...]
    [ CASCADE | RESTRICT ]
```

Description

La commande **REVOKE** retire des droits précédemment attribués à un ou plusieurs rôles. Le mot clé `PUBLIC` fait référence au groupe implicitement défini de tous les rôles.

Voir la description de la commande GRANT(7) pour connaître la signification des types de droits.

Notez qu'un rôle possède la somme des droits qui lui ont été donnés directement, des droits qui ont été donnés à un rôle dont il est membre et des droits donnés à `PUBLIC`. Du coup, par exemple, retirer les droits de `SELECT` à `PUBLIC` ne veut pas nécessairement dire que plus aucun rôle n'a le droit de faire de `SELECT` sur l'objet : ceux qui en avaient obtenu le droit directement ou via un autre rôle l'ont toujours. De même, révoquer `SELECT` d'un utilisateur ne l'empêchera peut-être pas d'utiliser `SELECT` si `PUBLIC` ou un autre de ses rôle a toujours les droits `SELECT`.

Si `GRANT OPTION FOR` est précisé, seul l'option de transmission de droit (grant option) est supprimée, pas le droit lui même. Sinon, le droit et l'option de transmission de droits sont révoqués.

Si un utilisateur détient un privilège avec le droit de le transmettre, et qu'il l'a transmis à d'autres utilisateurs, alors les droits de ceux-ci sont appelés des droits dépendants. Si les droits ou le droit de transmettre du premier utilisateur sont supprimés, et que des droits dépendants existent, alors ces droits dépendants sont aussi supprimés si l'option `CASCADE` est utilisée. Dans le cas contraire, la suppression de droits est refusée. Cette révocation récursive n'affecte que les droits qui avaient été attribués à travers une chaîne d'utilisateurs traçable jusqu'à l'utilisateur qui subit la commande `REVOKE`. Du coup, les utilisateurs affectés peuvent finalement garder le droit s'il avait aussi été attribué via d'autres utilisateurs.

En cas de révocation des droits sur une table, les droits sur les colonnes correspondantes (s'il y en a) sont automatiquement révoqués pour toutes les colonnes de la table en même temps.

Lors de la révocation de l'appartenance d'un rôle, `GRANT OPTION` est appelé `ADMIN OPTION` mais le comportement est similaire. Notez aussi que cette forme de la commande ne permet pas le mot `GROUP`.

Notes

Utilisez la commande **\dp** de psql(1) pour afficher les droits donnés sur des tables et colonnes. Voir GRANT(7) pour plus d'informations sur le format. Pour les objets qui ne sont pas des tables, il existe d'autres commandes **\d** qui peuvent afficher leurs droits.

Un utilisateur ne peut révoquer que les droits qu'il a donnés directement. Si, par exemple, un utilisateur A a donné un droit et la possibilité de le transmettre à un utilisateur B, et que B à son tour l'a donné à C, alors A ne peut pas retirer directement le droit de C. À la place, il peut supprimer le droit de transmettre à B et utiliser l'option `CASCADE` pour que le droit soit automatiquement supprimé à C. Autre exemple, si A et B ont donné le même droit à C, A peut révoquer son propre don de droit mais pas celui de B, donc C dispose toujours de ce droit.

Lorsqu'un utilisateur, non propriétaire de l'objet, essaie de révoquer (**REVOKE**) des droits sur l'objet, la commande échoue si l'utilisateur n'a aucun droit sur l'objet. Tant que certains droits sont disponibles, la commande s'exécute mais ne sont supprimés que les droits dont l'utilisateur a l'option de transmission. La forme **REVOKE ALL PRIVILEGES** affiche un message d'avertissement si les options de transmissions pour un des droits nommés spécifiquement dans la commande ne sont pas possédés. (En principe, ces instructions s'appliquent aussi au propriétaire de l'objet mais comme le propriétaire est toujours traité comme celui détenant toutes les options de transmission, ces cas n'arrivent jamais.)

Si un superutilisateur choisit d'exécuter une commande **GRANT** ou **REVOKE**, la commande est exécutée comme si elle était lancée par le propriétaire de l'objet affecté. Comme tous les droits proviennent du propriétaire d'un objet (directement ou via une chaîne de transmissions de droits), un superutilisateur peut supprimer tous les droits sur un objet mais cela peut nécessiter l'utilisation de `CASCADE` comme expliqué précédemment.

REVOKE peut aussi être effectué par un rôle qui n'est pas le propriétaire de l'objet affecté mais qui est un membre du rôle qui

possède l'objet ou qui est un membre d'un rôle qui détient les droits `WITH GRANT OPTION` sur cet objet. Dans ce cas, la commande est exécutée comme si elle avait été exécutée par le rôle qui possède réellement l'objet ou détient les droits `WITH GRANT OPTION`. Par exemple, si la table `t1` est possédée par le rôle `g1`, dont le rôle `u1` est membre, alors `u1` peut supprimer des droits sur `t1` qui sont enregistrés comme donnés par `g1`. Ceci incluera les dons de droits effectués par `u1` ainsi que ceux effectués par les autres membres du rôle `g1`.

Si le rôle exécutant **REVOKE** détient les droits indirectement via plus d'un chemin d'appartenance, le rôle indiqué comme ayant effectué la commande est non déterminable à l'avance. Dans de tels cas, il est préférable d'utiliser **SET ROLE** pour devenir le rôle que vous souhaitez voir exécuter la commande **REVOKE**. Ne pas faire cela peut avoir comme résultat de supprimer des droits autres que ceux que vous vouliez, voire même de ne rien supprimer du tout.

Exemples

Enlève au groupe public le droit d'insérer des lignes dans la table `films` :

```
REVOKE INSERT ON films FROM PUBLIC;
```

Supprime tous les droits de l'utilisateur `manuel` sur la vue `genres` :

```
REVOKE ALL PRIVILEGES ON genres FROM manuel;
```

Notez que ceci signifie en fait « révoque tous les droits que j'ai donné ».

Supprime l'appartenance de l'utilisateur `joe` au rôle `admins` :

```
REVOKE admins FROM joe;
```

Compatibilité

La note de compatibilité de la commande GRANT(7) s'applique par analogie à **REVOKE**. Les mots clés `RESTRICT` ou `CASCADE` sont requis d'après le standard, mais PostgreSQL™ utilise `RESTRICT` par défaut.

Voir aussi

GRANT(7)

Nom

ROLLBACK — annule la transaction en cours

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Description

ROLLBACK annule la transaction en cours et toutes les modifications effectuées lors de cette transaction.

Paramètres

`WORK`, `TRANSACTION`
: Mots clés optionnels. Ils sont sans effet.

Notes

L'utilisation de la commande COMMIT(7) permet de terminer une transaction avec succès.

Lancer **ROLLBACK** en dehors de toute transaction n'a pas d'autre conséquence que l'affichage d'un message d'avertissement.

Exemples

Pour annuler toutes les modifications :

```
ROLLBACK;
```

Compatibilité

Le standard SQL spécifie seulement les deux formes `ROLLBACK` et `ROLLBACK WORK`. à part cela, cette commande est totalement compatible.

Voir aussi

BEGIN(7), COMMIT(7), ROLLBACK TO SAVEPOINT(7)

Nom

ROLLBACK PREPARED — annule une transaction précédemment préparée en vue d'une validation en deux phases

Synopsis

```
ROLLBACK PREPARED id_transaction
```

Description

ROLLBACK PREPARED annule une transaction préparée.

Paramètres

id_transaction
: L'identifiant de la transaction à annuler.

Notes

Pour annuler une transaction préparée, il est impératif d'être soit l'utilisateur qui a initié la transaction, soit un superutilisateur. Il n'est, en revanche, pas nécessaire d'être dans la session qui a initié la transaction.

Cette commande ne peut pas être exécutée à l'intérieur d'un bloc de transaction. La transaction préparée est annulée immédiatement.

Toutes les transactions préparées disponibles sont listées dans la vue système pg_prepared_xacts.

Exemples

Annuler la transaction identifiée par `foobar` :

```
ROLLBACK PREPARED 'foobar';
```

Voir aussi

PREPARE TRANSACTION(7), COMMIT PREPARED(7)

Nom

ROLLBACK TO SAVEPOINT — annule les instructions jusqu'au point de sauvegarde

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] nom_pointsauvegarde
```

Description

Annule toutes les commandes qui ont été exécutées après l'établissement du point de sauvegarde. Le point de sauvegarde reste valide. Il est possible d'y d'y revenir encore si cela s'avérait nécessaire.

ROLLBACK TO SAVEPOINT détruit implicitement tous les points de sauvegarde établis après le point de sauvegarde indiqué.

Paramètres

nom_pointsauvegarde
: Le point de sauvegarde où retourner.

Notes

RELEASE SAVEPOINT(7) est utilisé pour détruire un point de sauvegarde sans annuler les effets de commandes exécutées après son établissement.

Spécifier un nom de point de sauvegarde inexistant est une erreur.

Les curseurs ont un comportement quelque peu non transactionnel en ce qui concerne les points de sauvegarde. Tout curseur ouvert à l'intérieur d'un point de sauvegarde est fermé lorsque le point de sauvegarde est rejoint. Si un curseur précédemment ouvert est affecté par une commande **FETCH** ou **MOVE** à l'intérieur d'un point de sauvegarde rejoint par la suite, la position du curseur reste celle obtenue par **FETCH** (c'est-à-dire que le déplacement du curseur dû au **FETCH** n'est pas annulé). La fermeture d'un curseur n'est pas non plus remise en cause par une annulation. Néanmoins, certains effets de bord causés par la requête du curseur (comme les effets de bord des fonctions volatiles appelées par la requête) *sont* annulés s'ils surviennent lors d'un point de sauvegarde qui est annulé plus tard. Un curseur dont l'exécution provoque l'annulation d'une transaction est placé dans un état non exécutable. De ce fait, alors même que la transaction peut être restaurée par **ROLLBACK TO SAVEPOINT**, le curseur ne peut plus être utilisé.

Exemples

Pour annuler les effets des commandes exécutées après l'établissement de `mon_pointsauvegarde` :

```
ROLLBACK TO SAVEPOINT mon_pointsauvegarde;
```

La position d'un curseur n'est pas affectée par l'annulation des points de sauvegarde :

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

Compatibilité

Le standard SQL spécifie que le mot clé `SAVEPOINT` est obligatoire mais PostgreSQL™ et Oracle™ autorisent son omission. SQL n'autorise que `WORK`, pas `TRANSACTION`, après `ROLLBACK`. De plus, SQL dispose d'une clause optionnelle `AND [ NO ] CHAIN` qui n'est actuellement pas supportée par PostgreSQL™. Pour le reste, cette commande est conforme au standard SQL.

Voir aussi

BEGIN(7), COMMIT(7), RELEASE SAVEPOINT(7), ROLLBACK(7), SAVEPOINT(7)

Nom

SAVEPOINT — définit un nouveau point de sauvegarde à l'intérieur de la transaction en cours

Synopsis

```
SAVEPOINT nom_pointsauvegarde
```

Description

SAVEPOINT établit un nouveau point de sauvegarde à l'intérieur de la transaction en cours.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

Paramètres

nom_pointsauvegarde
: Le nom du nouveau point de sauvegarde.

Notes

Utilisez ROLLBACK TO SAVEPOINT(7) pour annuler un point de sauvegarde. Utilisez RELEASE SAVEPOINT(7) pour détruire un point de sauvegarde, conservant l'effet des commandes exécutées après son établissement.

Les points de sauvegarde peuvent seulement être établis à l'intérieur d'un bloc de transaction. Plusieurs points de sauvegarde peuvent être définis dans une transaction.

Exemples

Pour établir un point de sauvegarde et annuler plus tard les effets des commandes exécutées après son établissement :

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

La transaction ci-dessus insère les valeurs 1 et 3, mais pas 2.

Pour établir puis détruire un point de sauvegarde :

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT mon_pointsauvegarde;
COMMIT;
```

La transaction ci-dessus insère à la fois les valeurs 3 et 4.

Compatibilité

SQL requiert la destruction automatique d'un point de sauvegarde quand un autre point de sauvegarde du même nom est créé. Avec PostgreSQL™, l'ancien point de sauvegarde est conservé, mais seul le plus récent est utilisé pour une annulation ou une libération. (Libérer avec **RELEASE SAVEPOINT** le point de sauvegarde le plus récent fait que l'ancien est de nouveau accessible aux commandes **ROLLBACK TO SAVEPOINT** et **RELEASE SAVEPOINT**.) Sinon, **SAVEPOINT** est totalement conforme à SQL.

Voir aussi

BEGIN(7), COMMIT(7), RELEASE SAVEPOINT(7), ROLLBACK(7), ROLLBACK TO SAVEPOINT(7)

Nom

SELECT, TABLE, WITH — récupère des lignes d'une table ou d'une vue

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] nom_d_affichage ] [, ...]
    [ FROM éléments_from [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW nom_window AS ( définition_window ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING opérateur ] [ NULLS { FIRST | LAST } ]
[, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET début ] [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ total ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF nom_table [, ...] ] [ NOWAIT ] [...] ]

avec éléments_from qui peut être :

    [ ONLY ] nom_table [ * ] [ [ AS ] alias [ ( alias_colonne [, ...] ) ] ]
    ( select ) [ AS ] alias [ ( alias_colonne [, ...] ) ]
    nom_requête_with [ [ AS ] alias [ ( alias_colonne [, ...] ) ] ]
    nom_fonction ( [ argument [, ...] ] ) [ AS ] alias [ ( alias_colonne [, ...] |
définition_colonne [, ...] ) ]
    nom_fonction ( [ argument [, ...] ] ) AS ( définition_colonne [, ...] )
    éléments_from [ NATURAL ] type_jointure éléments_from [ ON condition_jointure |
USING ( colonne_jointure [, ...] ) ]

et requête_with est :

    nom_requête_with [ ( nom_colonne [, ...] ) ] AS ( select )

TABLE { [ ONLY ] nom_table [ * ] | nom_requête_with }
```

Description

SELECT récupère des lignes de zéro ou plusieurs tables. Le traitement général de **SELECT** est le suivant :

1. Toutes les requêtes dans la liste `WITH` sont évaluées. Elles jouent le rôle de tables temporaires qui peuvent être référencées dans la liste `FROM`. Une requête `WITH` qui est référencée plus d'une fois dans `FROM` n'est calculée qu'une fois (voir la section intitulée « Clause `WITH` » ci-dessous).
2. Tous les éléments de la liste `FROM` sont calculés. (Chaque élément dans la liste `FROM` est une table réelle ou virtuelle.) Si plus d'un élément sont spécifiés dans la liste `FROM`, ils font l'objet d'une jointure croisée (cross-join). (Voir la section intitulée « Clause `FROM` » ci-dessous.)
3. Si la clause `WHERE` est spécifiée, toutes les lignes qui ne satisfont pas les conditions sont éliminées de l'affichage. (Voir la section intitulée « Clause `WHERE` » ci-dessous.)
4. Si la clause `GROUP BY` est spécifiée, l'affichage est divisé en groupes de lignes qui correspondent à une ou plusieurs valeurs. Si la clause `HAVING` est présente, elle élimine les groupes qui ne satisfont pas la condition donnée. (Voir la section intitulée « Clause `GROUP BY` » et la section intitulée « Clause `HAVING` » ci-dessous.)
5. Les lignes retournées sont traitées en utilisant les expressions de sortie de **SELECT** pour chaque ligne sélectionnée. (Voir la section intitulée « Liste **SELECT** » ci-dessous.)
6. En utilisant les opérateurs `UNION`, `INTERSECT` et `EXCEPT`, l'affichage de plusieurs instructions **SELECT** peut être combiné pour former un ensemble unique de résultats. L'opérateur `UNION` renvoie toutes les lignes qui appartiennent, au moins, à l'un des ensembles de résultats. L'opérateur `INTERSECT` renvoie toutes les lignes qui sont dans tous les ensembles de résultats. L'opérateur `EXCEPT` renvoie les lignes qui sont présentes dans le premier ensemble de résultats mais pas dans le deuxième. Dans les trois cas, les lignes dupliquées sont éliminées sauf si `ALL` est spécifié. (Voir la section intitulée « Clause `UNION` », la section intitulée « Clause `INTERSECT` » et la section intitulée « Clause `EXCEPT` » ci-dessous.)

7. Si la clause `ORDER BY` est spécifiée, les lignes renvoyées sont triées dans l'ordre spécifié. Si `ORDER BY` n'est pas indiqué, les lignes sont retournées dans l'ordre qui permet la réponse la plus rapide du système. (Voir la section intitulée « Clause `ORDER BY` » ci-dessous.)
8. `DISTINCT` élimine les lignes dupliquées du résultat. `DISTINCT ON` élimine les lignes qui correspondent à toutes les expressions données. `ALL` (la valeur par défaut) renvoie toutes les lignes candidates, y compris les lignes dupliquées. (Voir la section intitulée « Clause `DISTINCT` » ci-dessous.)
9. Si les clauses `LIMIT` (ou `FETCH FIRST`) ou `OFFSET` sont spécifiées, l'instruction **SELECT** ne renvoie qu'un sous-ensemble de lignes de résultats. (Voir la section intitulée « Clause `LIMIT` » ci-dessous.)
10. Si la clause `FOR UPDATE` ou `FOR SHARE` est spécifiée, l'instruction **SELECT** verrouille les lignes sélectionnées contre les mises à jour concurrentes. (Voir la section intitulée « Clause `FOR UPDATE`/`FOR SHARE` » ci-dessous.)

Le droit `SELECT` sur chaque colonne utilisée dans une commande **SELECT** est nécessaire pour lire ses valeurs. L'utilisation de `FOR UPDATE` ou de `FOR SHARE` requiert en plus le droit `UPDATE` (pour au moins une colonne de chaque table sélectionnée).

Paramètres

Clause `WITH`

La clause `WITH` vous permet de spécifier une ou plusieurs sous-requêtes qui peuvent être utilisées par leur nom dans la requête principale. Les sous-requêtes se comportent comme des tables temporaires ou des vues pendant la durée d'exécution de la requête principale.

Un nom (sans qualification de schéma) doit être spécifié pour chaque requête `WITH`. Optionnellement, une liste de noms de colonnes peut être spécifié ; si elle est omise, les noms de colonnes sont déduites de la sous-requête.

Si `RECURSIVE` est spécifié, la sous-requête peut se référencer elle même. Une sous-requête de ce type doit avoir la forme

```
terme_non_récursif UNION [ ALL ] terme_récursif
```

où l'auto-référence récursive doit apparaître dans la partie droite de l'`UNION`. Seule une auto-référence récursive est autorisée par requête.

Un autre effet de `RECURSIVE` est que les requêtes `WITH` n'ont pas besoin d'être ordonnées : une requête peut en référencer une autre qui se trouve plus loin dans la liste (toutefois, les références circulaires, ou récursion mutuelle, ne sont pas implémentées). Sans `RECURSIVE`, les requêtes `WITH` ne peuvent référencer d'autres requêtes `WITH` soœurs que si elles sont déclarées avant dans la liste `WITH`.

Une propriété utile des requêtes `WITH` est qu'elles ne sont évaluées qu'une seule fois par exécution de la requête principale, même si la requête principale les utilise plus d'une fois.

Voir Section 7.8, « Requêtes `WITH` (*Common Table Expressions*) » pour plus d'informations.

Clause `FROM`

La clause `FROM` spécifie une ou plusieurs tables source pour le **SELECT**. Si plusieurs sources sont spécifiées, le résultat est un produit cartésien (jointure croisée) de toutes les sources. Mais habituellement, des conditions de qualification sont ajoutées pour restreindre les lignes renvoyées à un petit sous-ensemble du produit cartésien.

La clause `FROM` peut contenir les éléments suivants :

`nom_table`
: Le nom (éventuellement qualifié par le nom du schéma) d'une table ou vue existante. Si `ONLY` est spécifié avant le nom de la table, seule cette table est parcourue. Dans le cas contraire, la table et toutes ses tables filles (s'il y en a) sont parcourues. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles sont inclues.

`alias`
: Un nom de substitution pour l'élément `FROM` contenant l'alias. Un alias est utilisé par brièveté ou pour lever toute ambiguïté lors d'auto-jointures (la même table est parcourue plusieurs fois). Quand un alias est fourni, il cache complètement le nom réel de la table ou fonction ; par exemple, avec `FROM foo AS`, le reste du **SELECT** doit faire référence à cet élément de `FROM` par `f` et non pas par `foo`. Si un alias est donné, une liste d'alias de colonnes peut aussi être saisi comme noms de substitution pour différentes colonnes de la table.

`select`
: Un sous-**SELECT** peut apparaître dans la clause `FROM`. Il agit comme si sa sortie était transformée en table temporaire pour la durée de cette seule commande **SELECT**. Le sous-**SELECT** doit être entouré de parenthèses et un alias *doit* lui être fourni.

Une commande VALUES(7) peut aussi être utilisée ici.

nom_requête_with
Une requête WITH est référencée par l'écriture de son nom, exactement comme si le nom de la requête était un nom de table (en fait, la requête WITH cache toutes les tables qui auraient le même nom dans la requête principale. Si nécessaire, vous pouvez accéder à une table réelle du même nom en précisant le schéma du nom de la table). Un alias peut être indiqué de la même façon que pour une table.

nom_fonction
Des appels de fonctions peuvent apparaître dans la clause FROM. (Cela est particulièrement utile pour les fonctions renvoyant des ensembles de résultats, mais n'importe quelle fonction peut être utilisée.) Un appel de fonction agit comme si la sortie était transformée en table temporaire pour la durée de cette seule commande **SELECT**. Un alias peut aussi être utilisé. Si un alias est donné, une liste d'alias de colonnes peut être ajoutée pour fournir des noms de substitution pour un ou plusieurs attributs du type composé de retour de la fonction. Si la fonction a été définie comme renvoyant le type de données record, alors un alias ou un mot clé AS doit être présent, suivi par une liste de définitions de colonnes de la forme (*nom_colonne type_données* [, ...]). La liste de définitions de colonnes doit correspondre au nombre réel et aux types réels des colonnes renvoyées par la fonction.

type_jointure
Un des éléments

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

Pour les types de jointures INNER et OUTER, une condition de jointure doit être spécifiée, à choisir parmi NATURAL, ON *condition_jointure* ou USING (*colonne_jointure* [, ...]). Voir ci-dessous pour la signification. Pour CROSS JOIN, aucune de ces clauses ne doit apparaître.

Une clause JOIN combine deux éléments FROM. Les parenthèses peuvent être utilisées pour déterminer l'ordre d'imbrication. En l'absence de parenthèses, les JOIN sont imbriqués de gauche à droite. Dans tous les cas, JOIN est plus prioritaire que les virgules séparant les éléments FROM.

CROSS JOIN et INNER JOIN produisent un simple produit cartésien. Le résultat est identique à celui obtenu lorsque les deux éléments sont listés au premier niveau du FROM, mais restreint par la condition de jointure (si elle existe). CROSS JOIN est équivalent à INNER JOIN ON (TRUE), c'est-à-dire qu'aucune ligne n'est supprimée par qualification. Ces types de jointure sont essentiellement une aide à la notation car ils ne font rien de plus qu'un simple FROM et WHERE.

LEFT OUTER JOIN renvoie toutes les lignes du produit cartésien qualifié (c'est-à-dire toutes les lignes combinées qui satisfont la condition de jointure), plus une copie de chaque ligne de la table de gauche pour laquelle il n'y a pas de ligne à droite qui satisfasse la condition de jointure. La ligne de gauche est étendue à la largeur complète de la table jointe par insertion de valeurs NULL pour les colonnes de droite. Seule la condition de la clause JOIN est utilisée pour décider des lignes qui correspondent. Les conditions externes sont appliquées après coup.

À l'inverse, RIGHT OUTER JOIN renvoie toutes les lignes jointes plus une ligne pour chaque ligne de droite sans correspondance (complétée par des NULL pour le côté gauche). C'est une simple aide à la notation car il est aisément convertible en LEFT en inversant les entrées gauche et droite.

FULL OUTER JOIN renvoie toutes les lignes jointes, plus chaque ligne gauche sans correspondance (étendue par des NULL à droite), plus chaque ligne droite sans correspondance (étendue par des NULL à gauche).

ON *condition_jointure*
condition_jointure est une expression qui retourne une valeur de type boolean (comme une clause WHERE) qui spécifie les lignes d'une jointure devant correspondre.

USING (*colonne_jointure* [, ...])
Une clause de la forme USING (a, b, ...) est un raccourci pour ON table_gauche.a = table_droite.a AND table_gauche.b = table_droite.b .... De plus, USING implique l'affichage d'une seule paire des colonnes correspondantes dans la sortie de la jointure.

NATURAL
NATURAL est un raccourci pour une liste USING qui mentionne toutes les colonnes de même nom dans les deux tables.

Clause WHERE

La clause WHERE optionnelle a la forme générale

```
WHERE condition
```

où *condition* est une expression dont le résultat est de type boolean. Toute ligne qui ne satisfait pas cette condition est éliminée de la sortie. Une ligne satisfait la condition si elle retourne vrai quand les valeurs réelles de la ligne sont substituées à toute référence de variable.

Clause GROUP BY

La clause GROUP BY optionnelle a la forme générale

```
GROUP BY expression [, ...]
```

GROUP BY condense en une seule ligne toutes les lignes sélectionnées qui partagent les mêmes valeurs pour les expressions regroupées. *expression* peut être le nom d'une colonne en entrée, le nom ou le numéro d'une colonne en sortie (élément de la liste **SELECT**), ou une expression quelconque formée de valeurs de colonnes en entrée. En cas d'ambiguïté, un nom de GROUP BY est interprété comme un nom de colonne en entrée, non en sortie.

Les fonctions d'agrégat, si utilisées, sont calculées pour toutes les lignes composant un groupe, produisant une valeur séparée pour chaque groupe (alors que sans GROUP BY, un agrégat produit une valeur unique calculée pour toutes les lignes sélectionnées). Quand GROUP BY est présent, les expressions du **SELECT** ne peuvent faire référence qu'à des colonnes groupées, sauf à l'intérieur de fonctions d'agrégat, la valeur de retour d'une colonne non-groupée n'étant pas unique.

Clause HAVING

La clause optionnelle HAVING a la forme générale

```
HAVING condition
```

où *condition* est identique à celle spécifiée pour la clause WHERE.

HAVING élimine les lignes groupées qui ne satisfont pas à la condition. HAVING est différent de WHERE : WHERE filtre les lignes individuelles avant l'application de GROUP BY alors que HAVING filtre les lignes groupées créées par GROUP BY. Chaque colonne référencée dans *condition* doit faire référence sans ambiguïté à une colonne groupée, sauf si la référence apparaît dans une fonction d'agrégat.

Même en l'absence de clause GROUP BY, la présence de HAVING transforme une requête en requête groupée. Cela correspond au comportement d'une requête contenant des fonctions d'agrégats mais pas de clause GROUP BY. Les lignes sélectionnées ne forment qu'un groupe, la liste du **SELECT** et la clause HAVING ne peuvent donc faire référence qu'à des colonnes à l'intérieur de fonctions d'agrégats. Une telle requête ne produira qu'une seule ligne si la condition HAVING est réalisée, aucune dans le cas contraire.

Clause WINDOW

La clause optionnelle WINDOW a la forme générale

```
WINDOW nom_window AS ( définition_window ) [, ...]
```

où *nom_window* est un nom qui peut être référencé par des clauses OVER ou des définitions de window, et *définition_window* est

```
[ nom_window_existante ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operateur ] [ NULLS { FIRST | LAST } ] [,
...] ]
[ clause_frame ]
```

Si un *nom_window_existante* est spécifié, il doit se référer à une entrée précédente dans la liste WINDOW ; la nouvelle Window copie sa clause de partitionnement de cette entrée, ainsi que sa clause de tri s'il y en a. Dans ce cas, la nouvelle Window ne peut pas spécifier sa propre clause PARTITION BY, et ne peut spécifier de ORDER BY que si la Window copiée n'en a pas. La nouvelle Window utilise toujours sa propre clause frame ; la Window copiée ne doit pas posséder de clause frame.

Les éléments de la liste PARTITION BY sont interprétés à peu près de la même façon que des éléments de la section intitulée « Clause GROUP BY », sauf qu'ils sont toujours des expressions simples et jamais le nom ou le numéro d'une colonne en sortie. Une autre différence est que ces expressions peuvent contenir des appels à des fonctions d'aggrégat, ce qui n'est pas autorisé dans une clause GROUP BY classique. Ceci est autorisé ici parce que le windowing se produit après le regroupement et l'aggrégation.

De façon similaire, les éléments de la liste ORDER BY sont interprétés à peu près de la même façon que les éléments d'un la section intitulée « Clause ORDER BY », sauf que les expressions sont toujours prises comme de simples expressions et jamais comme le nom ou le numéro d'une colonne en sortie.

La clause *clause_frame* optionnelle définit la *frame window* pour les fonctions window qui dépendent de la frame (ce n'est pas le cas de toutes). La frame window est un ensemble de lignes liées à chaque ligne de la requête (appelée la *ligne courante*). La *clause_frame* peut être une des clauses suivantes :

```
[ RANGE | ROWS ] début_frame
[ RANGE | ROWS ] BETWEEN début_frame AND fin_frame
```

où *début_frame* et *fin_frame* peuvent valoir

```
UNBOUNDED PRECEDING
valeur PRECEDING
CURRENT ROW
valeur FOLLOWING
UNBOUNDED FOLLOWING
```

Si *fin_frame* n'est pas précisé, il vaut par défaut CURRENT ROW. Les restrictions sont les suivantes : *début_frame* ne peut pas valoir UNBOUNDED FOLLOWING, *fin_frame* ne peut pas valoir UNBOUNDED PRECEDING, et le choix *fin_frame* ne peut apparaître avant le choix *début_frame* -- par exemple RANGE BETWEEN CURRENT ROW AND *valeur* PRECEDING n'est pas permis.

L'option par défaut pour la clause frame est RANGE UNBOUNDED PRECEDING, ce qui revient au même que RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ; il positionne la frame pour qu'il couvre toutes les lignes à partir du début de la partition jusqu'à la dernière ligne à égalité avec la ligne courante dans l'ordre défini par l' ORDER BY (ce qui signifie toutes les lignes s'il n'y a pas d' ORDER BY). Généralement, UNBOUNDED PRECEDING signifie que la frame commence à la première ligne de la partition, et de même UNBOUNDED FOLLOWING signifie que la frame se termine avec la dernière ligne de la partition (quel que soit le mode, RANGE ou bien ROWS). Dans le mode ROWS, CURRENT ROW signifie que la frame commence ou se termine sur la ligne courante ; mais dans le mode RANGE cela signifie que la frame débute ou se termine sur la première ou la dernière des lignes à égalité avec la ligne courante dans l'ordre de la clause ORDER BY. Les *valeur* PRECEDING et *valeur* FOLLOWING sont actuellement seulement permis en mode ROWS. Ils indiquent que la frame débute ou se termine autant de lignes avant ou après la ligne courante. *valeur* doit être une expression entière, ne contenant aucune variable, fonction d'aggrégat ni fonction window. La valeur ne doit être ni null ni négative ; mais elle peut être de zéro, ce qui sélectionne la ligne courante elle-même.

Attention, les options ROWS peuvent produire des résultats imprévisibles si l'ordre défini par l' ORDER BY n'ordonne pas les lignes de manière unique. Les options RANGE sont conçues pour s'assurer que les lignes qui sont à égalité suivant l'ordre de l' ORDER BY sont traitées de la même manière ; toutes les lignes à égalité seront ensemble dans la frame ou ensemble hors de la frame.

L'utilité d'une clause WINDOW est de spécifier le comportement des *fonctions window* apparaissant dans la clause la section intitulée « Liste **SELECT** » ou la clause la section intitulée « Clause ORDER BY » de la requête. Ces fonctions peuvent référencer les entrées de clauses WINDOW par nom dans leurs clauses OVER. Toutefois, il n'est pas obligatoire qu'une entrée de clause WINDOW soit référencée quelque part ; si elle n'est pas utilisée dans la requête, elle est simplement ignorée. Il est possible d'utiliser des fonctions window sans aucune clause WINDOW puisqu'une fonction window peut spécifier sa propre définition de window directement dans sa clause OVER. Toutefois, la clause WINDOW économise de la saisie quand la même définition window est utilisée pour plus d'une fonction window.

Les fonctions window sont décrites en détail dans Section 3.5, « Fonctions de fenêtrage », Section 4.2.8, « Appels de fonction de fenêtrage » et Section 7.2.4, « Traitement de fonctions Window ».

Liste SELECT

La liste **SELECT** (entre les mots clés SELECT et FROM) spécifie les expressions qui forment les lignes en sortie de l'instruction **SELECT**. Il se peut que les expressions fassent référence aux colonnes traitées dans la clause FROM. En fait, en général, elles le font.

Comme pour une table, chaque colonne de sortie d'un **SELECT** a un nom. Dans un **SELECT** simple, ce nom est juste utilisé pour donner un titre à la colonne pour l'affichage, mais quand le **SELECT** est une sous-requête d'une requête plus grande, le nom est vu par la grande requête comme le nom de colonne de la table virtuelle produite par la sous-requête. Pour indiquer le nom à utiliser pour une colonne de sortie, écrivez AS *nom_de_sortie* après l'expression de la colonne. (Vous pouvez omettre AS seulement si le nom de colonne souhaité n'est pas un mot clé réservé par PostgreSQL™ (voir Annexe C, Mots-clé SQL). Pour vous protéger contre l'ajout futur d'un mot clé, il est recommandé que vous écriviez toujours AS ou que vous mettiez le nom de sortie entre guillemets. Si vous n'indiquez pas de nom de colonne, un nom est choisi automatiquement par PostgreSQL™. Si l'expression

de la colonne est une simple référence à une colonne alors le nom choisi est le même que le nom de la colonne ; dans des cas plus complexes, un nom généré qui ressemblera à `?colonneN?` est habituellement choisi.

Un nom de colonne de sortie peut être utilisé pour se référer à la valeur de la colonne dans les clauses `ORDER BY` et `GROUP BY`, mais pas dans la clause`WHERE` ou `HAVING` ; à cet endroit, vous devez écrire l'expression.

`*` peut être utilisé, à la place d'une expression, dans la liste de sortie comme raccourci pour toutes les colonnes des lignes sélectionnées. De plus, *`nom_table`*`.*` peut être écrit comme raccourci pour toutes les colonnes de cette table. Dans ces cas, il est impossible de spécifier de nouveaux noms avec `AS` ; les noms des colonnes de sorties seront les même que ceux de la table.

Clause `UNION`

La clause `UNION` a la forme générale :

```
instruction_select UNION [ ALL ] instruction_select
```

`instruction_select` est une instruction **SELECT** sans clause `ORDER BY`, `LIMIT`, `FOR SHARE` ou `FOR UPDATE`. (`ORDER BY` et `LIMIT` peuvent être attachés à une sous-expression si elle est entourée de parenthèses. Sans parenthèses, ces clauses s'appliquent au résultat de l'`UNION`, non à l'expression à sa droite.)

L'opérateur `UNION` calcule l'union ensembliste des lignes renvoyées par les instructions **SELECT** impliquées. Une ligne est dans l'union de deux ensembles de résultats si elle apparaît dans au moins un des ensembles. Les deux instructions **SELECT** qui représentent les opérandes directes de l'`UNION` doivent produire le même nombre de colonnes et les colonnes correspondantes doivent être d'un type de données compatible.

Sauf lorsque l'option `ALL` est spécifiée, il n'y a pas de doublons dans le résultat de `UNION`. `ALL` empêche l'élimination des lignes dupliquées. `UNION ALL` est donc significativement plus rapide qu'`UNION`, et sera préféré.

Si une instruction **SELECT** contient plusieurs opérateurs `UNION`, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent.

Actuellement, `FOR UPDATE` et `FOR SHARE` ne peuvent pas être spécifiés pour un résultat d'`UNION` ou pour toute entrée d'un `UNION`.

Clause `INTERSECT`

La clause `INTERSECT` a la forme générale :

```
instruction_select INTERSECT [ ALL ] instruction_select
```

`instruction_select` est une instruction **SELECT** sans clause `ORDER BY`, `LIMIT`, `FOR UPDATE` ou `FOR SHARE`.

L'opérateur `INTERSECT` calcule l'intersection des lignes renvoyées par les instructions **SELECT** impliquées. Une ligne est dans l'intersection des deux ensembles de résultats si elle apparaît dans chacun des deux ensembles.

Le résultat d'`INTERSECT` ne contient aucune ligne dupliquée sauf si l'option `ALL` est spécifiée. Dans ce cas, une ligne dupliquée *m* fois dans la table gauche et *n* fois dans la table droite apparaît min(*m*,*n*) fois dans l'ensemble de résultats.

Si une instruction **SELECT** contient plusieurs opérateurs `INTERSECT`, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent. `INTERSECT` a une priorité supérieur à celle d'`UNION`. C'est-à-dire que `A UNION B INTERSECT C` est lu comme `A UNION (B INTERSECT C)`.

Actuellement, `FOR UPDATE` et `FOR SHARE` ne peuvent pas être spécifiés pour un résultat d'`INTERSECT` ou pour une entrée d'`INTERSECT`.

Clause `EXCEPT`

La clause `EXCEPT` a la forme générale :

```
instruction_select EXCEPT [ ALL ] instruction_select
```

`instruction_select` est une instruction **SELECT** sans clause `ORDER BY`, `LIMIT`, `FOR UPDATE` ou `FOR SHARE`.

L'opérateur `EXCEPT` calcule l'ensemble de lignes qui appartiennent au résultat de l'instruction **SELECT** de gauche mais pas à celui de droite.

Le résultat d'`EXCEPT` ne contient aucune ligne dupliquée sauf si l'option `ALL` est spécifiée. Dans ce cas, une ligne dupliquée *m* fois dans la table gauche et *n* fois dans la table droite apparaît max(*m*-*n*,0) fois dans l'ensemble de résultats.

Si une instruction **SELECT** contient plusieurs opérateurs `EXCEPT`, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent. `EXCEPT` a la même priorité qu'`UNION`.

Actuellement, `FOR UPDATE` et `FOR SHARE` ne peuvent pas être spécifiés dans un résultat `EXCEPT` ou pour une entrée d'un `EX-`

CEPT.

Clause ORDER BY

La clause optionnelle ORDER BY a la forme générale :

```
ORDER BY expression [ ASC | DESC | USING opérateur ] [ NULLS { FIRST | LAST } ] [, ...]
```

La clause ORDER BY impose le tri des lignes de résultat suivant les expressions spécifiées. Si deux lignes sont identiques suivant l'expression la plus à gauche, elles sont comparées avec l'expression suivante et ainsi de suite. Si elles sont identiques pour toutes les expressions de tri, elles sont renvoyées dans un ordre dépendant de l'implantation.

Chaque *expression* peut être le nom ou le numéro ordinal d'une colonne en sortie (élément de la liste **SELECT**). Elle peut aussi être une expression arbitraire formée à partir de valeurs des colonnes.

Le numéro ordinal fait référence à la position ordinale (de gauche à droite) de la colonne de résultat. Cette fonctionnalité permet de définir un ordre sur la base d'une colonne dont le nom n'est pas unique. Ce n'est pas particulièrement nécessaire parce qu'il est toujours possible d'affecter un nom à une colonne de résultat avec la clause AS.

Il est aussi possible d'utiliser des expressions quelconques dans la clause ORDER BY, ce qui inclut des colonnes qui n'apparaissent pas dans la liste résultat du **SELECT**. Ainsi, l'instruction suivante est valide :

```
SELECT nom FROM distributeurs ORDER BY code;
```

Il y a toutefois une limitation à cette fonctionnalité. La clause ORDER BY qui s'applique au résultat d'une clause UNION, INTERSECT ou EXCEPT ne peut spécifier qu'un nom ou numéro de colonne en sortie, pas une expression.

Si une expression ORDER BY est un nom qui correspond à la fois à celui d'une colonne résultat et à celui d'une colonne en entrée, ORDER BY l'interprète comme le nom de la colonne résultat. Ce comportement est à l'opposé de celui de GROUP BY dans la même situation. Cette incohérence est imposée par la compatibilité avec le standard SQL.

Un mot clé ASC (ascendant) ou DESC (descendant) peut être ajouté après toute expression de la clause ORDER BY. ASC est la valeur utilisée par défaut. Un nom d'opérateur d'ordre spécifique peut également être fourni dans la clause USING. Un opérateur de tri doit être un membre plus-petit-que ou plus-grand-que de certaines familles d'opérateur B-tree. ASC est habituellement équivalent à USING < et DESC à USING >. Le créateur d'un type de données utilisateur peut définir à sa guise le tri par défaut qui peut alors correspondre à des opérateurs de nom différent.

Si NULLS LAST est indiqué, les valeurs NULL sont listées après toutes les valeurs non NULL si NULLS FIRST est indiqué, les valeurs NULL apparaissent avant toutes les valeurs non NULL. Si aucune des deux n'est présente, le comportement par défaut est NULLS LAST quand ASC est utilisé (de façon explicite ou non) et NULLS FIRST quand DESC est utilisé (donc la valeur par défaut est d'agir comme si les NULL étaient plus grands que les non NULL). Quand USING est indiqué, le tri des NULL par défaut dépend du fait que l'opérateur est un plus-petit-que ou un plus-grand-que.

Notez que les options de tri s'appliquent seulement à l'expression qu'elles suivent. Par exemple, ORDER BY x, y DESC ne signifie pas la même chose que ORDER BY x DESC, y DESC.

Les données de chaînes de caractères sont triées suivant l'ordre spécifique à la locale, ordre établi au moment de la création de la base de données.

Clause DISTINCT

Si DISTINCT est spécifié, toutes les lignes dupliquées sont supprimées de l'ensemble de résultats (une ligne est conservée pour chaque groupe de lignes dupliquées). ALL spécifie le contraire : toutes les lignes sont conservées ; c'est la valeur par défaut.

DISTINCT ON (*expression* [, ...]) ne conserve que la première ligne de chaque ensemble pour lesquels il y a identité des expressions. Les expressions DISTINCT ON sont interprétées en utilisant les mêmes règles que pour ORDER BY (voir ci-dessus). La « première ligne » de chaque ensemble n'est pas prévisible sauf si ORDER BY est utilisé. Par exemple :

```
SELECT DISTINCT ON (emplacement) emplacement, heure, rapport
FROM rapports_meteo
ORDER BY emplacement, heure DESC;
```

récupère le rapport météo le plus récent de chaque emplacement. Si ORDER BY n'est pas utilisé pour forcer l'ordre descendant des valeurs heure de chaque emplacement, le rapport est ordonné suivant des temps aléatoires.

Le(s) expression(s) DISTINCT ON doi(ven)t correspondre à l'ordre des expression ORDER BY. La clause ORDER BY contient des expressions supplémentaires qui déterminent la précédence désirée des lignes à l'intérieur de chaque groupe DISTINCT ON.

Clause LIMIT

La clause LIMIT est constituée de deux sous-clauses indépendantes :

```
LIMIT { nombre | ALL }
OFFSET début
```

nombre spécifie le nombre maximum de lignes à renvoyer alors que *début* spécifie le nombre de lignes à passer avant de commencer à renvoyer des lignes. Lorsque les deux clauses sont spécifiées, *début* lignes sont passées avant de commencer à compter les *nombre* lignes à renvoyer.

Si l'expression de *compte* est évaluée à NULL, il est traité comme `LIMIT ALL`, c'est-à-dire sans limite. Si *début* est évalué à NULL, il est traité comme `OFFSET 0`.

SQL:2008 a introduit une sytaxe différente pour faire la même chose. PostgreSQL™ supporte aussi cette syntaxe.

```
OFFSET début { ROW | ROWS }
FETCH { FIRST | NEXT } [ compte ] { ROW | ROWS } ONLY
```

Avec cette syntaxe, pour écrire tout sauf une simple constant de type entier pour *début* ou *compte*, vous devez l'entourer de parenthèses. Si *compte* est omis dans une clause `FETCH`, il vaut 1 par défaut. `ROW` et `ROWS` ainsi que `FIRST` et `NEXT` sont des mots qui n'influencent pas les effets de ces clauses. D'après le standard, la clause `OFFSET` doit venir avant la clause `FETCH` si les deux sont présentes ; PostgreSQL™ est plus laxiste et autorise un ordre différent.

Avec `LIMIT`, utiliser la clause `ORDER BY` permet de contraindre l'ordre des lignes de résultat. Dans le cas contraire, le sous-ensemble obtenu n'est pas prévisible -- rien ne permet de savoir à quel ordre correspondent les lignes retournées. Celui-ci ne sera pas connu tant qu'`ORDER BY` n'aura pas été précisé.

Lors de la génération d'un plan de requête, le planificateur tient compte de `LIMIT`. Le risque est donc grand d'obtenir des plans qui diffèrent (ordres des lignes différents) suivant les valeurs utilisées pour `LIMIT` et `OFFSET`. Ainsi, sélectionner des sous-ensembles différents d'un résultat à partir de valeurs différentes de `LIMIT`/`OFFSET` *aboutit à des résultats incohérents* à moins d'avoir figé l'ordre des lignes à l'aide de la clause `ORDER BY`. Ce n'est pas un bogue, mais une conséquence du fait que SQL n'assure pas l'ordre de présentation des résultats sans utilisation d'une clause `ORDER BY`.

Il est même possible pour des exécutions répétées de la même requête `LIMIT` de renvoyer différents sous-ensembles des lignes d'une table s'il n'y a pas de clause `ORDER BY` pour forcer la sélection d'un sous-ensemble déterministe. Encore une fois, ce n'est pas un bogue ; le déterminisme des résultats n'est tout simplement pas garanti dans un tel cas.

Clause `FOR UPDATE`/`FOR SHARE`

La clause `FOR UPDATE` a la forme :

```
FOR UPDATE [ OF nom_table [, ...] ] [ NOWAIT ]
```

La clause liée, `FOR SHARE`, a la forme :

```
FOR SHARE [ OF nom_table [, ...] ] [ NOWAIT ]
```

`FOR UPDATE` verrouille pour modification les lignes récupérées par l'instruction **SELECT**. Cela les empêche d'être modifiées ou supprimées par les autres transactions jusqu'à la fin de la transaction en cours. Les autres transactions qui tentent des **UPDATE**, **DELETE** ou **SELECT FOR UPDATE** sur ces lignes sont bloquées jusqu'à la fin de la transaction courante. De plus, si un **UPDATE**, **DELETE** ou **SELECT FOR UPDATE** a déjà verrouillé une ligne ou un ensemble de lignes à partir d'une autre transaction, **SELECT FOR UPDATE** attend la fin de l'autre transaction puis verrouille et renvoie la ligne modifiée (ou aucune ligne si elle a été supprimée). Cependant, au sein d'une transaction `SERIALIZABLE`, une erreur est levée si une ligne à verrouiller à changé depuis le début de la transaction. Pour plus d'informations, voir Chapitre 13, Contrôle d'accès simultané.

`FOR SHARE` a un comportement similaire. La différence se situe dans le type de verrou acquis. Contrairement à `FOR UPDATE` qui pose un verrou exclusif, `FOR SHARE` pose un verrou partagé sur chaque ligne récupérée. Un verrou partagée bloque les instructions **UPDATE**, **DELETE** ou **SELECT FOR UPDATE** des transaction concurrentes accédant à ces lignes, mais il n'interdit pas les **SELECT FOR SHARE**.

Pour éviter à l'opération d'attendre la validation des autres transactions, on utilise l'option `NOWAIT`. **SELECT FOR UPDATE NOWAIT** rapporte une erreur si une ligne sélectionnée ne peut pas être verrouillée immédiatement. Il n'y a pas d'attente. `NOWAIT` s'applique seulement au(x) verrou(x) niveau ligne -- le verrou niveau table `ROW SHARE` est toujours pris de façon ordinaire (voir Chapitre 13, Contrôle d'accès simultané). L'option `NOWAIT` de LOCK(7) peut toujours être utilisée pour acquérir le verrou niveau table sans attendre.

Si des tables particulières sont nommées dans les clauses `FOR UPDATE` et `FOR SHARE`, alors seules les lignes provenant de ces tables sont verrouillées ; toute autre table utilisée dans le **SELECT** est simplement lue. Une clause `FOR UPDATE` ou `FOR SHARE` sans liste de tables affecte toute les tables utilisées dans l'instruction. Si `FOR UPDATE` ou `FOR SHARE` est appliquée à une vue ou à une sous-requête, cela affecte toutes les tables utilisées dans la vue ou la sous-requête. Néanmoins, `FOR UPDATE`/`FOR SHARE` ne s'appliquent pas aux requêtes `WITH` référencées par la clé primaire. Si vous voulez qu'un verrouillage

de lignes intervienne dans une requête `WITH`, spécifiez `FOR UPDATE` ou `FOR SHARE` à l'intérieur de la requête `WITH`.

Plusieurs clauses `FOR UPDATE` et `FOR SHARE` peuvent être données si il est nécessaire de spécifier différents comportements de verrouillage pour différentes tables. Si la même table est mentionné (ou affectée implicitement) par les clauses `FOR UPDATE` et `FOR SHARE`, alors elle est traitée comme un simple `FOR UPDATE`. De façon similaire, une table est traitée avec `NOWAIT` si c'est spécifiée sur au moins une des clauses qui l'affectent.

`FOR UPDATE` et `FOR SHARE` nécessitent que chaque ligne retournée soit clairement identifiable par une ligne individuelle d'une table ; ces options ne peuvent, par exemple, pas être utilisées avec des fonctions d'aggrégats.

Quand `FOR UPDATE` ou `FOR SHARE` apparaissent au niveau le plus élevé d'une requête **SELECT**, les lignes verrouillées sont exactement celles qui sont renvoyées par la requête ; dans le cas d'une requête avec jointure, les lignes verrouillées sont celles qui contribuent aux lignes jointes renvoyées. De plus, les lignes qui ont satisfait aux conditions de la requête au moment de la prise de son instantané sont verrouillées, bien qu'elles ne seront pas retournées si elles ont été modifiées après la prise du snapshot et ne satisfont plus les conditions de la requête. Si `LIMIT` est utilisé, le verrouillage cesse une fois que suffisamment de lignes ont été renvoyées pour satisfaire la limite (mais notez que les lignes ignorées à cause de la clause `OFFSET` seront verrouillées). De la même manière, si `FOR UPDATE` ou `FOR SHARE` est utilisé pour la requête d'un curseur, seules les lignes réellement récupérées ou parcourues par le curseur seront verrouillées.

Si `FOR UPDATE` ou `FOR SHARE` apparaissent dans un sous-**SELECT**, les lignes verrouillées sont celles renvoyées par la sous-requête à la requête externe. Cela peut concerner moins de lignes que l'étude de la sous-requête seule pourrait faire penser, parce que les conditions de la requête externe peuvent être utilisées pour optimiser l'exécution de la sous-requête. Par exemple,

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

verrouillera uniquement le lignes pour lesquelles `col1 = 5`, même si cette condition n'est pas écrite dans la sous-requête.

Attention

Évitez de verrouiller une ligne puis de la modifier après un nouveau point de sauvegarde ou après un bloc d'exception PL/pgSQL. L'annulation suivante pourrait causer la perte du verrou. Par exemple :

```
BEGIN;
SELECT * FROM ma_table WHERE cle = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE ma_table SET ... WHERE cle = 1;
ROLLBACK TO s;
```

Après le **ROLLBACK**, la ligne est réellement déverrouillée au lieu de retourner à son état avant le point de sauvegarde. Ceci peut arriver si une ligne verrouillée dans la transaction en cours est mise à jour ou supprimée, ou si un verrou partagé est passé en verrou exclusif : dans tous ces cas, l'état précédent du verrou est oublié. Si la transation est ensuite annulée à un état entre la commande de verrou initiale et la modification qui a suivi, la ligne n'apparaîtra plus verrouillée. Ceci est une déficience de l'implémentation qui sera corrigée dans une prochaine version de PostgreSQL™.

Attention

Il est possible qu'une commande **SELECT** utilisant `ORDER BY` et `FOR UPDATE/SHARE` renvoie les lignes dans le désordre. C'est possible car l' `ORDER BY` est appliqué en premier. La commande trie le résultat, mais peut alors être bloquée le temps d'obtenir un verrou sur une ou plusieurs des lignes. Une fois que le `SELECT` est débloqué, des valeurs sur la colonne qui sert à ordonner peuvent avoir été modifiées, ce qui entraîne ces lignes apparaissant dans le désordre (bien qu'elles soient dans l'ordre par rapport aux valeurs d'origine de ces colonnes). Ceci peut être contourné si besoin en plaçant la clause `FOR UPDATE/SHARE` dans une sous-requête, par exemple

```
SELECT * FROM (SELECT * FROM matable FOR UPDATE) ss ORDER BY column1;
```

Notez que cela entraîne le verrouillage de toutes les lignes de matable, alors que `FOR UPDATE` au niveau supérieur verrouillerait seulement les lignes réellement renvoyées. Cela peut causer une différence de performance significative, en particulier si l' `ORDER BY` est combiné avec `LIMIT` ou d'autres restrictions. Cette technique est donc recommandée uniquement si vous vous attendez à des mises à jour concurrentes sur les colonnes servant à l'ordonnancement et qu'un résultat strictement ordonné est requis.

Commande `TABLE`

La commande

```
TABLE nom
```

est complètement équivalente à

```
SELECT * FROM nom
```

Elle peut être utilisée comme commande principale d'une requête, ou bien comme une variante syntaxique permettant de gagner de la place dans des parties de requêtes complexes.

Exemples

Joindre la table `films` avec la table `distributeurs` :

```
SELECT f.titre, f.did, d.nom, f.date_prod, f.genre
    FROM distributeurs d, films f
    WHERE f.did = d.did

       titre       | did |     nom      | date_prod  |   genre
-------------------+-----+--------------+------------+------------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drame
 The African Queen | 101 | British Lion | 1951-08-11 | Romantique
 ...
```

Additionner la colonne `longueur` de tous les films, grouper les résultats par `genre` :

```
SELECT genre, sum(longueur) AS total FROM films GROUP BY genre;

   genre    | total
------------+-------
 Action     | 07:34
 Comédie    | 02:58
 Drame      | 14:28
 Musical    | 06:42
 Romantique | 04:38
```

Additionner la colonne `longueur` de tous les films, grouper les résultats par `genre` et afficher les groupes dont les totaux font moins de cinq heures :

```
SELECT genre, sum(longueur) AS total
    FROM films
    GROUP BY genre
    HAVING sum(longueur) < interval '5 hours';

   genre    | total
------------+-------
 Comedie    | 02:58
 Romantique | 04:38
```

Les deux exemples suivants représentent des façons identiques de trier les résultats individuels en fonction du contenu de la deuxième colonne (`nom`) :

```
SELECT * FROM distributeurs ORDER BY nom;
SELECT * FROM distributeurs ORDER BY 2;

 did |       nom
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
```

```
 112 | Warner Bros.
 108 | Westward
```

L'exemple suivant présente l'union des tables `distributeurs` et `acteurs`, restreignant les résultats à ceux de chaque table dont la première lettre est un W. Le mot clé `ALL` est omis, ce qui permet de n'afficher que les lignes distinctes.

```
distributeurs:                acteurs:
 did |     nom               id |     nom
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...

SELECT distributeurs.nom
    FROM distributeurs
    WHERE distributeurs.nom LIKE 'W%'
UNION
SELECT actors.nom
    FROM acteurs
    WHERE acteurs.nom LIKE 'W%';

      nom
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

L'exemple suivant présente l'utilisation d'une fonction dans la clause `FROM`, avec et sans liste de définition de colonnes :

```
CREATE FUNCTION distributeurs(int) RETURNS SETOF distributeurs AS $$
    SELECT * FROM distributeurs WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributeurs(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributeurs_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributeurs WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributeurs_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Cet exemple montre comment utiliser une clause `WITH` simple:

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t

         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

Notez que la requête WITH n'a été évaluée qu'une seule fois, ce qui fait qu'on a deux jeux contenant les mêmes trois valeurs.

Cet exemple utilise WITH RECURSIVE pour trouver tous les subordonnés (directs ou indirects) de l'employée Marie, et leur niveau de subordination, à partir d'une table qui ne donne que les subordonnés directs :

```
WITH RECURSIVE recursion_employes(distance, nom_employe, nom_manager) AS (
    SELECT 1, nom_employe, nom_manager
    FROM employe
    WHERE nom_manager = 'Marie'
  UNION ALL
    SELECT er.distance + 1, e.nom_employe, e.nom_manager
    FROM recursion_employes er, employe e
    WHERE er.nom_employe = e.nom_manager
  )
SELECT distance, nom_employe FROM recursion_employes;
```

Notez la forme typique des requêtes récursives : une condition initiale, suivie par UNION, suivis par la partie récursive de la requête. Assurez-vous que la partie récursive de la requête finira par ne plus retourner d'enregistrement, sinon la requête bouclera indéfiniment (Voir Section 7.8, « Requêtes WITH (*Common Table Expressions*) » pour plus d'exemples).

Compatibilité

L'instruction **SELECT** est évidemment compatible avec le standard SQL. Mais il y a des extensions et quelques fonctionnalités manquantes.

Clauses FROM omises

PostgreSQL™ autorise l'omission de la clause FROM. Cela permet par exemple de calculer le résultat d'expressions simples :

```
SELECT 2+2;

 ?column?
----------
        4
```

D'autres bases de données SQL interdisent ce comportement, sauf à introduire une table virtuelle d'une seule ligne sur laquelle exécuter la commande **SELECT**.

S'il n'y a pas de clause FROM, la requête ne peut pas référencer les tables de la base de données. La requête suivante est, ainsi, invalide :

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

Les versions antérieures à PostgreSQL™ 8.1 acceptaient les requêtes de cette forme en ajoutant une entrée implicite à la clause FROM pour chaque table référencée. Ce n'est plus autorisé.

Omettre le mot clé AS

Dans le standard SQL, le mot clé AS peut être omis devant une colonne de sortie à partir du moment où le nouveau nom de colonne est un nom valide de colonne (c'est-à-dire, différent d'un mot clé réservé). PostgreSQL™ est légèrement plus restrictif : AS est nécessaire si le nouveau nom de colonne est un mot clé quel qu'il soit, réservé ou non. Il est recommandé d'utiliser AS ou des colonnes de sortie entourées de guillemets, pour éviter tout risque de conflit en cas d'ajout futur de mot clé.

Dans les éléments de FROM, le standard et PostgreSQL™ permettent que AS soit omis avant un alias qui n'est pas un mot clé réservé. Mais c'est peu pratique pour les noms de colonnes, à causes d'ambiguïtés syntaxiques.

ONLY et l'héritage

Le standard SQL impose des parenthèses autour du nom de table après la clause ONLY, comme dans SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE .... PostgreSQL™ considère les parenthèses comme étant optionnelles.

PostgreSQL™ autorise une * en fin pour indiquer explicitement le comportement opposé de la clause ONLY (donc inclure les tables filles). Le standard ne le permet pas.

(Ces points s'appliquent de la même façon à toutes les commandes SQL supportant l'option ONLY.)

Espace logique disponible pour GROUP BY et ORDER BY

Dans le standard SQL-92, une clause ORDER BY ne peut utiliser que les noms ou numéros des colonnes en sortie, une clause GROUP BY que des expressions fondées sur les noms de colonnes en entrée. PostgreSQL™ va plus loin, puisqu'il autorise cha-

cune de ces clauses à utiliser également l'autre possibilité. En cas d'ambiguïté, c'est l'interprétation du standard qui prévaut. PostgreSQL™ autorise aussi l'utilisation d'expressions quelconques dans les deux clauses. Les noms apparaissant dans ces expressions sont toujours considérés comme nom de colonne en entrée, pas en tant que nom de colonne du résultat.

SQL:1999 et suivant utilisent une définition légèrement différente, pas totalement compatible avec le SQL-92. Néanmoins, dans la plupart des cas, PostgreSQL™ interprète une expression `ORDER BY` ou `GROUP BY` en suivant la norme SQL:1999.

Restrictions sur la clause `WINDOW`

Le standard SQL fournit des options additionnelles pour la *`clause_frame`* des window. PostgreSQL™ ne supporte à ce jour que les options mentionnées précédemment.

`LIMIT` et `OFFSET`

Les clauses `LIMIT` et `OFFSET` sont une syntaxe spécifique à PostgreSQL™, aussi utilisée dans MySQL™. La norme SQL:2008 a introduit les clauses `OFFSET ... FETCH {FIRST|NEXT}...` pour la même fonctionnalité, comme montré plus haut dans la section intitulée « Clause `LIMIT` ». Cette syntaxe est aussi utilisée par IBM DB2™. (Les applications écrites pour Oracle™ contournent fréquemment le problème par l'utilisation de la colonne autogénérée `rownum` pour obtenir les effets de ces clauses, qui n'est pas disponible sous PostgreSQL,)

`FOR UPDATE` and `FOR SHARE`

Bien que `FOR UPDATE` soit présent dans le standard SQL, le standard ne l'autorise que comme une option de **DECLARE CURSOR**. PostgreSQL™ l'autorise dans toute requête **SELECT** et dans toute sous-requête **SELECT**, mais c'est une extension. Ni la variante `FOR SHARE`, ni l'option `NOWAIT` n'apparaissent dans le standard.

Clauses non standard

La clause `DISTINCT ON` n'est pas définie dans le standard SQL.

Nom

SELECT INTO — définit une nouvelle table à partir des résultats d'une requête

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] nom_en_sortie ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] nouvelle_table
    [ FROM élément_from [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW nom_window AS ( définition_window ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING opérateur ] [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET début [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ nombre ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF nomtable [, ...] ] [ NOWAIT ] [...] ]
```

Description

SELECT INTO crée une nouvelle table en la remplissant avec des données récupérées par une requête. Les données ne sont pas renvoyées au client comme le fait habituellement l'instruction **SELECT**. Les nouvelles colonnes de la table ont les noms et les types de données associés avec les colonnes en sortie du **SELECT**.

Paramètres

`TEMPORARY` ou `TEMP`
: Si spécifié, la table est créée comme une table temporaire. Référez-vous à CREATE TABLE(7) pour plus de détails.

`new_table`
: Le nom de la table à créer (pouvant être qualifié par le nom du schéma).

Tous les autres paramètres sont décrits en détail dans SELECT(7).

Notes

CREATE TABLE AS(7) est fonctionnellement équivalent à **SELECT INTO**. **CREATE TABLE AS** est la syntaxe recommandée car cette forme de **SELECT INTO** n'est pas disponible dans ECPG ou PL/pgSQL. En effet, ils interprètent la clause `INTO` différemment. De plus, **CREATE TABLE AS** offre un ensemble de fonctionnalités plus important que celui de **SELECT INTO**.

Avant PostgreSQL™ 8.1, la table créée par **SELECT INTO** incluait des OID par défaut. Dans PostgreSQL™ 8.1, ce n'est plus le cas -- pour inclure des OID dans la nouvelle table, la variable de configuration default_with_oids doit être activée. Autrement, **CREATE TABLE AS** peut aussi être utilisé avec la clause `WITH OIDS`.

Exemples

Crée une nouvelle table `films_recent` ne contenant que les entrées récentes de la table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

Compatibilité

Le standard SQL utilise **SELECT INTO** pour représenter la sélection de valeurs dans des variables scalaires d'un programme hôte plutôt que la création d'une nouvelle table. Ceci est en fait l'utilisation trouvée dans ECPG (voir Chapitre 33, ECPG - SQL embarqué dans du C) et dans PL/pgSQL (voir Chapitre 39, PL/pgSQL - Langage de procédures SQL). L'usage de PostgreSQL™ de **SELECT INTO** pour représenter une création de table est historique. Il est préférable d'utiliser **CREATE TABLE AS** dans un nouveau programme.

Voir aussi

CREATE TABLE AS(7)

Nom

SET — change un paramètre d'exécution

Synopsis

```
SET [ SESSION | LOCAL ] paramètre_configuration { TO | = } { valeur | 'valeur' |
DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { fuseau-horaire | LOCAL | DEFAULT }
```

Description

La commande **SET** permet de modifier les paramètres d'exécution. Un grand nombre de paramètres d'exécution, listés dans Chapitre 18, Configuration du serveur, peuvent être modifiés à la volée avec la commande **SET**. **SET** ne modifie que les paramètres utilisés par la session courante.

Certains paramètres ne peuvent être modifiés que par le superutilisateur, d'autres ne peuvent plus être changés après le démarrage du serveur ou de la session.

Si **SET** ou **SET SESSION** sont utilisés dans une transaction abandonnée par la suite, les effets de la commande **SET** disparaissent dès l'annulation de la transaction. Lorsque la transaction englobant la commande est validée, les effets de la commande persistent jusqu'à la fin de la session, à moins qu'ils ne soient annulés par une autre commande **SET**.

Les effets de **SET LOCAL** ne durent que jusqu'à la fin de la transaction en cours, qu'elle soit validée ou non. Dans le cas particulier d'une commande **SET** suivie par **SET LOCAL** dans une même transaction, la valeur de **SET LOCAL** est utilisée jusqu'à la fin de la transaction, et celle de **SET** prend effet ensuite (si la transaction est validée).

Les effets de **SET** et **SET LOCAL** sont aussi annulés par le retour à un point de sauvegarde précédant la commande.

Si **SET LOCAL** est utilisé à l'intérieur d'une fonction qui comprend l'option `SET` pour la même variable (voir CREATE FUNCTION(7)), les effets de la commande **SET LOCAL** disparaîtront à la sortie de la fonction ; en fait, la valeur disponible lors de l'appel de la fonction est restaurée de toute façon. Ceci permet l'utilisation de **SET LOCAL** pour des modifications dynamiques et répétées d'un paramètre à l'intérieur d'une fonction, avec l'intérêt d'utiliser l'option `SET` pour sauvegarder et restaurer la valeur de l'appelant. Néanmoins, une commande **SET** standard surcharge toute option `SET` de la fonction ; son effet persistera sauf en cas d'annulation.

> **Note**
>
> De PostgreSQL™ version 8.0 à 8.2, les effets de **SET LOCAL** sont annulés suite au relachement d'un point de sauvegarde précédent, ou par une sortie avec succès d'un bloc d'exception PL/pgSQL. Ce comportement a été modifié car il n'était pas du tout intuitif.

Paramètres

`SESSION`
: Indique que la commande prend effet pour la session courante. C'est la valeur par défaut lorsque `SESSION` et `LOCAL` sont omis.

`LOCAL`
: Indique que la commande n'est effective que pour la transaction courante. Après **COMMIT** ou **ROLLBACK**, la valeur de session redevient effective. Une commande **SET LOCAL** est sans effet si elle est exécutée en dehors d'un bloc **BEGIN** car la transaction prend immédiatement fin.

`paramètre_configuration`
: Nom d'un paramètre ajustable pendant l'exécution. La liste des paramètres disponibles est documentée dans Chapitre 18, Configuration du serveur et ci-dessous.

`valeur`
: Nouvelle valeur du paramètre. Les valeurs peuvent être indiquées sous forme de constantes de chaîne, d'identifiants, de nombres ou de listes de ceux-ci, séparées par des virgules, de façon approprié pour ce paramètre. `DEFAULT` peut être utilisé pour repositionner le paramètre à sa valeur par défaut (c'est-à-dire quelque soit la valeur qu'il aurait eu si aucun **SET** n'avait été exécuté lors de cette session).

En plus des paramètres de configuration documentés dans Chapitre 18, Configuration du serveur, il y en a quelques autres qui ne

peuvent être initialisés qu'avec la commande **SET** ou ont une syntaxe spéciale.

`SCHEMA`
: `SET SCHEMA 'valeur'` est un alias pour `SET search_path TO valeur`. Seul un schéma peut être précisé en utilisant cette syntaxe.

`NAMES`
: `SET NAMES valeur` est un équivalent de `SET client_encoding TO valeur`.

`SEED`
: Précise la valeur interne du générateur de nombres aléatoires (la fonction `random`). Les valeurs autorisées sont des nombres à virgule flottante entre -1 et 1, qui sont ensuite multipliés par 2^{31}-1.

  Le générateur de nombres aléatoires peut aussi être initialisé en appelant la fonction `setseed` :

  ```
  SELECT setseed(valeur);
  ```

`TIME ZONE`
: `SET TIME ZONE valeur` est équivalent à `SET timezone TO valeur`. La syntaxe `SET TIME ZONE` permet d'utiliser une syntaxe spéciale pour indiquer le fuseau horaire. Quelques exemples de valeurs valides :

  `'PST8PDT'`
  Le fuseau horaire de Berkeley, Californie.

  `'Europe/Rome'`
  Le fuseau horaire de l'Italie.

  `-7`
  Le fuseau horaire situé 7 heures à l'ouest de l'UTC (équivalent à PDT). Les valeurs positives sont à l'est de l'UTC.

  `INTERVAL '-08:00' HOUR TO MINUTE`
  Le fuseau horaire situé 8 heures à l'ouest de l'UTC (équivalent à PST).

  `LOCAL`, `DEFAULT`
  Utilise le fuseau horaire local (c'est-à-dire la valeur `timezone` par défaut du serveur ; si cette dernière n'est pas explicitement configurée, il utilise la zone par défaut du système d'exploitation).
  Voir Section 8.5.3, « Fuseaux horaires » pour de plus amples informations sur les fuseaux horaires.

Notes

La fonction `set_config` propose des fonctionnalités équivalentes. Voir Section 9.24, « Fonctions d'administration système ». De plus, il est possible de mettre à jour (via UPDATE) la vue système pg_settings pour réaliser l'équivalent de **SET**.

Exemples

Mettre à jour le chemin de recherche :

```
SET search_path TO my_schema, public;
```

Utiliser le style de date traditionnel POSTGRES™ avec comme convention de saisie « les jours avant les mois » :

```
SET datestyle TO postgres, dmy;
```

Utiliser le fuseau horaire de Berkeley, Californie :

```
SET TIME ZONE 'PST8PDT';
```

Utiliser le fuseau horaire de l'Italie :

```
SET TIME ZONE 'Europe/Rome';
```

Compatibilité

`SET TIME ZONE` étend la syntaxe définie dans le standard SQL. Le standard ne permet que des fuseaux horaires numériques alors que PostgreSQL™ est plus souple dans les syntaxes acceptées. Toutes les autres fonctionnalités de `SET` sont des extensions de PostgreSQL™.

Voir aussi

RESET(7), SHOW(7)

Nom

SET CONSTRAINTS — initialise le moment de vérification de contrainte de la transaction en cours

Synopsis

```
SET CONSTRAINTS { ALL | nom [, ...] } { DEFERRED | IMMEDIATE }
```

Description

SET CONSTRAINTS initialise le comportement de la vérification des contraintes dans la transaction en cours. Les contraintes `IMMEDIATE` sont vérifiées à la fin de chaque instruction. Les contraintes `DEFERRED` ne sont vérifiées qu'à la validation de la transaction. Chaque contrainte a son propre mode `IMMEDIATE` ou `DEFERRED`.

À la création, une contrainte se voit donner une des trois caractéristiques : `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE` ou `NOT DEFERRABLE`. La troisième forme est toujours `IMMEDIATE` et n'est pas affectée par la commande **SET CONSTRAINTS**. Les deux premières classes commencent chaque transaction dans le mode indiqué mais leur comportement peut changer à l'intérieur d'une transaction par **SET CONSTRAINTS**.

SET CONSTRAINTS avec une liste de noms de contraintes modifie le mode de ces contraintes (qui doivent toutes être différables). Chaque nom de contrainte peut être qualifié d'un schéma. Le chemin de recherche des schémas est utilisé pour trouver le premier nom correspondant si aucun nom de schéma n'a été indiqué. **SET CONSTRAINTS ALL** modifie le mode de toutes les contraintes déférables.

Lorsque **SET CONSTRAINTS** modifie le mode d'une contrainte de `DEFERRED` à `IMMEDIATE`, le nouveau mode prend effet rétroactivement : toute modification de données qui aurait été vérifiée à la fin de la transaction est en fait vérifiée lors de l'exécution de la commande **SET CONSTRAINTS**. Si une contrainte est violée, la commande **SET CONSTRAINTS** échoue (et ne change pas le mode de contrainte). Du coup, **SET CONSTRAINTS** peut être utilisée pour forcer la vérification de contraintes à un point spécifique d'une transaction.

Actuellement, seules les contraintes `UNIQUE`, `PRIMARY KEY`, `REFERENCES` (clé étrangère) et `EXCLUDE` sont affectées par ce paramètre. Les contraintes `NOT NULL` et `CHECK` sont toujours vérifiées immédiatement quand une ligne est insérée ou modifiée (*pas* à la fin de l'instruction). Les contraintes uniques et d'exclusion qui n'ont pas été déclarées `DEFERRABLE` sont aussi vérifiées immédiatement.

Le déclenchement des triggers qui sont déclarés comme des « triggers de contraintes » est aussi contrôlé par ce paramètre -- ils se déclenchent au même moment que la contrainte associée devait être vérifiée.

Notes

Comme PostgreSQL™ ne nécessite pas les noms de contraintes d'être uniques à l'intérieur d'un schéma (mais seulement par tables), il est possible qu'il y ait plus d'une correspondance pour un nom de contrainte spécifié. Dans ce cas, **SET CONSTRAINTS** agira sur toutes les correspondances. Pour un nom sans qualification de schéma, une fois qu'une ou plusieurs correspondances ont été trouvées dans les schémas du chemin de recherche, les autres schémas du chemin ne sont pas testés.

Cette commande altère seulement le comportement des contraintes à l'intérieur de la transaction en cours. Du coup, si vous exécutez cette commande en dehors d'un bloc de transaction (pair **BEGIN**/**COMMIT**), elle ne semble pas avoir d'effet.

Compatibilité

Cette commande est compatible avec le comportement défini par le standard SQL en dehors du fait que, dans PostgreSQL™, il ne s'applique pas aux contraintes `NOT NULL` et `CHECK`. De plus, PostgreSQL™ vérifie les contraintes uniques non déferrables immédiatement, pas à la fin de l'instruction comme le standard le suggère.

Nom

SET ROLE — initialise l'identifiant utilisateur courant de la session en cours

Synopsis

```
SET [ SESSION | LOCAL ] ROLE nom_rôle
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

Description

Cette commande positionne l'identifiant utilisateur courant suivant la session SQL en cours à *nom_rôle*. Le nom du rôle peut être un identifiant ou une chaîne littérale. Après **SET ROLE**, la vérification des droits sur les commandes SQL est identique à ce qu'elle serait si le rôle nommé s'était lui-même connecté.

Il est obligatoire que l'utilisateur de la session courante soit membre du rôle *nom_rôle* (si l'utilisateur de la session est superutilisateur, tous les rôles sont utilisables).

Les modificateurs `SESSION` et `LOCAL` agissent de la même façon que pour la commande SET(7).

Les formes `NONE` et `RESET` réinitialisent l'identifiant de l'utilisateur à la valeur de session. Ces formes peuvent être exécutées par tout utilisateur.

Notes

L'utilisation de cette commande permet d'étendre ou de restreindre les privilèges d'un utilisateur. Si le rôle de l'utilisateur de la session comprend l'attribut `INHERITS`, alors il acquiert automatiquement les droits de chaque rôle qu'il peut prendre par la commande **SET ROLE** ; dans ce cas, **SET ROLE** supprime tous les droits affectés directement à l'utilisateur de la session et les autres droits des rôles dont il est membre, ne lui laissant que les droits disponibles sur le rôle nommé. A l'opposé, si le rôle session de l'utilisateur dispose de l'attribut `NOINHERITS`, **SET ROLE** supprime les droits affectés directement à l'utilisateur session et les remplace par les privilèges du rôle nommé.

En particulier, quand un utilisateur choisit un rôle autre que superutilisateur via **SET ROLE**, il perd les droits superutilisateur.

SET ROLE a des effets comparables à SET SESSION AUTHORIZATION(7) mais la vérification des droits diffère. De plus, **SET SESSION AUTHORIZATION** détermine les rôles autorisés dans les commandes **SET ROLE** ultérieures alors que **SET ROLE** ne modifie pas les rôles accessibles par un futur **SET ROLE**.

SET ROLE ne traite pas les variables de session indiqué par les paramètres du rôle (et configurés avec ALTER ROLE(7) ; cela ne survient qu'à la connexion.

SET ROLE ne peut pas être utilisé dans une fonction `SECURITY DEFINER`.

Exemples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

Compatibilité

PostgreSQL™ autorise la syntaxe identifiant (`"nom_role"`) alors que le SQL standard impose une chaîne littérale pour le nom du rôle. SQL n'autorise pas cette commande lors d'une transaction ; PostgreSQL™ n'est pas aussi restrictif, rien ne justifie cette interdiction. Les modificateurs `SESSION` et `LOCAL` sont des extensions PostgreSQL™ tout comme la syntaxe `RESET`.

Voir aussi

SET SESSION AUTHORIZATION(7)

Nom

SET SESSION AUTHORIZATION — Initialise l'identifiant de session de l'utilisateur et l'identifiant de l'utilisateur actuel de la session en cours

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION nom_utilisateur
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Description

Cette commande positionne l'identifiant de session de l'utilisateur et celui de l'utilisateur courant pour la session SQL en cours à *`nom_utilisateur`*. Le nom de l'utilisateur peut être un identifiant ou une chaîne littérale. En utilisant cette commande, il est possible, par exemple, de devenir temporairement un utilisateur non privilégié et de redevenir plus tard superutilisateur.

L'identifiant de session de l'utilisateur est initialement positionné au nom de l'utilisateur (éventuellement authentifié) fourni par le client. L'identifiant de l'utilisateur courant est habituellement identique à l'identifiant de session de l'utilisateur mais il peut être temporairement modifié par le contexte de fonctions `SECURITY DEFINER` ou de mécanismes similaires ; il peut aussi être changé par SET ROLE(7). L'identifiant de l'utilisateur courant est essentiel à la vérification des permissions.

L'identifiant de session de l'utilisateur ne peut être changé que si l'utilisateur de session initial (l'*utilisateur authentifié*) dispose des privilèges superutilisateur. Dans le cas contraire, la commande n'est acceptée que si elle fournit le nom de l'utilisateur authentifié.

Les modificateurs `SESSION` et `LOCAL` agissent de la même façon que la commande standard SET(7).

Les formes `DEFAULT` et `RESET` réinitialisent les identifiants courant et de session de l'utilisateur à ceux de l'utilisateur originellement authentifié. Tout utilisateur peut les exécuter.

Notes

SET SESSION AUTHORIZATION ne peut pas être utilisé dans une fonction `SECURITY DEFINER`.

Exemples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Compatibilité

Le standard SQL autorise l'apparition de quelques autres expressions à la place de *`nom_utilisateur`*. Dans la pratique, ces expressions ne sont pas importantes. PostgreSQL™ autorise la syntaxe de l'identifiant (`"nom_utilisateur"`) alors que SQL ne le permet pas. SQL n'autorise pas l'exécution de cette commande au cours d'une transaction ; PostgreSQL™ n'impose pas cette restriction parce qu'il n'y a pas lieu de le faire. Les modificateurs `SESSION` et `LOCAL` sont des extensions PostgreSQL™ tout comme la syntaxe `RESET`.

Le standard laisse la définition des droits nécessaires à l'exécution de cette commande à l'implantation.

Voir aussi

SET ROLE(7)

Nom

SET TRANSACTION — initialise les caractéristiques de la transaction actuelle

Synopsis

```
SET TRANSACTION mode_transaction [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION mode_transaction [, ...]

où mode_transaction fait
partie de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
```

Description

La commande **SET TRANSACTION** initialise les caractéristiques de la transaction courante. Elle est sans effet sur les transactions suivantes. **SET SESSION CHARACTERISTICS** positionne les caractéristiques par défaut pour toutes les transactions à venir d'une session. Ces valeurs peuvent ensuite être surchargées par **SET TRANSACTION** pour une transaction particulière.

Les caractéristiques de transaction disponibles sont le niveau d'isolation et le mode d'accès de la transaction (lecture/écriture ou lecture seule).

Le niveau d'isolation détermine les données que la transaction peut voir quand d'autres transactions fonctionnent concurrentiellement :

`READ COMMITTED`
: Une instruction ne peut voir que les lignes validées avant qu'elle ne commence. C'est la valeur par défaut.

`SERIALIZABLE`
: Toute instruction de la transaction en cours ne peut voir que les lignes validées avant que la première requête ou instruction de modification de données soit exécutée dans cette transaction.

Le standard SQL définit deux niveaux supplémentaires, `READ UNCOMMITTED` et `REPEATABLE READ`. Dans PostgreSQL™, `READ UNCOMMITTED` est traité comme `READ COMMITTED` alors que `REPEATABLE READ` est traité comme `SERIALIZABLE`.

Le niveau d'isolation de la transaction ne peut plus être modifié après l'exécution de la première requête ou instruction de modification de données (**SELECT**, **INSERT**, **DELETE**, **UPDATE**, **FETCH** ou **COPY**) d'une transaction. Voir Chapitre 13, Contrôle d'accès simultané pour plus d'informations sur l'isolation et le contrôle de concurrence.

La méthode d'accès de la transaction détermine si elle est en lecture/écriture ou en lecture seule. Lecture/écriture est la valeur par défaut. Quand une transaction est en lecture seule, les commandes SQL suivantes sont interdites : `INSERT`, `UPDATE`, `DELETE` et `COPY FROM` si la table modifiée n'est pas temporaire ; toutes les commandes `CREATE`, `ALTER` et `DROP` ; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE` ; `EXPLAIN ANALYZE` et `EXECUTE` si la commande exécutée figure parmi celles listées plus haut. C'est une notion de haut niveau de lecture seule qui n'interdit pas toutes les écritures sur disque.

Notes

Si **SET TRANSACTION** est exécuté sans **START TRANSACTION** ou **BEGIN** préalable, il est sans effet car la transaction se termine immédiatement.

Il est possible de se dispenser de **SET TRANSACTION** en spécifiant le *`mode_transaction`* désiré dans **BEGIN** ou **START TRANSACTION**.

Les modes de transaction par défaut d'une session peuvent aussi être configurés en initialisant les paramètres de configuration default_transaction_isolation et default_transaction_read_only. (En fait, **SET SESSION CHARACTERISTICS** est un équivalent verbeux de la configuration de ces variables avec **SET**.) Les valeurs par défaut peuvent ainsi être initialisées dans le fichier de configuration, via **ALTER DATABASE**, etc. Chapitre 18, Configuration du serveur fournit de plus amples informations.

Compatibilité

Les deux commandes sont définies dans le standard SQL. `SERIALIZABLE` y est le niveau d'isolation par défaut de la transac-

tion. Dans PostgreSQL™, la valeur par défaut est habituellement `READ COMMITTED`, mais elle peut être modifiée comme cela est mentionné ci-dessus. À défaut d'un prédicat vérouillé, le niveau `SERIALIZABLE` n'est pas vraiment sérialisable. Voir Chapitre 13, Contrôle d'accès simultané pour plus de détails.

Dans le standard SQL, il existe une autre caractéristique de transaction pouvant être configurée avec ces commandes : la taille de l'aire de diagnostique. Ce concept n'est valable que pour le SQL embarqué et, de fait, n'est pas implanté dans le serveur PostgreSQL™.

Le standard SQL requiert des virgules entre chaque *`mode_transaction`* mais, pour des raisons historiques, PostgreSQL™ autorise l'omission des virgules.

Nom

SHOW — affiche la valeur d'un paramètre d'exécution

Synopsis

```
SHOW nom
SHOW ALL
```

Description

SHOW affiche la configuration courante des paramètres d'exécution. Ces variables peuvent être initialisées à l'aide de l'instruction **SET**, par le fichier de configuration `postgresql.conf`, par la variable d'environnement `PGOPTIONS` (lors de l'utilisation de libpq ou d'une application fondée sur libpq), ou à l'aide d'options en ligne de commande lors du démarrage de **postgres**. Voir Chapitre 18, Configuration du serveur pour plus de détails.

Paramètres

nom

Le nom d'un paramètre d'exécution. Les paramètres disponibles sont documentés dans Chapitre 18, Configuration du serveur et sur la page de référence SET(7). De plus, il existe quelques paramètres qui peuvent être affichés mais ne sont pas initialisables :

`SERVER_VERSION`
Affiche le numéro de version du serveur.

`SERVER_ENCODING`
Affiche l'encodage des caractères côté serveur. À ce jour, ce paramètre peut être affiché mais pas initialisé parce que l'encodage est déterminé au moment de la création de la base de données.

`LC_COLLATE`
Affiche la locale de la base de données pour le tri de texte. À ce jour, ce paramètre est affichable mais pas initialisé parce que la configuration est déterminée lors de la création de la base de données.

`LC_CTYPE`
Affiche la locale de la base de données pour la classification des caractères. À ce jour, ce paramètre peut être affiché mais pas initialisé parce que la configuration est déterminée lors de la création de la base de données.

`IS_SUPERUSER`
Vrai si le rôle courant a des droits de super-utilisateur.

`ALL`

Affiche les valeurs de tous les paramètres de configuration avec leur description.

Notes

La fonction `current_setting` affiche les mêmes informations. Voir Section 9.24, « Fonctions d'administration système ». De plus, la vue système pg_settings propose la même information.

Exemples

Affiche la configuration courante du paramètre `datestyle` :

```
SHOW datestyle;
 datestyle
-----------
 ISO, MDY
(1 row)
```

Affiche la configuration courante du paramètre `geqo` :

```
SHOW geqo;
 geqo
------
 on
```

```
(1 row)
```

Affiche tous les paramètres :

```
            name          | setting |                 description
--------------------------+---------+-----------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption                | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages       | off     | Continues processing past damaged page headers.
(196 rows)
```

Compatibilité

La commande **SHOW** est une extension PostgreSQL™.

Voir aussi

SET(7), RESET(7)

Nom

START TRANSACTION — débute un bloc de transaction

Synopsis

```
START TRANSACTION [ mode_transaction [, ...] ]

où mode_transaction fait
partie de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
```

Description

Cette commande débute un nouveau bloc de transaction. Si le niveau d'isolation ou le mode lecture/écriture est spécifié, la nouvelle transaction adopte ces caractéristiques, comme si SET TRANSACTION(7) avait été exécuté. Cette commande est identique à la commande BEGIN(7).

Paramètres

Pour obtenir la signification des paramètres de cette instruction, on pourra se référer à SET TRANSACTION(7).

Compatibilité

Le standard SQL n'impose pas de lancer **START TRANSACTION** pour commencer un bloc de transaction : toute commande SQL débute implicitement un bloc. On peut considérer que PostgreSQL™ exécute implicitement un **COMMIT** après chaque commande non précédée de **START TRANSACTION** (ou **BEGIN**). Ce comportement est d'ailleurs souvent appelé « autocommit ». D'autres systèmes de bases de données relationnelles offrent une fonctionnalité de validation automatique.

Le standard SQL impose des virgules entre les *modes_transaction* successifs mais, pour des raisons historiques, PostgreSQL™ autorise l'omission des virgules.

Voir aussi la section de compatibilité de SET TRANSACTION(7).

Voir aussi

BEGIN(7), COMMIT(7), ROLLBACK(7), SAVEPOINT(7), SET TRANSACTION(7)

Nom

TRUNCATE — vide une table ou un ensemble de tables

Synopsis

```
TRUNCATE [ TABLE ] [ ONLY ] nom [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

Description

La commande **TRUNCATE** supprime rapidement toutes les lignes d'un ensemble de tables. Elle a le même effet qu'un **DELETE** non qualifié sur chaque table, mais comme elle ne parcourt par la table, elle est plus rapide. De plus, elle récupère immédiatement l'espace disque, évitant ainsi une opération **VACUUM**. Cette commande est particulièrement utile pour les tables volumineuses.

Paramètres

nom
: Le nom d'une table à vider (pouvant être qualifié par le schéma). Si la clause `ONLY` est précisée avant le nom de la table, seule cette table est tronquée. Dans le cas contraire, la table et toutes ses tables filles (si elle en a) sont tronquées. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles sont incluses.

`RESTART IDENTITY`
: Redémarre les séquences intégrées aux colonnes des tables tronquées.

`CONTINUE IDENTITY`
: Ne change pas la valeur des séquences. C'est la valeur par défaut.

`CASCADE`
: Vide toutes les tables qui ont des références de clés étrangères sur une des tables nommées et sur toute table ajoutée au groupe à cause du `CASCADE`.

`RESTRICT`
: Refuse le vidage si une des tables a des références de clés étrangères sur une table qui ne sont pas listées dans la commande. Cette option est active par défaut.

Notes

Vous devez avoir le droit `TRUNCATE` sur la table que vous voulez tronquer.

TRUNCATE nécessite un verrou d'accès exclusif (`ACCESS EXCLUSIVE`) sur chaque table qu'il traite, ce qui bloque toutes les autres opérations en parallèle sur cette table. Si un accès concurrent est nécessaire, alors la commande **DELETE** doit être utilisée.

TRUNCATE ne peut pas être utilisé sur une table référencée par d'autres tables au travers de clés étrangères, sauf si ces tables sont aussi comprises dans la commande. Dans le cas contraire, la vérification nécessiterait des parcours complets de tables, ce qui n'est pas le but de la commande **TRUNCATE**. L'option `CASCADE` est utilisable pour inclure automatiquement toutes les tables dépendantes -- faites attention lorsque vous utilisez cette option parce que vous pourriez perdre des données que vous auriez souhaitez conserver !

TRUNCATE ne déclenchera aucun trigger `ON DELETE` qui pourrait exister sur les tables. Par contre, il déclenchera les triggers `ON TRUNCATE`. Si des triggers `ON TRUNCATE` sont définis sur certaines des tables, alors tous les triggers `BEFORE TRUNCATE` sont déclenchés avant que le tronquage n'intervienne, et tous les triggers `AFTER TRUNCATE` sont déclenchés après la réalisation du dernier tronquage. Les triggers se déclencheront dans l'ordre de traitement des tables (tout d'abord celles listées dans la commande, puis celles ajoutées à cause des cascades).

Avertissement

TRUNCATE n'est pas compatible avec MVCC (voir Chapitre 13, Contrôle d'accès simultané pour des informations générales sur MVCC). Après un vidage, la table apparaîtra vide pour toutes les transactions en cours même si elles utilisent une image prise avant le début du vidage. Ceci est un problème pour une transaction qui n'a pas eu un accès à la table avant le vidage -- toute transaction qui y a accédé avant détient au moins un verrou `ACCESS SHARE` qui bloquera **TRUNCATE** jusqu'à la fin de la transaction. Donc le vidage ne causera aucune in-

cohérence apparente dans le contenu de la table pour des requêtes successives sur la même table, mais il causera une incohérence visble entre la table tronquée et les autres tables de la base.

Néanmoins, **TRUNCATE** est compatible avec le système des transactions : un tronquage pourrait être annulée si la transaction en cours n'est pas validée.

TRUNCATE est compatible avec le système des transactions. Les données seront toujours disponibles si la transaction est annulée.

Avertissement

Les opérations sur les séquences (**ALTER SEQUENCE RESTART**) réalisées en conséquence de l'utilisation de l'option `RESTART IDENTITY` sont non transactionnelles et ne seront pas annulées en cas d'échec. Pour minimiser les risques, ces opérations sont réalisées seulement après que tout le reste du travail de **TRUNCATE** ne soit terminé. Néanmoins, il y a toujours un risque si **TRUNCATE** est exécuté dans un bloc de transaction qui est annulé ultérieurement. Par exemple, avec :

```
BEGIN;
TRUNCATE TABLE foo RESTART IDENTITY;
COPY foo FROM ...;
COMMIT;
```

Si le **COPY** échoue, les données de la table seront restaurées mais les séquences seront laissées avec des valeurs probablement inférieures à celles qu'elles avaient avant, ce qui pourrait amener à des échecs de clés dupliquées ou à d'autres problèmes dans les transactions suivantes. Si cela peut devenir un problème, il est bien mieux d'éviter d'utiliser `RESTART IDENTITY`, et d'accepter que le nouveau contenu de la table aura des numéros de séquence bien supérieur à l'ancien.

Exemples

Vider les tables `grossetable` et `grandetable` :

```
TRUNCATE grossetable, grandetable;
```

La même chose, en réinitialisant les générateurs des séquences associées :

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

Vide la table `uneautretable`, et cascade cela à toutes les tables qui référencent `uneautretable` via des contraintes de clés étrangères :

```
TRUNCATE uneautretable CASCADE;
```

Compatibilité

Le standard SQL:2008 inclut une commande **TRUNCATE** avec la syntaxe `TRUNCATE TABLE` *`nom_table`*. Les clauses `CONTINUE IDENTITY`/`RESTART IDENTITY` font aussi partie du standard mais ont une signification légèrement différente, quoique en rapport. Certains des comportements de concurrence de cette commande sont laissés au choix de l'implémentation par le standard, donc les notes ci-dessus doivent être comprises et comparées avec les autres implémentations si nécessaire.

Nom

UNLISTEN — arrête l'écoute d'une notification

Synopsis

```
UNLISTEN { canal | * }
```

Description

UNLISTEN est utilisé pour supprimer un abonnement aux événements **NOTIFY**. **UNLISTEN** annule tout abonnement pour la session PostgreSQL™ en cours sur le canal de notification nommé *canal*. Le caractère générique * annule tous les abonnements de la session en cours.

NOTIFY(7) contient une discussion plus complète de l'utilisation de **LISTEN** et de **NOTIFY**.

Paramètres

canal
: Le nom d'un canal de notification (un identificateur quelconque).

*
: Tous les abonnements de cette session sont annulés.

Notes

Il est possible de se désabonner de quelque chose pour lequel il n'y a pas d'abonnement ; aucun message d'avertissement ou d'erreur n'est alors retourné.

À la fin de chaque session, **UNLISTEN *** est exécuté automatiquement.

Une transaction qui a exécuté **UNLISTEN** ne peut pas être préparée pour une validation en deux phases.

Exemples

Pour s'abonner :

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Une fois que **UNLISTEN** a été exécuté, les messages **NOTIFY** suivants sont ignorés :

```
UNLISTEN virtual;
NOTIFY virtual;
-- aucun événement NOTIFY n'est reçu
```

Compatibilité

Il n'y a pas de commande **UNLISTEN** dans le standard SQL.

Voir aussi

LISTEN(7), NOTIFY(7)

Nom

UPDATE — mettre à jour les lignes d'une table

Synopsis

```
UPDATE [ ONLY ] table [ * ] [ [ AS ] alias ]
    SET { colonne = { expression | DEFAULT } |
          ( colonne [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
    [ FROM liste_from ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | expression_sortie [ [ AS ] nom_sortie ] [, ...] ]
```

Description

UPDATE modifie les valeurs des colonnes spécifiées pour toutes les lignes qui satisfont la condition. Seules les colonnes à modifier doivent être mentionnées dans la clause `SET` ; les autres colonnes conservent leur valeur.

Il existe deux façons de modifier le contenu d'une table à partir d'informations contenues dans d'autres tables de la base de données : à l'aide de sous-requêtes ou en spécifiant des tables supplémentaires dans la clause `FROM`. Le contexte permet de décider de la technique la plus appropriée.

La clause `RETURNING` optionnelle fait que **UPDATE** calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours de mise à jour. Toute expression utilisant les colonnes de la table et/ou les colonnes d'autres tables mentionnées dans `FROM` peut être calculée. La syntaxe de la liste `RETURNING` est identique à celle de la commande **SELECT**.

L'utilisateur doit posséder le droit `UPDATE` sur la table, ou au moins sur les colonnes listées pour la mise à jour. Vous devez aussi avoir le droit `SELECT` sur toutes les colonnes dont les valeurs sont lues dans les *`expression`*s ou *`condition`*.

Paramètres

`table`
: Le nom de la table à mettre à jour (éventuellement qualifié du nom du schéma). Si `ONLY` est indiqué avant le nom de la table, les lignes modifiées ne concernent que la table nommée. Si `ONLY` n'est pas indiquée, les lignes modifiées font partie de la table nommée et de ses tables filles. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.

`alias`
: Un nom de substitution pour la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table. Par exemple, avec `UPDATE foo AS f`, le reste de l'instruction **UPDATE** doit référencer la table avec `f` et non plus `foo`.

`colonne`
: Le nom d'une colonne dans *`table`*. Le nom de la colonne peut être qualifié avec un nom de sous-champ ou un indice de tableau, si nécessaire. Ne pas inclure le nom de la table dans la spécification d'une colonne cible -- par exemple, `UPDATE tab SET tab.col = 1` est invalide.

`expression`
: Une expression à affecter à la colonne. L'expression peut utiliser les anciennes valeurs de cette colonne et d'autres colonnes de la table.

`DEFAULT`
: Réinitialise la colonne à sa valeur par défaut (qui vaut NULL si aucune expression par défaut ne lui a été affectée).

`liste_from`
: Une liste d'expressions de tables, qui permet aux colonnes des autres tables d'apparaître dans la condition `WHERE` et dans les expressions de mise à jour. Cela est similaire à la liste de tables pouvant être spécifiée dans la section intitulée « Clause `FROM` » d'une instruction **SELECT**. La table cible ne doit pas apparaître dans *`liste_from`*, sauf en cas d'auto-jointure (auquel cas elle doit apparaître avec un alias dans *`liste_from`*).

`condition`
: Une expression qui renvoie une valeur de type boolean. Seules les lignes pour lesquelles cette expression renvoie `true` sont mises à jour.

`nom_curseur`
: Le nom du curseur à utiliser dans une condition `WHERE CURRENT OF`. La ligne à mettre à jour est la dernière récupérée à partir de ce curseur. Le curseur doit être une requête sans regroupement sur la table cible de l'**UPDATE**. Notez que `WHERE`

CURRENT OF ne peut pas être spécifié avec une condition booléenne. Voir DECLARE(7) pour plus d'informations sur l'utilisation des curseurs avec WHERE CURRENT OF.

expression_sortie
Une expression à calculer et renvoyée par la commande **UPDATE** après chaque mise à jour de ligne. L'expression peut utiliser tout nom de colonne de la *table* ou des tables listées dans le FROM. Indiquez * pour que toutes les colonnes soient renvoyées.

nom_sortie
Un nom à utiliser pour une colonne renvoyée.

Sorties

En cas de succès, une commande **UPDATE** renvoie un message de la forme

```
UPDATE total
```

total est le nombre de lignes mises à jour. S'il vaut 0, c'est qu'aucune ligne ne correspondait à *condition* (ce qui n'est pas considéré comme une erreur).

Notes

Lorsqu'une clause FROM est précisée, la table cible est jointe aux tables mentionnées dans *liste_from*, et chaque ligne en sortie de la jointure représente une opération de mise à jour pour la table cible. Lors de l'utilisation de FROM, il faut s'assurer que la jointure produit au plus une ligne en sortie par ligne à modifier. En d'autres termes, une ligne cible ne doit pas être jointe à plus d'une ligne des autres tables. Le cas échéant, seule une ligne de jointure est utilisée pour mettre à jour la ligne cible, mais il n'est pas possible de prédire laquelle.

À cause de ce manque de déterminisme, il est plus sûr de ne référencer les autres tables qu'à l'intérieur de sous-requêtes. Même si c'est plus difficile à lire et souvent plus lent que l'utilisation d'une jointure.

Si la commande **UPDATE** contient une clause RETURNING, le résultat sera similaire à celui d'une instruction **SELECT** contenant les colonnes et les valeurs définies dans la liste RETURNING, à partir de la liste des lignes mises à jour par la commande.

Exemples

Changer le mot Drame en Dramatique dans la colonne *genre* de la table films :

```
UPDATE films SET genre = 'Dramatique' WHERE genre = 'Drame';
```

Ajuster les entrées de température et réinitialiser la précipitation à sa valeur par défaut dans une ligne de la table temps :

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse+15, prcp = DEFAULT
  WHERE ville = 'San Francisco' AND date = '2005-07-03';
```

Réaliser la même opération et renvoyer les lignes mises à jour :

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse+15, prcp = DEFAULT
  WHERE ville = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_basse, temp_haute, prcp;
```

Utiliser une autre syntaxe pour faire la même mise à jour :

```
UPDATE temps SET (temp_basse, temp_haute, prcp) = (temp_basse+1, temp_basse+15,
DEFAULT)
  WHERE ville = 'San Francisco' AND date = '2003-07-03';
```

Incrémenter le total des ventes de la personne qui gère le compte d'Acme Corporation, à l'aide de la clause FROM :

```
UPDATE employes SET total_ventes = total_ventes + 1 FROM comptes
  WHERE compte.nom = 'Acme Corporation'
  AND employes.id = compte.vendeur;
```

Réaliser la même opération en utilisant une sous-requête dans la clause WHERE :

```
UPDATE employes SET total_ventes = total_ventes + 1 WHERE id =
  (SELECT vendeur FROM comptes WHERE nom = 'Acme Corporation');
```

Tenter d'insérer un nouvel élément dans le stock avec sa quantité. Si l'élément existe déjà, mettre à jour le total du stock de

l'élément. Les points de sauvegarde sont utilisés pour ne pas avoir à annuler l'intégralité de la transaction en cas d'erreur :

```
BEGIN;
-- autres opérations
SAVEPOINT sp1;
INSERT INTO vins VALUES('Chateau Lafite 2003', '24');
-- A supposer que l'instruction ci-dessus échoue du fait d'une violation de clé
-- unique, les commandes suivantes sont exécutées :
ROLLBACK TO sp1;
UPDATE vins SET stock = stock + 24 WHERE nomvin = 'Chateau Lafite 2003';
-- continuer avec les autres opérations, et finir
COMMIT;
```

Modifier la colonne *genre* de la table films dans la ligne où le curseur `c_films` est actuellement positionné :

```
UPDATE films SET genre = 'Dramatic' WHERE CURRENT OF c_films;
```

Compatibilité

Cette commande est conforme au standard SQL, à l'exception des clauses `FROM` et `RETURNING` qui sont des extensions PostgreSQL™.

D'après le standard, la syntaxe de la liste de colonnes permet qu'une liste de colonnes soit affectée à une expression de ligne comme une sous-sélection :

```
UPDATE comptes SET (nom_contact, prenom_contact) =
    (SELECT nom, prenom FROM commerciaux
     WHERE commerciaux.id = comptes.vendeur_id);
```

Ceci n'est pas encore implémenté -- la source doit être une liste d'expressions indépendantes.

D'autres systèmes de bases de données offrent l'option `FROM` dans laquelle la table cible est supposée être à nouveau indiquée dans le `FROM`. PostgreSQL™ n'interprète pas la clause `FROM` ainsi. Il est important d'en tenir compte lors du portage d'applications qui utilisent cette extension.

Nom

VACUUM — récupère l'espace inutilisé et, optionnellement, analyse une base

Synopsis

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ table [ (colonne [,
...] ) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (colonne [, ...] ) ] ]
```

Description

VACUUM récupère l'espace de stockage occupé par des lignes mortes. Lors des opérations normales de PostgreSQL™, les lignes supprimées ou rendues obsolètes par une mise à jour ne sont pas physiquement supprimées de leur table. Elles restent présentes jusqu'à ce qu'un **VACUUM** soit lancé. C'est pourquoi, il est nécessaire de faire un **VACUUM** régulièrement, spécialement sur les tables fréquemment mises à jour.

Sans paramètre, **VACUUM** traite toutes les tables de la base de données courante pour lequel l'utilisateur connecté dispose du droit d'exécution du VACUUM. Avec un paramètre, **VACUUM** ne traite que cette table.

VACUUM ANALYZE fait un **VACUUM**, puis un **ANALYZE** sur chaque table sélectionnée. C'est une combinaison pratique pour les scripts de maintenance de routine. Voir ANALYZE(7) pour avoir plus de détails sur ce qu'il traite.

Le **VACUUM** standard (sans `FULL`) récupère simplement l'espace et le rend disponible pour une réutilisation. Cette forme de la commande peut opérer en parallèle avec les opérations normales de lecture et d'écriture de la table, car elle n'utilise pas de verrou exclusif. Néanmoins, l'espace récupéré n'est pas renvoyé au système de fichiers dans la plupart des cas ; il est conservé pour être réutilisé dans la même table. **VACUUM FULL** ré-écrit le contenu complet de la table dans un nouveau fichier sur disque sans perte d'espace, permettant à l'espace inutilisé d'être retourné au système d'exploitation. Cette forme est bien plus lente et nécessite un verrou exclusif sur chaque table le temps de son traitement.

Quand la liste d'options est entourée de parenthèses, les options peuvent être écrites dans n'importe quel ordre. Sans parenthèses, les options doivent être écrit dans l'ordre exact décrit ci-dessus. Avant PostgreSQL™ 9.0, la syntaxe sans parenthèses était la seule supportée. Il est attendu que toutes les nouvelles options ne seront supportées que par la syntaxe entre parenthèses.

Paramètres

`FULL`
: Choisit un vacuum « full », qui récupère plus d'espace, mais est beaucoup plus long et prend un verrou exclusif sur la table. Cette méthode requiert aussi un espace disque supplémentaire car il écrit une nouvelle copie de la table et ne supprime l'ancienne copie qu'à la fin de l'opération. Habituellement, cela doit seulement être utilisé quand une quantité importante d'espace doit être récupérée de la table.

`FREEZE`
: Choisit un « gel » agressif des lignes. Indiquer `FREEZE` est équivalent à réaliser un **VACUUM** avec le paramètre vacuum_freeze_min_age configuré à zéro. L'option `FREEZE` est obsolète et sera supprimée dans une version future ; configurez ce paramètre à la place.

`VERBOSE`
: Affiche un rapport détaillé de l'activité de vacuum sur chaque table.

`ANALYZE`
: Met à jour les statistiques utilisées par l'optimiseur pour déterminer la méthode la plus efficace pour exécuter une requête.

`table`
: Le nom (optionnellement qualifié par le nom d'un schéma) d'une table à traiter par vacuum. Par défaut, toutes les tables de la base de données courante sont traitées.

`colonne`
: Le nom d'une colonne spécifique à analyser. Par défaut, toutes les colonnes. Si une liste de colonnes est spécifiée, `ANALYZE` en est déduit.

Sorties

Lorsque `VERBOSE` est précisé, **VACUUM** indique sa progression par des messages indiquant la table en cours de traitement.

Différentes statistiques sur les tables sont aussi affichées.

Notes

Pour exécuter un VACUUM sur une table, vous devez habituellement être le propriétaire de la table ou un superutilisateur. Néanmoins, les propriétaires de la base de données sont autorisés à exécuter VACUUM sur toutes les tables de leurs bases de données, sauf sur les catalogues partagés. Cette restriction signifie qu'un vrai **VACUUM** sur une base complète ne peut se faire que par un superutilisateur.) **VACUUM** ignorera toutes les tables pour lesquelles l'utilisateur n'a pas le droit d'exécuter un VACUUM.

VACUUM ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Pour les tables ayant des index GIN, **VACUUM** (sous n'importe quelle forme) termine aussi toutes les insertions d'index en attente, en déplaçant les entrées d'index aux bons endroits dans la structure d'index GIN principale. Voir Section 53.3.1, « Technique GIN de mise à jour rapide » pour les détails.

Nous recommandons que les bases de données actives de production soient traitées par vacuum fréquemment (au moins toutes les nuits), pour supprimer les lignes mortes. Après avoir ajouté ou supprimé un grand nombre de lignes, il peut être utile de faire un **VACUUM ANALYZE** sur la table affectée. Cela met les catalogues système à jour de tous les changements récents et permet à l'optimiseur de requêtes de PostgreSQL™ de faire de meilleurs choix lors de l'optimisation des requêtes.

L'option `FULL` n'est pas recommandée en usage normal, mais elle peut être utile dans certains cas. Par exemple, si vous avez supprimé ou mis à jour l'essentiel des lignes d'une table et si vous voulez que la table diminue physiquement sur le disque pour n'occuper que l'espace réellement nécessaire et pour que les parcours de table soient plus rapides. Généralement, **VACUUM FULL** réduit plus la table qu'un simple **VACUUM**.

VACUUM peut engendrer une augmentation substantielle du trafic en entrées/sorties pouvant causer des performances diminuées pour les autres sessions actives. Du coup, il est quelque fois conseillé d'utiliser la fonctionnalité du délai du vacuum basé sur le coût. Voir Chapitre 17, Configuration du serveur et mise en place pour des informations supplémentaires.

PostgreSQL™ inclut un « autovacuum » qui peut automatiser la maintenance par VACUUM. Pour plus d'informations sur le VACUUM automatique et manuel, voir Section 23.1, « Nettoyages réguliers ».

Exemples

Ce qui suit est un exemple de lancement de **VACUUM** sur une table de la base de données regression.

```
regression=# VACUUM (VERBOSE, ANALYZE) onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_stringu1" now contains 1000 tuples in 48 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

Compatibilité

Il n'y a pas de commande **VACUUM** dans le standard SQL.

Voir aussi

vacuumdb(1), Section 18.4.3, « Report du VACUUM en fonction de son coût », Section 23.1.5, « Le démon auto-vacuum »

Nom

VALUES — calcule un ensemble de lignes

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY expression_de_tri [ ASC | DESC | USING operateur ] [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET debut ] [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ nombre ] { ROW | ROWS } ONLY ]
```

Description

VALUES calcule une valeur de ligne ou un ensemble de valeurs de lignes spécifiées par des expressions. C'est généralement utilisé pour générer une « table statique » à l'intérieur d'une commande plus large mais elle peut aussi être utilisée séparément.

Quand plus d'une ligne est indiquée, toutes les lignes doivent avoir le même nombre d'éléments. Les types de données des colonnes de la table résultante sont déterminés en combinant les types explicites et les types inférés des expressions apparaissant dans cette colonne, en utilisant les mêmes règles que pour l'`UNION` (voir Section 10.5, « Constructions `UNION`, `CASE` et constructions relatives »).

À l'intérieur de grosses commandes, **VALUES** est autorisé au niveau de la syntaxe partout où la commande **SELECT** l'est. Comme la grammaire traite cette commande comme un **SELECT**, il est possible d'utiliser les clauses `ORDER BY`, `LIMIT` (ou de façon équivalente `FETCH FIRST`) et `OFFSET` avec une commande **VALUES**.

Paramètres

`expression`
: Une constante ou une expression à calculer et à insérer à l'emplacement indiqué dans la table résultante (ensemble de lignes). Dans une liste **VALUES** apparaissant en haut d'une commande **INSERT**, une *`expression`* peut être remplacée par `DEFAULT` pour demander l'insertion de la valeur par défaut de la colonne de destination. `DEFAULT` ne peut pas être utilisé quand **VALUES** apparaît dans d'autres contextes.

`expression_de_tri`
: Une expression ou un entier indiquant comment trier les lignes de résultat. Cette expression peut faire référence aux colonnes de **VALUES** en tant que `column1`, `column2`, etc. Pour plus de détails, voir la section intitulée « Clause `ORDER BY` ».

`operateur`
: Un opérateur de tri. Pour plus de détails, voir la section intitulée « Clause `ORDER BY` ».

`nombre`
: Le nombre maximum de lignes à renvoyer. Pour plus de détails, voir la section intitulée « Clause `LIMIT` ».

`debut`
: Le nombre de lignes à échapper avant de commencer à renvoyer des lignes. Pour plus de détails, la section intitulée « Clause `LIMIT` ».

Notes

Évitez les listes **VALUES** comprenant un très grand nombre de lignes car vous pourriez rencontrer des problèmes comme un manque de mémoire et/ou des performances pauvres. Un **VALUES** apparaissant dans un **INSERT** est un cas spécial (parce que le type des colonnes est trouvé à partir de la table cible du **INSERT** et n'a donc pas besoin d'être deviné en parcourant la liste **VALUES**), du coup il peut gérer des listes plus importantes que dans d'autres contextes.

Exemples

Une simple commande **VALUES** :

```
VALUES (1, 'un'), (2, 'deux'), (3, 'trois');
```

Ceci renverra une table statique comprenant deux colonnes et trois lignes. En fait, c'est équivalent à :

```
SELECT 1 AS column1, 'un' AS column2
UNION ALL
SELECT 2, 'deux'
UNION ALL
SELECT 3, 'trois';
```

Plus généralement, **VALUES** est utilisé dans une commande SQL plus importante. L'utilisation la plus fréquente est dans un **INSERT** :

```
INSERT INTO films (code, titee, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drame');
```

Dans le contexte de la commande **INSERT**, les entrées d'une liste **VALUES** peuvent être `DEFAULT` pour indiquer que la valeur par défaut de la colonne ciblée doit être utilisée :

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comédie', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame', DEFAULT);
```

VALUES peut aussi être utilisé là où un sous-**SELECT** peut être écrit, par exemple dans une clause `FROM` :

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horreur'), ('UA', 'Sci-Fi')) AS t (studio, genre)
  WHERE f.studio = t.studio AND f.genre = t.genre;

UPDATE employes SET salaire = salaire * v.augmentation
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (no_dep, cible, augmentation)
  WHERE employees.no_dep = v.no_dep AND employees.ventes >= v.cible;
```

Notez qu'une clause `AS` est requise quand **VALUES** est utilisé dans une clause `FROM`, par exemple dans un **SELECT**. Il n'est pas nécessaire de spécifier les noms de toutes les colonnes dans une clause `AS` c'est une bonne pratique (les noms des colonnes par défaut pour **VALUES** sont `column1`, `column2`, etc dans PostgreSQL™ mais ces noms pourraient être différents dans d'autres SGBD).

Quand **VALUES** est utilisé dans **INSERT**, les valeurs sont toutes automatiquement converties dans le type de données de la colonne destination correspondante. Quand elle est utilisée dans d'autres contextes, il pourrait être nécessaire de spécifier le bon type de données. Si les entrées sont toutes des constantes litérales entre guillemets, convertir la première est suffisante pour déterminer le type de toutes :

```
SELECT * FROM machines
WHERE adresse_ip IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

Astuce

Pour de simples tests `IN`, il est préférable de se baser sur des listes de valeurs pour `IN` que d'écrire une requête **VALUES** comme indiquée ci-dessus. La méthode des listes de valeurs simples requiert moins d'écriture et est souvent plus efficace.

Compatibilité

VALUES est conforme au standard SQL. Les clauses `LIMIT` et `OFFSET` sont des extensions PostgreSQL™ ; voir aussi SELECT(7).

Voir aussi

INSERT(7), SELECT(7)

Applications client de PostgreSQL

Cette partie contient les informations de référence concernant les applications client et les outils de PostgreSQL™. Ces commandes ne sont pas toutes destinées à l'ensemble des utilisateurs. Certaines nécessitent des privilèges spécifiques. La caractéristique commune à toutes ces applications est leur fonctionnement sur toute machine, indépendemment du serveur sur lequel se trouve le serveur de base de données.

Nom

clusterdb — Grouper une base de données PostgreSQL™

Synopsis

`clusterdb` [*options_connexion*...] [[--verbose] | [-v]] [--table | -t *table*] [*nom_bd*]

`clusterdb` [*options_connexion*...] [[--verbose] | [-v]] [[--all] | [-a]]

Description

clusterdb est un outil de regroupage de tables au sein d'une base de données PostgreSQL™. Il trouve les tables précédemment groupées et les groupe à nouveau sur l'index utilisé lors du groupement initial. Les tables qui n'ont jamais été groupées ne sont pas affectées.

clusterdb est un enrobage de la commande SQL CLUSTER(7). Il n'y a pas de différence réelle entre le groupage de bases par cet outil ou par d'autres méthodes d'accès au serveur.

Options

clusterdb accepte les arguments suivants en ligne de commande :

`-a`, `--all`
: Grouper toutes les bases de données.

`[-d] nom_bd`, `[--dbname] nom_bd`
: Le nom de la base de données à grouper. Si ni ce nom, ni l'option `-a` (ou `--all`) ne sont précisés, le nom de la base de données est lu à partir de la variable d'environnement `PGDATABASE`. Si cette dernière n'est pas initialisée, le nom de l'utilisateur spécifié pour la connexion est utilisé.

`-e`, `--echo`
: Les commandes engendrées par clusterdb et envoyées au serveur sont affichées.

`-q`, `--quiet`
: Aucun message de progression n'est affiché.

`-t table`, `--table table`
: Seule la table *table* est groupée.

`-v`, `--verbose`
: Affiche des informations détaillées lors du traitement.

`-V`, `--version`
: Affiche la version de clusterdb puis quitte.

`-?`, `--help`
: Affiche l'aide sur les arguments en ligne de commande de clusterdb, puis quitte

clusterdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h hôte`, `--host hôte`
: Le nom de la machine hôte sur laquelle le serveur fonctionne. Si la valeur commence par une barre oblique (slash), elle est utilisée comme répertoire du socket de domaine Unix.

`-p port`, `--port port`
: Le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U nomutilisateur`, `--username nomutilisateur`
: Le nom de l'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
: Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
: Force clusterdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car clusterdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, clusterdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de difficulté, voir CLUSTER(7) et psql(1) qui présentent les problèmes et messages d'erreur éventuels. Le serveur de bases de données doit fonctionner sur l'hôte cible. De plus, toutes les configurations de connexion par défaut et variables d'environnement utilisées par la bibliothèque client libpq s'appliquent.

Exemples

Grouper la base de données `test` :

```
$ clusterdb test
```

Grouper la seule table `foo` de la base de données nommée `xyzzy` :

```
$ clusterdb --table foo xyzzy
```

Voir aussi

CLUSTER(7)

Nom

createdb — Créer une nouvelle base de données PostgreSQL™

Synopsis

`createdb` [*option_connexion*...] [*option*...] [*nombase*] [*description*]

Description

createdb crée une nouvelle base de données.

Normalement, l'utilisateur de la base de données qui exécute cette commande devient le propriétaire de la nouvelle base de données. Néanmoins, un propriétaire différent peut être spécifié via l'option `-O`, sous réserve que l'utilisateur qui lance la commande ait les droits appropriés.

createdb est un enrobage de la commande SQL CREATE DATABASE(7). Il n'y a pas de réelle différence entre la création de bases de données par cet outil ou à l'aide d'autres méthodes d'accès au serveur.

Options

createdb accepte les arguments suivants en ligne de commande :

nombase
: Le nom de la base de données à créer. Le nom doit être unique parmi toutes les bases de données PostgreSQL™ de ce groupe. La valeur par défaut est le nom de l'utilisateur courant.

description
: Le commentaire à associer à la base de données créée.

`-D` *tablespace*, `--location` *tablespace*
: Le tablespace par défaut de la base de données.

`-e`, `--echo`
: Les commandes engendrées par createdb et envoyées au serveur sont affichées.

`-l` *locale*, `--locale` *locale*
: Indique la locale à utiliser dans cette base de données. C'est équivalent à préciser à la fois `--lc-collate` et `--lc-ctype`.

`--lc-collate` *locale*
: Indique le paramètre LC_COLLATE utilisé pour cette base de données.

`--lc-ctype` *locale*
: Indique le paramètre LC_CTYPE utilisé pour cette base de données.

`-E` *locale*, `--encoding` *locale*
: L'encodage des caractères à utiliser dans la base de données. Les jeux de caractères supportés par le serveur PostgreSQL™ sont décrits dans Section 22.2.1, « Jeux de caractères supportés ».

`-O` *propriétaire*, `--owner` *propriétaire*
: Le propriétaire de la base de données.

`-T` *modèle*, `--template` *modèle*
: La base de données modèle.

`-V`, `--version`
: Affiche la version de createdb puis quitte.

`-?`, `--help`
: Affiche l'aide sur les arguments en ligne de commande de createdb, puis quitte

Les options `-D`, `-l`, `-E`, `-O` et `-T` correspondent aux options de la commande SQL sous-jacente CREATE DATABASE(7), à consulter pour plus d'informations sur ces options.

createdb accepte aussi les arguments suivants en ligne de commande, pour les paramètres de connexion :

`-h` *hôte*, `--host` *hôte*

Le nom de l'hôte sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash (NDT : barre oblique, /), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*, `--port` *`port`*
Le port TCP ou l'extension du fichier socket de domaine Unix local sur lequel le serveur attend les connexions.

`-U` *`nomutilisateur`*, `--username` *`nomutilisateur`*
Le nom de l'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force createdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car createdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, createdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`
S'il est configuré, précise le nom de la base de données à créer. Peut-être surchargé sur la ligne de commande.

`PGHOST`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut. `PGUSER` détermine aussi le nom de la base de données à créer si ce dernier n'est pas spécifié sur la ligne de commande ou par `PGDATABASE`.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de difficulté, on peut se référer à CREATE DATABASE(7) et psql(1) qui présentent les problèmes éventuels et les messages d'erreurs. Le serveur de bases de données doit être en cours d'exécution sur l'hôte cible. De plus, tous les paramètres de connexion et variables d'environnement par défaut utilisés par la bibliothèque d'interface libpq s'appliquent.

Exemples

Créer la base de données `demo` sur le serveur de bases de données par défaut :

```
$ createdb demo
```

Créer la base de données `demo` sur le serveur hébergé sur l'hôte `eden`, port 5000, en utilisant l'encodage `LATIN1` avec affichage de la commande engendrée :

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" ENCODING 'LATIN1'
```

Voir aussi

dropdb(1), CREATE DATABASE(7)

Nom

createlang — Définir un langage procédural sous PostgreSQL™

Synopsis

`createlang` [*`options_connexion`*...] *`nom_langage`* [*`nom_bd`*]
`createlang` [*`options_connexion`*...] [--list] | [-l] *`nom_bd`*

Description

createlang permet d'ajouter un nouveau langage de programmation à une base de données PostgreSQL™. createlang n'est qu'un enrobage de la commande CREATE LANGUAGE(7).

Options

createlang accepte les arguments suivants en ligne de commande :

`nom_langage`
Le nom du langage de programmation procédurale à définir.

`[-d]` *`nom_nd`*, `[--dbname]` *`nom_bd`*
La base de données à laquelle ajouter le langage. Par défaut, celle de même nom que l'utilisateur système.

`-e`, `--echo`
Les commandes SQL exécutées sont affichées.

`-l`, `--list`
La liste de langages installés sur la base de données cible est affichée.

`-V`, `--version`
Affiche la version de createlang puis quitte.

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de createlang, puis quitte

createlang accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*, `--host` *`hôte`*
Le nom de l'hôte de la machine sur laquelle le serveur fonctionne. Si la valeur commence par un slash (/), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*, `--port` *`port`*
Le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *`nomutilisateur`*, `--username` *`nomutilisateur`*
Le nom de l'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force createlang à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car createlang demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, createlang perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

La plupart des messages d'erreur s'expliquent d'eux-mêmes. Dans le cas contraire, createlang peut être lancée avec l'option `--echo` afin d'obtenir les commandes SQL à examiner. De plus, tout paramètre de connexion par défaut et toute variable d'environnement utilisé par la bibliothèque libpq s'appliqueront.

Notes

droplang(1) est utilisé pour supprimer un langage.

Exemples

Installer le langage `pltcl` dans la base de données `template1` :

```
$ createlang pltcl template1
```

Installer un langage dans `template1` l'installe automatiquement dans les bases de données créées ultérieurement.

Voir aussi

droplang(1), CREATE LANGUAGE(7), Variables d'environnement (Section 31.13, « Variables d'environnement »)

Nom

createuser — Définir un nouveau compte utilisateur PostgreSQL™

Synopsis

`createuser` [*option_connexion*...] [*option*...] [*nom_utilisateur*]

Description

createuser crée un nouvel utilisateur PostgreSQL™ (ou, plus précisément, un rôle). Seuls les superutilisateurs et les utilisateurs disposant du droit `CREATEROLE` peuvent créer de nouveaux utilisateurs. createuser ne peut de ce fait être invoqué que par un utilisateur pouvant se connecter en superutilisateur ou en utilisateur ayant le droit `CREATEROLE`.

Pour créer un superutilisateur, il est impératif de se connecter en superutilisateur ; la simple connexion en utilisateur disposant du droit `CREATEROLE` n'est pas suffisante. Être superutilisateur implique la capacité d'outrepasser toutes les vérifications de droits d'accès à la base de données ; les privilèges de superutilisateur ne peuvent pas être accordés à la légère.

createuser est un enrobage de la commande SQL CREATE ROLE(7). Il n'y a pas de différence réelle entre la création d'utilisateurs par cet outil ou au travers d'autres méthodes d'accès au serveur.

Options

createuser accepte les arguments suivant en ligne de commande

nom_utilisateur
: Le nom de l'utilisateur à créer. Ce nom doit être différent de tout rôle de l'instance courante de PostgreSQL™.

`-c` *numéro*, `--connection-limit` *numéro*
: Configure le nombre maximum de connexions simultanées pour le nouvel utilisateur. Par défaut, il n'y a pas de limite.

`-d`, `--createdb`
: Le nouvel utilisateur est autorisé à créer des bases de données.

`-D`, `--no-createdb`
: Le nouvel utilisateur n'est pas autorisé à créer des bases de données. Cela correspond au comportement par défaut.

`-e`, `--echo`
: Les commandes engendréee par createuser et envoyées au serveur sont affichées.

`-E`, `--encrypted`
: Le mot de passe de l'utilisateur est stocké chiffré dans la base. Si cette option n'est pas précisée, la gestion par défaut des mots de passe est utilisée.

`-i`, `--all`
: Le nouveau rôle hérite automatiquement des droits des rôles dont il est membre. Comportement par défaut.

`-I`, `--no-all`
: Le nouveau rôle n'hérite pas automatiquement des droits des rôles dont il est membre.

`-l`, `--login`
: Le nouvel utilisateur est autorisé à se connecter (son nom peut être utilisé comme identifiant initial de session). Comportement par défaut.

`-L`, `--no-login`
: Le nouvel utilisateur n'est pas autorisé à se connecter. (Un rôle sans droit de connexion est toujours utile pour gérer les droits de la base de données.)

`-N`, `--unencrypted`
: Le mot de passe de l'utilisateur n'est pas stocké chiffré. Si cette option n'est pas précisée, la gestion par défaut des mots de passe est utilisée.

`-P`, `--pwprompt`
: L'utilisation de cette option impose à createuser d'afficher une invite pour la saisie du mot de passe du nouvel utilisateur. Cela n'a pas d'utilité si l'authentification par mot de passe n'est pas envisagée.

`-r`, `--createrole`
: Le nouvel utilisateur est autorisé à créer de nouveaux rôles (il possède le privilège `CREATEROLE`).

`-R`, `--no-createrole`
Le nouvel utilisateur n'est pas autorisé à créer de nouveaux rôles. Cela correspond au comportement par défaut.

`-s`, `--superuser`
Le nouvel utilisateur a les privilèges superutilisateur.

`-S`, `--no-superuser`
Le nouvel utilisateur n'a pas les privilèges superutilisateur. Cela correspond au comportement par défaut.

Le nom, ou toute autre information manquante non fournie sur la ligne de commande, est automatiquement demandée.

createuser accepte aussi les arguments suivant en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*, `--host` *`hôte`*
Le nom de l'hôte sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash (/), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*, `--port` *`port`*
Le port TCP ou l'extension du fichier socket de domaine Unix sur lequel le serveur attend des connexions.

`-U` *`nomutilisateur`*, `--username` *`nomutilisateur`*
Nom de l'utilisateur utilisé pour la connexion (pas celui à créer).

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force createuser à demander un mot de passe (pour la connexion au serveur, pas pour le mot de passe du nouvel utilisateur).

Cette option n'est jamais obligatoire car createuser demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, createuser perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGHOST`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de problèmes, on peut consulter CREATE ROLE(7) et psql(1) qui fournissent des informations sur les problèmes potentiels et les messages d'erreur. Le serveur de la base de données doit être en cours d'exécution sur l'hôte cible. De plus, tout paramétrage de connexion par défaut et toute variable d'environnement utilisée par le client de la bibliothèque libpq s'applique.

Exemples

Créer un utilisateur `joe` sur le serveur de bases de données par défaut :

```
$ createuser joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

Créer le même utilisateur, `joe`, sur le serveur `eden`, port 5000, sans interaction, avec affichage de la commande sous-jacente :

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Créer l'utilisateur `joe`, superutilisateur, et lui affecter immédiatement un mot de passe :

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
```

```
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'xyzzy' SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
CREATE ROLE
```

Dans l'exemple ci-dessus, le nouveau mot de passe n'est pas affiché lorsqu'il est saisi. Il ne l'est ici que pour plus de clareté. Comme vous le voyez, le mot de passe est chiffré avant d'être envoyé au client. Si l'option `--unencrypted` est utilisé, le mot de passe *apparaîtra* dans la commande affichée (et aussi dans les journaux applicatifs et certainement ailleurs), donc vous ne devez pas utiliser `-e` dans ce cas, surtout si quelqu'un d'autre voit votre écran à ce moment.

Voir aussi

dropuser(1), CREATE ROLE(7)

Nom

dropdb — Supprimer une base de données PostgreSQL™

Synopsis

`dropdb` [*option_connexion*...] [*option*...] *nom_bd*

Description

dropdb détruit une base de données PostgreSQL™. L'utilisateur qui exécute cette commande doit être superutilisateur ou le propriétaire de la base de données.

dropdb est un enrobage de la commande SQL DROP DATABASE(7). Il n'y a aucune différence réelle entre la suppression de bases de données avec cet outil et celles qui utilisent d'autres méthodes d'accès au serveur.

Options

dropdb accepte les arguments suivants en ligne de commande :

nom_bd
: Le nom de la base de données à supprimer.

`-e`, `--echo`
: Les commandes engendrées et envoyées au serveur par dropdb sont affichées.

`-i`, `--interactive`
: Une confirmation préalable à toute destruction est exigée.

`-V`, `--version`
: Affiche la version de dropdb puis quitte.

`-?`, `--help`
: Affiche l'aide sur les arguments en ligne de commande de dropdb, puis quitte

dropdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*, `--host` *hôte*
: Le nom d'hôte de la machine sur laquelle le serveur fonctionne. Si la valeur débute par une barre oblique (/ ou slash), elle est utilisée comme répertoire de la socket de domaine Unix.

`-p` *port*, `--port` *port*
: Le port TCP ou l'extension du fichier de la socket locale de domaine Unix sur laquelle le serveur attend les connexions.

`-U` *nomutilisateur*, `--username` *nomutilisateur*
: Le nom de l'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
: Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
: Force dropdb à demander un mot de passe avant la connexion à une base de données.

  Cette option n'est jamais obligatoire car dropdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, dropdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGHOST`, `PGPORT`, `PGUSER`
: Paramètres de connexion par défaut

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la biblio-

thèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de difficultés, il peut être utile de consulter DROP DATABASE(7) et psql(1), sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Exemples

Détruire la base de données `demo` sur le serveur de bases de données par défaut :

```
$ dropdb demo
```

Détruire la base de données `demo` en utilisant le serveur hébergé sur l'hôte `eden`, qui écoute sur le port 5000, avec demande de confirmation et affichage de la commande sous-jacente :

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

Voir aussi

createdb(1), DROP DATABASE(7)

Nom

droplang — Supprimer un langage procédural

Synopsis

`droplang` [*option_connexion*...] *nom_langage* [*nom_bd*]

`droplang` [*option_connexion*...] [--list] | [-l] *nom_db*

Description

droplang permet de supprimer un langage procédural. Tout langage procédural, même s'il n'est pas fourni avec la distribution de PostgreSQL™ est candidat à la suppression par droplang.

Bien que les langages de programmation du serveur puissent être supprimés directement à l'aide de quelques commandes SQL, il est recommandé d'utiliser droplang. Non seulement, il réalise quelques vérifications, mais il est aussi plus simple à utiliser. Voir DROP LANGUAGE(7) pour plus d'informations.

Options

droplang accepte les arguments en ligne de commande :

nom_langage
: Le nom du langage de programmation à supprimer.

`[-d]` *nom_bd*, `[--dbname]` *nom_bd*
: La base de données qui contient le langage à supprimer. Par défaut, le nom de la base est équivalent à celui du nom de l'utilisateur système qui lance la commande.

`-e`, `--echo`
: Les commandes SQL exécutées sont affichées.

`-l`, `--list`
: La liste des langages installés sur la base de données cible est affiché.

`-V`, `--version`
: Affiche la version de droplang puis quitte.

`-?`, `--help`
: Affiche l'aide sur les arguments en ligne de commande de droplang, puis quitte

droplang accepte aussi les arguments suivants sur la ligne de commande pour les paramètres de connexion :

`-h` *hôte*, `--host` *hôte*
: Le nom d'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence par une barre oblique (/ ou slash), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*, `--port` *port*
: Le port TCP ou l'extension du fichier de la socket de domaine Unix sur lequel le serveur écoute les connexions.

`-U` *nomutilisateur*, `--username` *nomutilisateur*
: Le nom de l'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
: Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
: Force droplang à demander un mot de passe avant la connexion à une base de données.

  Cette option n'est jamais obligatoire car droplang demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, droplang perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
: Paramètres de connexion par défaut

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

La plupart des messages d'erreurs sont explicites. Dans le cas contraire, on peut utiliser droplang avec l'option `--echo` et regarder la commande SQL correspondante pour obtenir plus de détails. De plus, tout paramètre de connexion par défaut et toute variable d'environnement utilisé par la bibliothèque libpq s'appliqueront.

Notes

createlang(1) est utilisé pour ajouter un langage.

Exemples

Supprimer le langage `pltcl` :

```
$ droplang pltcl nomdb
```

Voir aussi

createlang(1), DROP LANGUAGE(7)

Nom

dropuser — Supprimer un compte utilisateur PostgreSQL™

Synopsis

`dropuser` [*option_connexion*...] [*option*...] [*nomutilisateur*]

Description

dropuser supprime un utilisateur. Seuls les superutilisateurs et les utilisateurs disposant du droit `CREATEROLE` peuvent supprimer des utilisateurs (seul un superutilisateur peut supprimer un superutilisateur).

dropuser est un enrobage de la commande SQL DROP ROLE(7). Il n'y a pas de différence réelle entre la suppression des utilisateurs à l'aide de cet outil ou à l'aide d'autres méthodes d'accès au serveur.

Options

dropuser accepte les arguments suivants en ligne de commande :

nomutilisateur
: Le nom de l'utilisateur PostgreSQL™ à supprimer. Un nom est demandé s'il n'est pas fourni sur la ligne de commande.

`-e`, `--echo`
: Les commandes engendrées et envoyées au serveur par dropuser sont affichées.

`-i`, `--interactive`
: Une confirmation est demandée avant la suppression effective de l'utilisateur.

`-V`, `--version`
: Affiche la version de dropuser puis quitte.

`-?`, `--help`
: Affiche l'aide sur les arguments en ligne de commande de dropuser, puis quitte

dropuser accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*, `--host` *hôte*
: Le nom d'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence par une barre oblique (/ ou slash), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *port*, `--port` *port*
: Le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *nomutilisateur*, `--username` *nomutilisateur*
: Le nom de l'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
: Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
: Force dropuser à demander un mot de passe avant la connexion à une base de données.

  Cette option n'est jamais obligatoire car dropuser demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, dropuser perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
: Paramètres de connexion par défaut.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la biblio-

thèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de difficultés, il peut être utile de consulter DROP ROLE(7) et psql(1), sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Exemples

Supprimer l'utilisateur `joe` de la base de données par défaut :

```
$ dropuser joe
```

Supprimer l'utilisateur `joe` sur le serveur hébergé sur l'hôte `eden`, qui écoute sur le port 5000, avec demande de confirmation et affichage de la commande sous-jacente :

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

Voir aussi

createuser(1), DROP ROLE(7)

Nom

ecpg — Préprocesseur C pour le SQL embarqué

Synopsis

`ecpg` [*option*...] *fichier*...

Description

ecpg est le préprocesseur du SQL embarqué pour les programmes écrits en C. Il convertit des programmes écrits en C contenant des instructions SQL embarqué en code C normal. Pour se faire, les appels au SQL sont remplacés par des appels spéciaux de fonctions. Les fichiers en sortie peuvent être traités par n'importe quel compilateur C.

ecpg convertit chaque fichier en entrée, donné sur la ligne de commande, en un fichier C correspondant. Les fichiers en entrée ont de préférence l'extension `.pgc`, auquel cas l'extension est remplacée par `.c` pour former le nom du fichier de sortie. Si l'extension du fichier en entrée n'est pas `.pgc`, alors le nom de fichier en sortie est obtenu en ajoutant `.c` au nom complet du fichier. Le nom de fichier en sortie peut aussi être surchargé en utilisant l'option `-o`.

Cette page de référence ne décrit pas le langage SQL embarqué. Voir Chapitre 33, ECPG - SQL embarqué dans du C pour plus d'informations sur ce thème.

Options

ecpg accepte les arguments suivants en ligne de commande :

`-c`
Engendre automatiquement du code C à partir de code SQL. Actuellement, cela fonctionne pour `EXEC SQL TYPE`.

`-C` *mode*
Initialise un mode de compatibilité. *mode* peut être `INFORMIX` ou `INFORMIX_SE`.

`-D` *symbol*
Définit un symbole du préprocesseur C.

`-i`
Les fichiers d'en-tête du système sont également analysés.

`-I` *répertoire*
Spécifie un chemin d'inclusion supplémentaire, utilisé pour trouver les fichiers inclus via `EXEC SQL INCLUDE`. Par défaut, il s'agit de `.` (répertoire courant), `/usr/local/include`, du répertoire de fichiers entêtes de PostgreSQL™ défini à la compilation (par défaut : `/usr/local/pgsql/include`), puis de `/usr/include`, dans cet ordre.

`-o` *nom_fichier*
Indique le nom du fichier de sortie, *nom_fichier*, utilisé par **ecpg**.

`-r` *option*
Sélectionne un comportement en exécution. à ce jour, la seule valeur possible pour *option* est `no_indicator`.

`-t`
Active la validation automatique (autocommit) des transactions. Dans ce mode, chaque commande SQL est validée automatiquement, sauf si elle est à l'intérieur d'un bloc de transaction explicite. Dans le mode par défaut, les commandes ne sont validées qu'à l'exécution de **EXEC SQL COMMIT**.

`-v`
Affiche des informations supplémentaires dont la version et le chemin des entêtes.

`--version`
Affiche la version de ecpg et quitte.

`--help`
Affiche l'aide sur les arguments en ligne de commande de ecpg et quitte.

Notes

Lors de la compilation de fichiers C prétraités, le compilateur a besoin de trouver les fichiers d'en-tête ECPG dans le répertoire des entêtes de PostgreSQL™. De ce fait, il faut généralement utiliser l'option `-I` lors de l'appel du compilateur (c'est-à-dire `-`

`I/usr/local/pgsql/include`).

Les programmes C qui utilisent du SQL embarqué doivent être liés avec la bibliothèque `libecpg`. Cela peut peut être effectué, par exemple, en utilisant les options de l'éditeur de liens `-L/usr/local/pgsql/lib -lecpg`.

La valeur réelle des répertoires, fonction de l'installation, peut être obtenue par l'utilisation de la commande pg_config(1).

Exemples

Soit un fichier source C contenant du SQL embarqué nommé `prog1.pgc`. Il peut être transformé en programme exécutable à l'aide des commandes suivantes :

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

Nom

pg_config — récupèrer des informations sur la version installée de PostgreSQL™

Synopsis

`pg_config` [*option*...]

Description

L'outil pg_config affiche les paramètres de configuration de la version installée de PostgreSQL™. Il peut, par exemple, d'être utilisé par des paquets logiciels qui souhaitent s'interfacer avec PostgreSQL™ pour faciliter la recherche des fichiers d'entêtes requis et des bibliothèques.

Options

Pour utiliser pg_config, une ou plusieurs des options suivantes doivent être fournies :

`--bindir`
: Afficher l'emplacement des exécutables utilisateur. Par exemple, pour trouver le programme **psql**. C'est aussi normalement l'emplacement du programme `pg_config`.

`--docdir`
: Afficher l'emplacement des fichiers de documentation.

`--htmldir`
: Affiche l'emplacement des fichiers de documentation HTML.

`--includedir`
: Afficher l'emplacement des fichiers d'entêtes C des interfaces clientes.

`--pkgincludedir`
: Afficher l'emplacement des autres fichiers d'entête C.

`--includedir-server`
: Afficher l'emplacement des fichiers d'entêtes C pour la programmation du serveur.

`--libdir`
: Afficher l'emplacement des bibliothèques.

`--pkglibdir`
: Afficher l'emplacement des modules chargeables dynamiquement ou celui que le serveur peut parcourir pour les trouver. (D'autres fichiers de données dépendant de l'architecture peuvent aussi être installés dans ce répertoire.)

`--localedir`
: Afficher l'emplacement des fichiers de support de la locale (c'est une chaîne vide si le support de la locale n'a pas été configuré lors de la construction de PostgreSQL™).

`--mandir`
: Afficher l'emplacement des pages de manuel.

`--sharedir`
: Afficher l'emplacement des fichiers de support qui ne dépendent pas de l'architecture.

`--sysconfdir`
: Afficher l'emplacement des fichiers de configuration du système.

`--pgxs`
: Afficher l'emplacement des fichiers makefile d'extensions.

`--configure`
: Afficher les options passées au script `configure` lors de la configuration de PostgreSQL™ en vue de sa construction. Cela peut être utilisé pour reproduire une configuration identique ou pour trouver les options avec lesquelles un paquet binaire a été construit. (Néanmoins, les paquets binaires contiennent souvent des correctifs personnalisés par le vendeur.) Voir aussi les exemples ci-dessous.

`--cc`
: Afficher la valeur de la macro `CC` utilisée lors de la construction de PostgreSQL™. Cela affiche le compilateur C utilisé.

`--cppflags`
: Afficher la valeur de la macro `CPPFLAGS` utilisée lors de la construction de PostgreSQL™. Cela affiche les options du compilateur C nécessaires pour l'exécution du préprocesseur (typiquement, les options `-I`).

`--cflags`
: Afficher la valeur de la macro `CFLAGS` utilisée lors de la construction de PostgreSQL™. Cela affiche les options du compilateur C.

`--cflags_sl`
: Afficher la valeur de la macro `CFLAGS_SL` utilisée lors de la construction de PostgreSQL™. Cela affiche les options supplémentaires du compilateur C utilisées pour construire les bibliothèques partagées.

`--ldflags`
: Afficher la valeur de la macro `LDFLAGS` utilisée lors de la construction de PostgreSQL™. Cela affiche les options de l'éditeur de liens.

`--ldflags_ex`
: Afficher la valeur de la variable `LDFLAGS_EX` utilisée lors de la construction de PostgreSQL™. Cela affiche les options de l'éditeur de liens uniquement pour la construction des exécutables.

`--ldflags_sl`
: Afficher la valeur de la macro `LDFLAGS_SL` utilisée lors de la construction de PostgreSQL™. Cela affiche les options de l'éditeur de liens utilisées pour construire seulement les bibliothèques partagées.

`--libs`
: Afficher la valeur de la macro `LIBS` utilisée lors de la construction de PostgreSQL™. Elle contient habituellement les options `-l` pour les bibliothèques externes auxquelles PostgreSQL™ est lié.

`--version`
: Afficher la version de PostgreSQL™.

Si plusieurs options sont données, l'information est affichée dans cet ordre, un élément par ligne. Si aucune option n'est donnée, toutes les informations disponibles sont affichées avec des étiquettes.

Notes

L'option `--includedir-server` est apparue dans PostgreSQL™ 7.2. Dans les versions précédentes, les fichiers d'entêtes du serveur étaient installés au même endroit que les entêtes client, qui pouvaient être récupérés avec l'option `--includedir`. Pour que le paquet gère les deux cas, la nouvelle option est tentée en premier, et le code de sortie est testé pour savoir si la commande a réussi.

Les options `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl` et `--libs` sont apparues avec PostgreSQL™ 8.1. L'option `--htmldir` n'est disponible qu'à partir de PostgreSQL™ 8.4. The option `--ldflags_ex` was added in PostgreSQL™ 9.0.

Dans les versions antérieures à PostgreSQL™ 7.1, avant que **pg_config** ne soit disponible, il n'existait aucune méthode de récupération de ces informations de configuration.

Exemple

Reproduire la configuration de construction de l'installation actuelle de PostgreSQL :

```
eval ./configure `pg_config --configure`
```

La sortie de `pg_config --configure` contient les guillemets du shell de sorte que les arguments contenant des espaces soient représentés correctement. Du coup, il est nécessaire d'utiliser `eval` pour obtenir des résultats corrects.

Nom

pg_dump — sauvegarder une base de données PostgreSQL™ dans un script ou tout autre fichier d'archive

Synopsis

`pg_dump` [*option_connexion*...] [*option*...] [*nom_base*]

Description

pg_dump est un outil de sauvegarde d'une base de données PostgreSQL™. Les sauvegardes réalisées sont cohérentes, même lors d'accès concurrents à la base de données. pg_dump ne bloque pas l'accès des autres utilisateurs (ni en lecture ni en écriture).

Les extractions peuvent être réalisées sous la forme de scripts ou de fichiers d'archive. Les scripts sont au format texte et contiennent les commandes SQL nécessaires à la reconstruction de la base de données dans l'état où elle était au moment de la sauvegarde. La restauration s'effectue en chargeant ces scrits avec psql(1). Ces scripts permettent de reconstruire la base de données sur d'autres machines et d'autres architectures, et même, au prix de quelques modifications, sur d'autres bases de données SQL.

La reconstruction de la base de données à partir d'autres formats de fichiers archive est obtenue avec pg_restore(1). pg_restore permet, à partir de ces formats, de sélectionner les éléments à restaurer, voire de les réordonner avant restauration. Les fichiers d'archive sont conçus pour être portables au travers d'architectures différentes.

Utilisé avec un des formats de fichier d'archive et combiné avec pg_restore, pg_dump fournit un mécanisme d'archivage et de transfert flexible. pg_dump peut être utilisé pour sauvegarder une base de données dans son intégralité ; pg_restore peut alors être utilisé pour examiner l'archive et/ou sélectionner les parties de la base de données à restaurer. Le format de fichier de sortie le plus flexible est le format « personnalisé » (*custom* en anglais, `-Fc`). Compressé par défaut, il permet de sélectionner et réordonner les éléments archivés. Non compressé, le format tar (`-Ft`) a des restrictions sur la façon de réordonner les données au chargement, mais est, somme toute, assez flexible. Qui plus est, il est utilisable par des outils Unix standard comme **tar**.

Lors de l'exécution de pg_dump, il est utile de surveiller les messages d'avertissement (affichés sur la sortie erreur standard), en particulier en ce qui concerne les limitations indiquées ci-dessous.

Options

Les options suivantes de la ligne de commande contrôlent le contenu et le format de la sortie.

nom_base
: Le nom de la base de données à sauvegarder. En l'absence de précision, la variable d'environnement `PGDATABASE` est utilisée. Si cette variable n'est pas positionnée, le nom de l'utilisateur de la connexion est utilisé.

`-a`, `--data-only`
: Seules les données sont sauvegardées, pas le schéma (définition des données).

  Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option peut être précisée lors de l'appel de **pg_restore**.

`-b`, `--blobs`
: Inclut les objets larges dans la sauvegarde. C'est le comportement par défaut, sauf si une des options suivantes est ajoutée : `--schema`, `--table` ou `--schema-only`. L'option `-b` n'est utile que pour ajouter les objets larges aux sauvegardes sélectives.

`-c`, `--clean`
: Les commandes de nettoyage (suppression) des objets de la base sont écrites avant les commandes de création.

  Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option est précisée à l'appel de **pg_restore**.

`-C`, `--create`
: La sortie débute par une commande de création de la base de données et de connexion à cette base. Peu importe, dans ce cas, la base de données à laquelle la connexion est faite avant la restauration.

  Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option est précisée à l'appel de **pg_restore**.

`-f` *file*, `--file=`*file*
: La sortie est redirigée vers le fichier indiqué. Sans précision, la sortie standard est utilisée.

`-E` *codage*, `--encoding=`*codage*
: La sauvegarde est créée dans l'encodage indiqué. Par défaut, la sauvegarde utilise celui de la base de données. Le même ré-

sultat peut être obtenu en positionnant la variable d'environnement `PGCLIENTENCODING` avec le codage désiré pour la sauvegarde.

`-F` *`format`*, `--format=`*`format`*
Le format de la sortie. *`format`* correspond à un des éléments suivants :

`p`
fichier de scripts SQL en texte simple (défaut) ;

`c`
archive personnalisée utilisable par pg_restore. C'est le format le plus flexible puisqu'il permet une sélection manuelle et d'ordonner les éléments archivés. Ce format est compressé par défaut.

`t`
archive **tar** utilisable par pg_restore. Ce format en sortie permet une sélection manuelle et un réordonnancement des éléments archivés lors de la restauration, mais il existe une restriction : l'ordre relatif des éléments de données d'une table ne peut pas être changé lors de la restauration. De plus, le format **tar** ne supporte pas la compression et a une limite de 8 Go sur la taille de tables individuelles.

`-i`, `--ignore-version`
Une option obsolète qui est maintenant ignorée.

`-n` *`schéma`*, `--schema=`*`schéma`*
Sauvegarde uniquement les schémas correspondant à *`schema`* ; la sélection se fait à la fois sur le schéma et sur les objets qu'il contient. Quand cette option n'est pas indiquée, tous les schémas non système de la base cible sont sauvegardés. Plusieurs schémas peuvent être indiqués en utilisant plusieurs fois l'option `-n`. De plus, le paramètre *`schéma`* est interprété comme un modèle selon les règles utilisées par les commandes `\d` de psql (voir la section intitulée « motifs »). Du coup, plusieurs schémas peuvent être sélectionnés en utilisant des caractères joker dans le modèle. Lors de l'utilisation de ces caractères, il faut faire attention à placer le modèle entre guillemets, si nécessaire, pour empêcher le shell de remplacer les jokers.

Note

Quand `-n` est indiqué, pg_dump ne sauvegarde aucun autre objet de la base que ceux dont les schémas sélectionnés dépendent. Du coup, il n'est pas garanti que la sauvegarde d'un schéma puisse être restaurée avec succès dans une base vide.

Note

Les objets qui ne font pas partie du schéma comme les objets larges ne sont pas sauvegardés quand `-n` est précisé. Ils peuvent être rajouter avec l'option `--blobs`.

`-N` *`schéma`*, `--exclude-schema=`*`schéma`*
Ne sauvegarde pas les schémas correspondant au modèle *`schéma`*. Le modèle est interprété selon les même règles que `-n`. `-N` peut aussi être indiqué plus d'une fois pour exclure des schémas correspondant à des modèles différents.

Quand les options `-n` et `-N` sont indiquées, seuls sont sauvegardés les schémas qui correspondent à au moins une option `-n` et à aucune option `-N`. Si `-N` apparaît sans `-n`, alors les schémas correspondant à `-N` sont exclus de ce qui est une sauvegarde normale.

`-o`, `--oids`
Les identifiants d'objets (OID) sont sauvegardés comme données des tables. Cette option est utilisée dans le cas d'applications utilisant des références aux colonnes OID (dans une contrainte de clé étrangère, par exemple). Elle ne devrait pas être utilisée dans les autres cas.

`-O`, `--no-owner`
Les commandes d'initialisation des possessions des objets au regard de la base de données originale ne sont pas produites. Par défaut, pg_dump engendre des instructions **ALTER OWNER** ou **SET SESSION AUTHORIZATION** pour fixer ces possessions. Ces instructions échouent lorsque le script n'est pas lancé par un superutilisateur (ou par l'utilisateur qui possède tous les objets de ce script). L'option `-O` est utilisée pour créer un script qui puisse être restauré par n'importe quel utilisateur. En revanche, c'est cet utilisateur qui devient propriétaire de tous les objets.

Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option est précisée à l'appel de **pg_restore**.

`-R`, `--no-reconnect`
Cette option, obsolète, est toujours acceptée pour des raisons de compatibilité ascendante.

`-s`, `--schema-only`

Seule la définition des objets (le schéma) est sauvegardée, pas les données.

`-S` *`nomutilisateur`*, `--superuser=`*`nomutilisateur`*
Le nom du superutilisateur à utiliser lors de la désactivation des déclencheurs. Cela n'a d'intérêt que si l'option `--disable-triggers` est précisée. (En règle générale, il est préférable de ne pas utiliser cette option et de lancer le script produit en tant que superutilisateur.)

`-t` *`table`*, `--table=`*`table`*
Sauvegarde uniquement les tables (ou vues ou séquences) correspondant à *`table`*. Plusieurs tables sont sélectionnables en utilisant plusieurs fois l'option `-t`. De plus, le paramètre *`table`* est interprété comme un modèle suivant les règles utilisées par les commandes `\d` de psql (voir la section intitulée « motifs »). Du coup, plusieurs tables peuvent être sélectionnées en utilisant des caractères joker dans le modèle. Lors de l'utilisation de ces caractères, il faut faire attention à placer le modèle entre guillemets, si nécessaire, pour empêcher le shell de remplacer les jokers.

Les options `-n` et `-N` n'ont aucun effet quand l'option `-t` est utilisée car les tables sélectionnées par `-t` sont sauvegardées quelle que soit la valeur des options relatives aux schémas. Les objets qui ne sont pas des tables ne sont pas sauvegardés.

Note

Quand `-t` est indiqué, pg_dump ne sauvegarde aucun autre objet de la base dont la (ou les) table(s) sélectionnée(s) pourrai(en)t dépendre. Du coup, il n'est pas garanti que la sauvegarde spécifique d'une table puisse être restaurée avec succès dans une base vide.

Note

Le comportement de l'option `-t` n'est pas entièrement compatible avec les versions de PostgreSQL™ antérieures à la 8.2. Auparavant, écrire `-t tab` sauvegardait toutes les tables nommées `tab`, mais maintenant, seules sont sauvegardées celles qui sont visibles dans le chemin de recherche des objets. Pour retrouver l'ancien comportement, il faut écrire `-t '*.tab'`. De plus, il faut écrire quelque chose comme `-t sch.tab` pour sélectionner une table dans un schéma particulier plutôt que l'ancienne syntaxe `-n sch -t tab`.

`-T` *`table`*, `--exclude-table=`*`table`*
Ne sauvegarde pas les tables correspondant au modèle *`table`*. Le modèle est interprété selon les même règles que `-t`. `-T` peut aussi être indiqué plusieurs pour exclure des tables correspondant à des modèles différents.

Quand les options `-t` et `-T` sont indiquées, seules sont sauvegardées les tables qui correspondent à au moins une option `-t` et à aucune option `-T`. Si `-T` apparaît sans `-t`, alors les tables correspondant à `-T` sont exclues de ce qui est une sauvegarde normale.

`-v`, `--verbose`
Mode verbeux. pg_dump affiche des commentaires détaillés sur les objets et les heures de début et de fin dans le fichier de sauvegarde. Des messages de progression sont également affichés sur la sortie d'erreur standard.

`-V`, `--version`
Affiche la version de pg_dump puis quitte.

`-x`, `--no-privileges`, `--no-acl`
Les droits d'accès (commandes grant/revoke) ne sont pas sauvegardés.

`-Z` *`0..9`*, `--compress=`*`0..9`*
Indique le niveau de compression à utiliser. Zéro signifie sans compression. Pour le format d'archive personnalisé, cela signifie la compression des segments individuels des données des tables. Par défaut, la compression se fait à un niveau modéré. Pour le format texte, indiquer une valeur différente de zéro implique une compression du fichier complet, comme s'il était passé à gzip ; mais par défaut, la sortie n'est pas compressée. Le format d'archive tar ne supporte pas du tout la compression.

`--binary-upgrade`
Cette option est destinée à être utilisée pour une mise à jour en ligne. Son utilisation dans d'autres buts n'est ni recommandée ni supportée. Le comportement de cette option peut changer dans les futures versions sans avertissement.

`--inserts`
Extraire les données en tant que commandes **INSERT** (plutôt que **COPY**). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL™. Notez que la restauration peut échouer complètement si vous avez changé l'ordre des colonnes. L'option `--column-inserts` est plus sûre, mais encore plus lente.

`--column-inserts`, `--attribute-inserts`

Extraire les données en tant que commandes **INSERT** avec des noms de colonnes explicites (`INSERT INTO table (colonne, ...) VALUES ...`). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL™.

`--disable-dollar-quoting`
Cette option désactive l'utilisation du caractère dollar comme délimiteur de corps de fonctions, et force leur délimitation en tant que chaîne SQL standard.

`--disable-triggers`
Cette option ne s'applique que dans le cas d'une extraction de données seules. Ceci demande à pg_dump d'inclure des commandes pour désactiver temporairement les triggers sur les tables cibles pendant que les données sont rechargées. Utilisez ceci si, sur les tables, vous avez des contraintes d'intégrité ou des triggers que vous ne voulez pas invoquer pendant le rechargement.

À l'heure actuelle, les commandes émises pour `--disable-triggers` doivent être exécutées en tant que superutilisateur. Par conséquent, vous devez aussi spécifier un nom de superutilisateur avec `-S`, ou préférablement faire attention à lancer le script résultat en tant que superutilisateur.

Cette option n'a de sens que pour le format texte simple. Pour les formats d'archive, vous pouvez spécifier cette option quand vous appelez **pg_restore**.

`--lock-wait-timeout=`*`expiration`*
Ne pas attendre indéfiniment l'acquisition de verrous partagés sur table au démarrage de l'extraction. Échouer à la place s'il est impossible de verrouiller une table dans le temps d'*`expiration`* indiqué. L'expiration peut être spécifiée dans tous les formats acceptés par **SET statement_timeout**, les valeurs autorisées dépendant de la version du serveur sur laquelle vous faites l'extraction, mais une valeur entière en millisecondes est acceptée par toutes les versions depuis la 7.3. Cette option est ignorée si vous exportez d'une version antérieure à la 7.3.

`--no-tablespaces`
Ne pas générer de commandes pour créer des tablespace, ni sélectionner de tablespace pour les objets. Avec cette option, tous les objets seront créés dans le tablespace par défaut durant la restauration.

Cette option n'a de sens que pour le format texte simple. Pour les formats d'archive, vous pouvez spécifier cette option quand vous appelez **pg_restore**.

`--use-set-session-authorization`
Émettre des commandes SQL standard **SET SESSION AUTHORIZATION** à la place de commandes **ALTER OWNER** pour déterminer l'appartenance d'objet. Ceci rend l'extraction davantage compatible avec les standards, mais, suivant l'historique des objets de l'extraction, peut ne pas se restaurer correctement. Par ailleurs, une extraction utilisant **SET SESSION AUTHORIZATION** nécessitera certainement des droits superutilisateur pour se restaurer correctement, alors que **ALTER OWNER** nécessite des droits moins élevés.

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de pg_dump, puis quitte

Les options de ligne de commande suivantes gèrent les paramètres de connexion :

`-h` *`hôte`*, `--host` *`hôte`*
Le nom d'hôte de la machine sur laquelle le serveur de bases de données est exécuté. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour le socket de domaine Unix. La valeur par défaut est fournie par la variable d'environnement `PGHOST`, si elle est initialisée. Dans le cas contraire, une connexion sur la socket de domaine Unix est tentée.

`-p` *`port`*, `--port` *`port`*
Le port TCP ou le fichier local de socket de domaine Unix sur lequel le serveur écoute les connexions. La valeur par défaut est fournie par la variable d'environnement `PGPORT`, si elle est initialisée. Dans le cas contraire, il s'agit de la valeur fournie à la compilation.

`-U` *`nomutilisateur`*, `--username` *`nomutilisateur`*
Le nom d'utilisateur utilisé pour la connexion.

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force pg_dump à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais nécessaire car pg_dump demande automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_dump perd une tentative de connexion pour tester si le serveur demande un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--role=nomrole`
Spécifie un rôle à utiliser pour créer l'extraction. Avec cette option, pg_dump émet une commande **SET ROLE** *nomrole* après s'être connecté à la base. C'est utile quand l'utilisateur authentifié (indiqué par `-U`) n'a pas les droits dont pg_dump a besoin, mais peut basculer vers un rôle qui les a. Certaines installations ont une politique qui est contre se connecter directement en tant que superutilisateur, et l'utilisation de cette option permet que les extractions soient faites sans violer cette politique.

Environnement

`PGDATABASE`, `PGHOST`, `PGOPTIONS`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

pg_dump exécute intrinsèquement des instructions **SELECT**. Si des problèmes apparaissent à l'exécution de pg_dump, psql(1) peut être utilisé pour s'assurer qu'il est possible de sélectionner des informations dans la base de données. De plus, tout paramètre de connexion par défaut et toute variable d'environnement utilisé par la bibliothèque libpq s'appliquent.

L'activité générée par pg_dump dans la base de données est normalement collectée par le collecteur de statistiques. Si c'est gênant, vous pouvez positionner le paramètre `track_counts` à false via `PGOPTIONS` ou la commande `ALTER USER`.

Notes

Si des ajouts locaux à la base `template1` ont été effectués, il est impératif de s'assurer que la sortie de pg_dump est effectivement restaurée dans une base vide ; dans le cas contraire, il est fort probable que la duplication des définitions des objets ajoutés engendre des erreurs. Pour obtenir une base vide de tout ajout local, on utilise `template0` à la place de `template1` comme modèle. Par exemple :

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Quand une sauvegarde des seules données est sélectionnée et que l'option `--disable-triggers` est utilisée, pg_dump engendre des commandes de désactivation des déclencheurs sur les tables utilisateur avant l'insertion des données, puis après coup, des commandes de réactivation après l'insertion. Si la restauration est interrompue, il se peut que les catalogues systèmes conservent cette position.

Les fichiers d'une archive tar sont limités à une taille inférieure à 8 Go. (C'est une limitation inhérente au format des fichiers tar.) Ce format ne peut donc pas être utilisé si la représentation textuelle d'une table dépasse cette taille. La taille totale d'une archive tar et de tout autre format de sortie n'est pas limitée, sauf peut-être par le système d'exploitation.

Le fichier de sauvegarde produit par pg_dump ne contient pas les statistiques utilisées par l'optimiseur pour la planification des requêtes. Il est donc conseillé, pour assurer des performances optimales, de lancer **ANALYZE** après la restauration d'une sauvegarde ; voir Section 23.1.3, « Maintenir les statistiques du planificateur » et Section 23.1.5, « Le démon auto-vacuum » pour plus d'informations. Le fichier de sauvegarde ne contient pas non plus de commandes **ALTER DATABASE ... SET** ; ces paramètres sont sauvegardés par pg_dumpall(1), avec les utilisateurs et les paramètres globaux à l'installation.

Parce que pg_dump est utilisé pour transférer des données vers des nouvelles versions de PostgreSQL™, la sortie de pg_dump devra pouvoir se charger dans les versions du serveur PostgreSQL™ plus récentes que la version de pg_dump. pg_dump peut aussi extraire des données de serveurs PostgreSQL™ plus anciens que sa propre version. (À l'heure actuelle, les versions de serveurs supportées vont jusqu'à la 7.0.) Toutefois, pg_dump ne peut pas réaliser d'extraction de serveurs PostgreSQL™ plus récents que sa propre version majeure ; il refusera même d'essayer, plutôt que de risquer de fournir une extraction invalide. Par ailleurs, il n'est pas garanti que la sortie de pg_dump puisse être chargée dans un serveur d'une version majeure plus ancienne -- pas même si l'extraction a été faite à partir d'un serveur dans cette version. Charger un fichier d'extraction dans un serveur de version plus ancienne pourra requérir une édition manuelle du fichier pour supprimer les syntaxe incomprises de l'ancien serveur.

Exemples

Sauvegarder une base appelée `ma_base` dans un script SQL :

```
$ pg_dump ma_base > base.sql
```

Charger ce script dans une base nouvellement créée et nommée `nouvelle_base`:

```
$ psql -d nouvelle_base -f base.sql
```

Sauvegarder une base dans un fichier au format personnalisé :

```
$ pg_dump -Fc ma_base > base.dump
```

Charger un fichier d'archive dans une nouvelle base nommée `nouvelle_base` :

```
$ pg_restore -d nouvelle_base base.dump
```

Sauvegarder la table nommée `mytab` :

```
$ pg_dump -t ma_table ma_base > base.sql
```

Sauvegarder toutes les tables du schéma `detroit` et dont le nom commence par `emp` sauf la table nommée `traces_employes` :

```
$ pg_dump -t 'detroit.emp*' -T detroit.traces_employes ma_base > base.sql
```

Sauvegarder tous les schémas dont le nom commence par `est` ou `ouest` et se termine par `gsm`, en excluant les schémas dont le nom contient le mot `test` :

```
$ pg_dump -n 'est*gsm' -n 'ouest*gsm' -N '*test*' ma_base > base.sql
```

Idem mais en utilisant des expressions rationnelles dans les options :

```
$ pg_dump -n '(est|ouest)*gsm' -N '*test*' ma_base > base.sql
```

Sauvegarder tous les objets de la base sauf les tables dont le nom commence par `ts_` :

```
$ pg_dump -T 'ts_*' ma_base > base.sql
```

Pour indiquer un nom qui comporte des majuscules dans les options `-t` et assimilées, il faut ajouter des guillemets doubles ; sinon le nom est converti en minuscules (voirla section intitulée « motifs »). Les guillemets doubles sont interprétés par le shell et doivent dont être placés entre guillemets. Du coup, pour sauvegarder une seule table dont le nom comporte des majuscules, on utilise une commande du style :

```
$ pg_dump -t '"NomAMajuscule"' ma_base > ma_base.sql
```

Voir aussi

pg_dumpall(1), pg_restore(1), psql(1)

Nom

pg_dumpall — extraire une grappe de bases de données PostgreSQL™ dans un fichier de script

Synopsis

`pg_dumpall` [*option_connexion*...] [*option*...]

Description

pg_dumpall est un outil d'extraction (« sauvegarde ») de toutes les bases de données PostgreSQL™ d'une grappe vers un fichier script. Celui-ci contient les commandes SQL utilisables pour restaurer les bases de données avec psql(1). Cela est obtenu en appelant pg_dump(1) pour chaque base de données de la grappe. pg_dumpall sauvegarde aussi les objets globaux, communs à toutes les bases de données. (pg_dump ne sauvegarde pas ces objets.) Cela inclut aussi les informations concernant les utilisateurs et groupes des bases de données, ainsi que les tablespaces et les propriétés telles que les droits d'accès s'y appliquant.

Puisque pg_dumpall lit les tables de toutes les bases de données, il est préférable d'avoir les droits de superutilisateur de la base de données pour obtenir une sauvegarde complète. De plus, il faut détenir des droits superutilisateur pour exécuter le script produit, afin de pouvoir créer les utilisateurs, les groupes et les bases de données.

Le script SQL est écrit sur la sortie standard. Utilisez l'option [-f|fichier] ou les opérateurs shell pour la rediriger vers un fichier.

pg_dumpall se connecte plusieurs fois au serveur PostgreSQL™ (une fois par base de données). Si l'authentification par mot de passe est utilisé, un mot de passe est demandé à chaque tentative de connexion. Il est intéressant de disposer d'un fichier `~/.pgpass` dans de tels cas. Voir Section 31.14, « Fichier de mots de passe » pour plus d'informations.

Options

Les options suivantes, en ligne de commande, contrôlent le contenu et le format de la sortie.

`-a`, `--data-only`
: Seules les donnéessont sauvegardées, pas le schéma (définition des données).

`-c`, `--clean`
: Les commandes SQL de nettoyage (suppression) des bases de données avant leur recréation sont incluses. Des commandes **DROP** sont également ajoutées pour les rôles et les tablespaces.

`-f` *nomfichier*, `--file=`*nomfichier*
: Envoie le résultat dans le fichier indiqué. Si cette option n'est pas utilisée, la sortie standard est utilisée.

`-g`, `--globals-only`
: Seuls les objets globaux sont sauvegardés (rôles et tablespaces), pas les bases de données.

`-i`, `--ignore-version`
: Une option obsolète qui est maintenant ignorée.

  pg_dumpall peut gérer des bases de données de versions précédentes de PostgreSQL™, mais les très anciennes versions ne sont plus supportées (avant la 7.0). Cette option peut être utilisée pour surcharger la vérification de la version (si pg_dumpall échoue, il ne faudra pas nier avoir été averti).

`-o`, `--oids`
: Les identifiants des objets (OID) sont sauvegardés comme faisant partie des données de chaque table. Cette option est utilisée si l'application référence les colonnes OID (par exemple, dans une contrainte de clé étrangère). Sinon, cette option ne doit pas être utilisée.

`-O`, `--no-owner`
: Les commandes permettant de positionner les propriétaires des objets à ceux de la base de données originale. Par défaut, pg_dumpall lance les instructions **ALTER OWNER** ou **SET SESSION AUTHORIZATION** pour configurer le propriétaire des éléments créés. Ces instructions échouent lorsque le script est lancé par un utilisateur ne disposant pas des droits de superutilisateur (ou ne possédant pas les droits du propriétaire de tous les objets compris dans ce script). Pour que ce qui devient alors propriétaire de tous les objets créés, l'option `-O` doit être utilisée.

`--lock-wait-timeout=`*expiration*
: Ne pas attendre indéfiniment l'acquisition de verrous partagés sur table au démarrage de l'extraction. Échouer à la place s'il est impossible de verrouiller une table dans le temps d'*expiration* spécifié. L'expiration peut être indiquée dans tous les formats acceptés par **SET statement_timeout**, les valeurs autorisées dépendant de la version du serveur sur laquelle vous faites l'extraction, mais une valeur entière en millisecondes est acceptée par toutes les versions depuis la 7.3. Cette option

est ignorée si vous exportez d'une version antérieure à la 7.3.

`--no-tablespaces`
Ne pas générer de commandes pour créer des tablespace, ni sélectionner de tablespace pour les objets. Avec cette option, tous les objets seront créés dans le tablespace par défaut durant la restauration.

`-r`, `--roles-only`
Sauvegarde seulement les rôles, pas les bases ni les tablespaces.

`-s`, `--schema-only`
Seules les définitions des objets (schéma), sans les données, sont sauvegardées.

`-S` *`username`*, `--superuser=`*`username`*
Précise le nom du superutilisateur à utiliser pour la désactivation des déclencheurs. Cela n'a d'intérêt que lorsque `--disable-triggers` est utilisé. (Il est en général préférable de ne pas utiliser cette option et de lancer le script résultant en tant que superutilisateur.)

`-t`, `--tablespaces-only`
Sauvegarde seulement les tablespaces, pas les bases ni les rôles.

`-v`, `--verbose`
Indique l'utilisation du mode verbeux. Ainsi pg_dumpall affiche les heures de démarrage/arrêt dans le fichier de sauvegarde et les messages de progression sur la sortie standard. Il active également le mode verbeux dans pg_dump.

`-V`, `--version`
Affiche la version de pg_dumpall puis quitte.

`-x`, `--no-privileges`, `--no-acl`
Les droits d'accès (commandes grant/revoke) ne sont pas sauvegardés.

`--binary-upgrade`
Cette option est destinée à être utilisée pour une mise à jour en ligne. Son utilisation dans d'autres buts n'est ni recommandée ni supportée. Le comportement de cette option peut changer dans les versions futures sans avertissement.

`--inserts`
Extraire les données en tant que commandes **INSERT** (plutôt que **COPY**). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL™. Notez que la restauration peut échouer complètement si vous avez changé l'ordre des colonnes. L'option `--column-inserts` est plus sûre, mais encore plus lente.

`--column-inserts`, `--attribute-inserts`
Extraire les données en tant que commandes **INSERT** avec des noms de colonnes explicites (`INSERT INTO` *`table`* `(`*`colonne`*`, ...) VALUES ...`). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL™.

`--disable-dollar-quoting`
L'utilisation du dollar comme guillemet dans le corps des fonctions est désactivée. Celles-ci sont mises entre guillemets en accord avec la syntaxe du standard SQL.

`--disable-triggers`
Cette option n'est utile que lors de la création d'une sauvegarde des seules données. pg_dumpall inclut les commandes de désactivation temporaire des déclencheurs sur les tables cibles pendant le rechargement des données. Cette option est utile lorsqu'il existe des vérifications d'intégrité référentielle ou des déclencheurs sur les tables qu'on ne souhaite pas voir appelés lors du rechargement des données.

Actuellement, les commandes émises par `--disable-triggers` nécessitent d'être lancées par un superutilisateur. Il est donc impératif de préciser le nom du superutilisateur avec `-S` ou, préférentiellement, de lancer le script résultant en tant que superutilisateur.

`--use-set-session-authorization`
Les commandes **SET SESSION AUTHORIZATION** du standard SQL sont affichées à la place des commandes **ALTER OWNER** pour préciser le propriétaire de l'objet. Cela améliore la compatibilité de la sauvegarde vis-à-vis des standard. Toutefois, du fait de l'ordre d'apparition des objets dans la sauvegarde, la restauration peut ne pas être correcte.

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de pg_dumpall, puis quitte

Les options suivantes en ligne de commande contrôlent les paramètres de connexion à la base de données.

`-h` *`hôte`*, `--host=`*`hôte`*

Précise le nom d'hôte de la machine sur laquelle le serveur de bases de données est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix. La valeur par défaut est prise à partir de la variable d'environnement `PGHOST`, si elle est initialisée, sinon une connexion socket de domaine Unix est tentée.

`-l` *`dbname`*, `--database=`*`dbname`*
Spécifie le nom de la base où se connecter pour la sauvegarde des gobjets globaux et pour découvrir les bases qui devraient être sauvegardées. Si cette option n'est pas utilisée, la base `postgres` est utilisé et, si elle n'est pas, `template1` sera utilisée.

`-p` *`port`*, `--port=`*`port`*
Précise le port TCP ou l'extension du fichier socket de domaine Unix local sur lequel le serveur est en écoute des connexions. La valeur par défaut est la variable d'environnement `PGPORT`, si elle est initialisée, ou la valeur utilisée lors de la compilation.

`-U` *`nomutilisateur`*, `--username=`*`nomutilisateur`*
Utilisateur utilisé pour initier la connexion.

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force pg_dumpall à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car pg_dumpall demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_dumpall perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Notez que le mot de passe sera demandé pour chaque base de données à sauvegarder. Habituellement, il est préférable de configurer un fichier `~/.pgpass` pour que de s'en tenir à une saisie manuelle du mot de passe.

`--role=`*`nomrole`*
Spécifie un rôle à utiliser pour créer l'extraction. Avec cette option, pg_dumpall émet une commande **SET ROLE** *`nomrole`* après s'être connecté à la base. C'est utile quand l'utilisateur authentifié (indiqué par `-U`) n'a pas les droits dont pg_dumpall a besoin, mais peut basculer vers un rôle qui les a. Certaines installations ont une politique qui est contre se connecter directement en tant que superutilisateur, et l'utilisation de cette option permet que les extractions soient faites sans violer cette politique.

Environnement

`PGHOST`, `PGOPTIONS`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Notes

Comme pg_dumpall appelle pg_dump en interne, certains messages de diagnostique se réfèrent en fait à pg_dump.

Une fois la restauration effectuée, il est conseillé de lancer **ANALYZE** sur chaque base de données, de façon à ce que l'optimiseur dispose de statistiques utiles. **vacuumdb -a -z** peut également être utilisé pour analyser toutes les bases de données.

pg_dumpall requiert que tous les tablespaces nécessaires existent avant la restauration. Dans le cas contraire, la création de la base échouera pour une base qui ne se trouve pas dans l'emplacement par défaut.

Exemples

Sauvegarder toutes les bases de données :

```
$ pg_dumpall > db.out
```

Pour recharger les bases de données à partir de ce fichier, vous pouvez utiliser :

```
$ psql -f db.out postgres
```

(La base de données utilisée pour la connexion initiale n'a pas d'importance ici car le fichier de script créé par pg_dumpall contient les commandes nécessaires à la création et à la connexion aux bases de données sauvegardées.)

Voir aussi

Vérifier pg_dump(1) pour des détails sur les conditions d'erreur possibles.

Nom

pg_restore — restaure une base de données PostgreSQL™ à partir d'un fichier d'archive créé par pg_dump

Synopsis

`pg_restore` [*option_connexion*...] [*option*...] [*nom_fichier*]

Description

pg_restore est un outil pour restaurer une base de données PostgreSQL™ à partir d'une archive créée par pg_dump(1) dans un des formats non textuel. Il lance les commandes nécessaires pour reconstruire la base de données dans l'état où elle était au moment de sa sauvegarde. Les fichiers d'archive permettent aussi à pg_restore d'être sélectif sur ce qui est restauré ou même de réordonner les éléments à restaurer. Les fichiers d'archive sont conçus pour être portables entre les architectures.

pg_restore peut opérer dans deux modes. Si un nom de base de données est spécifié, pg_restore se connecte à cette base de données et restaure le contenu de l'archive directement dans la base de données. Sinon, un script contenant les commandes SQL nécessaires pour reconstruire la base de données est créé et écrit dans un fichier ou sur la sortie standard. La sortie du script est équivalente à celles créées par le format en texte plein de pg_dump. Quelques-unes des options contrôlant la sortie sont du coup analogues aux options de pg_dump.

De toute évidence, pg_restore ne peut pas restaurer l'information qui ne se trouve pas dans le fichier d'archive. Par exemple, si l'archive a été réalisée en utilisant l'option donnant les « données sauvegardées par des commandes **INSERT** », pg_restore ne sera pas capable de charger les données en utilisant des instructions **COPY**.

Options

pg_restore accepte les arguments suivants en ligne de commande.

nom_fichier
: Spécifie l'emplacement du fichier d'archive à restaurer. S'il n'est pas spécifié, l'entrée standard est utilisée.

`-a`, `--data-only`
: Restaure seulement les données, pas les schémas (définitions des données).

`-c`, `--clean`
: Nettoie (supprime) les objets de la base de données avant de les créer.

`-C`, `--create`
: Crée la base de données avant de la restaurer. (Quand cette option est utilisée, la base de données nommée avec `-d` est utilisée seulement pour lancer la commande initiale **CREATE DATABASE**. Toutes les données sont restaurées dans le nom de la base de données qui apparaît dans l'archive.)

`-d` *nom_base*, `--dbname=`*nom_base*
: Se connecte à la base de données *nom_base* et restaure directement dans la base de données.

`-e`, `--exit-on-error`
: Quitte si une erreur est rencontrée lors de l'envoi des commandes SQL à la base de données. La valeur par défaut est de continuer et d'afficher le nombre d'erreurs à la fin de la restauration.

`-f` *nom_fichier*, `--file=`*filename*
: Spécifie le fichier en sortie pour le script généré ou pour la liste lorsqu'elle est utilisée avec `-l`. Par défaut, il s'agit de la sortie standard.

`-F` *format*, `--format=`*format*
: Spécifie le format de l'archive. Il n'est pas nécessaire de le spécifier car pg_restore détermine le format automatiquement. Si spécifié, il peut être un des suivants :

  `t`, `tar`
  : L'archive est une archive **tar**.

  `c`, `custom`
  : L'archive est dans le format personnalisé de pg_dump.

`-i`, `--ignore-version`
: Une option obsolète qui est maintenant ignorée.

`-I` *index*, `--index=`*index*

Restaure uniquement la définition des index nommés.

`-j` *`nombre-de-jobs`*, `--jobs=`*`nombre-de-jobs`*
Exécute les parties les plus consommatrices en temps de pg_restore -- celles des chargements de données, créations d'index et créations de contraintes -- en utilisant plusieurs jobs concurrents. Cette option peut réduire de beaucoup le temps pour restaurer une grosse base de données pour un serveur fonctionnant sur une machine multi-processeus.

Chaque job est un processus ou un thread, suivant le système d'exploitation, et utilise une connexion séparée au serveur.

La valeur optimale pour cette option dépend de la configuration matérielle du serveur, du client et du réseau. Les facteurs incluent le nombre de cœurs CPU et la configuration disque. Un bon moyen pour commencer est le nombre de cœurs CPU du serveur, mais une valeur plus grande que ça peut amener des temps de restauration encore meilleurs dans de nombreux cas. Bien sûr, les valeurs trop hautes apporteront des performances en baisse.

Seul le format d'archivage personnalisé est supporté avec cette option. Le fichier en entrée doit être un fichier standard (pas un tube par exemple). Cette option est ignorée lors de la création d'un script plutôt qu'une connexion à la base de données. De plus, plusieurs jobs ne peuvent pas être utilisés ensemble si vous voulez l'option `--single-transaction`.

`-l`, `--list`
Liste le contenu de l'archive. Le résultat de cette opération peut être utilisé en entrée de l'option `-L`. Notez que, si vous utilisez des options de filtre telles que `-n` ou `-t` avec l'option `-l`, elles restreignent les éléments listés.

`-L` *`fichier_liste`*, `--use-list=`*`fichier_liste`*
Restaure seulement les objets qui sont listés dans le fichier *`fichier_liste`*, et les restaure dans l'ordre où elles apparaissent dans le fichier. Notez que, si des options de filtre comme `-n` et `-t` sont utilisées avec `-L`, elles ajouteront cette restriction aux éléments restaurés.

`fichier_liste` est normalement créé en éditant la sortie d'une précédente opération `-l`. Les lignes peuvent être déplacées ou supprimées, et peuvent aussi être mise en commentaire en ajoutant un point-virgule (`;`) au début de la ligne. Voir ci-dessous pour des exemples.

`-n` *`nom_schema`*, `--schema=`*`nom_schema`*
Restaure seulement les objets qui sont dans le schéma nommé. Elle peut être combinée avec l'option `-t` pour ne restaurer qu'une seule table.

`-O`, `--no-owner`
Ne pas donner les commandes initialisant les propriétaires des objets pour correspondre à la base de données originale. Par défaut, pg_restore lance des instructions **ALTER OWNER** ou **SET SESSION AUTHORIZATION** pour configurer le propriétaire des éléments du schéma créé. Ces instructions échouent sauf si la connexion initiale à la base de données est réalisée par un superutilisateur (ou le même utilisateur que le propriétaire des objets du script). Avec `-O`, tout nom d'utilisateur peut être utilisé pour la connexion initiale et cet utilisateur est le propriétaire des objets créés.

`--no-tablespaces`
Ne sélectionne pas les tablespaces. Avec cette option, tous les objets seront créés dans le tablespace par défaut lors de la restauration.

`-P` *`nom_fonction(argtype [, ...])`*, `--function=`*`nom_fonction(argtype [, ...])`*
Restaure seulement la fonction nommée. Faites attention à épeler le nom de la fonction et les arguments exactement comme ils apparaissent dans la table des matières du fichier de sauvegarde.

`-r`, `--no-reconnect`
Cette option est obsolète mais est toujours acceptée pour des raisons de compatibilité ascendante.

`-s`, `--schema-only`
Restaure uniquement le schéma (définition des données), et non pas les données elles-même (contenu de la table). Les valeurs actuelles des séquences ne seront pas restaurées. À ne pas confondre avec l'option `--schema` qui utilise le mot schéma avec une autre signification).

`-S` *`nom_utilisateur`*, `--superuser=`*`nom_utilisateur`*
Spécifie le nom d'utilisateur du superutilisateur à utiliser pour désactiver les déclencheurs. Ceci est seulement nécessaire si `--disable-triggers` est utilisé.

`-t` *`table`*, `--table=`*`table`*
Restaure uniquement la définition et/ou les données de la table nommée. Ceci est combinable avec l'option `-n` pour spécifier un schéma.

`-T` *`trigger`*, `--trigger=`*`trigger`*
Restaure uniquement le déclencheur nommé.

`-v`, `--verbose`
Spécifie le mode verbeux.

`-V`, `--version`
Affiche la version de pg_restore, puis quitte.

`-x`, `--no-privileges`, `--no-acl`
Empêche la restauration des droits d'accès (commandes grant/revoke).

`--disable-triggers`
Cette option n'est pertinente que lors d'une restauration des données seules. Elle demande à pg_restore d'exécuter des commandes pour désactiver temporairement les déclencheurs sur les tables cibles pendant que les données sont rechargées. Utilisez ceci si vous avez des vérifications d'intégrité référentielle sur les tables que vous ne voulez pas appeler lors du rechargement des données.

Actuellement, les commandes émises pour `--disable-triggers` doivent être exécutées par un superutilisateur. Donc, vous devriez aussi spécifier un nom de superutilisateur avec `-S` ou, de préférence, lancer pg_restore en tant que superutilisateur PostgreSQL™.

`--use-set-session-authorization`
Affiche les commandes **SET SESSION AUTHORIZATION** du standard SQL à la place des commandes **ALTER OWNER** pour déterminer le propriétaire de l'objet. Ceci rend la sauvegarde plus compatible avec les standards mais, suivant l'historique des objets dans la sauvegarde, pourrait restaurer correctement.

`--no-data-for-failed-tables`
Par défaut, les données de la table sont restaurées même si la commande de création de cette table a échoué (par exemple parce qu'elle existe déjà). Avec cette option, les données de cette table seront ignorées. Ce comportement est utile si la base cible contient déjà des données pour cette table. Par exemple, les tables supplémentaires des extensions de PostgreSQL™ comme PostGIS™ pourraient avoir déjà été créées et remplies sur la base cible ; indiquer cette option empêche l'ajout de données dupliquées ou obsolètes.

Cette option est seulement efficace lors de la restauration directe d'une base, pas lors de la réalisation d'une sortie de script SQL.

`-1`, `--single-transaction`
Exécute la restauration en une seule transaction (autrement dit, toutes les commandes de restauration sont placées entre un **BEGIN** et un **COMMIT**). Ceci assure l'utilisateur que soit toutes les commandes réussissent, soit aucun changement n'est appliqué. Cette option implique `--exit-on-error`.

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de pg_restore, puis quitte.

pg_restore accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*, `--host=`*`hôte`*
Spécifie le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence par un slash, elle est utilisée comme répertoire du socket de domaine Unix. La valeur par défaut est prise dans la variable d'environnement `PGHOST`, si elle est initialisée, sinon une connexion socket de domaine Unix est tentée.

`-p` *`port`*, `--port=`*`port`*
Spécifie le port TCP ou l'extension du fichier socket de domaine Unix sur lequel le serveur écoute les connexions. Par défaut, l'outil utilise la variable d'environnement `PGPORT`, si elle est configurée, sinon il utilise la valeur indiquée à la compilation.

`-U` *`nom_utilisateur`*, `--username=`*`nom_utilisateur`*
Se connecte en tant que cet utilisateur

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force pg_restore à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car pg_restore demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_restore perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--role=`*`nom_rôle`*
Indique un nom de rôle utilisé pour la restauration. Cette option fait que pg_restore exécute un **SET ROLE** *`nom_rôle`* après connexion à la base de données. C'est utile quand l'utilisateur authentifié (indiqué par l'option `-U`) n'a pas les droits demandés

par pg_restore, mais peut devenir le rôle qui a les droits requis. Certains installations ont une politique contre la connexion en super-utilisateur directement, et utilisent cette option pour permettre aux restaurations de se faire sans violer cette règle.

Environnement

`PGHOST`, `PGOPTIONS`, `PGPORT`, `PGUSER`
: Paramètres de connexion par défaut

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

Quand une connexion directe à la base de données est spécifiée avec l'option `-d`, pg_restore exécute en interne des instructions SQL. Si vous avez des problèmes en exécutant pg_restore, assurez-vous d'être capable de sélectionner des informations à partir de la base de données en utilisant, par exemple à partir de psql(1). De plus, tout paramètre de connexion par défaut et toute variable d'environnement utilisé par la bibliothèque libpq s'appliqueront.

Notes

Si votre installation dispose d'ajouts locaux à la base de données `template1`, faites attention à charger la sortie de pg_restore dans une base de données réellement vide ; sinon, vous avez des risques d'obtenir des erreurs dûes aux définitions dupliquées des objets ajoutés. Pour créer une base de données vide sans ajout local, copiez à partir de `template0`, et non pas de `template1`, par exemple :

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Les limitations de pg_restore sont détaillées ci-dessous.

- Lors de la restauration des données dans une table pré-existante et que l'option `--disable-triggers` est utilisée, pg_restore émet des commandes pour désactiver les déclencheurs sur les tables utilisateur avant d'insérer les données, puis émet les commandes pour les réactiver après l'insertion des données. Si la restauration est stoppée en plein milieu, les catalogues système pourraient être abandonnés dans le mauvais état.
- pg_restore ne peut pas restaurer les « large objects » de façon sélective, par exemple seulement ceux d'une table précisée. Si une archive contient des « large objects », alors tous les « large objects » seront restaurées (ou aucun s'ils sont exclus avec l'option `-L`, l'option `-t` ou encore d'autres options.

Voir aussi la documentation de pg_dump(1) pour les détails sur les limitations de pg_dump.

Une fois la restauration terminée, il est conseillé de lancer **ANALYZE** sur chaque table restaurée de façon à ce que l'optimiseur dispose de statistiques utiles ; voir Section 23.1.3, « Maintenir les statistiques du planificateur » et Section 23.1.5, « Le démon auto-vacuum » pour plus d'informations.

Exemples

Supposons que nous avons sauvegardé une base nommée `ma_base` dans un fichier de sauvegarde au format personnalisé :

```
$ pg_dump -Fc ma_base > ma_base.dump
```

Pour supprimer la base et la re-créer à partir de la sauvegarde :

```
$ dropdb ma_base
$ pg_restore -C -d postgres ma_base.dump
```

La base nommée avec l'option `-d` peut être toute base de données existante dans le cluster ; pg_restore l'utilise seulement pour exécuter la commande **CREATE DATABASE** pour `ma_base`. Avec `-C`, les données sont toujours restaurées dans le nom de la base qui apparaît dans le fichier de sauvegarde.

Pour charger la sauvegarde dans une nouvelle base nommée `nouvelle_base` :

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Notez que nous n'utilisons pas `-C` et que nous nous sommes connectés directement sur la base à restaurer. De plus, notez que nous clonons la nouvelle base à partir de `template0` et non pas de `template1`, pour s'assurer qu'elle est vide.

Pour réordonner les éléments de la base de données, il est tout d'abord nécessaire de sauvegarder la table des matières de l'archive :

```
$ pg_restore -l ma_base.dump > ma_base.liste
```

Le fichier de liste consiste en un en-tête et d'une ligne par élément, par exemple :

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

Les points virgules commencent un commentaire et les numéros au début des lignes se réfèrent à l'ID d'archive interne affectée à chaque élément.

Les lignes dans le fichier peuvent être commentées, supprimées et réordonnées. Par exemple :

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

peut être utilisé en entrée de pg_restore et ne restaure que les éléments 10 et 6 dans cet ordre :

```
$ pg_restore -L mabase.liste mabase.fichier
```

Voir aussi

pg_dump(1), pg_dumpall(1), psql(1)

Nom

psql — terminal interactif PostgreSQL™

Synopsis

`psql` [*option*...] [*nombase* [*nomutilisateur*]]

Description

psql est une interface en mode texte pour PostgreSQL™. Il vous permet de saisir des requêtes de façon interactive, de les exécuter sur PostgreSQL™ et de voir les résultats de ces requêtes. Alternativement, les entrées peuvent êtres lues à partir d'un fichier. De plus, il fournit un certain nombre de méta-commandes et plusieurs fonctionnalités style shell pour faciliter l'écriture des scripts et automatiser un nombre varié de tâches.

Options

`-a`, `--echo-all`
Affiche toutes les lignes en entrée sur la sortie standard lorsqu'elles sont lues. Ceci est plus utile dans le traitement de scripts que dans le mode interactif. C'est équivalent à initialiser la variable `ECHO` à `all`.

`-A`, `--no-align`
Bascule dans le mode d'affichage non aligné. (Le mode d'affichage par défaut est aligné.)

`-c` *commande*, `--command` *commande*
Indique que psql doit exécuter une chaîne de commande, *commande*, puis s'arrêter. Cette option est utile dans les scripts shell. Les fichiers de démarrage (`psqlrc` et `~/.psqlrc`) sont ignorés avec cette option.

commande doit être soit une chaîne de commandes complètement analysable par le serveur (c'est-à-dire qui ne contient aucune des fonctionnalités spécifiques de psql), soit une seule commande antislash. Du coup, vous ne pouvez pas mixer les commandes SQL et les méta-commandes psql avec cette option. Pour réussir ceci, vous pouvez envoyer la chaîne dans un tube vers psql, par exemple : `echo "\x \\ SELECT * FROM foo;" | psql`. (`\\` est la méta-commande séparateur.)

Si la chaîne de commandes contient plusieurs commandes SQL, elles sont traitées dans une seule transaction sauf si des commandes **BEGIN**/**COMMIT** explicites sont inclues dans la chaîne pour la diviser en plusieurs transactions. Ceci est différent du comportement adopté quand la même chaîne est envoyée dans l'entrée standard de psql. De plus, seul le résultat de la dernière commande SQL est renvoyé.

À cause de ces comportements historiques, placer plus d'une commande dans la chaîne fournie à l'option `-c` a souvent des résultats inattendus. Il est préférable de fournir plusieurs commandes sur l'entrée standard de psql, soit en utilisant echo comme illustré ci-dessus, soit avec un document shell en ligne, par exemple :

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

`-d` *nombase*, `--dbname` *nombase*
Indique le nom de la base de données où se connecter. Ceci est équivalent à spécifier *nombase* comme premier argument de la ligne de commande qui n'est pas une option.

Si ce paramètre contient un signe =, il est traité comme une chaîne *conninfo*. Voir Section 31.1, « Fonctions de contrôle de connexion à la base de données » pour plus d'informations.

`-e`, `--echo-queries`
Copie toutes les commandes qui sont envoyées au serveur sur la sortie standard. Ceci est équivalent à initialiser la variable `ECHO` à `queries`.

`-E`, `--echo-hidden`
Affiche les requêtes réelles générées par **\d** et autres commandes antislash. Vous pouvez utiliser ceci pour étudier les opérations internes de psql. Ceci est équivalent à initialiser la variable `ECHO_HIDDEN` dans psql.

`-f` *nomfichier*, `--file` *nomfichier*
Utilise le fichier *nomfichier* comme source des commandes au lieu de lire les commandes de façon interactive. Une fois

que le fichier est entièrement traité, psql se termine. Ceci est globalement équivalent à la commande interne **\i**.

Si *nomfichier* est un `-` (tiret), alors l'entrée standard est lue.

Utiliser cette option est légèrement différent d'écrire `psql < nomfichier`. En général, les deux feront ce que vous souhaitez mais utiliser `-f` active certaines fonctionnalités intéressantes comme les messages d'erreur avec les numéros de ligne. Il y a aussi une légère chance qu'utiliser cette option réduira la surcharge du lancement. D'un autre côté, la variante utilisant la redirection de l'entrée du shell doit (en théorie) pour ramener exactement le même affichage que celui que vous auriez eu en saisissant tout manuellement.

`-F` *séparateur*, `--field-separator` *séparateur*
Utilisez *séparateur* comme champ séparateur pour un affichage non aligné. Ceci est équivalent à **\pset fieldsep** ou **\f**.

`-h` *nomhôte*, `--host` *nomhôte*
Indique le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-H`, `--html`
Active l'affichage en tableau HTML. Ceci est équivalent à `\pset format html` ou à la commande **\H**.

`-l`, `--list`
Liste toutes les bases de données disponibles puis quitte. Les autres option non relatives à la connexion sont ignorées. Ceci est similaire à la commande interne **\list**.

`-L` *nomfichier*, `--log-file` *nomfichier*
Écrit tous les résultats des requêtes dans le fichier *nomfichier* en plus de la destination habituelle.

`-n`, `--no-readline`
N'utilise pas readline pour l'édition de ligne et n'utilise pas l'historique. Ceci est utile quand on veut désactiver la gestion de la tabulation pour l'action copie/colle.

`-o` *nomfichier*, `--output` *nomfichier*
Dirige tous les affichages de requêtes dans le fichier *nomfichier*. Ceci est équivalent à la commande **\o**.

`-p` *port*, `--port` *port*
Indique le port TCP ou l'extension du fichier socket de domaine local Unix sur lequel le serveur attend les connexions. Par défaut, il s'agit de la valeur de la variable d'environnement `PGPORT` ou, si elle n'est pas initialisée, le port spécifié au moment de la compilation, habituellement 5432.

`-P` *affectation*, `--pset` *affectation*
Vous permet de spécifier les options d'affichage dans le style de **\pset** sur la ligne de commande. Notez que, ici, vous devez séparer nom et valeur avec un signe égal au lieu d'un espace. Du coup, pour initialiser le format d'affichage en LaTeX, vous devez écrire `-P format=latex`.

`-q`, `--quiet`
Indique que psql doit travailler silencieusement. Par défaut, il affiche des messages de bienvenue et des informations diverses. Si cette option est utilisée, rien de ceci n'est affiché. C'est utile avec l'option `-c`. À l'intérieur de psql, vous pouvez aussi initialiser la variable `QUIET` pour arriver au même effet.

`-R` *séparateur*, `--record-separator` *séparateur*
Utilisez *séparateur* comme séparateur d'enregistrement pour un affichage non aligné. Ceci est équivalent à la commande **\pset recordsep**.

`-s`, `--single-step`
S'exécute en mode étape par étape. Ceci signifie qu'une intervention de l'utilisateur est nécessaire avant l'envoi de chaque commande au serveur, avec une option pour annuler l'exécution. Utilisez cette option pour déboguer des scripts.

`-S`, `--single-line`
S'exécute en mode simple ligne où un retour à la ligne termine une commande SQL, de la même façon qu'un point-virgule.

Note

Ce mode est fourni pour ceux qui insistent pour l'avoir, mais vous n'êtes pas nécessairement encouragé à l'utiliser. En particulier, si vous mixez SQL et méta-commandes sur une ligne, l'ordre d'exécution n'est pas toujours clair pour l'utilisateur non expérimenté.

`-t`, `--tuples-only`
Désactive l'affichage des noms de colonnes et le pied de page contenant le nombre de résultats, etc. Ceci est équivalent à la méta-commande **\t**.

`-T` *`options_table`*, `--table-attr` *`options_table`*
Permet d'indiquer les options à placer à l'intérieur d'une balise `table` en HTML. Voir **\pset** pour plus de détails.

`-U` *`nomutilisateur`*, `--username` *`nomutilisateur`*
Se connecte à la base de données en tant que l'utilisateur *`nomutilisateur`* au lieu de celui par défaut. (Vous devez aussi avoir le droit de le faire, bien sûr.)

`-v` *`affectation`*, `--set` *`affectation`*, `--variable` *`affectation`*
Réalise une affectation de variable comme la commande interne **\set**. Notez que vous devez séparer nom et valeur par un signe égal sur la ligne de commande. Pour désinitialiser une variable, enlevez le signe d'égalité. Pour simplement initialiser une variable sans valeur, utilisez le signe égal sans passer de valeur. Ces affectations sont réalisées lors de la toute première étape du lancement, du coup les variables réservées à des buts internes peuvent être écrasées plus tard.

`-V`, `--version`
Affiche la version de psql et quitte.

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

Notez que cette option restera positionnée pour l'ensemble de la session, et qu'elle affecte aussi l'utilisation de la méta-commande **\connect** en plus de la tentative de connexion initiale.

`-W`, `--password`
Force psql à demander un mot de passe avant de se connecter à une base de données.

Cette option n'est jamais obligatoire car psql demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, psql perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Notez que cette option sera conservée pour la session entière, et que du coup elle affecte l'utilisation de la méta-commande **\connect** ainsi que la tentative de connexion initiale.

`-x`, `--expanded`
Active le mode de formatage de table étendu. Ceci est équivalent à la commande **\x**.

`-X`,, `--no-psqlrc`
Ne lit pas le fichier de démarrage (ni le fichier système `psqlrc` ni celui de l'utilisateur `~/.psqlrc`).

`-1`, `--single-transaction`
Quand psql exécute un script avec l'option `-f`, ajouter cette option englobe le script avec les instructions **BEGIN**/**COMMIT** pour tout faire dans une seule transaction. Ceci nous assure que soit toutes les commandes se terminent avec succès, soit aucune modification n'est enregistrée.

Si le script utilise lui-même **BEGIN**, **COMMIT** ou **ROLLBACK**, cette option n'aura pas les effets désirés. De plus, si le script contient toute commande qui ne peut pas être exécutée à l'intérieur d'un bloc de transaction, indiquer cette option provoquera l'échec de cette commande (et du coup de la transaction entière).

`-?`, `--help`
Affiche de l'aide sur les arguments en ligne de commande de psql et quitte.

Code de sortie

psql renvoie 0 au shell s'il se termine normalement, 1 s'il y a eu une erreur fatale de son fait (pas assez de mémoire, fichier introuvable), 2 si la connexion au serveur s'est interrompue ou a été annulée, 3 si une erreur est survenue dans un script et si la variable `ON_ERROR_STOP` a été initialisée.

Usage

Se connecter à une base de données

psql est une application client PostgreSQL™ standard. Pour se connecter à une base de données, vous devez connaître le nom de votre base de données cible, le nom de l'hôte et le numéro de port du serveur ainsi que le nom de l'utilisateur que vous souhaitez connecter. psql peut connaître ces paramètres à partir d'options en ligne de commande, respectivement `-d`, `-h`, `-p` et `-U`. Si un argument autre qu'une option est rencontré, il est interprété comme le nom de la base de données (ou le nom de l'utilisateur si le nom de la base de données est déjà donné). Toutes les options ne sont pas requises, des valeurs par défaut sont applicables. Si vous omettez le nom de l'hôte, psql se connecte via un socket de domaine Unix à un serveur sur l'hôte local ou via TCP/IP sur `local-`

host pour les machines qui n'ont pas sockets de domaine Unix. Le numéro de port par défaut est déterminé au moment de la compilation. Comme le serveur de bases de données utilise la même valeur par défaut, vous n'aurez pas besoin de spécifier le port dans la plupart des cas. Le nom de l'utilisateur par défaut est votre nom d'utilisateur Unix, de même pour le nom de la base de données par défaut. Notez que vous ne pouvez pas simplement vous connecter à n'importe quelle base de données avec n'importe quel nom d'utilisateur. Votre administrateur de bases de données doit vous avoir informé de vos droits d'accès.

Quand les valeurs par défaut ne sont pas correctes, vous pouvez vous simplifier la vie en configurant les variables d'environnement PGDATABASE, PGHOST, PGPORT et/ou PGUSER avec les valeurs appropriées (pour les variables d'environnement supplémentaires, voir Section 31.13, « Variables d'environnement »). Il est aussi intéressant d'avoir un fichier ~/.pgpass pour éviter d'avoir régulièrement à saisir des mots de passe. Voir Section 31.14, « Fichier de mots de passe » pour plus d'informations.

Une autre façon d'indiquer les paramètres de connexion est dans une chaîne *conninfo* qui est utilisée à la place du nom d'une base de données. Ce mécanisme vous donne un grand contrôle sur la connexion. Par exemple :

```
$ psql "service=monservice sslmode=require"
```

De cette façon, vous pouvez aussi utiliser LDAP pour la recherche de paramètres de connexion, comme décrit dans Section 31.16, « Recherches LDAP des paramètres de connexion ». Voir Section 31.1, « Fonctions de contrôle de connexion à la base de données » pour plus d'informations sur toutes les options de connexion disponibles.

Si la connexion ne peut pas se faire, quelle qu'en soit la raison (c'est-à-dire droits non suffisants, serveur arrêté sur l'hôte cible, etc.), psql renvoie une erreur et s'arrête.

Saisir des commandes SQL

Dans le cas normal, psql fournit une invite avec le nom de la base de données sur laquelle psql est connecté suivi par la chaîne =>. Par exemple

```
$ psql basetest
psql (9.0.18)
Type "help" for help.

basetest=>
```

À l'invite l'utilisateur peut saisir des commandes SQL. Ordinairement, les lignes en entrée sont envoyées vers le serveur quand un point-virgule de fin de commande est saisi. Une fin de ligne ne termine pas une commande. Du coup, les commandes peuvent être saisies sur plusieurs lignes pour plus de clarté. Si la commande est envoyée et exécutée sans erreur, les résultats de la commande sont affichés sur l'écran.

À chaque fois qu'une commande est exécutée, psql vérifie aussi les événements de notification générés par LISTEN(7) et NOTIFY(7).

Meta-commandes

Tout ce que vous saisissez dans psql qui commence par un antislash non échappé est une méta-commande psql qui est traitée par psql lui-même. Ces commandes aident à rendre psql plus utile pour l'administration ou pour l'écriture de scripts. Les méta-commandes sont plus souvent appelées les commandes slash ou antislash.

Le format d'une commande psql est l'antislash suivi immédiatement d'un verbe de commande et de ses arguments. Les arguments sont séparés du verbe de la commande et les uns des autres par un nombre illimité d'espaces blancs.

Pour inclure des espaces blancs dans un argument, vous devez l'envelopper dans des guillemets simples. Pour inclure un guillemet simple dans un argument, utilisez deux guillemets simples. Tout ce qui est contenu entre des guillemets simples est de plus sujet à des substitutions style C pour \n (nouvelle ligne), \t (tabulation), *chiffres* (octal) et \x*chiffres* (hexadécimal).

Si un argument sans guillemets commence avec un caractère :, il est pris pour une variable psql et la valeur de la variable est utilisée à la place de l'argument.

Les arguments placés entre guillemets arrières (`) sont pris comme une ligne de commande passée au shell. L'affichage de la commande (sans l'éventuel saut de ligne à la fin) est pris comme valeur de l'argument. Cela s'applique aussi aux séquences d'échappement ci-dessus.

Quelques commandes prennent un identifiant SQL (comme un nom de table) en argument. Ces arguments suivent les règles de la syntaxe SQL : les lettres sans guillemets sont forcées en minuscule alors que les guillemets doubles (") protègent les lettres de la conversion de casse et autorisent l'incorporation d'espaces blancs dans l'identifiant. À l'intérieur des guillemets doubles, les guillemets doubles en paire se réduisent à un seul guillemet double dans le nom résultant. Par exemple, FOO"BAR"BAZ est interprété comme fooBARbaz et "Un nom ""bizarre" devient Un nom "bizarre.

L'analyse des arguments se termine quand d'autres antislash non entre guillemets surviennent. Ceci est pris pour le début d'une nouvelle méta-commande. La séquence spéciale `\\` (deux antislashes) marque la fin des arguments et continue l'analyse des commandes SQL, si elles existent. De cette façon, les commandes SQL et psql peuvent être mixées librement sur une ligne. Mais dans tous les cas, les arguments d'une meta-commande ne peuvent pas continuer après la fin de la ligne.

Les meta-commandes suivantes sont définies :

`\a`
Si le format actuel d'affichage d'une table est non aligné, il est basculé à aligné. S'il n'est pas non aligné, il devient non aligné. Cette commande est conservée pour des raisons de compatibilité. Voir **\pset** pour une solution plus générale.

`\cd [ répertoire ]`
Modifie le répertoire courant par *répertoire*. Sans argument, le répertoire personnel de l'utilisateur devient le répertoire courant.

Astuce

Pour afficher votre répertoire courant, utilisez `\! pwd`.

`\C [ titre ]`
Initialise ou supprime le titre des tables affichées en résultat d'une requête. Cette commande est équivalente à `\pset title titre`. (Le nom de cette commande provient de « caption », car elle avait précédemment pour seul but d'initialiser l'en-tête dans une table HTML.)

`\connect` (or `\c`) `[ nom_base [ nom_utilisateur ] [ hôte ] [ port ] ]`
Établie une nouvelle connexion à un serveur PostgreSQL™. Si la nouvelle connexion est réussie, la connexion précédente est fermée. Si une option parmi *nom_base*, *nom_utilisateur*, *hôte* et *port* est omise ou vaut `-`, la valeur de ce même paramètre de la connexion précédente est utilisée. S'il ny avait pas encore de connexion, la valeur par défaut dans libpq est utilisée.

Si la tentative de connexion échoue (mauvais nom d'utilisateur, accès refusé, etc.), la connexion précédente est conservée si psql est en mode interactif. Lors de l'exécution d'un script non interactif, le traitement s'arrêtera immédiatement avec une erreur. Cette distinction a été choisie pour deux raisons : aider l'utilisateur face aux fautes de frappe et en tant que mesure de précaution pour qu'un script n'agisse pas par erreur sur la mauvaise base.

`\copy { table [(liste_colonnes)] | (requête) } { from | to } { nomfichier | stdin | stdout | pstdin | pstdout } [with] [binary] [oids] [delimiter [as] 'caractère'] [null [as] 'chaîne'] [csv [header] [quote [as] 'caractère'] [escape [as] 'caractère'] [force quote liste_colonnes] [force not null liste_colonnes]]`
Réalise une opération de copy côté client. C'est une opération qui exécute une commande SQL, COPY(7), mais au lieu que le serveur lise ou écrive le fichier spécifié, psql lit ou écrit le fichier en faisant le routage des données entre le serveur et le système de fichiers local. Ceci signifie que l'accès et les droits du fichier sont ceux de l'utilisateur local, pas celui du serveur, et qu'aucun droit de superutilisateur n'est requis.

La syntaxe de la commande est similaire à celle de la commande COPY(7) SQL. Notez que, à cause de cela, des règles spéciales d'analyse s'appliquent à la commande **\copy**. En particulier, les règles de substitution de variable et d'échappement antislash ne s'appliquent pas.

`\copy ... from stdin | to stdout` lit/écrit basé sur l'entrée standard de la commande ou sa sortie standard respectivement. Toutes les lignes sont lues à partir de la même source qui a lancé la commande, en continuant jusqu'à ce que `\.` soit lu ou que le flux parvienne à EOF. La sortie est envoyée au même endroit que la sortie de la commande. Pour lire/écrire à partir de l'entrée et de la sortie standard de psql, utilisez `pstdin` ou `pstdout`. Cette option est utile pour peupler des tables en ligne à l'intérieur d'un fichier script SQL.

Astuce

Cette opération n'est pas aussi efficace que la commande **COPY** en SQL parce que toutes les données doivent passer au travers de la connexion client/serveur. Pour les grosses masses de données, la commande SQL est préférable.

`\copyright`
Affiche le copyright et les termes de distribution de PostgreSQL™.

`\d[S+] [ motif ]`
Pour chaque relation (table, vue, index ou séquence) correspondant au *motif*, affiche toutes les colonnes, leur types, le ta-

blespace (s'il ne s'agit pas du tablespace par défaut) et tout attribut spécial tel que `NOT NULL` ou les valeurs par défaut. Les index, contraintes, règles et déclencheurs associés sont aussi affichés, ainsi que la définition de la vue si la relation est une vue. (Ce qui « Correspond au motif » est défini ci-dessous.)

Le forme de la commande `\d+` est identique, sauf que des informations plus complètes sont affichées : tout commentaire associé avec les colonnes de la table est affiché, ainsi que la présence d'OID dans la table.

Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

> **Note**
>
> Si **\d** est utilisé sans argument *`motif`*, c'est équivalent, en plus commode, à **\dtvs** qui affiche une liste de toutes les tables, vues et séquences.

`\da[S] [ motif ]`
Liste toutes les fonctions d'agrégat disponibles, avec lee type de retour et les types de données sur lesquels elles opèrent. Si *`motif`* est spécifié, seuls les agrégats dont les noms commencent par le motif sont affichés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

`\db[+] [ motif ]`
Liste tous les tablespaces disponibles. Si *`motif`* est spécifié, seuls les tablespaces dont le nom correspond au motif sont affichés. Si `+` est ajouté à la fin de la commande, chaque objet est listé avec les droits associés.

`\dc[S] [ motif ]`
Liste les conversions entre les encodages de jeux de caractères. Si *`motif`* est spécifié, seules les conversions dont le nom correspond au motif sont listées. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

`\dC [ motif ]`
Liste les conversions de types. Si *`motif`* est indiqué, seules sont affichées les conversions dont le type source ou cible correspond au motif.

`\dd[S] [ motif ]`
Affiche les descriptions des objets correspondant au *`motif`* ou de tous les objets si aucun argument n'est donné. Mais dans tous les cas, seuls les objets qui ont une description sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes. Le terme « objet » couvre les agrégats, les fonctions, les opérateurs, les types, les relations (tables, vues, index, séquences, objets larges), les règles et les déclencheurs. Par exemple, :

```
=> \dd version
                        Object descriptions
   Schema   |  Name   |  Object  |        Description
------------+---------+----------+---------------------------
 pg_catalog | version | function | PostgreSQL version string
(1 row)
```

Les descriptions des objets peuvent être ajoutées avec la commande SQL COMMENT(7).

`\ddp [ motif ]`
Liste les paramètres par défaut pour les privilèges d'accès. Une entrée est affichée pour chaque rôle (et schéma, si c'est approprié) pour lequel les paramètres par défaut des privilèges ont été modifiés par rapport aux paramètres par défaut intégrés. Si *`motif`* est spécifié, seules les entrées dont le nom de rôle ou le nom de schéma correspond au motif sont listées.

La commande ALTER DEFAULT PRIVILEGES(7) sert à positionner les privilèges d'accès par défaut. Le sens de l'affichage des privilèges est expliqué à la page de GRANT(7).

`\dD[S] [ motif ]`
Liste les domaines. Si *`motif`* est spécifié, seuls les domaines dont le nom correspond au motif sont affichés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

`\des[+] [ motif ]`
Liste les serveurs distants (mnémonique : « external servers »). Si *`motif`* est spécifié, seuls les serveurs dont le nom correspond au motif sont affichés. Si la forme `\des+` est utilisée, une description complète de chaque serveur est affichée, incluant la liste de contrôle d'accès du serveur (ACL), type, version et options.

`\deu[+] [ motif ]`
Liste les correspondances d'utilisateurs (mnémonique : « external users »). Si *`motif`* est spécifié, seules les correspondances dont le nom correspond au motif sont affichées. Si la forme `\deu+` est utilisée, des informations supplémentaires sur chaque

correspondance d'utilisateur sont affichées.

Attention

`\deu+` risque aussi d'afficher le nom et le mot de passe de l'utilisateur distant, il est donc important de faire attention à ne pas les divulguer.

`\dew[+] [ motif ]`
Liste les wrappers de données distants (mnémonique : « external wrappers »). Si *motif* est spécifié, seuls les wrappers dont le nom correspond au motif sont affichés. Si la forme `\dew+` est utilisée, les ACL et options du wrapper sont aussi affichées.

`\df[antwS+] [ pattern ]`
Liste les fonctions, ainsi que leurs arguments, types de retour, et types de fonctions, qui sont classés comme « agg » (agrégat), « normal », « trigger », or « window ». Afin de n'afficher que les fonctions d'un type spécifié, ajoutez les lettres correspondantes, respectivement `a`, `n`, `t`, or `w` à la commande. Si *motif* est spécifié, seules les fonctions dont le nom correspond au motif sont affichées. Si la forme `\df+` est utilisée, des informations supplémentaires sur chaque fonction, dont la volatibilité, le langage, le code source et la description, sont proposées. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

Astuce

Pour rechercher des fonctions prenant des arguments ou des valeurs de retour d'un type spécifique, utilisez les capacités de recherche du paginateur pour parcourir la sortie de `\df`.

`\dF[+] [ motif ]`
Liste les configurations de la recherche plein texte. Si *motif* est spécifié, seules les configurations dont le nom correspond au motif seront affichées. Si la forme `\dF+` est utilisée, une description complète de chaque configuration est affichée, ceci incluant l'analyseur de recherche plein texte et la liste de dictionnaire pour chaque type de jeton de l'analyseur.

`\dFd[+] [ motif ]`
Liste les dictionnaires de la recherche plein texte. Si *motif* est spécifié, seuls les dictionnaires dont le nom correspond au motif seront affichés. Si la forme `\dFd+` est utilisée, des informations supplémentaires sont affichées pour chaque dictionnaire, ceci incluant le motif de recherche plein texte et les valeurs des options.

`\dFp[+] [ motif ]`
Liste les analyseurs de la recherche plein texte. Si *motif* est spécifié, seuls les analyseurs dont le nom correspond au motif seront affichés. Si la forme `\dFp+` est utilisée, une description complète de chaque analyseur est affichée, ceci incluant les fonctions sous-jacentes et les types de jeton reconnu.

`\dFt[+] [ motif ]`
Liste les motifs de la recherche plein texte. Si *motif* est spécifié, seuls les motifs dont le nom correspond au motif seront affichés. Si la forme `\dFt+` est utilisée, des informations supplémentaires sont affichées pour chaque motif, ceci incluant les noms des fonctions sous-jacentes.

`\dg[+] [ motif ]`
Liste les rôles des bases de données. Si *motif* est spécifié, seuls les rôles dont le nom correspond au motif sont listés. (Cette commande est maintenant réellement identique à `\du`.) Si la forme `\dg+` est utilisée, des informations supplémentaires sont affichées pour chaque rôle, par exemple le commentaire.

`\di[S+] [ motif ]`, `\ds[S+] [ motif ]`, `\dt[S+] [ motif ]`, `\dv[S+] [ motif ]`
Dans ce groupe de commandes, les lettres `i`, `s`, `t` et `v` correspondent respectivement à index, séquence, table et vue. Vous pouvez indiquer n'importe quelle combinaison de ces lettres, dans n'importe quel ordre, pour obtenir la liste de tous les objets de ces types. Par exemple, `\dit` liste les index et tables. Si `+` est ajouté à la fin de la commande, chaque objet est listé avec sa taille physique sur disque et sa description associée s'il y en a une. Si *motif* est spécifié, seuls les objets dont les noms correspondent au motif sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

`\dl`
Ceci est un alias pour **\lo_list**, qui affiche une liste des objets larges.

`\dn[+] [ motif ]`
Liste les schémas. Si *motif* est spécifié, seuls les schémas dont le nom correspond au motif sont listés. Tout schéma temporaire non local est supprimé. Si `+` est ajouté à la fin de la commande, chaque objet est listé avec ses droits et description associés.

`\do[S] [ motif ]`

Liste les opérateurs avec leur opérande et type en retour. Si *motif* est spécifié, seuls les opérateurs dont le nom correspond au motif sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

`\dp [ motif ]`
Liste les tables, vues et séquences avec leur droits d'accès associés. Si *motif* est spécifié, seules les tables, vues et séquences dont le nom correspond au motif sont listées.

Les commandes GRANT(7) et REVOKE(7) sont utilisées pour configurer les droits d'accès. Les explications sur le sens de l'affichage des privilèges sont sous GRANT(7).

`\drds [ role-pattern [ database-pattern ] ]`
Liste les paramètres de configuration définis. Ces paramètres peuvent être spécifiques à un rôle, spécifiques à une base, ou les deux. *role-pattern* et *database-pattern* servent à choisir sur quels rôles spécifiques ou quelles bases de données - respectivement - les paramètres sont listés. Si ces options sont omises, ou si on spécifie `*`, tous les paramètres sont listés, y compris ceux qui ne sont pas spécifiques à un rôle ou à une base, respectivement.

Les commande ALTER ROLE(7) et ALTER DATABASE(7) servent à définir les paramètres de configuration par rôle et par base de données.

`\dT[S+] [ motif ]`
Liste les types de données. Si *motif* est spécifié, seuls les types dont le nom correspond au motif sont affichés. Si `+` est ajouté à la fin de la commande, chaque type est listé avec son nom interne et sa taille, ainsi que ses valeurs autorisées si c'est un type enum. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets systèmes.

`\du [ motif ]`
Liste les rôles de la base de données. Si *motif* est spécifié, seuls les rôles dont le nom correspond au motif sont affichés. Si la forme `\du+` est utilisée, des informations supplémentaires sont affichées pour chaque rôle, par exemple le commentaire.

`\edit` (ou `\e`) `[ nomfichier ]`
Si *nomfichier* est spécifié, le fichier est édité ; en quittant l'éditeur, son contenu est recopié dans le tampon de requête. Si aucun argument n'est fourni, le tampon de requête actuel est copié dans un fichier temporaire qui est ensuite édité de la même façon.

Le nouveau tampon de requête est ensuite ré-analysé suivant les règles habituelles de psql, où le tampon complet est traité comme une seule ligne. (Du coup, vous ne pouvez pas faire de scripts de cette façon. Utilisez **\i** pour cela.) Ceci signifie aussi que si la requête se termine avec (ou plutôt contient) un point-virgule, elle est immédiatement exécutée. Dans les autres cas, elle attend simplement dans le tampon de requête.

Astuce

psql recherche les variables d'environnement `PSQL_EDITOR`, `EDITOR` et `VISUAL` (dans cet ordre) pour connaître l'éditeur à utiliser. Si aucun n'est initialisé, `vi` est utilisé sur les systèmes Unix, `notepad.exe` sur les systèmes Windows.

`\ef [ description_fonction ]`
Cette commande récupère et édite la définition de la fonction désignée au moyen d'une commande **CREATE OR REPLACE FUNCTION**. L'édition est faite de la même façon que pour `\e`. Après que l'éditeur se soit fermé, la commande mise à jour attend dans le tampon de requête ; tapez `;` ou `\g` pour l'envoyer, ou `\r` pour l'annuler.

La fonction cible peut être spécifiée par son nom seul, ou par son nom et ses arguments, par exemple `foo(integer, text)`. Les types d'arguments doivent être fournis s'il y a plus d'une fonction du même nom.

Si aucune fonction n'est spécifiée, un modèle d'ordre **CREATE FUNCTION** vierge est affiché pour édition.

`\echo texte [ ... ]`
Affiche les arguments sur la sortie standard séparés par un espace et suivi par une nouvelle ligne. Ceci peut être utile pour intégrer des informations sur la sortie des scripts. Par exemple :

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

Si le premier argument est `-n` sans guillemets, alors la fin de ligne n'est pas écrite.

Astuce

Si vous utilisez la commande **\o** pour rediriger la sortie de la requête, vous pourriez souhaiter utiliser **\qecho** au

lieu de cette commande.

`\encoding [ codage ]`
Initialise l'encodage du jeu de caractères du client. Sans argument, cette commande affiche l'encodage actuel.

`\f [ chaîne ]`
Initialise le champ séparateur pour la sortie de requête non alignée. La valeur par défaut est la barre verticale (`|`). Voir aussi **\pset** comme moyen générique de configuration des options d'affichage.

`\g [ { nomfichier | |commande } ]`
Envoie le tampon de requête en entrée vers le serveur et stocke en option la sortie de la requête dans *nomfichier* ou envoie dans un tube la sortie vers un autre shell Unix exécutant *commande*. Un simple `\g` est virtuellement équivalent à un point-virgule. Un `\g` avec argument est une alternative en « un coup » à la commande **\o**.

`\help` (ou `\h`) `[ commande ]`
Donne la syntaxe sur la commande SQL spécifiée. Si *commande* n'est pas spécifiée, alors psql liste toutes les commandes pour lesquelles une aide en ligne est disponible. Si *commande* est un astérisque (`*`), alors l'aide en ligne de toutes les commandes SQL est affichée.

Note

Pour simplifier la saisie, les commandes qui consistent en plusieurs mots n'ont pas besoin d'être entre guillemets. Du coup, il est correct de saisir **`\help alter table`**.

`\H`
Active le format d'affichage HTML des requêtes. Si le format HTML est déjà activé, il est basculé au format d'affichage défaut (texte aligné). Cette commande est pour la compatibilité mais voir **\pset** pour configurer les autres options d'affichage.

`\i nomfichier`
Lit l'entrée à partir du fichier *nomfichier* et l'exécute comme si elle avait été saisie sur le clavier.

Note

Si vous voulez voir les lignes sur l'écran au moment de leur lecture, vous devez initialiser la variable `ECHO` à `all`.

`\l` (ou `\list`), `\l+` (ou `\list+`)
Liste les noms, propriétaires, encodage de jeux de caractères et droits d'accès de toutes les bases du serveur. Si `+` est ajouté à la fin de la commande, la taille des bases, les tablespaces par défaut et les descriptions sont aussi affichées. (Les tailles ne sont disponibles que pour les bases auxquelles l'utilisateur courant a le droit de se connecter.)

`\lo_export loid nomfichier`
Lit l'objet large d'OID *loid* à partir de la base de données et l'écrit dans *nomfichier*. Notez que ceci est subtilement différent de la fonction serveur `lo_export`, qui agit avec les droits de l'utilisateur avec lequel est exécuté le serveur de base de données et sur le système de fichiers du serveur.

Astuce

Utilisez **\lo_list** pour trouver l'OID de l'objet large.

`\lo_import nomfichier [ commentaire ]`
Stocke le fichier dans un objet large PostgreSQL™. En option, il associe le commentaire donné avec l'objet. Exemple :

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'une
photo de moi'
lo_import 152801
```

La réponse indique que l'objet large a reçu l'ID 152801, qui peut être utilisé pour accéder de nouveau à l'objet créé. Pour une meilleure lisibilité, il est recommandé de toujours associer un commentaire compréhensible par un humain avec chaque objet. Les OID et les commentaires sont visibles avec la commande **\lo_list**.

Notez que cette commande est subtilement différente de la fonction serveur `lo_import` car elle agit en tant qu'utilisateur local sur le système de fichier local plutôt qu'en tant qu'utilisateur du serveur et de son système de fichiers.

`\lo_list`
Affiche une liste de tous les objets larges PostgreSQL™ actuellement stockés dans la base de données, avec tous les commen-

taires fournis par eux.

`\lo_unlink` *`loid`*
Supprime l'objet large d'OID *`loid`* de la base de données.

Astuce

Utilisez **\lo_list** pour trouver l'OID d'un objet large.

`\o` [{*`nomfichier`* | `|`*`commande`*}]
Sauvegarde les résultats des requêtes suivantes dans le fichier *`nomfichier`* ou envoie via un tube les résultats à venir dans un shell Unix séparé pour exécuter *`command`*. Si aucun argument n'est spécifié, l'affichage de la requête est redirigé vers la sortie standard.

Les « résultats de requête » incluent toutes les tables, réponses de commande et messages d'avertissement obtenus du serveur de bases de données, ainsi que la sortie de différentes commandes antislash qui envoient des requêtes à la base de données (comme **\d**), mais sans message d'erreur.

Astuce

Pour intégrer du texte entre les résultats de requête, utilisez **\qecho**.

`\p`
Affiche le tampon de requête actuel sur la sortie standard.

`\password [ nom_utilisateur ]`
Modifie le mot de passe de l'utilisateur indiqué (par défaut, l'utilisateur en cours). Cette commande demande le nouveau mot de passe, le chiffre et l'envoie au serveur avec la commande **ALTER ROLE**. Ceci vous assure que le nouveau mot de passe n'apparaît pas en clair dans l'historique de la commande, les traces du serveur ou encore ailleurs.

`\prompt [ texte ] nom`
Demande à l'utilisateur la valeur pour la variable *`nom`*. Un affichage optionnel, *`texte`*, peut être proposé. (Pour des invites à plusieurs mots, utilisez les guillemets simples.)

Par défaut, `\prompt` utilise le terminal pour les entrées et sorties. Néanmoins, si la bascule `-f` est utilisée, `\prompt` utilise l'entrée et la sortie standard.

`\pset option [ valeur ]`
Cette commande initialise les options affectant l'affichage des tables résultat de la requête. *`option`* décrit l'option à initialiser. La sémantique de *`valeur`* varie en fonction de l'option sélectionnée. Pour certaines options, omettre *`valeur`* a pour conséquence de basculer ou désactiver l'option, tel que cela est décrit pour chaque option. Si aucun comportement de ce type n'est mentionné, alors omettre *`valeur`* occasionne simplement l'affichage de la configuration actuelle.

Les options ajustables d'affichage sont :

`format`
Initialise le format d'affichage parmi `unaligned`, `aligned`, `wrapped`, `html`, `latex` ou `troff-ms`. Les abréviations uniques sont autorisées. (ce qui signifie qu'une lettre est suffisante.)

Le format `unaligned` écrit toutes les colonnes d'un enregistrement sur une seule ligne, séparées par le séparateur de champ courant. Ceci est utile pour créeer des sorties qui doivent être lues par d'autres programmes (au format séparé par des caractère tabulation ou par des virgules, par exemple).

Le format `aligned` est le format de sortie texte standard, lisible par les humains, joliement formaté ; c'est le format par défaut.

Le format `wrapped` est comme `aligned`, sauf qu'il retourne à la ligne dans les données de grande taille afin que la sortie tienne dans la largeur de colonne cible. La largeur cible est déterminée comme cela est décrit à l'option `columns`. Notez que psql n'essaie pas de revenir à la ligne dans les titres de colonnes. Par conséquent, si la largeur totale nécessaire pour le titre de colonne est plus grande que la largeur cible, le format `wrapped` se comporte de la même manière que `aligned`.

Les formats `html`, `latex`, and `troff-ms` produisent des tables destinées à être inclues dans des documents utilisant le langage de marques respectif. Ce ne sont pas des documents complets ! (Ce n'est pas dramatique en HTML mais en LaTeX vous devez avoir une structure de document complet.)

`border`
Le *`valeur`* doit être un nombre. En général, plus grand est ce nombre, plus les tables ont de bordure et de ligne mais ceci dépend du format. Dans le mode HTML, ceci sera traduit directement avec l'attribut `border=...`. Avec les autres, seules les

valeurs 0 (sans bordure), 1 (lignes internes de division) et 2 (forme de table) ont un sens.

`columns`
Positionne la largeur pour le format `wrapped` , ainsi que la largeur à partir de laquelle la sortie est suffisemment longue pour nécessiter le pager. Si l'option est positionnée à zéro (la valeur par défaut), la largeur de la colonne est contrôlée soit par la variable d'environnement `COLUMNS`, soit par la largeur d'écran détectée si `COLUMNS` n'est pas positionnée. De plus, si `columns` vaut zero, alors le format `wrapped` affecte seulement la sortie écran. Si `columns` ne vaut pas zéro, alors les sorties fichier et tubes (pipe) font l'objet de retours à la ligne à cette largeur également.

`linestyle`
Positionne le style des lignes de bordure sur `ascii`, `old-ascii` `unicode`. Les abréviations uniques sont autorisées. (Cela signifie qu'une lettre suffit.) La valeur par défaut est `ascii`. Cette option affecte seulement les formats de sortie `aligned` et `wrapped`.

Le style `ascii` utilise les caractères basiques ASCII . Les retours à la ligne dans les données sont représentés par un symbole `+` dans la marge de droite. Si le format `wrapped` est sélectionné, un retour chariot est ajouté à l'affichage pour les valeurs dont la taille à l'affichage est trop importante pour tenir dans une cellule de la colonne associée. Un point (`.`) est affiché dans la marge droite de la ligne avant le retour chariot et un autre point est affiché dans la marge gauche de la ligne suivante.

Le style `old-ascii` utilise des caractères basiques ASCII, utilisant le style de formatage utilisé dans PostgreSQL™ 8.4 and et les versions plus anciennes. Les retours à la ligne dans les données sont représentés par un symbole `:` à la place du séparateur de colonnes placé à gauche. Quand les données sont réparties sur plusieurs lignes sans qu'il y ait de caractère de retour à la ligne dans les données, un symbole `;` est utilisé à la place du séparateur de colonne de gauche.

Le style `unicode` utilise les caractères Unicode de dessin de boîte. Les retours à la ligne dans les données sont représentés par un symbole de retour à la ligne dans la marge de droite. Lorsque les données sont réparties sur plusieurs lignes, sans qu'il y ait de caractère de retour à la ligne dans les données, le symbole ellipse est affiché dans la marge de droite de la première ligne, et également dans la marge de gauche de la ligne suivante.

Quand le paramètre `border` vaut plus que zéro, cette option détermine également les caractères utilisés pour dessiner les lignes de bordure. Les simples caractères ASCII fonctionnent partout, mais les caractères Unicode sont plus jolis sur les affichages qui les reconnaissent.

`expanded` (ou `x`)
Si le paramètre *`valeur`* est précisé, il peut valoir soit `on` soit `off` qui activera ou désactivera le mode étendu. Si *`valeur`* est omis, la commande bascule entre le mode normal et le mode étendu. Quand le mode étendu est activé, les résultats d'une requête sont affichés sur deux colonnes, avec le nom de la colonne dans la partie gauche et les données dans la partie droite. Ce mode est utile si les données ne tiennent pas sur l'écran dans le mode normal, « horizontal ».

`null`
Positionne la chaîne de caractères à afficher à la place d'une valeur null. Par défaut, rien n'est affiché, ce qui peut facilement être confondu avec une chaîne de caractères vide. Par exemple, vous pouvez préférer afficher `\pset null '(null)'`.

`fieldsep`
Indique le séparateur de champ à utiliser dans le mode d'affichage non aligné. De cette façon, vous pouvez créer, par exemple, une sortie séparée par des tabulations ou des virgules, que d'autres programmes pourraient préférer. Pour configurer une tabulation comme champ séparateur, saisissez `\pset fieldsep '\t'`. Le séparateur de champ par défaut est `'|'` (une barre verticale).

`footer`
Si le paramètre *`valeur`* est précisé, il doit valoir soit `on`, soit `off`, ce qui a pour effet d'activer ou de désactiver l'affichage du pied de table (le compte : `(`*`n`*` rows)`). Si le paramètre *`valeur`* est omis, la commande bascule entre l'affichage du pied de table ou sa désactivation.

`numericlocale`
Si *`valeur`* est précisée, elle doit valoir soit `on`, soit `off` afin d'activer ou désactiver l'affichage d'un caractère dépendant de la locale pour séparer des groupes de chiffres à gauche du séparateur décimal. Si *`valeur`* est omise, la commande bascule entre la sortie numérique classique et celle spécifique à la locale.

`recordsep`
Indique le séparateur d'enregistrement (ligne) à utiliser dans le mode d'affichage non aligné. La valeur par défaut est un caractère de retour chariot.

`tuples_only` (ou `t`)
Si *`valeur`* est spécifiée, elle doit valoir soit `on`, soit `off`, ce qui va activer ou désactiver le mode "tuples seulement". Si *`valeur`* est omise, la commande bascule entre la sortie normale et la sortie "tuples seulement". La sortie normale comprend des informations supplémentaires telles que les entêtes de colonnes, les titres, et différents pieds. Dans le mode "tuples seulement", seules les données de la table sont affichées.

`title [ texte ]`
Initialise le titre de la table pour toutes les tables affichées ensuite. Ceci peut être utilisé pour ajouter des balises de description à l'affichage. Si aucun *valeur* n'est donné, le titre n'est pas initialisé.

`tableattr` (or `T`)
Précise les attributs qui seront placés à l'intérieur des balises HTML `table` tag, dans le format de sortie `html`. Ceci pourrait être par exemple `cellpadding` ou `bgcolor`. Notez que vous ne voulez probablement pas spécifier `border` car c'est pris en compte par `\pset border`. Si *valeur* n'est pas précisée, aucun attribut de table n'est positionné.

`pager`
Contrôle l'utilisation d'un paginateur pour les requêtes et les affichages de l'aide de psql. Si la variable d'environnement `PAGER` est configurée, la sortie est envoyée via un tube dans le programme spécifié. Sinon, une valeur par défaut dépendant de la plateforme (comme `more`) est utilisée.

Quand l'option `pager` vaut `off`, le paginateur n'est pas utilisé. Quand l'option `pager` vaut `on`, et que cela est approprié, c'est à dire quand la sortie est dirigée vers an terminal, et ne tient pas dans l'écran, le paginateur est utilisé. L'option `pager` peut également être positionnée à `always`, ce qui a pour effet d'utiliser le paginateur pour toutes les sorties terminal, que ces dernières tiennent ou non dans l'écran. `\pset pager` sans préciser *valeur* bascule entre les états "paginateur activé" et "paginateur désactivé".

Des exemples d'utilisation de ces différents formats sont disponibles dans la section la section intitulée « Exemples ».

Astuce

Il existe plusieurs raccourcis de commandes pour **\pset**. Voir **\a**, **\C**, **\H**, **\t**, **\T** et **\x**.

Note

C'est une erreur d'appeler **\pset** sans argument. Dans le futur, cet appel pourrait afficher le statut actuel de toutes les options d'affichage.

`\q` ou `\quit`
Quitte le programme psql. Avec un script, seule l'exécution du script est terminée.

`\qecho texte [ ... ]`
Cette commande est identique à **\echo** sauf que les affichages sont écrits dans le canal d'affichage des requêtes, configuré par **\o**.

`\r`
Réinitialise (efface) le tampon de requêtes.

`\s [ nomfichier ]`
Affiche ou sauvegarde l'historique des lignes de commandes dans *nomfichier*. Si *nomfichier* est omis, l'historique est écrit sur la sortie standard. Cette option est seulement disponible si psql est configuré pour utiliser la bibliothèque GNU Readline.

`\set [ nom [ valeur [ ... ]]]`
Initialise la variable interne *nom* à *valeur* ou, si plus d'une valeur est donnée, à la concaténation de toutes les valeurs. Si aucun second argument n'est donné, la variable est simplement initialisée sans valeur. Pour désinitialiser une variable, utilisez la commande **\unset**.

Les noms de variables valides peuvent contenir des caractères, chiffres et tirets bas. Voir la section la section intitulée « Variables » ci-dessous pour les détails. Les noms des variables sont sensibles à la casse.

Bien que vous puissiez configurer toute variable comme vous le souhaitez, psql traite certaines variables de façon spéciale. Elles sont documentées dans la section sur les variables.

Note

Cette commande est totalement séparée de la commande SQL SET(7).

`\t`
Bascule l'affichage des en-têtes de nom de colonne en sortie et celle du bas de page indiquant le nombre de lignes. Cette commande est équivalente à `\pset tuples_only` et est fournie pour en faciliter l'accès.

`\T options_table`

Spécifie les attributs qui seront placés dans le tag `table` pour le format de sortie HTML. Cette commande est équivalente à `\pset tableattr` *`options_table`*.

`\timing [ on | off ]`
Sans paramètre, affiche le temps pris par chaque instruction SQL, en millisecondes, ou arrête cet affichage. Avec paramètre, force la valeur au paramètre.

`\w` *`nomfichier`*, `\w |`*`commande`*
Place le tampon de requête en cours dans le fichier *`nomfichier`* ou l'envoie via un tube à la commande Unix *`commande`*.

`\x`
Bascule le mode étendu de formatage en table. C'est équivalent à `\pset expanded`.

`\z [ motif ]`
Liste les tables, vues et séquences avec leur droit d'accès associé. Si un *`motif`* est spécifié, seules les tables, vues et séquences dont le nom correspond au motif sont listées.

Ceci est un alias pour **\dp** (« affichage des droits »).

`\! [ commande ]`
Lance un shell Unix séparé ou exécute la commande Unix *`commande`*. Les arguments ne sont pas interprétés, le shell les voit tel quel.

`\?`
Affiche l'aide sur les commandes antislash.

motifs

Les différentes commandes `\d` acceptent un paramètre *`motif`* pour spécifier le(s) nom(s) d'objet à afficher. Dans le cas le plus simple, un motif est seulement le nom exact de l'objet. Les caractères à l'intérieur du motif sont normalement mis en minuscule comme pour les noms SQL ; par exemple, `\dt FOO` affichera la table nommée `foo`. Comme pour les noms SQL, placer des guillemets doubles autour d'un motif empêchera la mise en minuscule. Si vous devez inclure un guillemet double dans un motif, écrivez-le en double en accord avec les règles sur les identifiants SQL. Par exemple, `\dt "FOO""BAR"` affichera la table nommée `FOO"BAR` (et non pas `foo"bar`). Contrairement aux règles normales pour les noms SQL, vous pouvez placer des guillemets doubles simplement autour d'une partie d'un motif, par exemple `\dt FOO"FOO"BAR` affichera la table nommée `fooFOObar`.

Lorsque le paramètre *`motif`* est complètement absent, la commande `\d` affiche tous les objets visibles dans le chemin de recherche courant -- cela est équivalent à l'utilisation du motif `*`. (Un objet est dit *visible* si le schéma qui le contient est dans le chemin de recherche et qu'aucun objet de même type et même nom n'apparaît en priorité dans le chemin de recherche. Cela est équivalent à dire que l'objet peut être référencé par son nom sans préciser explicitement le schéma.) Pour voir tous les objets de la base quelle que soit leur visibilité, utilisez le motif `*.*` .

À l'intérieur d'un motif, `*` correspond à toute séquence de caractères (et aussi à aucun) alors que `?` ne correspond qu'à un seul caractère. (Cette notation est comparable à celle des motifs de nom de fichier Unix.) Par exemple, `\dt int*` affiche les tables dont le nom commence avec `int`. Mais à l'intérieur de guillemets doubles, `*` et `?` perdent leurs significations spéciales et sont donc traités directement.

Un motif qui contient un point (`.`) est interprété comme le motif d'un nom de schéma suivi par celui d'un nom d'objet. Par exemple, `\dt foo*.*bar*` affiche toutes les tables dont le nom inclut `bar` et qui sont dans des schémas dont le nom commence avec `foo`. Sans point, le motif correspond seulement avec les objets qui sont visibles dans le chemin de recherche actuel des schémas. De nouveau, un point dans des guillemets doubles perd sa signification spéciale et est traité directement.

Les utilisateurs avancés peuvent utiliser des expressions rationnelles comme par exemple les classes de caractère (`[0-9]` pour tout chiffre). Tous les caractères spéciaux d'expression rationnelle fonctionnent de la façon indiquée dans Section 9.7.3, « Expressions rationnelles POSIX », sauf pour le `.` qui est pris comme séparateur (voir ci-dessus), l'étoile (`*`) qui est transformée en l'expression rationnelle `.*` et `?` qui est transformée en `.`, et `$` qui est une correspondance littérale. Vous pouvez émuler ces caractères si besoin en écrivant `?` pour `.`, `(`*`R`*`+|)` pour *`R`*`*` et `(`*`R`*`|)` pour *`R`*`?`. `$` n'est pas nécessaire en tant que caractère d'une expression rationnelle car le motif doit correspondre au nom complet, contrairement à l'interprétation habituelle des expressions rationnelles (en d'autres termes, `$` est ajouté automatiquement à votre motif). Écrivez `*` au début et/ou à la fin si vous ne souhaitez pas que le motif soit ancré. Notez qu'à l'intérieur de guillemets doubles, tous les caractères spéciaux des expressions rationnelles perdent leur signification spéciale et sont traités directement. De plus, ces caractères sont traités littéralement dans les motifs des noms d'opérateurs (par exemple pour l'argument de `\do`).

Fonctionnalités avancées

Variables

psql fournit des fonctionnalités de substitution de variable similaire aux shells de commandes Unix. Les variables sont simplement

des paires nom/valeur où la valeur peut être toute chaîne, quel que soit sa longueur. Pour initialiser des variables, utilisez la méta-commande psql **\set** :

```
basetest=> \set foo bar
```

initialise la variable `foo` avec la valeur `bar`. Pour récupérer le contenu de la variable, précédez le nom avec un caractère deux-points. Vous pouvez l'utiliser comme argument de toute commande slash :

```
basetest=> \echo :foo
bar
```

> **Note**
>
> Les arguments de **\set** sont sujets aux même règles de substitution que les autres commandes. Du coup, vous pouvez construire des références intéressantes comme `\set :foo 'quelquechose'` et obtenir des « liens doux » ou des « variables de variables » comme, respectivement, Perl™ ou PHP™. Malheureusement (ou heureusement ?), on ne peut rien faire d'utile avec ces constructions. D'un autre côté, `\set bar :foo` est un moyen parfaitement valide de copier une variable.

Si vous appelez **\set** sans second argument, la variable est initialisée avec une chaîne vide. Pour désinitialiser (ou supprimer) une variable, utilisez la commande **\unset**.

Les noms de variables internes de psql peuvent être constitués de lettres, nombres et tirets bas dans n'importe quel ordre et autant de fois que vous le voulez. Un certain nombre de ces variables sont traitées spécialement par psql. Elles indiquent certaines options qui peuvent changer au moment de l'exécution en modifiant la valeur de la variable ou représentent un certain état de l'application. Bien que vous puissiez utiliser ces variables dans n'importe quel but, ce n'est pas recommandé car le comportement du programme pourrait devenir très rapidement vraiment étrange. Par convention, toutes les variables traitées spécialement sont uniquement composées de lettres majuscules (et peut-être aussi de chiffres et de tirets bas). Pour s'assurer d'une compatibilité maximum dans le futur, évitez l'utilisation de tels noms de variables pour vos propres besoins. Une liste de toutes les variables traitées spécialement suit.

`AUTOCOMMIT`

Si actif (`on`, valeur par défaut), chaque commande SQL est automatiquement validée si elle se termine avec succès. Pour suspendre la validation dans ce mode, vous devez saisir une commande SQL **BEGIN** ou **START TRANSACTION**. Lorsqu'elle est désactivée (`off`) ou non initialisée, les commandes SQL ne sont plus validées tant que vous ne lancez pas explicitement **COMMIT** ou **END**. Le mode sans autocommit fonctionne en lançant implicitement un **BEGIN**, juste avant toute commande qui n'est pas déjà dans un bloc de transaction et qui n'est pas elle-même un **BEGIN** ou une autre commande de contrôle de transaction, ou une commande qui ne peut pas être exécutée à l'intérieur d'un bloc de transaction (comme **VACUUM**).

> **Note**
>
> Dans le mode sans autocommit, vous devez annuler explicitement toute transaction échouée en saisissant **ABORT** ou **ROLLBACK**. Gardez aussi en tête que si vous sortez d'une session sans validation, votre travail est perdu.

> **Note**
>
> Le mode auto-commit est le comportement traditionnel de PostgreSQL™ alors que le mode sans autocommit est plus proche des spécifications SQL. Si vous préférez sans autocommit, vous pouvez le configurer dans le fichier `psqlrc` global du système ou dans votre fichier `~/.psqlrc`.

`DBNAME`

Le nom de la base de données à laquelle vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

`ECHO`

Si cette variable est initialisée à `all`, toutes les lignes saisies au clavier ou provenant d'un script sont écrites sur la sortie standard avant d'être analysées ou exécutées. Pour sélectionner ce comportement au lancement du programme, utilisez l'option `-a`. Si `ECHO` vaut `queries`, psql affiche simplement toutes les requêtes au moment de leur envoi au serveur. L'option pour ceci est `-e`.

`ECHO_HIDDEN`

Quand cette variable est initialisée et qu'une commande antislash est envoyée à la base de données, la requête est d'abord affi-

chée. De cette façon, vous pouvez étudier le fonctionnement interne de PostgreSQL™ et fournir des fonctionnalités similaires dans vos propres programmes. (Pour sélectionner ce comportement au lancement du programme, utilisez l'option `-E`.) Si vous configurez la variable avec la valeur `noexec`, les requêtes sont juste affichées mais ne sont pas réellement envoyées au serveur ni exécutées.

`ENCODING`
Le codage courant du jeu de caractères du client.

`FETCH_COUNT`
Si cette variable est un entier positif, les résultats de la requête **SELECT** sont récupérés et affichés en groupe de ce nombre de lignes, plutôt que par le comportement par défaut (récupération de l'ensemble complet des résultats avant l'affichage). Du coup, seule une petite quantité de mémoire est utilisée, quelle que soit la taille de l'ensemble des résultats. Une configuration entre 100 et 1000 est habituellement utilisée lors de l'activation de cette fonctionnalité. Gardez en tête que lors de l'utilisation de cette fonctionnalité, une requête pourrait échouer après avoir affiché quelques lignes.

Astuce

Bien que vous puissiez utiliser tout format de sortie avec cette fonctionnalité, le format par défaut, `aligned`, rend mal car chaque groupe de `FETCH_COUNT` lignes sera formaté séparément, modifiant ainsi les largeurs de colonnes suivant les lignes du groupe. Les autres formats d'affichage fonctionnent mieux.

`HISTCONTROL`
Si cette variable est configurée à `ignorespace`, les lignes commençant avec un espace n'entrent pas dans la liste de l'historique. Si elle est initialisée avec la valeur `ignoredups`, les lignes correspondant aux précédentes lignes de l'historique n'entrent pas dans la liste. Une valeur de `ignoreboth` combine les deux options. Si elle n'est pas initialisée ou si elle est configurée avec une autre valeur que celles-ci, toutes les lignes lues dans le mode interactif sont sauvegardées dans la liste de l'historique.

Note

Cette fonctionnalité a été plagiée sur Bash.

`HISTFILE`
Le nom du fichier utilisé pour stocker l'historique. La valeur par défaut est `~/.psql_history`. Par exemple, utiliser :

```
\set HISTFILE ~/.psql_history- :DBNAME
```

dans `~/.psqlrc` fera que psql maintiendra un historique séparé pour chaque base de données.

Note

Cette fonctionnalité a été plagiée sans honte à partir de Bash.

`HISTSIZE`
Le nombre de commandes à stocker dans l'historique des commandes. La valeur par défaut est 500.

Note

Cette fonctionnalité a été plagiée sur Bash.

`HOST`
L'hôte du serveur de la base de données où vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

`IGNOREEOF`
Si non initialisé, envoyer un caractère EOF (habituellement **Ctrl**+**D**) dans une session interactive de psql ferme l'application. Si elle est configurée avec une valeur numérique, ce nombre de caractères EOF est ignoré avant la fin de l'application. Si la variable est configurée mais n'a pas de valeur numérique, la valeur par défaut est de 10.

Note

Cette fonctionnalité a été plagiée sur Bash.

`LASTOID`

La valeur du dernier OID affecté, renvoyée à partir d'une commande **INSERT** ou **lo_import**. La validité de cette variable est seulement garantie jusqu'à l'affichage du résultat de la commande SQL suivante.

`ON_ERROR_ROLLBACK`

Lorsqu'il est actif (`on`), si une instruction d'un bloc de transaction génère une erreur, cette dernière est ignorée et la transaction continue. Lorsqu'il vaut `interactive`, ces erreurs sont seulement ignorées lors des sessions interactives, mais ne le sont pas lors de la lecture de scripts. Lorsqu'il vaut `off` (valeur par défaut), une instruction générant une erreur dans un bloc de transaction annule la transaction complète. Le mode on_error_rollback-on fonctionne en exécutant un **SAVEPOINT** implicite pour vous, juste avant chaque commande se trouvant dans un bloc de transaction et annule jusqu'au dernier point de sauvegarde en cas d'erreur.

`ON_ERROR_STOP`

Par défaut, si les scripts non interactifs rencontrent une erreur, comme une commande SQL mal formée ou une méta-commande interne, le traitement continue. Ceci a été le comportement traditionnel de psql mais il est quelque fois indésirable. Si cette variable est configurée, le traitement du script s'arrête immédiatement. Si le script a été appelé à partir d'un autre script, il se termine de la même façon. Si le script le plus externe n'a pas été appelé à partir d'une session interactive de psql mais plutôt en utilisant l'option `-f`, psql renvoie le code erreur 3 pour distinguer ce cas des conditions d'erreurs fatales (code d'erreur 1).

`PORT`

Le port du serveur de la base de données sur lequel vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

`PROMPT1`, `PROMPT2`, `PROMPT3`

Ils spécifient à quoi doit ressembler l'invite psql. Voir la section intitulée « Invite » ci-dessous.

`QUIET`

Cette variable est équivalente à l'option `-q` en ligne de commande. Elle n'est probablement pas très utile en mode interactif.

`SINGLELINE`

Cette variable est équivalente à l'option `-S` en ligne de commande.

`SINGLESTEP`

Cette variable est équivalente à l'option `-s` en ligne de commande.

`USER`

L'utilisateur de la base de données où vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être désinitialisé.

`VERBOSITY`

Cette variable peut être configurée avec les valeurs `default`, `verbose` (bavard) ou `terse` (succinct) pour contrôler la verbosité des rapports d'erreurs.

Interpolation SQL

Une fonctionnalité utile supplémentaire des variables psql est que vous pouvez les substituer (« interpoler ») dans les instructions SQL standards. psql offre des fonctionalités spéciales pour garantir que les valeurs utilisées comme chaînes SQL litérales ou comme identifiants sont correctement échappées. La syntaxe pour interpoler une valeur sans échappement spécial est de nouveau de précéder le nom de la variable avec un caractère deux-points (`:`) :

```
basetest=> \set foo 'ma_table'
basetest=> SELECT * FROM :foo;
```

envoie alors la requête pour la table `ma_table`. Notez que cela peut être dangereux ; la valeur de la variable est copiée de façon litérale, elle peut même contenir des guillemets non fermés, ou bien des commandes backslash. Vous devez vous assurer que cela a du sens à l'endroit où vous les utilisez.

Lorsqu'une valeur doit être utilisée comme une chaîne SQL litérale ou un identifiant, il est plus sûr de s'arranger pour qu'elle soit échappée. Afin d'échapper la valeur d'une variable en tant que chaîne SQL litérale, écrivez un caractère deux-points, suivi du nom de la variable entouré par des guillemets simples. Pour échapper la valeur en tant qu'identifiant SQL, écrivez un caractère deux-points suivi du nom de la valeur entouré de guillemets doubles. L'exemple précédent peut s'écrire de façon plus sûre ainsi :

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

L'interpolation de variable n'est pas effectuée dans les entités SQL entourées de guillemets. SQL

Une utilisation possible de ce mécanisme est de copier le contenu d'un fichier dans une colonne d'une table. Tout d'abord, chargez le fichier dans une variable puis procédez ainsi :

```
basetest=> \set contenu `cat mon_fichier.txt`
basetest=> INSERT INTO ma_table VALUES (:'contenu');
```

(Notez que cela ne fonctionnera par si le fichier `mon_fichier.txt` contient des octets nuls. psql ne gère pas les octets nuls inclus dans les valeurs de variable.)

Comme les caractères deux-points peuvent légitimement apparaître dans les commandes SQL, une tentative apparente d'interpolation (comme `:nom`, `:'nom'`, or `:"nom"`) n'est pas modifiée, sauf si la variable nommée est actuellement positionnée. Dans tous les cas, vous pouvez échapper un caractère deux-points avec un backslash pour le protéger des substitutions. (La syntaxe deux-points pour les variables est du SQL standard pour les langages de requête embarqués, comme ECPG. La syntaxe avec les deux-points pour les tranches de tableau et les conversions de types sont des extensions PostgreSQL™ extensions, d'où le conflit. La syntaxe avec le caractère deux-points pour échapper la valeur d'une variable en tant que chaîne SQL litérale ou identifiant est une extension psql .)

Invite

Les invites psql peuvent être personnalisées suivant vos préférences. Les trois variables `PROMPT1`, `PROMPT2` et `PROMPT3` contiennent des chaînes et des séquences d'échappement spéciales décrivant l'apparence de l'invite. L'invite 1 est l'invite normale qui est lancée quand psql réclame une nouvelle commande. L'invite 2 est lancée lorsqu'une saisie supplémentaire est attendue lors de la saisie de la commande parce que la commande n'a pas été terminée avec un point-virgule ou parce qu'un guillemet n'a pas été fermé. L'invite 3 est lancée lorsque vous exécutez une commande SQL **COPY** et que vous devez saisir les valeurs des lignes sur le terminal.

La valeur de la variable prompt sélectionnée est affichée littéralement sauf si un signe pourcentage (`%`) est rencontré. Suivant le prochain caractère, certains autres textes sont substitués. Les substitutions définies sont :

`%M`
: Le nom complet de l'hôte (avec le nom du domaine) du serveur de la base de données ou `[local]` si la connexion est établie via une socket de domaine Unix ou `[local:`*`/répertoire/nom`*`]`, si la socket de domaine Unix n'est pas dans l'emplacement par défaut défini à la compilation.

`%m`
: Le nom de l'hôte du serveur de la base de données, tronqué au premier point ou `[local]` si la connexion se fait via une socket de domaine Unix.

`%>`
: Le numéro de port sur lequel le serveur de la base de données écoute.

`%n`
: Le nom d'utilisateur de la session. (L'expansion de cette valeur peut changer pendant une session après une commande **SET SESSION AUTHORIZATION**.)

`%/`
: Le nom de la base de données courante.

`%~`
: Comme `%/` mais l'affichage est un `~` (tilde) si la base de données est votre base de données par défaut.

`%#`
: Si l'utilisateur de la session est un superutilisateur, alors un `#` sinon un `>`. (L'expansion de cette valeur peut changer durant une session après une commande **SET SESSION AUTHORIZATION**.)

`%R`
: Pour l'invite 1, affiche normalement `=` mais affiche `^` si on est en mode simple ligne et `!` si la session est déconnectée de la base de données (ce qui peut arriver si **\connect** échoue). Pour l'invite 2, la séquence est remplacée par `-`, `*`, un simple guillemet, un double ou un signe dollar, suivant si psql attend une saisie supplémentaire parce que la commande n'est pas terminée, parce que vous êtes à l'intérieur d'un commentaire `/* ... */`, ou parce que vous n'avez pas terminé un guillemet ou une chaîne échappée avec des dollars. Pour l'invite 3, la séquence ne produit rien.

`%x`
: État de la Transaction : une chaîne vide lorsque vous n'êtes pas dans un bloc de transaction ou `*` si vous vous y trouvez, ou `!` si vous êtes dans une transaction échouée, ou enfin `?` lorsque l'état de la transaction est indéterminé (par exemple à cause d'une rupture de la connexion).

`%`*`chiffres`*
: Le caractère avec ce code numérique est substitué.

`%:`*`nom`*`:`
: La valeur de la variable *`nom`* de psql. Voir la section la section intitulée « Variables » pour les détails.

`%`commande``
la sortie de la *commande*, similaire à la substitution par « guillemets inverse » classique.

`%[ ... %]`
Les invites peuvent contenir des caractères de contrôle du terminal qui, par exemple, modifient la couleur, le fond ou le style du texte de l'invite, ou modifient le titre de la fenêtre du terminal. Pour que les fonctionnalités d'édition de ligne de Readline fonctionnent correctement, les caractères de contrôle non affichables doivent être indiqués comme invisibles en les entourant avec `%[` et `%]`. Des pairs multiples de ceux-ci pourraient survenir à l'intérieur de l'invite. Par exemple :

```
basetest=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

a pour résultat une invite en gras (`1;`), jaune sur noir (`33;40`) sur les terminaux compatibles VT100.

Pour insérer un pourcentage dans votre invite, écrivez `%%`. Les invites par défaut sont `'%/%R%# '` pour les invites 1 et 2 et `'>> '` pour l'invite 3.

> **Note**
>
> Cette fonctionnalité a été plagiée sur tcsh.

Édition de la ligne de commande

psql supporte la bibliothèque Readline pour une édition et une recherche simplifiée et conviviale de la ligne de commande. L'historique des commandes est automatiquement sauvegardé lorsque psql quitte et est rechargé quand psql est lancé. La complétion par tabulation est aussi supportée bien que la logique de complétion n'ait pas la prétention d'être un analyseur SQL. Si pour quelques raisons que ce soit, vous n'aimez pas la complétion par tabulation, vous pouvez la désactiver en plaçant ceci dans un fichier nommé `.inputrc` de votre répertoire personnel :

```
$if psql
set disable-completion on
$endif
```

(Ceci n'est pas une fonctionnalité psql mais Readline. Lisez sa documentation pour plus de détails.)

Environnement

`COLUMNS`
Si `\pset columns` vaut zéro, contrôle la largeur pour le format `wrapped` et la largeur pour déterminer si une sortie large a besoin du pager.

`PAGER`
Si les résultats d'une requête ne tiennent pas sur l'écran, ils sont envoyés via un tube sur cette commande. Les valeurs typiques sont `more` ou `less`. La valeur par défaut dépend de la plateforme. L'utilisation du paginateur peut être désactivée en utilisant la commande **\pset**.

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut (voir Section 31.13, « Variables d'environnement »).

`PSQL_EDITOR`, `EDITOR`, `VISUAL`
Éditeur utilisé par la commande **\e**. Les variables sont examinées dans l'ordre donné ; la première initialisée est utilisée.

`SHELL`
Commande exécutée par la commande **\!**.

`TMPDIR`
Répertoire pour stocker des fichiers temporaires. La valeur par défaut est `/tmp`.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Fichiers

- Sauf si une option `-X` ou `-c` est fournie, psql tente de lire et exécuter les commandes provenant du fichier global au système `psqlrc` ou du fichier utilisateur `~/.psqlrc` avant de démarrer. (Sur Windows, le fichier de démarrage de l'utilisateur est nommé `%APPDATA%\postgresql\psqlrc.conf`.) Voir *`PREFIX`*`/share/psqlrc.sample` pour plus d'informations sur la configuration du fichier global au système. Il pourrait être utilisé pour configurer le client et le serveur à votre goût (en utilisant les commandes **\set** et **SET**).

- À la fois le fichier `psqlrc` global au système et le fichier `~/.psqlrc` de l'utilisateur peuvent être créés en étant spécifiques à une version si vous leur ajouter un tiret et le numéro de version de `~/.psqlrc-9.0.18`. Un fichier correspondant à une version spécifique est préféré à un fichier sans indication de version.
- L'historique de la ligne de commandes est stocké dans le fichier `~/.psql_history` ou `%APPDATA%\postgresql\psql_history` sur Windows.

Notes

- Dans une vie précédente, psql permettait au premier argument d'une commande antislash à une seule lettre de commencer directement après la commande, sans espace séparateur. À partir de PostgreSQL™ 8.4, ce n'est plus autorisé.
- Le fonctionnement de psql n'est garanti qu'avec des serveurs de même version. Cela ne signifie pas que d'autres combinaisons vont complètement échouer, mais des problèmes subtils, voire moins subtils, pourraient apparaître. Les méta-commandes sont particulièrements fragiles si le serveur est d'une version plus récente que psql lui-même. Toutefois, les commandes backslash de la famille `\d` devraient fonctionner avec des serveurs 7.4 jusqu'à la version courante, même si pas nécessairement avec des serveurs plus récents que psql lui-même.

Notes pour les utilisateurs sous Windows

psql est construit comme une « application de type console ». Comme les fenêtres console de windows utilisent un codage différent du reste du système, vous devez avoir une attention particulière lors de l'utilisation de caractères sur 8 bits à l'intérieur de psql. Si psql détecte une page de code problématique, il vous avertira au lancement. Pour modifier la page de code de la console, deux étapes sont nécessaires :

- Configurez la page code en saisissant **`cmd.exe /c chcp 1252`**. (1252 est une page code appropriée pour l'Allemagne ; remplacez-la par votre valeur.) Si vous utilisez Cygwin, vous pouvez placer cette commande dans `/etc/profile`.
- Configurez la police de la console par `Lucida Console` parce que la police raster ne fonctionne pas avec la page de code ANSI.

Exemples

Le premier exemple montre comment envoyer une commande sur plusieurs lignes d'entrée. Notez le changement de l'invite :

```
basetest=> CREATE TABLE ma_table (
basetest(>  premier integer not NULL default 0,
basetest(>  second text)
basetest-> ;
CREATE TABLE
```

Maintenant, regardons la définition de la table :

```
basetest=> \d ma_table
             Table "ma_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 premier   | integer | not null default 0
 second    | text    |
```

Maintenant, changeons l'invite par quelque chose de plus intéressant :

```
basetest=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost basetest=>
```

Supposons que nous avons rempli la table de données et que nous voulons les regarder :

```
peter@localhost basetest=> SELECT * FROM ma_table;
 premier | second
---------+--------
       1 | one
       2 | two
       3 | three
       4 | four
(4 rows)
```

Vous pouvez afficher cette table de façon différente en utilisant la commande **\pset** :

```
peter@localhost basetest=> \pset border 2
Border style is 2.
peter@localhost basetest=> SELECT * FROM ma_table;
+---------+--------+
| premier | second |
+---------+--------+
|       1 | one    |
|       2 | two    |
|       3 | three  |
|       4 | four   |
+---------+--------+
(4 rows)

peter@localhost basetest=> \pset border 0
Border style is 0.
peter@localhost basetest=> SELECT * FROM ma_table;
premier second
------- ------
      1 one
      2 two
      3 three
      4 four
(4 rows)

peter@localhost basetest=> \pset border 1
Border style is 1.
peter@localhost basetest=> \pset format unaligned
Output format is unaligned.
peter@localhost basetest=> \pset fieldsep ","
Field separator is ",".
peter@localhost basetest=> \pset tuples_only
Showing only tuples.
peter@localhost basetest=> SELECT second, first FROM
ma_table;
one,1
two,2
three,3
four,4
```

Vous pouvez aussi utiliser les commandes courtes :

```
peter@localhost basetest=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost basetest=> SELECT * FROM ma_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Nom

reindexdb — reindexe une base de données PostgreSQL™

Synopsis

`reindexdb` [*option-connexion*...] [--table | -t *table*] [--index | -i *index*] [*nombase*]

`reindexdb` [*option-connexion*...] [--all | -a]

`reindexdb` [*option-connexion*...] [--system | -s] [*nombase*]

Description

reindexdb permet de reconstruire les index d'une base de données PostgreSQL™.

reindexdb est un enrobage de la commande REINDEX(7). Il n'y a pas de différence entre la réindexation des bases de données par cette méthode et par celles utilisant d'autres méthodes d'accès au serveur.

Options

reindexdb accepte les arguments suivants en ligne de commande :

`-a`, `--all`
: Réindexe toutes les bases de données.

`[-d] base`, `[--dbname] base`
: Spécifie le nom de la base à réindexer. Si cette option n'est pas présente et que l'option `-a` (ou `--all`) n'est pas utilisée, le nom de la base est lu à partir de la variable d'environnement `PGDATABASE`. Si elle n'est pas configurée, le nom de l'utilisateur pour la connexion est utilié.

`-e`, `--echo`
: Affiche les commandes que reindexdb génère et envoie au serveur.

`-i index`, `--index index`
: Ne recrée que l'index *index*.

`[-d] nombase`, `[--dbname] nombase`
: Spécifie le nom de la base de données à réindexer. Si elle n'est pas spécifiée et que l'option `-a` (ou `--all`) n'est pas utilisée, le nom de la base de données est lu à partir de la variable d'environnement `PGDATABASE`. Si elle n'est pas initialisée, le nom de l'utilisateur spécifié pour la connexion est utilisé.

`-e`, `--echo`
: Affiche les commandes que reindexdb engendre et envoie au serveur.

`-s`, `--system`
: Réindexe les catalogues système de la base de données.

`-t table`, `--table table`
: Ne réindexe que la table *table*.

reindexdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h hôte`, `--host hôte`
: Précise le nom d'hôte de la machine hébergeant le serveur. Si cette valeur débute par une barre oblique ('/' ou slash), elle est utilisée comme répertoire de socket UNIX.

`-p port`, `--port port`
: Précise le port TCP ou le fichier de socket UNIX d'écoute.

`-U username`, `--username username`
: Nom de l'utilisateur à utiliser pour la connexion.

`-w`, `--no-password`
: Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force reindexdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car reindexdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, reindexdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`-V`, `--version`
Affiche la version de reindexdb puis quitte.

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de reindexdb, puis quitte

Environnement

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
Paramètres par défaut pour la connexion

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de difficultés, il peut être utile de consulter REINDEX(7) et psql(1), sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Notes

reindexdb peut avoir besoin de se connecter plusieurs fois au serveur PostgreSQL™. Afin d'éviter de saisir le mot de passe à chaque fois, on peut utiliser un fichier `~/.pgpass`. Voir Section 31.14, « Fichier de mots de passe » pour plus d'informations.

Exemples

Pour réindexer la base de données `test` :

```
$ reindexdb test
```

Pour réindexer la table `foo` et l'index `bar` dans une base de données nommée `abcd` :

```
$ reindexdb --table foo --index bar abcd
```

Voir aussi

REINDEX(7)

Nom

vacuumdb — récupère l'espace inutilisé et, optionnellement, analyse une base de données PostgreSQL™

Synopsis

`vacuumdb` [*option-de-connexion*...] [[--full] | [-f]] [[--freeze] | [-F]] [[--verbose] | [-v]] [[--analyze] | [-z]] [[--analyze-only] | [-Z]] [--table | -t *table* [(*colonne* [,...])]] [*base-de-donnees*]

`vacuumdb` [*options-de-connexion*...] [[--full] | [-f]] [[--freeze] | [-F]] [[--verbose] | [-v]] [[--analyze] | [-z]] [[--analyze-only] | [-Z]] [[--all] | [-a]]

Description

vacuumdb est un outil de nettoyage d'une base de données. vacuumdb peut également engendrer des statistiques internes utilisées par l'optimiseur de requêtes de PostgreSQL™.

vacuumdb est une surcouche de la commande VACUUM(7). Il n'y a pas de différence réelle entre exécuter des VACUUM et des ANALYZE sur les bases de données via cet outil et via d'autres méthodes pour accéder au serveur.

Options

vacuumdb accepte les arguments suivants sur la ligne de commande :

`-a`, `--all`
Nettoie toutes les bases de données.

`[-d]` *`base-de-donnees`*, `[--dbname]` *`base-de-donnees`*
Indique le nom de la base de données à nettoyer ou à analyser. Si aucun nom n'est pas précisé et si `-a` (ou `--all`) n'est pas utilisé, le nom de la base de données est récupéré dans la variable d'environnement `PGDATABASE`. Si cette variable n'est pas initialisée, c'est le nom d'utilisateur précisé pour la connexion qui est utilisé.

`-e`, `--echo`
Affiche les commandes que vacuumdb engendre et envoie au serveur.

`-f`, `--full`
Exécute un nettoyage « complet ».

`-F`, `--freeze`
« Gèle » agressivement les lignes.

`-q`, `--quiet`
N'affiche pas de message de progression.

`-t` *`table`* `[ (`*`colonne`*` [,...]) ]`, `--table` *`table`* `[ (`*`colonne`*` [,...]) ]`
Ne nettoie ou n'analyse que la table *`table`*. Des noms de colonnes peuvent être précisés en conjonction avec les options `--analyze` ou `--analyze-only`.

> **Astuce**
>
> Lorsque des colonnes sont indiquées, il peut être nécessaire d'échapper les parenthèses. (Voir les exemples plus bas.)

`-v`, `--verbose`
Affiche des informations détaillées durant le traitement.

`-V`, `--version`
Affiche la version de vacuumdb, puis quitte.

`-z`, `--analyze`
Calcule aussi les statistiques utilisées par le planificateur.

`-Z`, `--analyze-only`
Calcule seulement les statistiques utilisées par le planificateur (donc pas de VACUUM).

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de vacuumdb, puis quitte.

vacuumdb accepte aussi les arguments suivants comme paramètres de connexion :

`-h` *`hôte`*, `--host` *`hôte`*
Indique le nom d'hôte de la machine qui héberge le serveur de bases de données. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour la socket de domaine Unix.

`-p` *`port`*, `--port` *`port`*
Indique le port TCP ou le fichier local de socket de domaine Unix sur lequel le serveur attend les connexions.

`-U` *`utilisateur`*, `--username` *`utilisateur`*
Nom d'utilisateur pour la connexion.

`-w`, `--no-password`
Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`, `--password`
Force vacuumdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car vacuumdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, vacuumdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`, `PGHOST`, `PGPORT`, `PGUSER`
Paramètres de connexion par défaut.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Diagnostiques

En cas de difficultés, il peut être utile de consulter VACUUM(7) et psql(1), sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Notes

vacuumdb peut avoir besoin de se connecter plusieurs fois au serveur PostgreSQL™. Afin d'éviter de saisir le mot de passe à chaque fois, on peut utiliser un fichier `~/.pgpass`. Voir Section 31.14, « Fichier de mots de passe » pour plus d'informations.

Exemples

Pour nettoyer la base de données `test` :

```
$ vacuumdb test
```

Pour nettoyer et analyser une base de données nommée `grossebase` :

```
$ vacuumdb --analyze grossebase
```

Pour nettoyer la seule table `foo` dans une base de données nommée `xyzzy` et analyser la seule colonne `bar` de la table :

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

Voir aussi

VACUUM(7)

Applications relatives au serveur PostgreSQL

Cette partie contient des informations de référence concernant les applications et les outils relatifs au serveur PostgreSQL™. Ces commandes n'ont d'utilité que lancées sur la machine sur laquelle le serveur fonctionne. D'autres programmes utilitaires sont listés dans la Applications client de PostgreSQL.

Nom

initdb — Créer un nouveau « cluster »

Synopsis

`initdb` [*option*...] [--pgdata] | [-D]*répertoire*

Description

initdb crée une nouvelle grappe de bases de données, ou « cluster », PostgreSQL™. Un cluster est un ensemble de bases de données gérées par une même instance du serveur.

Créer un cluster consiste à :

- créer les répertoires dans lesquels sont stockées les données de la base ;
- créer les tables partagées du catalogue (tables partagées par tout le cluster) ;
- créer les bases de données `template1` et `postgres`.

Lors de la création ultérieure d'une base de données, tout ce qui se trouve dans la base `template1` est copié. (Ce qui implique que tout ce qui est installé dans `template1` est automatiquement copié dans chaque base de données créée par la suite.) La base de données `postgres` est une base de données par défaut à destination des utilisateurs, des outils et des applications tiers.

initdb tente de créer le répertoire de données indiqué. Il se peut que la commande n'est pas les droits nécessaires si le répertoire parent du répertoire de données indiqué est possédé par root. Dans ce cas, pour réussir l'initialisation, il faut créer un répertoire de données vide en tant que root, puis utiliser **chown** pour en donner la possession au compte utilisateur de la base de données. **su** peut alors être utilisé pour prendre l'identité de l'utilisateur de la base de données et exécuter **initdb**.

initdb doit être exécuté par l'utilisateur propriétaire du processus serveur parce que le serveur doit avoir accès aux fichiers et répertoires créés par **initdb**. Comme le serveur ne peut pas être exécuté en tant que root, il est impératif de ne pas lancer **initdb** en tant que root. (En fait, **initdb** refuse de se lancer dans ces conditions.)

initdb initialise la locale et l'encodage de jeu de caractères par défaut du cluster. L'encodage du jeu de caractères, l'ordre de tri (`LC_COLLATE`) et les classes d'ensembles de caractères (`LC_CTYPE`, c'est-à-dire majuscule, minuscule, chiffre) peuvent être configurés séparément pour chaque base de données à sa création. **initdb** détermine ces paramètres à partir de la base de données `template1` qui servira de valeur par défaut pour toutes les autres bases de données.

Pour modifier l'ordre de tri ou les classes de jeu de caractères par défaut, utilisez les options `--lc-collate` et `--lc-ctype`. Les ordres de tri autres que `C` et `POSIX` ont aussi un coût en terme de performance. Pour ces raisons, il est important de choisir la bonne locale lors de l'exécution d'**initdb**.

Les catégories de locale restantes peuvent être modifiées plus tard, lors du démarrage du serveur. Vous pouvez aussi utiliser `--locale` pour configurer les valeurs par défaut de toutes les catégories de locale, ceci incluant l'ordre de tri et les classes de jeu de caractères. Toutes les valeurs de locale côté serveur (`lc_*`) peuvent être affichées via la commande **SHOW ALL**. Il y a plus d'informations sur ce point sur Section 22.1, « Support des locales ».

Pour modifier l'encodage par défaut, utilisez l'option `--encoding`. Section 22.2, « Support des jeux de caractères » propose plus d'options.

Options

`-A` *méthode_auth*, `--auth=`*méthode_auth*
: Précise la méthode d'authentification utilisée dans `pg_hba.conf` pour les utilisateurs locaux. `trust` ne doit être utilisé que lorsque tous les utilisateurs locaux du système sont dignes de confiance. Pour faciliter l'installation, `trust` est la valeur par défaut.

`-D` *répertoire*, `--pgdata=`*répertoire*
: Indique le répertoire de stockage de la grappe de bases de données. C'est la seule information requise par **initdb**. Il est possible d'éviter de préciser cette option en configurant la variable d'environnement `PGDATA`. Cela permet, de plus, au serveur de bases de données (**postgres**) de trouver le répertoire par cette même variable.

`-E` *codage*, `--encoding=`*codage*
: Définit l'encodage de la base de données modèle (*template*). C'est également l'encodage par défaut des bases de données créées ultérieurement. Cette valeur peut toutefois être surchargée. La valeur par défaut est déduite de la locale. Dans le cas

où cela n'est pas possible, `SQL_ASCII` est utilisé. Les jeux de caractères supportés par le serveur PostgreSQL™ sont décrits dans Section 22.2.1, « Jeux de caractères supportés ».

`--locale=locale`
Configure la locale par défaut pour le cluster. Si cette option n'est pas précisée, la locale est héritée de l'environnement d'exécution d'**initdb**. Le support des locales est décrit dans Section 22.1, « Support des locales ».

`--lc-collate=locale`, `--lc-ctype=locale`, `--lc-messages=locale`, `--lc-monetary=locale`, `--lc-numeric=locale`, `--lc-time=locale`
Même principe que `--locale`, mais seule la locale de la catégorie considérée est configurée.

`--no-locale`
Équivalent à `--locale=C`.

`-X directory`, `--xlogdir=directory`
Définit le répertoire de stockage des journaux de transaction.

`-U nomutilisateur`, `--username=nomutilisateur`
Précise le nom de l'utilisateur défini comme superutilisateur de la base de données. Par défaut, c'est le nom de l'utilisateur qui lance **initdb**. Le nom du superutilisateur importe peu, mais `postgres` peut être conservé, même si le nom de l'utilisateur système diffère.

`-W`, `--pwprompt`
Force **initdb** à demander un mot de passe pour le superutilisateur de la base de données. Cela n'a pas d'importance lorsqu'aucune authentification par mot de passe n'est envisagée. Dans le cas contraire, l'authentification par mot de passe n'est pas utilisable tant qu'un mot de passe pour le superutilisateur n'est pas défini.

`--pwfile=nomfichier`
Incite **initdb** à lire le mot de passe du superutilisateur à partir d'un fichier. La première ligne du fichier est utilisée comme mot de passe.

`--text-search-config=CFG`
Sélectionne la configuration par défaut de la recherche plein texte. Voir default_text_search_config pour plus d'informations.

D'autres paramètres, moins utilisés, sont disponibles :

`-d`, `--debug`
Affiche les informations de débogage du processus amorce et quelques autres messages de moindre intérêt pour le grand public. Le processus amorce est le programme qu'**initdb** lance pour créer les tables catalogues. Cette option engendre une quantité considérable de messages ennuyeux.

`-L répertoire`
Indique à **initdb** où trouver les fichiers d'entrée nécessaires à l'initialisation du cluster. En temps normal, cela n'est pas nécessaire. Un message est affiché lorsque leur emplacement doit être indiqué de manière explicite.

`-n`, `--noclean`
Par défaut, lorsqu'**initdb** rencontre une erreur qui l'empêche de finaliser la création du cluster, le programme supprime tous les fichiers créés avant l'erreur. Cette option désactive le nettoyage. Elle est utile pour le débogage.

`-V`, `--version`
Affiche la version de initdb puis quitte.

`-?`, `--help`
Affiche l'aide sur les arguments en ligne de commande de initdb, puis quitte

Environnement

`PGDATA`
Indique le répertoire de stockage de la grappe de bases de données ; peut être surchargé avec l'option `-D`.

Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Notes

initdb peut aussi être appelé avec **pg_ctl initdb**.

Voir aussi

postgres(1)

Nom

pg_controldata — afficher les informations de contrôle d'un groupe de bases de données PostgreSQL™

Synopsis

`pg_controldata` [*répertoire_données*]

Description

pg_controldata affiche les informations initialisées lors d'**initdb**, telles que la version du catalogue. Il affiche aussi des informations sur le traitement des WAL et des points de vérification. Cette information, qui porte sur le groupe complet, n'est pas spécifique à une base de données.

Cet outil ne peut être lancé que par l'utilisateur qui a initialisé le groupe. Il est, en effet, nécessaire d'avoir le droit de lire le répertoire des données. Le répertoire des données peut être spécicifé sur la ligne de commande ou à l'aide de la variable d'environnement `PGDATA`. Cet outil accepte les options `-V` et `--version`, qui affiche la version de pg_controldata puis arrête l'application. Il accepte aussi les options `-?` et `--help`, qui affichent les arguments acceptés.

Environnement

`PGDATA`
: Emplacement du répertoire de données par défaut

Nom

pg_ctl — initialiser, démarrer, arrêter ou redémarrer le serveur PostgreSQL™

Synopsis

`pg_ctl` init[db] [-s] [-D *répertoire_données*] [-o *options*]

`pg_ctl` start [-w] [-t *secondes*] [-s] [-D *répertoire_données*] [-l *nomfichier*] [-o *options*] [-p *chemin*] [-c]

`pg_ctl` stop [-W] [-t *secondes*] [-s] [-D *répertoire_données*] [-m [s[mart]] | [f[ast]] | [i[mmediate]]]

`pg_ctl` restart [-w] [-t *secondes*] [-s] [-D *répertoire_données*] [-c] [-m [s[mart]] | [f[ast]] | [i[mmediate]]] [-o *options*]

`pg_ctl` reload [-s] [-D *répertoire_données*]

`pg_ctl` status [-D *répertoire_données*]

`pg_ctl` kill *nom_signal id_processus*

`pg_ctl` register [-N *nom_service*] [-U *nom_utilisateur*] [-P *mot_de_passe*] [-D *répertoire_données*] [-w] [-t *secondes*] [-s] [-o *options*]

`pg_ctl` unregister [-N *nom_service*]

Description

pg_ctl est un outil qui permet d'initialiser une instance, de démarrer, d'arrêter, ou de redémarrer un serveur PostgreSQL™ (postgres(1)). Il permet également d'afficher le statut d'un serveur en cours d'exécution.

Bien que le serveur puisse être démarré manuellement, pg_ctl encapsule les tâches comme la redirection des traces ou le détachement du terminal et du groupe de processus. Il fournit également des options intéressantes pour contrôler l'arrêt.

Le mode `init` ou `initdb` crée une nouvelle grappe PostgreSQL™. Une grappe est un ensemble de bases contrôlées par une même instance du serveur. Ce mode invoque la commande **initdb**. Voir initdb(1) pour les détails.

En mode `start`, un nouveau serveur est démarré. Le serveur est démarré en tâche de fond et l'entrée standard est attachée à `/dev/null` (or `nul` on Windows). On Unix-like systems, by default, the server's standard output and standard error are send to pg_ctl's standard output (not standard error). The standard output of pg_ctl should then be redirected to a file or piped to another process such as a log rotating program like rotatelogs; otherwise **postgres** will write its output to the controlling terminal (from the background) and will not leave the shell's process group. On Windows, by default the server's standard output and standard error are sent to the terminal. These default behaviors can be changed by using `-l` to append server output to a log file.

En mode `stop`, le serveur en cours d'exécution dans le répertoire indiqué est arrêté. Trois méthodes différentes d'arrêt peuvent être choisies avec l'option `-m` : le mode « smart » attend la fin de la sauvegarde en ligne (PITR) et la déconnexion de tous les clients. C'est la valeur par défaut. Si le serveur est en mode récupération (*recovery*), la récupération et la réplication en continu sont arrêtées dès que tous les clients se sont déconnectés. Le mode « fast » n'attend pas la déconnexion des clients et stoppe la sauvegarde en ligne (PITR). Toutes les transactions actives sont annulées et les clients sont déconnectés. Le serveur est ensuite arrêté. Le mode « immediate » tue tous les processus serveur sans leur laisser la possibilité de s'arrêter proprement. Cela conduit à une récupération au redémarrage.

Le mode `restart` exécute un arrêt suivi d'un démarrage. Ceci permet de modifier les options en ligne de commande de **postgres**.

Le mode `reload` envoie simplement au processus **postgres** un signal `SIGHUP`. Le processus relit alors ses fichiers de configuration (`postgresql.conf`, `pg_hba.conf`, etc.). Cela permet de modifier les options des fichiers de configuration qui ne requièrent pas un redémarrage complet pour être prises en compte.

Le mode `status` vérifie si un serveur est toujours en cours d'exécution sur le répertoire de données indiqué. Si c'est le cas, le PID et les options en ligne de commande utilisées lors de son démarrage sont affichés.

Le mode `kill` permet d'envoyer un signal à un processus spécifique. Ceci est particulièrement utile pour Microsoft Windows™, qui ne possède pas de commande kill. `--help` permet d'afficher la liste des noms de signaux supportés.

Le mode `register` permet d'enregistrer un service système sur Microsoft Windows™.

Le mode `unregister` permet, sur Microsoft Windows™, d'annuler un service système précédemment enregistré avec la commande `register`.

Options

`-c`
Tente d'autoriser la création de fichiers core suite à un arrêt brutal du serveur, sur les plateformes où cette fonctionnalité est disponible, en augmentant la limite logicielle qui en dépend. C'est utile pour le déboguage et pour diagnostiquer des problèmes en permettant la récupération d'une trace de la pile d'un processus serveur en échec.

`-D` *`répertoire_données`*
Indique l'emplacement des fichiers de la base de données sur le système de fichiers. Si cette option est omise, la variable d'environnement `PGDATA` est utilisée.

`-l` *`nomfichier`*
Ajoute la sortie des traces du serveur dans *`nomfichier`*. Si le fichier n'existe pas, il est créé. L'`umask` est configuré à 077, donc l'accès au journal des traces est, par défaut, interdit aux autres utilisateurs.

`-m` *`mode`*
Précise le mode d'arrêt. *`mode`* peut être `smart`, `fast` ou `immediate`, ou la première lettre d'un de ces trois mots.

`-o` *`options`*
Indique les options à passer directement à la commande **postgres**.

Ces options sont habituellement entourées par des guillemets simples ou doubles pour s'assurer qu'elles soient passées groupées.

`-p` *`chemin`*
Indique l'emplacement de l'exécutable `postgres`. Par défaut, l'exécutable `postgres` est pris à partir du même répertoire que **pg_ctl** ou, si cela échoue, à partir du répertoire d'installation codé en dur. Il n'est pas nécessaire d'utiliser cette option sauf cas inhabituel, comme lorsque des erreurs concernant l'impossibilité de trouver l'exécutable `postgres` apparaissent.

Dans le mode `init`, cette option indique de manière analogue la localisation de l'exécutable `initdb`.

`-s`
Affichage des seules erreurs, pas de messages d'information.

`-t`
Le nombre de secondes à attendre pour la fin du lancement ou de l'arrêt.

`-w`
Attendre la fin du démarrage ou de l'arrêt. Le temps d'attente par défaut est de 60 secondes, option par défaut pour les arrêts. Un arrêt réussi est indiqué par la suppression du fichier PID. Pour le démarrage, un **psql -l** se terminant avec succès indique que tout va bien. **pg_ctl** tente d'utiliser le bon port pour psql. Si la variable d'environnement `PGPORT` existe, elle est utilisée. Sinon, un port configuré dans le fichier `postgresql.conf` est cherché. Si aucun des deux n'est utilisé, le port par défaut avec lequel PostgreSQL™ a été compilé (5432 par défaut) est utilisé. Lorsqu'il attend, **pg_ctl** renvoie un code de sortie précis fondé sur le succès du démarrage ou de l'arrêt.

`-W`
Ne pas attendre la fin du démarrage ou de l'arrêt. C'est la valeur par défaut pour les démarrages et redémarrages.

Options Windows

`-N` *`nom_service`*
Nom du service système à enregistrer. Le nom est utilisé à la fois comme nom de service et comme nom affiché.

`-P` *`mot_de_passe`*
Mot de passe de l'utilisateur qui démarre le service.

`-U` *`nom_utilisateur`*
Nom de l'utilisateur qui démarre le service. Pour les utilisateurs identifiés sur un domaine, on utilise le format `DOMAIN\nom_utilisateur`.

Environnement

`PGDATA`
Emplacement par défaut du répertoire des données.

`PGHOST`
Nom d'hôte par défaut ou localisation de la socket de domaine Unix pour psql(1) (utilisée par l'option `-w`).

`PGPORT`
Port par défaut pour psql(1) (utilisé par l'option `-w`).

Pour les autres paramètres serveur, voir postgres(1). Cet outil, comme la plupart des autres outils PostgreSQL™, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 31.13, « Variables d'environnement »).

Fichiers

`postmaster.pid`
Ce fichier, situé dans le répertoire des données, est utilisé pour aider pg_ctl à déterminer si le serveur est actuellement en cours d'exécution.

`postmaster.opts`
Si ce fichier existe dans le répertoire des données, pg_ctl (en mode `restart`) passe le contenu du fichier comme options de postgres, sauf en cas de surcharge par l'option `-o`. Le contenu de ce fichier est aussi affiché en mode `status`.

`postgresql.conf`
Ce fichier, situé dans le répertoire des données, est analysé pour trouver le port à utiliser avec psql lorsque `-w` est donné en mode `start`.

Notes

Attendre le démarrage complet n'est pas une opération bien définie et peut échouer si le contrôle d'accès est configuré pour qu'un client local ne puisse pas se connecter sans interaction manuelle (par exemple, une authentification par mot de passe). Pour des variables de connexion supplémentaires, voir Section 31.13, « Variables d'environnement ». Pour les mots de passe, voir aussi Section 31.14, « Fichier de mots de passe ».

Exemples

Lancer le serveur

Démarrer un serveur :

```
$ pg_ctl start
```

Démarrer un serveur, avec blocage tant que le serveur n'est pas complètement démarré :

```
$ pg_ctl -w start
```

Pour un serveur utilisant le port 5433 et s'exécutant sans `fsync`, on utilise :

```
$ pg_ctl -o "-F -p 5433" start
```

Arrêt du serveur

```
$ pg_ctl stop
```

arrête le serveur. Utiliser l'option `-m` permet de contrôler *comment* le serveur s'arrête.

Redémarrage du serveur

Redémarrer le serveur est pratiquement équivalent à l'arrêter puis à le démarrer à nouveau si ce n'est que **pg_ctl** sauvegarde et réutilise les options en ligne de commande qui étaient passées à l'instance précédente. Pour redémarrer le serveur de la façon la plus simple, on utilise :

```
$ pg_ctl restart
```

Redémarrer le serveur, en attendant l'arrêt et le redémarrage :

```
$ pg_ctl -w restart
```

Redémarrer en utilisant le port 5433 et en désactivant `fsync` après redémarrage :

```
$ pg_ctl -o "-F -p 5433" restart
```

Affichage de l'état du serveur

Exemple de statut affiché à partir de pg_ctl :

```
$ pg_ctl status
pg_ctl : le serveur est en cours d'exécution (pid : 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

C'est la ligne de commande qui sera appelée en mode redémarrage.

Voir aussi

initdb(1), postgres(1)

Nom

pg_resetxlog — réinitialiser les WAL et les autres informations de contrôle d'une grappe de bases de données PostgreSQL™

Synopsis

`pg_resetxlog` [-f] [-n] [-o*oid*] [-x *xid*] [-e *xid_epoch*] [-m *mxid*] [-O *mxoff*] [-l *timelineid*,*fileid*,*seg*] *rép_données*

Description

pg_resetxlog efface les jouranux d'écritures anticipées (*Write-Ahead Log* ou WAL) et réinitialise optionnellement quelques autres informations de contrôle stockées dans le fichier `pg_control`. Cette fonction est parfois nécessaire si ces fichiers ont été corrompus. Elle ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption.

À la suite de cette commande, le serveur doit pouvoir redémarrer. Toutefois, il ne faut pas perdre de vue que la base de données peut contenir des données inconsistantes du fait de transactions partiellement validées. Il est alors opportun de sauvegarder les données, lancer **initdb** et de les recharger. Après cela, les inconsistances doivent être recharchées et le cas échéant corrigées.

Seul l'utilisateur qui a installé le serveur peut utiliser cet outil. Il requiert, en effet, un accès en lecture/écriture au répertoire des données. Pour des raisons de sécurité, **pg_resetxlog** n'utilise pas la variable d'environnement `PGDATA`. Le répertoire des données doit donc être précisé sur la ligne de commande.

Si **pg_resetxlog** se plaint de ne pas pouvoir déterminer de données valides pour `pg_control`, vous pouvez malgré tout le forcer à continuer en spécifiant l'option `-f` (force). Dans ce cas, des valeurs probables sont substituées aux données manquantes. La plupart des champs correspondent mais une aide manuelle pourrait être nécessaire pour le prochain OID, le prochain TID et sa date, le prochain identifiant multi-transaction et son décalage, l'adresse de début des journaux de transactions. Ces champs peuvent être configurés en utilisant les options indiquées ci-dessus. Si vous n'êtes pas capable de déterminer les bonnes valeurs pour tous ces champs, `-f` peut toujours être utilisé mais la base de données récupérée doit être traitée avec encore plus de suspicion que d'habitude : une sauvegarde immédiate et un rechargement sont impératifs. *Ne pas* exécuter d'opérations de modifications de données dans la base avant de sauvegarder ; ce type d'action risque de faire empirer la corruption.

Les options `-o`, `-x`, `-e`, `-m`, `-O` et `-l` permettent d'initialiser manuellement le prochain OID, le prochain TID et sa date, le prochain ID multitransaction, son décalage et l'adresse de début des WAL. Elles sont seulement nécessaires si **pg_resetxlog** est incapable de déterminer les valeurs appropriées en lisant `pg_control`. Les valeurs saines pour le prochain ID en transaction peuvent se déterminer ainsi :

- Une bonne valeur du prochain ID de transaction (`-x`) peut être déterminée en recherchant le fichier le plus large numériquement dans le répertoire `pg_clog` sous le répertoire des données, en ajoutant un et en multipliant par 1048576. Notez que les noms de fichiers sont en hexadécimal. Il est habituellement plus facile de spécifier la valeur de l'option en hexadécimal aussi. Par exemple, si `0011` est l'entrée la plus large dans `pg_clog`, `-x 0x1200000` fonctionnera (cinq zéros à la fin fournissent le bon multiplieur).
- Une valeur saine pour le prochain ID multitransaction (`-m`) peut être déterminée en recherchant le nom de fichier le plus important numériquement dans le sous-répertoire `pg_multixact/offsets` du répertoire data, en lui ajoutant un, puis en le multipliant par 65536. Comme ci-dessus, les noms de fichiers sont en hexadécimal, donc la façon la plus simple de le faire est de spécifier la valeur de l'option en hexadécimal et de lui ajouter quatre zéros.
- Une valeur saine pour le prochain décalage multitransaction (`-O`) peut être déterminée en recherchant le nom de fichier le plus important numériquement dans le sous-répertoire `pg_multixact/members` du répertoire data, en lui ajoutant un, puis en le multipliant par 65536. Comme ci-dessus, les noms de fichiers sont en hexadécimal, donc la façon la plus simple de le faire est de spécifier la valeur de l'option en hexadécimal et de lui ajouter quatre zéros.
- L'adresse de début des WAL (`-l`) doit être plus large que tout nom de fichier de segment WAL déjà existant dans le répertoire `pg_xlog` sous le répertoire des données. Ces noms sont aussi en hexadécimal et ont trois parties. La première est le « timeline ID » et doit habituellement être conservée identique. Ne choisissez pas de valeur plus importante que 255 (`0xFF`) pour la troisième partie ; à la place, incrémentez la deuxième partie et réinitialisez la troisième partie à 0. Par exemple, si `000000010000000320000004A` est l'entrée la plus importante de `pg_xlog`, `-l 0x1,0x32,0x4B` fonctionnera ; mais si l'entrée la plus importante est `000000010000003A000000FF`, choisissez `-l 0x1,0x3B,0x0` ou plus.

Note

pg_resetxlog lui-même recherche les fichiers dans `pg_xlog` et choisit un paramètre `-l` par défaut au-delà du dernier fichier existant. Du coup, un ajustement manuel de `-l` sera seulement nécessaire si vous avez connaissance de fichiers WAL qui ne sont actuellement pas présents dans le répertoire `pg_xlog`, comme des

entrées dans une archive hors ligne ou si le contenu de `pg_xlog` avait complètement disparu.

- Il n'y a pas de façon plus simple pour déterminer un OID suivant qui se trouve après celui de la base de données mais, heureusement, il n'est pas critique d'obtenir le bon OID suivant.
- La valeur epoch de l'identifiant de transaction n'est pas réellement stockée dans la base, sauf dans le champ qui est initialisé par **pg_resetxlog**, donc toute valeur sera correcte en ce qui concerne la base de données elle-même. Vous pourrez avoir besoin d'ajuster cette valeur pour vous assurer que les systèmes de réplication comme Slony-I fonctionnent correctement -- dans ce cas, une valeur appropriée doit être obtenue à partir de l'état de la base répliquée.

L'option `-n` (sans opération) demande à **pg_resetxlog** d'afficher les valeurs reconstruites à partir de `pg_control` puis quitte sans rien modifier. C'est principalement un outil de débogage mais qui peut être utile pour une vérification avant de permettre à **pg_resetxlog** de traiter réellement la base.

Les options `-V` et `--version` affichent la version de pg_resetxlog puis arrêtent l'application. Les options `-?` et `--help` affichent les arguments acceptés.

Notes

Cette commande ne doit pas être utilisée si le serveur est en cours d'exécution. **pg_resetxlog** refuse de se lancer s'il trouve un fichier de verrouillage du serveur dans le répertoire des données ; Si le serveur s'est arrêté brutalement, il peut subsister un tel fichier. Dans ce cas, vous pouvez supprimer le fichier de verrouillage pour permettre à **pg_resetxlog** de se lancer. Mais avant de le faire, soyez doublement certain qu'il n'y a pas de processus serveur en cours.

Nom

postgres — Serveur de bases de données PostgreSQL™

Synopsis

`postgres` [*option*...]

Description

postgres est le serveur de bases de données PostgreSQL™. Pour qu'une application cliente puisse accéder à une base de données, elle se connecte (soit via le réseau soit localement) à un processus **postgres** en cours d'exécution. L'instance **postgres** démarre ensuite un processus serveur séparé pour gérer la connexion.

Une instance **postgres** gère toujours les données d'un seul cluster. Un cluster est un ensemble de bases de données stocké à un même emplacement dans le système de fichiers (le « répertoire des données »). Plus d'un processus **postgres** peut être en cours d'exécution sur un système à un moment donné, s'ils utilisent des répertoires différents et des ports de communication différents (voir ci-dessous). Quand **postgres** se lance, il a besoin de connaître l'emplacement du répertoire des données. Cet emplacement doit être indiquée par l'option `-D` ou par la variable d'environnement `PGDATA` ; il n'y a pas de valeur par défaut. Typiquement, `-D` ou `PGDATA` pointe directement vers le répertoire des données créé par initdb(1). D'autres dispositions de fichiers possibles sont discutés dans Section 18.2, « Emplacement des fichiers ». Un répertoire de données est créé avec initdb(1). Un répertoire de données est créé avec initdb(1).

Par défaut, **postgres** s'exécute en avant-plan et affiche ses messages dans le flux standard des erreurs. En pratique, **postgres** devrait être exécuté en tant que processus en arrière-plan, par exemple au lancement.

La commande **postgres** peut aussi être appelé en mode mono-utilisateur. L'utilisation principal de ce mode est lors du « bootstrap » utilisé par initdb(1). Quelque fois, il est utilisé pour du débogage et de la récupération suite à un problème (mais noter qu'exécuter un serveur en mode mono-utilisateur n'est pas vraiment convenable pour déboguer le serveur car aucune communication inter-processus réaliste et aucun verrouillage n'interviennent.) Quand il est appelé en mode interactif à partir du shell, l'utilisateur peut saisir des requêtes et le résultat sera affiché à l'écran mais dans une forme qui est plus utile aux développeurs qu'aux utilisateurs. Dans le mode mono-utilisateur, la session ouverte par l'utilisateur sera configurée avec l'utilisateur d'identifiant 1 et les droits implicites du superutilisateur lui sont donnés. Cet utilisateur n'a pas besoin d'exister, donc le mode mono-utilisateur peut être utilisé pour récupérer manuellement après certains types de dommages accidentels dans les catalogues systèmes.

Options

postgres accepte les arguments suivants en ligne de commande. Pour une discussion détaillée des options, consultez Chapitre 18, Configuration du serveur. Vous pouvez éviter de saisir la plupart de ces options en les initialisant dans le fichier de configuration. Certaines options (sûres) peuvent aussi être configurées à partir du client en cours de connexion d'une façon dépendante de l'application, configuration qui ne sera appliquée qu'à cette session. Par exemple si la variable d'environnement `PGOPTIONS` est configurée, alors les clients basés sur libpq passeront cette chaîne au serveur qui les interprétera comme les options en ligne de commande de **postgres**.

Général

`-A 0|1`
Active les vérifications d'assertion à l'exécution, donc une aide au débogage pour détecter les erreurs de programmation. Cette option est seulement disponible si les assertions on été activées lors de la compilation de PostgreSQL™. Dans ce cas, la valeur par défaut est « on ».

`-B` *`ntampons`*
Configure le nombre de tampons partagés utilisés par les processus serveur. La valeur par défaut de ce paramètre est choisi automatiquement par initdb. Indiquer cette option est équivalent à configurer le paramètre shared_buffers.

`-c` *`nom=valeur`*
Configure un parammètre d'exécution nommé. Les paramètres de configuration supportés par PostgreSQL™ sont décrits dans Chapitre 18, Configuration du serveur. La plupart des autres options en ligne de commande sont en fait des formes courtes d'une affectation de paramères. `-c` peut apparaître plusieurs fois pour configurer différents paramètres.

`-d` *`niveau-débogage`*
Configure le niveau de débogage. Plus haute est sa valeur, plus importante seront les traces écrites dans les journaux. Les valeurs vont de 1 à 5. Il est aussi possible de passer `-d 0` pour une session spécifique qui empêchera le niveau des traces serveur du processus **postgres** parent d'être propagé jusqu'à cette session.

`-D` *`repdonnées`*
Indique le répertoire des données ou des fichier(s) de configuration. Voir Section 18.2, « Emplacement des fichiers » pour les détails.

`-e`
Configure le style de date par défaut à « European », c'est-à-dire l'ordre `DMY` pour les champs en entrée. Ceci cause aussi l'affichage de la date avant le mois dans certains formats de sortie de date. Voir Section 8.5, « Types date/heure » pour plus d'informations.

`-F`
Désactive les appels `fsync` pour améliorer les performances au risque de corrompre des données dans l'idée d'un arrêt brutal du système. Spécifier cette option est équivalent à désactiver le paramètre de configuration fsync. Lisez la documentation détaillée avant d'utiliser ceci !

`-h` *`hôte`*
Indique le nom d'hôte ou l'adresse IP sur lequel **postgres** attend les connexions TCP/IP d'applications clientes. La valeur peut aussi être une liste d'adresses séparées par des virgules ou `*` pour indiquer l'attente sur toutes les interfaces disponibles. Une valeur vide indique qu'il n'attend sur aucune adresse IP, auquel cas seuls les sockets de domaine Unix peuvent être utilisés pour se connecter au serveur. Par défaut, attend les connexions seulement sur `localhost`. Spécifier cette option est équivalent à la configurer dans le paramètre listen_addresses.

`-i`
Autorise les clients distants à se connecter via TCP/IP (domaine Internet). Sans cette option, seules les connexions locales sont autorisées. Cette option est équivalent à la configuration du paramètre `listen_addresses` à `*` dans `postgresql.conf` ou via `-h`.

Cette option est obsolète car il ne permet plus l'accès à toutes les fonctionnalités de listen_addresses. Il est généralement mieux de configurer directement `listen_addresses`.

`-k` *`directory`*
Indique le répertoire de la socket de domaine Unix sur laquelle **postgres** est en attente des connexions des applications clients. La valeur par défaut est habituellement `/tmp` mais cela peut se changer au moment de la construction.

`-l`
Active les connexions sécurisées utilisant SSL. PostgreSQL™ doit avoir été compilé avec SSL pour que cette option soit disponible. Pour plus d'informations sur SSL, référez-vous à Section 17.8, « Connexions tcp/ip sécurisées avec ssl ».

`-N` *`max-connections`*
Initialise le nombre maximum de connexions clientes que le serveur acceptera. La valeur par défaut de ce paramètre est choisi automatiquement par initdb. Indiquer cette option est équivalent à configurer le paramètre max_connections.

`-o` *`extra-options`*
Les options en ligne de commande indiquées dans *`extra-options`* sont passées à tous les processus serveur exécutés par ce processus **postgres**. Si la chaîne d'option contient des espaces, la chaîne entière doit être entre guillemets.

Cette option est obsolète ; toutes les options en ligne de commande des processus serveur peuvent être spécifiées directement sur la ligne de commande de **postgres**.

`-p` *`port`*
Indique le port TCP/IP ou l'extension du fichier socket de domaine Unix sur lequel **postgres** attend les connexions des applications clientes. Par défaut, la valeur de la variable d'environnement `PGPORT` environment ou, si cette variable n'est pas configurer, la valeur connue à la compilation (habituellement 5432). Si vous indiquez un port autre que celui par défaut, alors toutes les applications clientes doivent indiquer le même numéro de port soit dans les options en ligne de commande soit avec `PGPORT`.

`-s`
Affiche une information de temps et d'autres statistiques à la fin de chaque commande. Ceci est utile pour créer des rapports de performance ou pour configurer finement le nombre de tampons.

`-S` *`work-mem`*
Indique la quantité de mémoire à utiliser par les tris internes et par les hachages avant d'utiliser des fichiers disque temporaires. Voir la description du paramètre `work_mem` dans Section 18.4.1, « Mémoire ».

`--nom=valeur`
Configure un paramètre à l'exécution ; c'est une version courte de `-c`.

`--describe-config`
Cette option affiche les variables de configuration internes du serveur, leurs descriptions et leurs valeurs par défaut dans un format **COPY** délimité par des tabulations. Elle est conçue principalement pour les outils d'administration.

Options semi-internes

Les options décrites ici sont utilisées principalement dans un but de débogage et pouvant quelque fois aider à la récupération de bases de données très endommagées/ Il n'y a aucune raison pour les utiliser dans la configuration d'un système en production. Elles sont listées ici à l'intention des développeurs PostgreSQL™. De plus, une de ces options pourrait disparaître ou changer dans le futur sans avertissement.

`-f { s | i | m | n | h }`
: Interdit l'utilisation de parcours et de méthode de jointure particulières. `s` and `i` désactivent respectivement les parcours séquentiels et d'index alors que `n`, `m` et `h` désactivent respectivement les jointures de boucles imbriquées, jointures de fusion et hash.

  Ni les parcours séquentiels ni les jointures de boucles imbriquées ne peuvent être désactivés complètement ; les options `-fs` et `-fn` ne font que décourager l'optimiseur d'utiliser ce type de plans.

`-n`
: Cette option est présente pour les problèmes de débogage du genre mort brutal d'un processus serveur. La stratégie habituelle dans cette situation est de notifier tous les autres processus serveur qu'ils doivent se terminer, puis réinitialiser la mémoire partagée et les sémaphores. Tout ceci parce qu'un processus serveur errant peut avoir corrompu certains états partagés avant de terminer. Cette option spécifie seulement que **postgres** ne réinitialisera pas les structures de données partagées. Un développeur système avec quelques connaissances peut utiliser un débogueur pour examiner l'état de la mémoire partagée et des sémaphores.

`-O`
: Autorise la modification de la structure des tables système. C'est utilisé par **initdb**.

`-P`
: Ignore les index système lors de la lecture des tables système (mais les met à jour lors de la modification des tables). Ceci est utile lors de la récupération d'index système endommagés.

`-t pa[rser] | pl[anner] | e[xecutor]`
: Affiche les statistiques en temps pour chaque requête en relation avec un des modules majeurs du système. Cette option ne peut pas être utilisée avec l'option `-s`.

`-T`
: Cette option est présente pour les problèmes de débogage du genre mort brutal d'un processus serveur. La stratégie habituelle dans cette situation est de notifier tous les autres processus serveur qu'ils doivent se terminer, puis réinitialiser la mémoire partagée et les sémaphores. Tout ceci parce qu'un processus serveur errant peut avoir corrompu certains états partagés avant de terminer. Cette option spécifie seulement que **postgres** arrêtera tous les autres processus serveur en leur envoyant le signal `SIGSTOP` mais ne les arrêtera pas. Ceci permet aux développeurs système de récupérer manuellement des « core dumps » de tous les processus serveur.

`-v` *`protocole`*
: Indique le numéro de version utilisé par le protocole interface/moteur pour une session particulière. Cette option est uniquement utilisée en interne.

`-W` *`secondes`*
: Un délai de ce nombre de secondes survient quand un nouveau processus serveur est lancé, une fois la procédure d'authentification terminée. Ceci a pour but de permettre au développeur d'attacher un débogueur au processus serveur.

Options en mode mono-utilisateur

Les options suivantes s'appliquent uniquement en mode mono-utilisateur.

`--single`
: Sélectionne le mode mono-utilisateur. Cette option doit être la première sur la ligne de commande.

`base`
: Indique le nom de la base à accèder. Il doit être le dernier argument. Si elle est omise, le nom de l'utilisateur est utilisé par défaut.

`-E`
: Affiche toutes les commandes.

`-j`
: Désactive l'utilisation du retour chariot comme délimiteur d'instruction.

`-r` *`fichier`*

Envoie toute la sortie des traces du serveur dans *fichier*. Dans le mode normal, cette option est ignorée et `stderr` est utilisé par tous les processus.

Environnement

`PGCLIENTENCODING`
Jeu de caractères utilisé par défaut par tous les clients. (Les clients peuvent surcharger ce paramètre individuellement.) Cette valeur est aussi configurable dans le fichier de configuration.

`PGDATA`
Emplacement du répertoire des données par défaut

`PGDATESTYLE`
Valeur par défaut du paramètre en exécution datestyle. (Cette variable d'environnement est obsolète.)

`PGPORT`
Numéro de port par défaut (à configurer de préférence dans le fichier de configuration)

`TZ`
Fuseau horaire du serveur

Diagnostiques

Un message d'erreur mentionnant `semget` ou `shmget` indique probablement que vous devez configurer votre noyau pour fournir la mémoire partagée et les sémaphores adéquates. Pour plus de discussion, voir Section 17.4, « Gérer les ressources du noyau ». Vous pouvez aussi repousser la configuration du noyau en diminuant shared_buffers pour réduire la consommation de la mémoire partagée utilisée par PostgreSQL™, et/ou en diminuant max_connections pour réduire la consommation de sémaphores.

Un message d'erreur suggérant qu'un autre serveur est déjà en cours d'exécution devra vous demander une vérification attentive, par exemple en utilisant la commande ps should be checked carefully, for example by using the command

```
$ ps ax | grep postgres
```

ou

```
$ ps -ef | grep postgres
```

suivant votre système. Si vous êtes certain qu'il n'y a aucun serveur en conflit, vous pouvez supprimer le fichier verrou mentionné dans le message et tenter de nouveau.

Un message d'erreur indiquant une incapacité à se lier à un port indique que ce port est déjà utilisé par des processus autres que PostgreSQL™. Vous pouvez aussi obtenir cette erreur si vous quittez **postgres** et le relancez immédiatement en utilisant le même port ; dans ce cas, vous devez tout simplement attendre quelques secondes pour que le système d'exploitation ferme bien le port avant de tenter de nouveau. Enfin, vous pouvez obtenir cette erreur si vous indiquez un numéro de port que le système considère comme réservé. Par exemple, beaucoup de versions d'Unix considèrent les numéros de port sous 1024 comme de « confiance » et permettent seulement leur accès par le superutilisateur Unix.

Notes

L'outil pg_ctl(1) est utilisable pour lancer et arrêter le serveur **postgres** de façon sûre et confortable.

Si possible, *ne pas* utiliser `SIGKILL` pour tuer le serveur **postgres** principal. Le fait empêchera **postgres** de libérer les ressources système (c'est-à-dire mémoire partagée et sémaphores) qu'il détient avant de s'arrêter. Ceci peut poser problèmes lors du lancement d'un **postgres** frais.

Pour terminer le serveur **postgres** normalement, les signaux `SIGTERM`, `SIGINT` ou `SIGQUIT` peuvent être utilisés. Le premier attendra que tous les clients terminent avant de quitter, le second forcera la déconnexion de tous les clients et le troisième quittera immédiatement sans arrêt propre. Ce dernier amènera une récupération lors du redémarrage.

Le signal `SIGHUP` rechargera les fichiers de configuration du serveur. Il est aussi possible d'envoyer `SIGHUP` à un processus serveur individuel mais ce n'est pas perceptible.

Pour annuler une requête en cours d'exécution, envoyez le signal `SIGINT` au processus exécutant cette commande.

Le serveur **postgres** utilise `SIGTERM` pour indiquer aux processus serveur de quitter normalement et `SIGQUIT` pour terminer sans le nettoyage habituel. Ces signaux *ne devraient pas* être utilisés par les utilisateurs. Il est aussi déconseillé d'envoyer `SIGKILL` à un processus serveur -- le serveur **postgres** principal interprétera ceci comme un arrêt brutal et forcera tous les autres processus serveur à quitter dans le cas d'une procédure standard de récupération après arrêt brutal.

Bogues

Les options `--` ne fonctionneront pas sous `FreeBSD` et `OpenBSD`. Utilisez `-c` à la place. C'est un bogue dans les systèmes d'exploitation affectés ; une prochaine version de PostgreSQL™ fournira un contournement si ce n'est pas corrigé.

Utilisation

Pour démarrer un serveur en mode mono-utilisateur, utilisez une commande comme

```
postgres --single -D /usr/local/pgsql/data autres-options ma_base
```

Fournissez le bon chemin vers le répertoire des bases avec l'option `-D` ou assurez-vous que la variable d'environnement `PGDATA` est configurée. De plus, spécifiez le nom de la base particulière avec laquelle vous souhaitez travailler.

Habituellement, le serveur en mode mono-utilisateur traite le retour chariot comme le terminateur d'une commande ; il n'y a pas le concept du point-virgule contraitement à psql. Pour saisir une commande sur plusieurs lignes, vous devez saisir un antislash juste avant un retour chariot, sauf pour le dernier.

Mais si vous utilisez l'option `-j`, alors le retour chariot ne termine pas une commande. Dans ce cas, le serveur lira l'entrée standard jusqu'à une marque de fin de fichier (EOF), puis il traitera la saisie comme une seule chaîne de commande. Les retours chariot avec antislash ne sont pas traités spécialement dans ce cas.

Pour quitter la session, saisissez EOF (habituellement, **Control**+**D**). Si vous avez utilisé l'option `-j`, deux EOF consécutifs sont nécessaires pour quitter.

Notez que le serveur en mode mono-utilisateur ne fournit pas de fonctionnalités avancées sur l'édition de lignes (par exemple, pas d'historique des commandes).

Exemples

Pour lancer **postgres** en tâche de fond avec les valeurs par défaut, saisissez :

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

Pour lancer **postgres** avec un port spécifique :

```
$ postgres -p 1234
```

Cette commande exécutera un **postgres** qui communiquera sur le port 1234. Pour se connecter à ce serveur en utilisant psql, vous aurez besoin de saisir :

```
$ psql -p 1234
```

ou de configurer la variable d'environnement `PGPORT` :

```
$ export PGPORT=1234
$ psql
```

Les paramètres nommés peuvent être configurés suivant deux façons :

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Ces deux formes surchargent le paramétrage qui pourrait exister pour `work_mem` dans `postgresql.conf`. Notez que les tirets bas dans les noms de paramètres sont écrits avec soir des tirets bas soit des tirets sur la ligne de commande. Sauf pour les expériences à court terme, il est probablement mieux de modifier le paramétrage dans `postgresql.conf` que de se baser sur une option en ligne de commande.

Voir aussi

initdb(1), pg_ctl(1)

Nom

postmaster — Serveur de bases de données PostgreSQL™

Synopsis

`postmaster` [*`option`*...]

Description

postmaster est un alias obsolète de **postgres**.

Voir aussi

postgres(1)

Partie VII. Internes

Cette partie contient des informations diverses utiles aux développeurs.

Chapitre 44. Présentation des mécanismes internes de PostgreSQL

> **Auteur**
>
> Ce chapitre est extrait de sim98, mémoire de maîtrise (Master's Thesis) de Stefan Simkovics. Cette maîtrise a été préparée à l'université de technologie de Vienne sous la direction du professeur (O.Univ.Prof.Dr.) Georg Gottlob et de l'assistante d'université (Univ.Ass.) Mag. Katrin Seyr.

Ce chapitre présente la structure interne du serveur PostgreSQL™. La lecture des sections qui suivent permet de se faire une idée de la façon dont une requête est exécutée ; les opérations internes ne sont pas décrites dans le détail. Ce chapitre a, au contraire, pour but d'aider le lecteur à comprendre la suite des opérations effectuées sur le serveur depuis la réception d'une requête jusqu'au retour des résultats.

44.1. Chemin d'une requête

Ceci est un rapide aperçu des étapes franchies par une requête pour obtenir un résultat.

1. Une connexion au serveur est établie par une application. Elle transmet une requête et attend le retour des résultats.
2. L'*étape d'analyse* (*parser stage*) vérifie la syntaxe de la requête et crée un *arbre de requête* (*query tree*).
3. Le *système de réécriture* (*rewrite system*) recherche les *règles* (stockées dans les *catalogues système*) à appliquer à l'arbre de requête. Il exécute les transformations indiquées dans le *corps des règles* (*rule bodies*).

   La réalisation des *vues* est une application du système de réécriture. Toute requête utilisateur impliquant une vue (c'est-à-dire une *table virtuelle*), est réécrite en une requête qui accède aux *tables de base*, en fonction de la *définition de la vue*.
4. Le *planificateur/optimiseur* (*planner/optimizer*) transforme l'arbre de requête (réécrit) en un *plan de requête* (*query plan*) passé en entrée de l'*exécuteur*.

   Il crée tout d'abord tous les *chemins* possibles conduisant au résultat. Ainsi, s'il existe un index sur une relation à parcourir, il existe deux chemins pour le parcours. Le premier consiste en un simple parcours séquentiel, le second utilise l'index. Le coût d'exécution de chaque chemin est estimé ; le chemin le moins coûteux est alors choisi. Ce dernier est étendu en un plan complet que l'exécuteur peut utiliser.
5. L'exécuteur traverse récursivement les étapes de l'*arbre de planification* (*plan tree*) et retrouve les lignes en fonction de ce plan. L'exécuteur utilise le *système de stockage* lors du parcours des relations, exécute les *tris* et *jointures*, évalue les *qualifications* et retourne finalement les lignes concernées.

Les sections suivantes présentent en détail les éléments brièvement décrits ci-dessus.

44.2. Établissement des connexions

PostgreSQL™ est écrit suivant un simple modèle client/serveur « processus par utilisateur ». Dans ce modèle, il existe un *processus client* connecté à un seul *processus serveur*. Comme le nombre de connexions établies n'est pas connu à l'avance, il est nécessaire d'utiliser un *processus maître* qui lance un processus serveur à chaque fois qu'une connexion est demandée. Ce processus maître s'appelle `postgres` et écoute les connexions entrantes sur le port TCP/IP indiqué. À chaque fois qu'une demande de connexion est détectée, le processus `postgres` lance un nouveau processus serveur. Les tâches du serveur communiquent entre elles en utilisant des *sémaphores* et de la *mémoire partagée* pour s'assurer de l'intégrité des données lors d'accès simultanés aux données.

Le processus client est constitué de tout programme comprenant le protocole PostgreSQL™ décrit dans le Chapitre 46, Protocole client/serveur. De nombreux clients s'appuient sur la bibliothèque C libpq, mais il existe différentes implantations indépendantes du protocole, tel que le pilote Java JDBC.

Une fois la connexion établie, le processus client peut envoyer une requête au serveur (*backend*). La requête est transmise en texte simple, c'est-à-dire qu'aucune analyse n'a besoin d'être réalisée au niveau de l'*interface* (client). Le serveur analyse la requête, crée un *plan d'exécution*, exécute le plan et renvoie les lignes trouvées au client par la connexion établie.

44.3. Étape d'analyse

L'*étape d'analyse* est constituée de deux parties :

- l'*analyseur*, défini dans `gram.y` et `scan.l`, est construit en utilisant les outils Unix bison et flex ;
- le *processus de transformation* fait des modifications et des ajouts aux structures de données renvoyées par l'analyseur.

44.3.1. Analyseur

L'analyseur doit vérifier que la syntaxe de la chaîne de la requête (arrivant comme un texte ASCII) est valide. Si la syntaxe est correcte, un *arbre d'analyse* est construit et renvoyé, sinon une erreur est retournée. Les analyseur et vérificateur syntaxiques sont développés à l'aide des outils Unix bien connus bison et flex.

L'*analyseur lexical*, défini dans le fichier `scan.l`, est responsable de la reconnaissance des *identificateurs*, des *mots clés SQL*, etc. Pour chaque mot clé ou identificateur trouvé, un *jeton* est engendré et renvoyé à l'analyseur.

L'analyseur est défini dans le fichier `gram.y` et consiste en un ensemble de *règles de grammaire* et en des *actions* à exécuter lorsqu'une règle est découverte. Le code des actions (qui est en langage C) est utilisé pour construire l'arbre d'analyse.

Le fichier `scan.l` est transformé en fichier source C `scan.c` en utilisant le programme flex et `gram.y` est transformé en `gram.c` en utilisant bison. Après avoir réalisé ces transformations, un compilateur C normal peut être utilisé pour créer l'analyseur. Il est inutile de modifier les fichiers C engendrés car ils sont écrasés à l'appel suivant de flex ou bison.

> **Note**
>
> Les transformations et compilations mentionnées sont normalement réalisées automatiquement en utilisant les *makefile* distribués avec les sources de PostgreSQL™.

La description détaillée de bison ou des règles de grammaire données dans `gram.y` dépasse le cadre de ce document. Il existe de nombreux livres et documentations en relation avec flex et bison. Il est préférable d'être familier avec bison avant de commencer à étudier la grammaire donnée dans `gram.y`, au risque de ne rien y comprendre.

44.3.2. Processus de transformation

L'étape d'analyse crée un arbre d'analyse qui n'utilise que les règles fixes de la structure syntaxique de SQL. Il ne fait aucune recherche dans les catalogues système. Il n'y a donc aucune possibilité de comprendre la sémantique détaillée des opérations demandées. Lorsque l'analyseur a fini, le *processus de transformation* prend en entrée l'arbre résultant de l'analyseur et réalise l'interprétation sémantique nécessaire pour connaître les tables, fonctions et opérateurs référencés par la requête. La structure de données construite pour représenter cette information est appelée l'*arbre de requête*.

La séparation de l'analyse brute et de l'analyse sémantique résulte du fait que les recherches des catalogues système ne peuvent se dérouler qu'à l'intérieur d'une transaction. Or, il n'est pas nécessaire de commencer une transaction dès la réception d'une requête. L'analyse brute est suffisante pour identifier les commandes de contrôle des transactions (**BEGIN**, **ROLLBACK**, etc.). Elles peuvent de plus être correctement exécutées sans analyse complémentaire. Lorsqu'il est établi qu'une vraie requête doit être gérée (telle que **SELECT** ou **UPDATE**), une nouvelle transaction est démarrée si aucune n'est déjà en cours. Ce n'est qu'à ce moment-là que le processus de transformation peut être invoqué.

La plupart du temps, l'arbre d'une requête créé par le processus de transformation a une structure similaire à l'arbre d'analyse brute mais, dans le détail, de nombreuses différences existent. Par exemple, un nœud FuncCall dans l'arbre d'analyse représente quelque chose qui ressemble syntaxiquement à l'appel d'une fonction. Il peut être transformé soit en nœud FuncExpr soit en nœud Aggref selon que le nom référencé est une fonction ordinaire ou une fonction d'agrégat. De même, des informations sur les types de données réels des colonnes et des résultats sont ajoutées à l'arbre de la requête.

44.4. Système de règles de PostgreSQL™

PostgreSQL™ supporte un puissant *système de règles* pour la spécification des *vues* et des *mises à jour de vues* ambiguës. À l'origine, le système de règles de PostgreSQL™ était constitué de deux implantations :

- la première, qui fonctionnait au *niveau des lignes*, était implantée profondément dans l'*exécuteur*. Le système de règles était appelé à chaque fois qu'il fallait accéder une ligne individuelle. Cette implantation a été supprimée en 1995 quand la dernière version officielle du projet Berkeley Postgres™ a été transformée en Postgres95™ ;
- la deuxième implantation du système de règles est une technique appelée *réécriture de requêtes*. Le *système de réécriture* est un module qui existe entre l'*étape d'analyse* et le *planificateur/optimiseur*. Cette technique est toujours implantée.

Le système de réécriture de requêtes est vu plus en détails dans le Chapitre 37, Système de règles. Il n'est donc pas nécessaire d'en parler ici. Il convient simplement d'indiquer qu'à la fois l'entrée et la sortie du système sont des arbres de requêtes. C'est-à-dire qu'il n'y a pas de changement dans la représentation ou le niveau de détail sémantique des arbres. La réécriture peut être imaginée comme une forme d'expansion de macro.

44.5. Planificateur/Optimiseur

La tâche du *planificateur/optimiseur* est de créer un plan d'exécution optimal. En fait, une requête SQL donnée (et donc, l'arbre d'une requête) peut être exécutée de plusieurs façons, chacune arrivant au même résultat. Si ce calcul est possible, l'optimiseur de la requête examinera chacun des plans d'exécution possibles pour sélectionner le plan d'exécution estimé comme le plus rapide.

> **Note**
>
> Dans certaines situations, examiner toutes les façons d'exécuter une requête prend beaucoup de temps et de mémoire. En particulier, lors de l'exécution de requêtes impliquant un grand nombre de jointures. Pour déterminer un plan de requête raisonnable (mais pas forcément optimal) en un temps raisonnable, PostgreSQL™ utilise un *Genetic Query Optimizer* (voir Chapitre 50, Optimiseur génétique de requêtes (*Genetic Query Optimizer*)) dès lors que le nombre de jointures dépasse une certaine limite (voir geqo_threshold).

La procédure de recherche du planificateur fonctionne avec des structures de données appelés *chemins*, simples représentations minimales de plans ne contenant que l'information nécessaire au planificateur pour prendre ses décisions. Une fois le chemin le moins coûteux déterminé, un *arbre plan* est construit pour être passé à l'exécuteur. Celui-ci représente le plan d'exécution désiré avec suffisamment de détails pour que l'exécuteur puisse le lancer. Dans le reste de cette section, la distinction entre chemins et plans est ignorée.

44.5.1. Engendrer les plans possibles

Le planificateur/optimiseur commence par engendrer des plans de parcours de chaque relation (table) invididuelle utilisée dans la requête. Les plans possibles sont déterminés par les index disponibles pour chaque relation. Un parcours séquentiel de relation étant toujours possible, un plan de parcours séquentiel est systématiquement créé. Soit un index défini sur une relation (par exemple un index B-tree) et une requête qui contient le filtre `relation.attribut OPR constante`. Si `relation.attribut` correspond à la clé de l'index B-tree et `OPR` est un des opérateurs listés dans la *classe d'opérateurs* de l'index, un autre plan est créé en utilisant l'index B-tree pour parcourir la relation. S'il existe d'autres index et que les restrictions de la requête font correspondre une clé à un index, d'autres plans sont considérés. Des plans de parcours d'index sont également créés pour les index dont l'ordre de tri peut correspondre à la clause `ORDER BY` de la requête (s'il y en a une), ou dont l'ordre de tri peut être utilisé dans une jointure de fusion (cf. plus bas).

Si la requête nécessite de joindre deux, ou plus, relations, les plans de jointure des relations sont considérés après la découverte de tous les plans possibles de parcours des relations uniques. Les trois stratégies possibles de jointure sont :

- *jointure de boucle imbriquée* (*nested loop join*) : la relation de droite est parcourue une fois pour chaque ligne trouvée dans la relation de gauche. Cette stratégie est facile à implanter mais peut être très coûteuse en temps. (Toutefois, si la relation de droite peut être parcourue à l'aide d'un index, ceci peut être une bonne stratégie. Il est possible d'utiliser les valeurs issues de la ligne courante de la relation de gauche comme clés du parcours d'index à droite.)
- *jointure de fusion* (*merge join*) : chaque relation est triée selon les attributs de la jointure avant que la jointure ne commence. Puis, les deux relations sont parcourues en parallèle et les lignes correspondantes sont combinées pour former des lignes jointes. Ce type de jointure est plus intéressant parce que chaque relation n'est parcourue qu'une seule fois. Le tri requis peut être réalisé soit par une étape explicite de tri soit en parcourant la relation dans le bon ordre en utilisant un index sur la clé de la jointure ;
- *jointure de hachage* (*hash join*) : la relation de droite est tout d'abord parcourue et chargée dans une table de hachage en utilisant ses attributs de jointure comme clés de hachage. La relation de gauche est ensuite parcourue et les valeurs appropriées de chaque ligne trouvée sont utilisées comme clés de hachage pour localiser les lignes correspondantes dans la table.

Quand la requête implique plus de deux relations, le résultat final doit être construit à l'aide d'un arbre d'étapes de jointure, chacune à deux entrées. Le planificateur examine les séquences de jointure possibles pour trouver le moins cher.

Si la requête implique moins de geqo_threshold relations, une recherche quasi-exhaustive est effectuée pour trouver la meilleure séquence de jointure. Le planificateur considère préférentiellement les jointures entre paires de relations pour lesquelles existe une clause de jointure correspondante dans la qualification `WHERE` (i.e. pour lesquelles existe une restriction de la forme `where rel1.attr1=rel2.attr2`). Les paires jointes pour lesquelles il n'existe pas de clause de jointure ne sont considérées que lorsqu'il n'y a plus d'autre choix, c'est-à-dire qu'une relation particulière n'a pas de clause de jointure avec une autre relation. Tous les plans possibles sont créés pour chaque paire jointe considérée par le planificateur. C'est alors celle qui est (estimée) la moins

coûteuse qui est choisie.

Lorsque `geqo_threshold` est dépassé, les séquences de jointure sont déterminées par heuristique, comme cela est décrit dans Chapitre 50, Optimiseur génétique de requêtes (*Genetic Query Optimizer*). Pour le reste, le processus est le même.

L'arbre de plan terminé est composé de parcours séquentiels ou d'index des relations de base, auxquels s'ajoutent les nœuds des jointures en boucle, des jointures de tri fusionné et des jointures de hachage si nécessaire, ainsi que toutes les étapes auxiliaires nécessaires, telles que les nœuds de tri ou les nœuds de calcul des fonctions d'agrégat. La plupart des types de nœud de plan ont la capacité supplémentaire de faire une *sélection* (rejet des lignes qui ne correspondent pas à une condition booléenne indiquée) et une *projection* (calcul d'un ensemble dérivé de colonnes fondé sur des valeurs de colonnes données, par l'évaluation d'expressions scalaires si nécessaire). Une des responsabilités du planificateur est d'attacher les conditions de sélection issues de la clause `WHERE` et le calcul des expressions de sortie requises aux nœuds les plus appropriés de l'arbre de plan.

44.6. Exécuteur

L'*exécuteur* prend le plan créé par le planificateur/optimiseur et l'exécute récursivement pour extraire l'ensemble requis de lignes. Il s'agit principalement d'un mécanisme de pipeline en demande-envoi. Chaque fois qu'un nœud du plan est appelé, il doit apporter une ligne supplémentaire ou indiquer qu'il a fini d'envoyer des lignes.

Pour donner un exemple concret, on peut supposer que le nœud supérieur est un nœud `MergeJoin`. Avant de pouvoir faire une fusion, deux lignes doivent être récupérées (une pour chaque sous-plan). L'exécuteur s'appelle donc récursivement pour exécuter les sous-plans (en commençant par le sous-plan attaché à l'`arbre gauche`). Le nouveau nœud supérieur (le nœud supérieur du sous-plan gauche) est, par exemple, un nœud `Sort` (NdT : Tri) et un appel récursif est une nouvelle fois nécessaire pour obtenir une ligne en entrée. Le nœud fils de `Sort` pourrait être un nœud `SeqScan`, représentant la lecture réelle d'une table. L'exécution de ce nœud impose à l'exécuteur de récupérer une ligne à partir de la table et de la retourner au nœud appelant. Le nœud `Sort` appelle de façon répétée son fils pour obtenir toutes les lignes à trier. Quand l'entrée est épuisée (indiqué par le nœud fils renvoyant un NULL au lieu d'une ligne), le code de `Sort` est enfin capable de renvoyer sa première ligne en sortie, c'est-à-dire le premier suivant l'ordre de tri. Il conserve les lignes restantes en mémoire de façon à les renvoyer dans le bon ordre en réponse à des demandes ultérieures.

Le nœud `MergeJoin` demande de la même façon la première ligne à partir du sous-plan droit. Ensuite, il compare les deux lignes pour savoir si elles peuvent être jointes ; si c'est le cas, il renvoie la ligne de jointure à son appelant. Au prochain appel, ou immédiatement, s'il ne peut pas joindre la paire actuelle d'entrées, il avance sur la prochaine ligne d'une des deux tables (suivant le résultat de la comparaison), et vérifie à nouveau la correspondance. Éventuellement, un des sous-plans est épuisé et le nœud `MergeJoin` renvoie NULL pour indiquer qu'il n'y a plus de lignes jointes à former.

Les requêtes complexes peuvent nécessiter un grand nombre de niveaux de nœuds pour les plans, mais l'approche générale est la même : chaque nœud est exécuté et renvoie sa prochaine ligne en sortie à chaque fois qu'il est appelé. Chaque nœud est responsable aussi de l'application de toute expression de sélection ou de projection qui lui a été confiée par le planificateur.

Le mécanisme de l'exécuteur est utilisé pour évaluer les quatre types de requêtes de base en SQL : **SELECT**, **INSERT**, **UPDATE** et **DELETE**. Pour **SELECT**, le code de l'exécuteur de plus haut niveau a uniquement besoin d'envoyer chaque ligne retournée par l'arbre plan de la requête vers le client. Pour **INSERT**, chaque ligne renvoyée est insérée dans la table cible indiquée par **INSERT**. Cela se fait dans un nœud spécial haut niveau du plan appelé `ModifyTable`. (Une simple commande **INSERT ... VALUES** crée un arbre plan trivial constitué d'un seul nœud, `Result`, calculant une seule ligne de résultat, et `ModifyTable` au-dessus pour réaliser l'insertion. Mais **INSERT ... SELECT** peut demander la pleine puissance du mécanisme de l'exécuteur.) Pour **UPDATE**, le planificateur s'arrange pour que chaque ligne calculée inclue toutes les valeurs mises à jour des colonnes, plus le *TID* (tuple ID ou l'identifiant de la ligne) de la ligne de la cible originale ; cette donnée est envoyée dans un nœud `ModifyTable`, qui utilise l'information pour créer une nouvelle ligne mise à jour et marquer l'ancienne ligne comme supprimée. Pour **DELETE**, la seule colonne renvoyée par le plan est le TID, et `ModifyTable` node utilise simplement le TID pour visiter chaque ligne cible et la marquer comme supprimée.

Chapitre 45. Catalogues système

Les catalogues système représentent l'endroit où une base de données relationnelle stocke les métadonnées des schémas, telles que les informations sur les tables et les colonnes, et des données de suivi interne. Les catalogues système de PostgreSQL™ sont de simples tables. Elle peuvent être supprimées et recrées. Il est possible de leur ajouter des colonnes, d'y insérer et modifier des valeurs, et de mettre un joyeux bazar dans le système. En temps normal, l'utilisateur n'a aucune raison de modifier les catalogues système, il y a toujours des commandes SQL pour le faire. (Par exemple, **CREATE DATABASE** insère une ligne dans le catalogue pg_database -- et crée physiquement la base de données sur le disque.) Il y a des exceptions pour certaines opérations particulièrement ésotériques, comme l'ajout de méthodes d'accès aux index.

45.1. Aperçu

Tableau 45.1, « Catalogues système » liste les catalogues système. Une documentation plus détaillée des catalogues système suit.

La plupart des catalogues système sont recopiés de la base de données modèle lors de la création de la base de données et deviennent alors spécifiques à chaque base de données. Un petit nombre de catalogues sont physiquement partagés par toutes les bases de données d'une installation de PostgreSQL™. Ils sont indiqués dans les descriptions des catalogues.

Tableau 45.1. Catalogues système

Nom du catalogue	Contenu
pg_aggregate	fonctions d'agrégat
pg_am	méthodes d'accès aux index
pg_amop	opérateurs des méthodes d'accès
pg_amproc	procédures de support des méthodes d'accès
pg_attrdef	valeurs par défaut des colonnes
pg_attribute	colonnes des tables (« attributs »)
pg_authid	identifiants d'autorisation (rôles)
pg_auth_members	relations d'appartenance aux identifiants d'autorisation
pg_cast	conversions de types de données (*cast*)
pg_class	tables, index, séquences, vues (« relations »)
pg_constraint	contraintes de vérification, contraintes uniques, contraintes de clés primaires, contraintes de clés étrangères
pg_conversion	informations de conversions de codage
pg_database	bases de données du cluster PostgreSQL™
pg_db_role_setting	configuration par rôle et par base de données
pg_default_acl	droits par défaut sur des types d'objets
pg_depend	dépendances entre objets de la base de données
pg_description	descriptions ou commentaires des objets de base de données
pg_enum	définitions des labels et des valeurs des enum
pg_foreign_data_wrapper	définitions des wrappers de données distantes
pg_foreign_server	définitions des serveurs distants
pg_index	informations supplémentaires des index
pg_inherits	hiérarchie d'héritage de tables
pg_language	langages d'écriture de fonctions
pg_largeobject	pages de données pour les « Large Objects »
pg_largeobject_metadata	métadonnées pour les « Large Objects »
pg_namespace	schémas
pg_opclass	classes d'opérateurs de méthodes d'accès
pg_operator	opérateurs

Nom du catalogue	Contenu
pg_opfamily	familles d'opérateurs de méthodes d'accès
pg_pltemplate	données modèles pour les langages procéduraux
pg_proc	fonctions et procédures
pg_rewrite	règles de réécriture de requêtes
pg_shdepend	dépendances sur les objets partagés
pg_shdescription	commentaires sur les objets partagés
pg_statistic	statistiques de l'optimiseur de requêtes
pg_tablespace	*tablespaces* du cluster de bases de données
pg_trigger	déclencheurs
pg_ts_config	configuration de la recherche plein texte
pg_ts_config_map	configuration de la recherche plein texte pour la correspondance des lexèmes (*token*)
pg_ts_dict	dictionnaires de la recherche plein texte
pg_ts_parser	analyseurs de la recherche plein texte
pg_ts_template	modèles de la recherche plein texte
pg_type	types de données
pg_user_mapping	correspondance d'utilisateurs sur des serveurs distants

45.2. pg_aggregate

Le catalogue pg_aggregate stocke les informations concernant les fonctions d'agrégat. Une fonction d'agrégat est une fonction qui opère sur un ensemble de données (typiquement une colonne de chaque ligne qui correspond à une condition de requête) et retourne une valeur unique calculée à partir de toutes ces valeurs. Les fonctions d'agrégat classiques sont `sum` (somme), `count` (compteur) et `max` (plus grande valeur). Chaque entrée de pg_aggregate est une extension d'une entrée de pg_proc. L'entrée de pg_proc contient le nom de l'agrégat, les types de données d'entrée et de sortie, et d'autres informations similaires aux fonctions ordinaires.

Tableau 45.2. Les colonnes de pg_aggregate

Nom	Type	Références	Description
`aggfnoid`	regproc	`pg_proc.oid`	OID pg_proc de la fonction d'agrégat
`aggtransfn`	regproc	`pg_proc.oid`	Fonction de transition
`aggfinalfn`	regproc	`pg_proc.oid`	Fonction finale (0 s'il n'y en a pas)
`aggsortop`	oid	`.oi` `pg_operatord`	Opérateur de tri associé (0 s'il n'y en a pas)
`aggtranstype`	oid	`pg_type.oid`	Type de la donnée interne de transition (état) de la fonction d'agrégat
`agginitval`	text		Valeur initiale de la fonction de transition. C'est un champ texte qui contient la valeur initiale dans sa représentation externe en chaîne de caractères. Si ce champ est NULL, la valeur d'état de transition est initialement NULL.

Les nouvelles fonctions d'agrégat sont enregistrées avec la commande CREATE AGGREGATE(7). La Section 35.10, « Agrégats utilisateur » fournit de plus amples informations sur l'écriture des fonctions d'agrégat et sur la signification des fonctions de transition.

45.3. pg_am

Le catalogue pg_am stocke les informations concernant les méthodes d'accès aux index. On trouve une ligne par méthode d'accès supportée par le système. Le contenu de ce catalogue est discuté en détails dans Chapitre 51, Définition de l'interface des méthodes d'accès aux index.

Tableau 45.3. Colonnes de pg_am

Nom	Type	Références	Description
amname	name		Nom de la méthode d'accès
amstrategies	int2		Nombre de stratégies d'opérateur pour cette méthode d'accès, ou zéro si la méthode d'accès n'a pas un ensemble fixe de stratégies opérateurs
amsupport	int2		Nombre de routines de support pour cette méthode d'accès
amcanorder	bool		La méthode d'accès supporte-t-elle les parcours ordonnés ?
amcanbackward	bool		La méthode d'accès supporte-t-elle les parcours en arrière ?
amcanunique	bool		La méthode d'accès supporte-t-elle les index uniques ?
amcanmulticol	bool		La méthode d'accès supporte-t-elle les index multicolonnes ?
amoptionalkey	bool		La méthode d'accès supporte-t-elle un parcours sans contrainte pour la première colonne de l'index ?
amindexnulls	bool		La méthode d'accès supporte-t-elle les entrées d'index NULL ?
amsearchnulls	bool		La méthode d'accès supporte-t-elle les recherches `IS NULL`/`NOT NULL` ?
amstorage	bool		Le type de données de stockage d'index peut-il différer du type de données de la colonne ?
amclusterable	bool		La commande CLUSTER peut-elle être utilisée avec un index de ce type ?
amkeytype	oid	`pg_type.oid`	Type de donnée stockée dans l'index, ou zéro si le type n'est pas de taille fixe
aminsert	regproc	`pg_proc.oid`	Fonction « insérer cette ligne »
ambeginscan	regproc	`pg_proc.oid`	Fonction « commencer un nouveau parcours »
amgettuple	regproc	`pg_proc.oid`	Fonction « prochaine ligne valide », ou zéro si aucune
amgetbitmap	regproc	`pg_proc.oid`	Fonction « récupérer toutes les lignes valides »
amrescan	regproc	`pg_proc.oid`	Fonction « redémarrer ce parcours »
amendscan	regproc	`pg_proc.oid`	Fonction « arrêter ce parcours »
ammarkpos	regproc	`pg_proc.oid`	Fonction « marquer la position actuelle du parcours »
amrestrpos	regproc	`pg_proc.oid`	Fonction « restaurer une position de parcours marquée »
ambuild	regproc	`pg_proc.oid`	Fonction « construire un nouvel index »
ambulkdelete	regproc	`pg_proc.oid`	Fonction de destruction en masse
amvacuumcleanup	regproc	`pg_proc.oid`	Fonction de nettoyage post-**VACUUM**
amcostestimate	regproc	`pg_proc.oid`	Fonction d'estimation du coût de parcours d'un index
amoptions	regproc	`pg_proc.oid`	Fonction d'analyse et de validation du champ *reloptions* d'un index

45.4. pg_amop

Le catalogue pg_amop stocke les informations concernant les opérateurs associés aux familles d'opérateurs des méthodes d'accès aux index. Il y a une ligne pour chaque opérateur membre d'une famille. Un opérateur peut apparaître dans plus d'une famille, mais ne peut pas apparaître dans plus d'une position à l'intérieur d'une famille.

Tableau 45.4. Colonnes de pg_amop

Nom	Type	Références	Description
amopfamily	oid	.oi pg_opfamilyd	La famille d'opérateur
amoplefttype	oid	pg_type.oid	Type de données en entrée, côté gauche, de l'opérateur
amoprighttype	oid	pg_type.oid	Type de données en entrée, côté droit, de l'opérateur
amopstrategy	int2		Numéro de stratégie d'opérateur
amopopr	oid	.oi pg_operatord	OID de l'opérateur
amopmethod	oid	pg_am.oid	Méthode d'accès à l'index pour cette famille d'opérateur

Une entrée dans *amopmethod* doit correspondre au opfmethod de sa famille d'opérateur parent (l'inclusion de *amopmethod* à ce niveau est une dénormalisation intentionnelle de la structure du catalogue pour des raisons de performance). De plus, *amoplefttype* et *amoprighttype* doivent correspondre aux champs *oprleft* et *oprright* de l'entrée pg_operator référencée.

45.5. pg_amproc

Le catalogue pg_amproc stocke les informations concernant les procédures de support associées aux familles d'opérateurs de méthodes d'accès. Il y a une ligne pour chaque procédure de support appartenant à une famille.

Tableau 45.5. Colonnes de pg_amproc

Nom	Type	Références	Description
amprocfamily	oid	.oi pg_opfamilyd	La famille d'opérateur
amproclefttype	oid	pg_type.oid	Type de données en entrée, côté gauche, de l'opérateur associé
amprocrighttype	oid	pg_type.oid	Type de données en entrée, côté droit, de l'opérateur associé
amprocnum	int2		Numéro de procédure de support
amproc	regproc	pg_proc.oid	OID de la procédure

On interprète habituellement les champs *amproclefttype* et *amprocrighttype* comme identifiant les types de données des côtés gauche et droit d'opérateur(s) supporté(s) par une procédure particulière. Pour certaines méthodes d'accès, ils correspondent aux types de données en entrée de la procédure elle-même. Il existe une notion de procédures de support par « défaut » pour un index, procédures pour lesquelles *amproclefttype* et *amprocrighttype* sont tous deux équivalents à l'*opcintype* de l'opclass de l'index.

45.6. pg_attrdef

Le catalogue pg_attrdef stocke les valeurs par défaut des colonnes. Les informations principales des colonnes sont stockées dans pg_attribute (voir plus loin). Seules les colonnes pour lesquelles une valeur par défaut est explicitement indiquée (quand la table est créée ou quand une colonne est ajoutée) ont une entrée dans pg_attrdef.

Tableau 45.6. Colonnes de pg_attrdef

Nom	Type	Références	Description
adrelid	oid	pg_class.oid	La table à laquelle appartient la colonne
adnum	int2	pg_attribute.attnum	Numéro de la colonne
adbin	text		Représentation interne de la valeur par défaut de la colonne
adsrc	text		Représentation lisible de la valeur par défaut

Le champ *adsrc* est historique. Il est préférable de ne pas l'utiliser parce qu'il ne conserve pas de trace des modifications qui peuvent affecter la représentation de la valeur par défaut. La compilation inverse du champ *adbin* (avec pg_get_expr par exemple) est une meilleure façon d'afficher la valeur par défaut.

45.7. pg_attribute

Le catalogue pg_attribute stocke les informations concernant les colonnes des tables. Il y a exactement une ligne de pg_attribute par colonne de table de la base de données. (Il y a aussi des attributs pour les index et, en fait, tous les objets qui possèdent des entrées dans pg_class.)

Le terme attribut, équivalent à colonne, est utilisé pour des raisons historiques.

Tableau 45.7. Colonnes de pg_attribute

Nom	Type	Références	Description
attrelid	oid	pg_class.oid	La table à laquelle appartient la colonne
attname	name		Le nom de la colonne
atttypid	oid	pg_type.oid	Le type de données de la colonne
attstattarget	int4		Contrôle le niveau de détail des statistiques accumulées pour la colonne par ANALYZE(7). Une valeur 0 indique qu'aucune statistique ne doit être collectée. Une valeur négative indique d'utiliser l'objectif de statistiques par défaut. Le sens exact d'une valeur positive dépend du type de données. Pour les données scalaires, *attstattarget* est à la fois le nombre de « valeurs les plus courantes » à collecter et le nombre d'histogrammes à créer.
attlen	int2		Une copie de pg_type.typlen pour le type de la colonne.
attnum	int2		Le numéro de la colonne. La numérotation des colonnes ordinaires démarre à 1. Les colonnes système, comme les *oid*, ont des numéros négatifs arbitraires.
attndims	int4		Nombre de dimensions, si la colonne est de type tableau, sinon 0. (Pour l'instant, le nombre de dimensions des tableaux n'est pas contrôlé, donc une valeur autre que 0 indique que « c'est un tableau ».)
attcacheoff	int4		Toujours -1 sur disque, mais peut être mis à jour lorsque la ligne est chargée en mémoire, pour mettre en cache l'emplacement de l'attribut dans la ligne.
atttypmod	int4		Stocke des données spécifiques au type de données précisé lors de la création de la table (par exemple, la taille maximale d'une colonne de type varchar). Il est transmis aux fonctions spécifiques au type d'entrée de données et de vérification de taille. La valeur est généralement -1 pour les types de données qui n'ont pas besoin de *atttypmod*.

Nom	Type	Références	Description
attbyval	bool		Une copie de pg_type.typbyval du type de la colonne.
attstorage	char		Contient normalement une copie de pg_type.typstorage du type de la colonne. Pour les types de données TOASTables, cette valeur peut être modifiée après la création de la colonne pour en contrôler les règles de stockage.
attalign	char		Une copie de pg_type.typalign du type de la colonne.
attnotnull	bool		Indique une contrainte de non-nullité de colonne. Il est possible de changer cette colonne pour activer ou désactiver cette contrainte.
atthasdef	bool		Indique que la colonne a une valeur par défaut. Dans ce cas, il y a une entrée correspondante dans le catalogue pg_attrdef pour définir cette valeur.
attisdropped	bool		Indique que la colonne a été supprimée et n'est plus valide. Une colonne supprimée est toujours présente physiquement dans la table, mais elle est ignorée par l'analyseur de requête et ne peut être accédée en SQL.
attislocal	bool		La colonne est définie localement dans la relation. Une colonne peut être simultanément définie localement et héritée.
attinhcount	int4		Nombre d'ancêtres directs de la colonne. Une colonne qui a au moins un ancêtre ne peut être ni supprimée ni renommée.
attacl	aclitem[]		Droits d'accès niveau colonne, s'il y en a qui ont été spécifiquement accordés à cette colonne
attoptions	text[]		Options au niveau colonne, en tant que chaînes du type « motclé=valeur »

Dans l'entrée pg_attribute d'une colonne supprimée, *atttypid* est réinitialisée à 0 mais *attlen* et les autres champs copiés à partir de pg_type sont toujours valides. Cet arrangement est nécessaire pour s'adapter à la situation où le type de données de la colonne supprimée a été ensuite supprimé et qu'il n'existe donc plus de ligne pg_type. *attlen* et les autres champs peuvent être utilisés pour interpréter le contenu d'une ligne de la table.

45.8. pg_authid

Le catalogue pg_authid contient les informations concernant les identifiants pour les autorisations d'accès aux bases de données (rôles). Un rôle englobe les concepts d'« utilisateur » et de « groupe ». Un utilisateur est essentiellement un rôle qui a l'attribut de connexion (*rolcanlogin*). Tout rôle (avec ou sans *rolcanlogin*) peut avoir d'autres rôles comme membres ; voir pg_auth_members.

Comme ce catalogue contient les mots de passe, il ne doit pas être lisible par tout le monde. pg_roles est une vue, lisible par tout le monde, de pg_authid qui masque le champ du mot de passe.

Chapitre 20, Rôles et droits de la base de données contient des informations détaillées sur les utilisateurs et sur la gestion des droits.

Comme l'identité des utilisateurs est identique pour tout le cluster de bases de données, pg_authid est partagé par toutes les bases du cluster ; il n'existe qu'une seule copie de pg_authid par cluster, non une par base de données.

Tableau 45.8. Colonnes de pg_authid

Nom	Type	Description
rolname	name	Nom du rôle
rolsuper	bool	Le rôle est superutilisateur

Nom	Type	Description
rolall	bool	Le rôle hérite automatiquement des droits des rôles dont il est membre
rolcreaterole	bool	Le rôle peut créer d'autres rôles
rolcreatedb	bool	Le rôle peut créer des bases de données
rolcatupdate	bool	Le rôle peut mettre à jour les catalogues système directement. (Même un superutilisateur ne peut le faire si cette colonne n'est pas à `true`.)
rolcanlogin	bool	Le rôle peut se connecter, c'est-à-dire qu'il peut être donné comme identifiant d'autorisation de session.
rolconnlimit	int4	Pour les rôles qui peuvent se connecter, indique le nombre maximum de connexions concurrentes que le rôle peut initier. -1 signifie qu'il n'y a pas de limite.
rolpassword	text	Le mot de passe (éventuellement chiffré) ; NULL si aucun. Si le mot de passe est chiffré, cette colonne contient la chaîne « md5 » suivi par un hachage md5 hexadécimal sur 32 caractères. Le hachage md5 correspondra au mot de passe de l'utilisateur concaténé à son nom (par exemple si l'utilisateur joe a le mot de passe xyzzy, PostgreSQL™ enregistrera le hachage md5 du mot xyzzyjoe).
rolvaliduntil	timestamptz	Date d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe) ; NULL si indéfiniment valable

45.9. pg_auth_members

Le catalogue pg_auth_members contient les relations d'appartenance entre les rôles. Tout ensemble non circulaire d'appartenances est autorisé.

Parce que les identités de l'utilisateur sont valables sur l'ensemble du cluster, pg_auth_members est partagé par toutes les bases de données d'un cluster : il n'existe qu'une seule copie de pg_auth_members par cluster, pas une par base de données.

Tableau 45.9. Colonnes de pg_auth_members

Nom	Type	Références	Description
roleid	oid	`pg_authid.oid`	Identifiant d'un rôle qui a un membre
member	oid	`pg_authid.oid`	Identifiant d'un rôle qui est membre d'un *roleid*
grantor	oid	`pg_authid.oid`	Identifiant du rôle qui a autorisé cette appartenance
admin_option	bool		Vrai si *member* peut donner l'appartenance à *roleid* aux autres

45.10. pg_cast

Le catalogue pg_cast stocke les chemins de conversion de type de donnée, qu'il s'agisse de ceux par défaut ou ceux définis avec la commande CREATE CAST(7).

pg_cast ne représente pas toutes les conversions de type que le système connaît, seulement celles qui ne peuvent pas se déduire à partir de règles génériques. Par exemple, la conversion entre un domaine et son type de base n'est pas représentée explicitement dans pg_cast. Autre exception importante : « les conversions automatiques d'entrée/sortie », celles réalisées en utilisant les propres fonctions d'entrée/sortie du type de données pour convertir vers ou à partir du text ou des autres types de chaînes de caractères, ne sont pas représentées explicitement dans pg_cast.

Tableau 45.10. Colonnes de pg_cast

Nom	Type	Références	Description
castsource	oid	`pg_type.oid`	OID du type de données source

Nom	Type	Références	Description
casttarget	oid	pg_type.oid	OID du type de données cible
castfunc	oid	pg_proc.oid	OID de la fonction à utiliser pour réaliser la conversion. 0 si la méthode ne requiert pas une fonction.
castcontext	char		Indique dans quel contexte la conversion peut être utilisée. e si seules les conversions explicites sont autorisées (avec CAST ou ::). a si les conversions implicites lors de l'affectation à une colonne sont autorisées, en plus des conversions explicites. i si les conversions implicites dans les expressions sont autorisées en plus des autres cas.
castmethod	char		Indique comment la conversion est effectuée. f signifie que la fonction indiquée dans le champ *castfunc* est utilisée. i signifie que les fonctions d'entrée/sortie sont utilisées. b signifie que les types sont binairement coercibles, et que par conséquent aucune conversion n'est nécessaire.

Les fonctions de transtypage listées dans pg_cast doivent toujours prendre le type source de la conversion comme type du premier argument et renvoyer le type de destination de la conversion comme type de retour. Une fonction de conversion peut avoir jusqu'à trois arguments. Le deuxième argument, s'il est présent, doit être de type integer ; il reçoit le modificateur de type associé avec le type de destination ou -1 s'il n'y en a pas. Le troisième argument, s'il est présent, doit être de type boolean ; il reçoit true si la conversion est une conversion explicite, false sinon.

Il est possible de créer une entrée pg_cast dans laquelle les types source et cible sont identiques si la fonction associée prend plus d'un argument. De telles entrées représentent les « fonctions de forçage de longueur » qui forcent la validité des valeurs de ce type pour une valeur particulière du modificateur de type.

Quand une entrée pg_cast possède des types différents pour la source et la cible et une fonction qui prend plus d'un argument, le transtypage et le forçage de longueur s'effectuent en une seule étape. Lorsqu'une telle entrée n'est pas disponible, le forçage vers un type qui utilise un modificateur de type implique deux étapes, une de transtypage, l'autre pour appliquer le modificateur.

45.11. pg_class

Le catalogue pg_class liste les tables, et à peu près tout ce qui contient des colonnes ou ressemble de près ou de loin à une table. Cela inclut les index (mais il faut aussi aller voir dans pg_index), les séquences, les vues, les types composites et les tables TOAST ; voir *relkind*. Par la suite, lorsque l'on parle de « relation », on sous-entend tous ces types d'objets. Les colonnes ne sont pas toutes significatives pour tous les types de relations.

Tableau 45.11. Colonnes de pg_class

Nom	Type	Références	Description
relname	name		Nom de la table, vue, index, etc.
relnamespace	oid	pg_namespace.oid	OID du *namespace* qui contient la relation.
reltype	oid	pg_type.oid	OID du type de données qui correspond au type de ligne de la table, s'il y en a un. 0 pour les index qui n'ont pas d'entrée dans pg_type.
reloftype	oid	pg_type.oid	Pour les tables typées, l'OID du type composite sous-jacent. Sinon, 0 dans tous les autres cas.
relowner	oid	pg_authid.oid	Propriétaire de la relation.
relam	oid	pg_am.oid	S'il s'agit d'un index, OID de la méthode d'accès utilisée (B-tree, hash, etc.)
relfilenode	oid		Nom du fichier disque de la relation ; zéro signifie que c'est une relation « mapped » dont le nom de fichier est déterminé par un statut de bas niveau.
reltablespace	oid	pg_tablespace.oid	Le *tablespace* dans lequel est stocké la relation. Si 0, il s'agit du *tablespace* par défaut de la base de données.

Nom	Type	Références	Description
			(Sans intérêt si la relation n'est pas liée à un fichier disque.)
relpages	int4		Taille du fichier disque, exprimée en pages (de taille BLCKSZ). Ce n'est qu'une estimation utilisée par le planificateur. Elle est mise à jour par les commandes **VACUUM**, **ANALYZE** et quelques commandes DDL comme **CREATE INDEX**.
reltuples	float4		Nombre de lignes de la table. Ce n'est qu'une estimation utilisée par le planificateur. Elle est mise à jour par les commandes **VACUUM**, **ANALYZE** et quelques commandes DDL comme **CREATE INDEX**.
reltoastrelid	oid	`pg_class.oid`	OID de la table TOAST associée à cette table. 0 s'il n'y en a pas. La table TOAST stocke les attributs de grande taille « hors ligne » dans une table secondaire.
reltoastidxid	oid	`pg_class.oid`	Pour une table TOAST, OID de son index. 0 si ce n'est pas une table TOAST.
relhasindex	bool		Vrai si c'est une table et qu'elle possède (ou possédait encore récemment) quelque index.
relisshared	bool		Vrai si cette table est partagée par toutes les bases de données du cluster. Seuls certains catalogues système (comme pg_database) sont partagés.
relistemp	bool		Vrai si cette table est une relation temporaire. Si c'est le cas, seule la session créatrice peut accéder sans danger à son contenu.
relkind	char		`r` = table ordinaire, `i` = index, `S` = séquence, `v` = vue, `c` = type composite, `t` = table TOAST.
relnatts	int2		Nombre de colonnes utilisateur dans la relation (sans compter les colonnes système). Il doit y avoir le même nombre d'entrées dans pg_attribute. Voir aussi `pg_attribute.attnum`.
relchecks	int2		Nombre de contraintes de vérification (`CHECK`) sur la table ; voir le catalogue pg_constraint.
relhasoids	bool		Vrai si un OID est engendré pour chaque ligne de la relation.
relhaspkey	bool		Vrai si la table a (ou a eu) une clé primaire.
relhasexclusion	bool		Pour une table, true si la table a (ou a eu) des contraintes d'exclusion ; pour un index, true si l'index supporte une contrainte d'exclusion
relhasrules	bool		Vrai si la table contient (ou a contenu) des règles ; voir le catalogue pg_rewrite.
relhastriggers	bool		Vrai si la table a (ou a eu) des triggers ; voir le catalogue pg_trigger
relhassubclass	bool		Vrai si au moins une table hérite ou a hérité de la table considérée.
relfrozenxid	xid		Tous les ID de transaction avant celui-ci ont été remplacés par un ID de transaction permanent (« frozen »). Ceci est utilisé pour déterminer si la table doit être nettoyée (VACUUM) pour éviter un bouclage des ID de transaction (*ID wraparound*) ou pour compacter `pg_clog`. 0 (InvalidTransactionId) si la relation n'est pas une table.
relacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour plus de détails.
reloptions	text[]		Options spécifiques de la méthode d'accès, représentées par des chaînes du type « motclé=valeur »

Plusieurs des drapeaux booléens dans pg_class sont maintenus faiblement : la valeur true est garantie s'il s'agit du bon état, mais elle pourrait ne pas être remise à false immédiatement quand la condition n'est plus vraie. Par exemple, `relhasindex` est configurée par **CREATE INDEX** mais n'est jamais remise à false par **DROP INDEX**. C'est **VACUUM** qui le fera `relhasindex` s'il découvre que la table n'a pas d'index. Cet arrangement évite des fenêtres de vulnérabilité et améliore la concurrence.

45.12. pg_constraint

Le catalogue pg_constraint stocke les vérifications, clés primaires, clés uniques, étrangères et d'exclusion des tables. (Les contraintes de colonnes ne sont pas traitées de manière particulière. Elles sont équivalentes à des contraintes de tables.) Les contraintes NOT NULL sont représentées dans le catalogue pg_attribute, pas ici.

Les triggers de contraintes définies par des utilisateurs (créés avec **CREATE CONSTRAINT TRIGGER**) ont aussi une entrée dans cette table.

Les contraintes de vérification de domaine sont également stockées dans ce catalogue.

Tableau 45.12. Colonnes de pg_constraint

Nom	Type	Références	Description
conname	name		Nom de la contrainte (pas nécessairement unique !)
connamespace	oid	`.o` `pg_namespaceid`	OID du *namespace* qui contient la contrainte.
contype	char		`c` = contrainte de vérification, `f` = contrainte de clé étrangère, `p` = contrainte de clé primaire, `u` = contrainte d'unicité, `t` = contrainte trigger, `x` = contrainte d'exclusion
condeferrable	bool		La contrainte peut-elle être retardée (*deferable*) ?
condeferred	bool		La contrainte est-elle retardée par défaut ?
conrelid	oid	`pg_class.oid`	Table à laquelle appartient la contrainte ; 0 si ce n'est pas une contrainte de table.
contypid	oid	`pg_type.oid`	Domaine auquel appartient la contrainte ; 0 si ce n'est pas une contrainte de domaine.
conindid	oid	`pg_class.oid`	L'index qui force cette contrainte (unique, clé primaire, clé étrangère, d'exclusion) ; sinon 0
confrelid	oid	`pg_class.oid`	Si c'est une clé étrangère, la table référencée ; sinon 0
confupdtype	char		Code de l'action de mise à jour de la clé étrangère : `a` = no action, `r` = restrict, `c` = cascade, `n` = set null, `d` = set default
confdeltype	char		Code de l'action de suppression de clé étrangère : `a` = no action, `r` = restrict, `c` = cascade, `n` = set null, `d` = set default
confmatchtype	char		Type de concordance de la clé étrangère : `f` = full, `p` = partial, `u` = simple (non spécifié)
conislocal	bool		Cette contrainte est définie localement dans la relation. Notez qu'une contrainte peut être définie localement et héritée simultanément
coninhcount	int4		Le nombre d'ancêtres d'héritage directs que cette contraite possède. Une contrainte avec un nombre non nul d'ancêtres ne peut être ni supprimée ni renommée.

conkey	int2[]	`.a`
		`tt`
		`nu`
		`pg_attributem`
			S'il s'agit d'une contrainte de table (incluant les clés étrangères mais pas les triggers de contraintes), liste des colonnes contraintes

confkey	int2[]	`.a`	S'il s'agit d'une clé étrangère, liste des colonnes ré-

Nom	Type	Références	Description
		ttnum	férencées
conpfeqop	oid[]	.oi pg_operatord	S'il s'agit d'une clé étrangère, liste des opérateurs d'égalité pour les comparaisons clé primaire/clé étrangère
conppeqop	oid[]	.oi pg_operatord	S'il s'agit d'une clé étrangère, liste des opérateurs d'égalité pour les comparaisons clé primaire/clé primaire
conffeqop	oid[]	.oi pg_operatord	S'il s'agit d'une clé étrangère, liste des opérateurs d'égalité pour les comparaisons clé étrangère/clé étrangère
conexclop	oid[]	.oi pg_operatord	Si une contrainte d'exclusion, liste les opérateurs d'exclusion par colonne
conbin	text		S'il s'agit d'une contrainte de vérification, représentation interne de l'expression
consrc	text		S'il s'agit d'une contrainte de vérification, représentation compréhensible de l'expression

Dans le cas d'une contrainte d'exclusion, *conkey* est seulement utile pour les éléments contraints qui sont de simples références de colonnes. Dans les autres cas, un zéro apparaît dans *conkey* et l'index associé doit être consulté pour découvrir l'expression contrainte. (du coup, *conkey* a le même contenu que pg_index.*indkey* pour l'index.)

> **Note**
>
> *consrc* n'est pas actualisé lors de la modification d'objets référencés ; par exemple, il ne piste pas les renommages de colonnes. Plutôt que se fier à ce champ, il est préférable d'utiliser pg_get_constraintdef() pour extraire la définition d'une contrainte de vérification.

> **Note**
>
> pg_class.relchecks doit accepter le même nombre de contraintes de vérification pour chaque relation. De plus, pg_class.relhasexclusion doit être vrai s'il n'y a pas d'entrées de contraintes d'exclusion pour la relation.

45.13. pg_conversion

Le catalogue pg_conversion décrit les procédures de conversion de codage disponibles. Voir la commande CREATE CONVERSION(7) pour plus d'informations.

Tableau 45.13. Colonnes de pg_conversion

Nom	Type	Références	Description
conname	name		Nom de la conversion (unique au sein d'un *namespace*)
connamespace	oid	.o pg_namespaceid	OID du *namespace* qui contient la conversion.
conowner	oid	pg_authid.oid	Propriétaire de la conversion
conforencoding	int4		ID du codage source
contoencoding	int4		ID du codage de destination
conproc	regproc	pg_proc.oid	Procédure de conversion
condefault	bool		Vrai s'il s'agit de la conversion par défaut

45.14. pg_database

Le catalogue pg_database stocke les informations concernant les bases de données disponibles. Celles-ci sont créées avec la commande CREATE DATABASE(7). Consulter le Chapitre 21, Administration des bases de données pour les détails sur la signification de certains paramètres.

Contrairement à la plupart des catalogues système, pg_database est partagé par toutes les bases de données d'un cluster : il n'y a qu'une seule copie de pg_database par cluster, pas une par base.

Tableau 45.14. Colonnes de pg_database

Nom	Type	Références	Description
datname	name		Nom de la base de données
datdba	oid	pg_authid.oid	Propriétaire de la base, généralement l'utilisateur qui l'a créée
encoding	int4		Encodage de la base de données (la fonction pg_encoding_to_char() peut convertir ce nombre en nom de l'encodage)
datcollate	name		LC_COLLATE pour cette base de données
datctype	name		LC_CTYPE pour cette base de données
datistemplate	bool		Si ce champ est vrai, alors la base peut être utilisée dans la clause TEMPLATE de la commande **CREATE DATABASE** pour créer une nouvelle base comme clone de celle-ci.
datallowconn	bool		Si ce champ est faux, alors personne ne peut se connecter à cette base de données. Ceci est utilisé pour interdire toute modification de la base template0.
datconnlimit	int4		Nombre maximum de connexions concurrentes autorisées sur la base de données. -1 indique l'absence de limite.
datlastsysoid	oid		Dernier OID système de la base de données ; utile en particulier pour pg_dump.
datfrozenxid	xid		Tous les ID de transaction avant celui-ci ont été remplacés par un ID de transaction permanent (« frozen »). Ceci est utilisé pour déterminer si la table doit être nettoyée (VACUUM) pour éviter un bouclage des ID de transaction (*ID wraparound*) ou pour compacter pg_clog. C'est la valeur minimale des valeurs par table de pg_class.*relfrozenxid*.
dattablespace	oid	pg_tablespace.oid	Le *tablespace* par défaut de la base de données. Dans cette base de données, toutes les tables pour lesquelles pg_class.*reltablespace* vaut 0 sont stockées dans celui-ci ; en particulier, tous les catalogues système non partagés s'y trouvent.
datacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour les détails.

45.15. pg_db_role_setting

Le catalogue pg_db_role_setting enregistre les valeurs par défaut qui ont été configurées pour les variables de configuration, pour chaque combinaison de rôle et de base.

Contrairement à la plupart des catalogues systèmes, pg_db_role_setting est partagé parmi toutes les bases de données de l'instance : il n'existe qu'une copie de pg_db_role_setting par instance, pas une par base de données.

Tableau 45.15. Colonnes de pg_db_role_setting

Nom	Type	Références	Description
setdatabase	oid	pg_database.oid	L'OID de la base de données

Nom	Type	Références	Description
			pour laquelle la configuration est applicable ; zéro si cette configuration n'est pas spécifique à une base de données
setrole	oid	pg_authid.oid	L'OID du rôle pour laquelle la configuration est applicable ; zéro si cette configuration n'est pas spécifique à un rôle
setconfig	text[]		Valeurs par défaut pour les variables de configuration

45.16. pg_default_acl

Le catalogue pg_default_acl enregistre les droits initiaux à affecter aux nouveaux objets créés.

Tableau 45.16. Colonnes de pg_default_acl

Nom	Type	Références	Description
defaclrole	oid	pg_authid.oid	OID du rôle associé à cette entrée
defaclnamespace	oid	pg_namespace.oid	OID du schéma associé à cette entrée, 0 si aucun
defaclobjtype	char		Type de l'objet pour cette entrée : r = relation (table, vue), S = séquence, f = fonction
defaclacl	aclitem[]		Droits d'accès qu'auront les nouveaux objets de ce type

Une entrée pg_default_acl affiche les droits initiaux affectés à un objet appartenant à l'utilisateur indiqué. Il existe actuellement deux types d'entrées : des entrées « globales » avec *defaclnamespace* = 0, et des entrées « par schéma » qui référencent un schéma. Si une entrée globale est présente, alors elle *surcharge* les droits par défaut codés en dur pour le type de l'objet. Une entrée par schéma, si présente, représente les droits à *ajouter* aux droits par défaut globaux ou aux droits codés en dur.

Notez que quand une entrée de droits (ACL) dans un autre catalogue est NULL, cela veut dire que les droits par défaut codés en dur sont utilisés pour cet objet, et *non pas* ce qui pourrait être dans pg_default_acl à ce moment. pg_default_acl est seulement consulté durant la création de l'objet.

45.17. pg_depend

Le catalogue pg_depend enregistre les relations de dépendances entre les objets de la base de données. Cette information permet à la commande **DROP** de trouver les objets qui doivent être supprimés conjointement par la commande **DROP CASCADE** ou au contraire empêchent la suppression dans le cas de **DROP RESTRICT**.

Voir aussi pg_shdepend, qui remplit la même fonction pour les dépendances impliquant des objets partagés sur tout le cluster.

Tableau 45.17. Colonnes de pg_depend

Nom	Type	Références	Description
classid	oid	pg_class.oid	OID du catalogue système dans lequel l'objet dépendant se trouve.
objid	oid	toute colonne OID	OID de l'objet dépendant
objsubid	int4		Pour une colonne de table, ce champ indique le numéro de colonne (les champs *objid* et *classid* font référence à la table elle-même). Pour tous les autres types d'objets, cette colonne est à 0.
refclassid	oid	pg_class.oid	OID du catalogue système dans lequel l'objet réfé-

Nom	Type	Références	Description
			rencé se trouve.
refobjid	oid	toute colonne OID	OID de l'objet référencé
refobjsubid	int4		Pour une colonne de table, ce champ indique le numéro de colonne (les champs *refobjid* et *refclassid* font référence à la table elle même). Pour tous les autres types d'objets, cette colonne est à 0.
deptype	char		Code définissant la sémantique particulière de la relation de dépendance. Voir le texte.

Dans tous les cas, une entrée de pg_depend indique que l'objet de référence ne peut pas être supprimé sans supprimer aussi l'objet dépendant. Néanmoins, il y a des nuances, identifiées par *deptype* :

DEPENDENCY_NORMAL (n)
: Une relation normale entre des objets créés séparément. L'objet dépendant peut être supprimé sans affecter l'objet référencé. Ce dernier ne peut être supprimé qu'en précisant l'option CASCADE, auquel cas l'objet dépendant est supprimé lui-aussi. Exemple : une colonne de table a une dépendance normale avec ses types de données.

DEPENDENCY_AUTO (a)
: L'objet dépendant peut être supprimé séparément de l'objet référencé, mais il l'est automatiquement avec la suppression de ce dernier, quel que soit le mode RESTRICT ou CASCADE. Exemple : une contrainte nommée sur une table est auto-dépendante de la table, elle est automatiquement supprimée avec celle-ci.

DEPENDENCY_INTERNAL (i)
: L'objet dépendant est créé conjointement à l'objet référencé et fait partie intégrante de son implantation interne. Un **DROP** de l'objet dépendant est interdit (l'utilisateur est averti qu'il peut effectuer un **DROP** de l'objet référencé à la place). La suppression de l'objet référencé est propagée à l'objet dépendant que **CASCADE** soit précisé ou non. Exemple : un trigger créé pour vérifier une contrainte de clé étrangère est rendu dépendant de l'entrée de la contrainte dans pg_constraint.

DEPENDENCY_PIN (p)
: Il n'y a pas d'objet dépendant ; ce type d'entrée signale que le système lui-même dépend de l'objet référencé, et donc que l'objet ne doit jamais être supprimé. Les entrées de ce type sont créées uniquement par **initdb**. Les colonnes de l'objet dépendant contiennent des zéros.

D'autres types de dépendance peuvent apparaître dans le futur.

45.18. pg_description

Le catalogue pg_description stocke les descriptions (commentaires) optionnelles de chaque objet de la base de données. Les descriptions sont manipulées avec la commande COMMENT(7) et lues avec les commandes \d de psql. pg_description contient les descriptions prédifinies de nombreux objets internes.

Voir aussi pg_shdescription, qui offre la même fonction pour les descriptions des objets partagés au sein d'un cluster.

Tableau 45.18. Colonnes de pg_description

Nom	Type	Références	Description
objoid	oid	toute colonne OID	OID de l'objet commenté
classoid	oid	pg_class.oid	OID du catalogue système dans lequel apparaît l'objet
objsubid	int4		Pour un commentaire de colonne de table, le numéro de la colonne. Les champs *objoid* et *classoid* font référence à la table elle-même. Pour tous les autres types de données, cette colonne est à 0.
description	text		Texte quelconque commentant l'objet

45.19. pg_enum

Le catalogue système pg_enum contient des entrées qui correspondent aux types enum précisant les valeurs et labels associés. La représentation interne d'une valeur enum donnée est en fait l'OID de sa ligne associée dans pg_enum. Les OID pour un type enum particulier précisent aussi l'ordre du tri. Il n'y a toutefois aucune garantie quant au tri des OID de types enum non relatifs.

Tableau 45.19. Colonnes de pg_enum

Nom	Type	Références	Description
enumtypid	oid	`pg_type.oid`	OID de l'entrée pg_type correspondant à cette valeur d'enum
enumlabel	name		Le label texte pour cette valeur d'enum

45.20. pg_foreign_data_wrapper

Le catalogue pg_foreign_data_wrapper stocke les définitions des wrappers de données distantes. Un wrapper de données distantes est le mécanisme par lequel des données externes, stockées sur des serveurs distants, sont accédées.

Tableau 45.20. Colonnes de pg_foreign_data_wrapper

Nom	Type	Références	Description
fdwname	name		Nom du wrapper de données distantes
fdwowner	oid	`pg_authid.oid`	Propriétaire du wrapper de données distantes
fdwvalidator	oid	`pg_proc.oid`	Référence le validateur de fonction qui est responsable de vérifier la validité des options génériques passées au wrapper de données distantes, ainsi que les serveurs distants et les données de correspondance du wrapper de données distantes. Zéro si aucun validateur n'est fourni.
fdwacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour plus de détails
fdwoptions	text[]		Options spécifiques pour un wrapper de données distantes, sous la forme de chaînes « motclé=valeur »

45.21. pg_foreign_server

Le catalogue pg_foreign_server stocke les définitions de serveurs distants. Un serveur distant décrit la connexion à un serveur distant, qui stocke des données externes. Les serveurs distants sont accédés via des wrappers de données distantes.

Tableau 45.21. Colonnes pg_foreign_server

Nom	Type	Référence	Description
srvname	name		Nom du serveur distant
srvowner	oid	`pg_authid.oid`	Propriétaire du serveur distant
srvfdw	oid	`pg_foreign_data_wrapper.oid`	OID du wrapper de données distantes pour ce serveur distant
srvtype	text		Type du serveur (optionnel)

Nom	Type	Reférence	Description
srvversion	text		Version du serveur (optionnel)
srvacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour les détails
srvoptions	text[]		Options pour le serveur distant, sous la forme de chaînes « motclé=valeur ».

45.22. pg_index

Le catalogue pg_index contient une partie des informations concernant les index. Le reste se trouve pour l'essentiel dans pg_class.

Tableau 45.22. Colonnes de pg_index

Nom	Type	Références	Description
indexrelid	oid	`pg_class.oid`	OID de l'entrée dans pg_class de l'index
indrelid	oid	`pg_class.oid`	OID de l'entrée dans pg_class de la table sur laquelle porte l'index
indnatts	int2		Nombre de colonnes de l'index (duplique `pg_class.relnatts`)
indisunique	bool		Vrai s'il s'agit d'un index d'unicité
indisprimary	bool		Vrai s'il s'agit de l'index de clé primaire de la table (*indisunique* doit toujours être vrai quand ce champ l'est.)
indimmediate	bool		Si vrai, la vérification de l'unicité est forcée immédiatement lors de l'insertion (*indisunique* doit toujours être vrai quand ce champ l'est)
indisclustered	bool		Vrai si la table a été réorganisée en fonction de l'index
indisvalid	bool		Si vrai, l'index est valide pour les requêtes. Faux signifie que l'index peut être incomplet : les opérations INSERT/UPDATE peuvent toujours l'utiliser, mais il ne peut pas être utilisé sans risque pour les requêtes, et, dans le cas d'un index d'unicité, celle-ci n'est plus non-plus garantie.
indcheckxmin	bool		Si vrai, les requêtes ne doivent pas utiliser l'index tant que le *xmin* de cette ligne de pg_index est en-dessous de leur horizon d'événements TransactionXmin, car la table peut contenir des chaînes HOT cassées avec des lignes incompatibles qu'elles peuvent voir.
indisready	bool		Si vrai, l'index est actuellement prêt pour les insertions. Faux indique que l'index doit être ignoré par les opérations **INSERT/UPDATE**
indkey	int2vector	`pg_attribute.attnum`	C'est un tableau de valeurs *indnatts* qui indique les colonnes de la table indexées. Par exemple, une valeur `1 3` signifie que la première et la troisième colonne de la table composent la clé de l'index. Un 0 dans ce tableau indique que l'attribut de l'index correspondant est une expression sur les colonnes de la table plutôt qu'une simple référence de colonne.
indclass	oidvector	`pg_opclass.oid`	Pour chaque colonne de la clé d'indexation, contient l'OID de la classe d'opérateur à utiliser. Voir pg_opclass pour plus de détails.
indoption	int2vector		C'est un tableau de valeurs *indnatts* qui enregistrent des drapeaux d'information par colonne. La signification de ces drapeaux est définie par la méthode d'accès à

Nom	Type	Références	Description
			l'index.
indexprs	text		Arbres d'expression (en représentation `nodeToString()`) pour les attributs d'index qui ne sont pas de simples références de colonnes. Il s'agit d'une liste qui contient un élément par entrée à 0 dans *indkey*. Nul si tous les attributs d'index sont de simples références.
indpred	text		Arbre d'expression (en représentation `nodeToString()`) pour les prédicats d'index partiels. Nul s'il ne s'agit pas d'un index partiel.

45.23. pg_inherits

Le catalogue pg_alls enregistre l'information sur la hiérarchie d'héritage des tables. Il existe une entrée pour chaque table enfant direct dans la base de données. (L'héritage indirect peut être déterminé en suivant les chaînes d'entrées.)

Tableau 45.23. Colonnes de pg_alls

Nom	Type	Références	Description
inhrelid	oid	`pg_class.oid`	OID de la table fille
inhparent	oid	`pg_class.oid`	OID de la table mère
inhseqno	int4		S'il y a plus d'un parent direct pour une table fille (héritage multiple), ce nombre indique dans quel ordre les colonnes héritées doivent être arrangées. Le compteur commence à 1.

45.24. pg_language

Le catalogue pg_language enregistre les langages utilisables pour l'écriture de fonctions ou procédures stockées. Voir CREATE LANGUAGE(7) et dans le Chapitre 38, Langages de procédures pour plus d'information sur les gestionnaires de langages.

Tableau 45.24. Colonnes de pg_language

Nom	Type	Références	Description
lanname	name		Nom du langage
lanowner	oid	`pg_authid.oid`	Propriétaire du langage
lanispl	bool		Faux pour les langages internes (comme SQL) et vrai pour les langages utilisateur. À l'heure actuelle, pg_dump utilise ce champ pour déterminer les langages à sauvegarder mais cela peut être un jour remplacé par un mécanisme différent.
lanpltrusted	bool		Vrai s'il s'agit d'un langage de confiance (*trusted*), ce qui signifie qu'il est supposé ne pas donner accès à ce qui dépasse l'exécution normale des requêtes SQL. Seuls les superutilisateurs peuvent créer des fonctions dans des langages qui ne sont pas dignes de confiance.
lanplcallfoid	oid	`pg_proc.oid`	Pour les langages non-internes, ceci référence le gestionnaire de langage, fonction spéciale en charge de l'exécution de toutes les fonctions écrites dans ce langage.
laninline	oid	`pg_proc.oid`	Ceci référence une fonction qui est capable d'exécuter des blocs de code anonyme « en ligne » (blocs DO(7)). Zéro si les blocs en ligne ne sont pas supportés

Nom	Type	Références	Description
lanvalidator	oid	pg_proc.oid	Ceci référence une fonction de validation de langage, en charge de vérifier la syntaxe et la validité des nouvelles fonctions lors de leur création. 0 si aucun validateur n'est fourni.
lanacl	aclitem[]		Droits d'accès ;; voir GRANT(7) et REVOKE(7) pour les détails.

45.25. pg_largeobject

Le catalogue pg_largeobject contient les données qui décrivent les « objets volumineux » (*large objects*). Un objet volumineux est identifié par un OID qui lui est affecté lors de sa création. Chaque objet volumineux est coupé en segments ou « pages » suffisamment petits pour être facilement stockés dans des lignes de pg_largeobject. La taille de données par page est définie par LOBLKSIZE, qui vaut actuellement BLCKSZ/4, soit habituellement 2 Ko).

Avant PostgreSQL™ 9.0, il n'existait pas de droits associés aux « Large Objects ». Du coup, pg_largeobject était lisible par tout le monde et pouvait être utilisé pour obtenir les OID (et le contenu) de tous les « Large Objects » du système. Ce n'est plus le cas ; utilisez pg_largeobject_metadata pour obtenir une liste des OID des « Large Objects ».

Tableau 45.25. Colonnes de pg_largeobject

Nom	Type	References	Description
loid	oid	pg_largeobject_metadata.oid	Identifiant de l'objet volumineux qui contient la page
pageno	int4		Numéro de la page au sein de l'objet volumineux, en partant de 0
data	bytea		Données effectivement stockées dans l'objet volumineux. Il ne fait jamais plus de LOBLKSIZE mais peut faire moins.

Chaque ligne de pg_largeobject contient les données d'une page de l'objet volumineux, en commençant au décalage d'octet (pageno * LOBLKSIZE) dans l'objet. Ceci permet un stockage diffus : des pages peuvent manquer, d'autres faire moins de LOBLKSIZE octets même s'il ne s'agit pas de la dernière de l'objet. Les parties manquantes sont considérées comme des suites de zéro.

45.26. pg_largeobject_metadata

Le catalogue pg_largeobject_metadata contient des méta-données associées aux « Larges Objects ». Les données des « Larges Objects » sont réellement stockées dans pg_largeobject.

Tableau 45.26. Colonnes de pg_largeobject_metadata

Nom	Type	Description
lomowner	oid	pg_authid.oid
lomacl	aclitem[]	

45.27. pg_namespace

Le catalogue pg_namespace stocke les *namespace*. Un *namespace* est la structure sous-jacente aux schémas SQL : chaque *namespace* peut contenir un ensemble séparé de relations, types, etc. sans qu'il y ait de conflit de nommage.

Tableau 45.27. Colonnes de pg_namespace

Nom	Type	Références	Description
nspname	name		Nom du *namespace*
nspowner	oid	pg_authid.oid	Propriétaire du *namespace*
nspacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour les détails.

45.28. pg_opclass

Le catalogue pg_opclass définit les classes d'opérateurs de méthodes d'accès aux index. Chaque classe d'opérateurs définit la sémantique pour les colonnes d'index d'un type particulier et d'une méthode d'accès particulière. Une classe d'opérateur définit essentiellement qu'une famille d'opérateur particulier est applicable à un type de données indexable particulier. L'ensemble des opérateurs de la famille actuellement utilisables avec la colonne indexée sont tous ceux qui acceptent le type de données de la colonne en tant qu'entrée du côté gauche.

Les classes d'opérateurs sont longuement décrites dans la Section 35.14, « Interfacer des extensions d'index ».

Tableau 45.28. Colonnes de pg_opclass

Nom	Type	Références	Description
opcmethod	oid	pg_am.oid	Méthode d'accès à l'index pour laquelle est définie la classe d'opérateurs
opcname	name		Nom de la classe d'opérateurs
opcnamespace	oid	.o pg_namespaceid	*Namespace* de la classe d'opérateurs
opcowner	oid	pg_authid.oid	Propriétaire de la classe d'opérateurs
opcfamily	oid	.oi pg_opfamilyd	Famille d'opérateur contenant la classe d'opérateur
opcintype	oid	pg_type.oid	Type de données que la classe d'opérateurs indexe
opcdefault	bool		Vrai si la classe d'opérateurs est la classe par défaut pour *opcintype*
opckeytype	oid	pg_type.oid	Type de données stocké dans l'index ou 0 s'il s'agit du même que *opcintype*

L'*opcmethod* d'une classe d'opérateurs doit coïncider avec l'opfmethod de la famille d'opérateurs qui le contient. Il ne doit pas non plus y avoir plus d'une ligne pg_opclass pour laquelle opcdefault est vrai, quelque soit la combinaison de opcmethod et opcintype.

45.29. pg_operator

Le catalogue pg_operator stocke les informations concernant les opérateurs. Voir la commande CREATE OPERATOR(7) et la Section 35.12, « Opérateurs définis par l'utilisateur » pour plus d'informations.

Tableau 45.29. Colonnes de pg_operator

Nom	Type	Références	Description
oprname	name		Nom de l'opérateur
oprnamespace	oid	.o pg_namespaceid	OID du *namespace* qui contient l'opérateur
oprowner	oid	pg_authid.oid	Propriétaire de l'opérateur
oprkind	char		b = infix (« les deux »), l = prefix (« gauche »), r = postfix (« droit »)
oprcanmerge	bool		L'opérateur supporte les jointures de fusion
oprcanhash	bool		L'opérateur supporte les jointures par découpage
oprleft	oid	pg_type.oid	Type de l'opérande de gauche

Nom	Type	Références	Description
oprright	oid	pg_type.oid	Type de l'opérande de droite
oprresult	oid	pg_type.oid	Type du résultat
oprcom	oid	.oi pg_operatord	Commutateur de l'opérateur, s'il existe
oprnegate	oid	.oi pg_operatord	Négateur de l'opérateur, s'il existe
oprcode	regproc	pg_proc.oid	Fonction codant l'opérateur
oprrest	regproc	pg_proc.oid	Fonction d'estimation de la sélectivité de restriction de l'opérateur
oprjoin	regproc	pg_proc.oid	Fonction d'estimation de la sélectivité de jointure de l'opérateur

Les colonnes inutilisées contiennent des zéros. *oprleft* vaut, par exemple, 0 pour un opérateur préfixe.

45.30. pg_opfamily

Le catalogue pg_opfamily définit les familles d'opérateur. Chaque famille d'opérateur est un ensemble d'opérateurs et de routines de support associées codant les sémantiques définies pour une méthode d'accès particulière de l'index. De plus, les opérateurs d'une famille sont tous « compatibles », au sens défini par la méthode d'accès. Le concept de famille d'opérateur autorise l'utilisation des opérateurs inter-type de données avec des index et l'utilisation des sémantiques de méthode d'accès.

Les familles d'opérateur sont décrites dans Section 35.14, « Interfacer des extensions d'index ».

Tableau 45.30. Colonnes de pg_opfamily

Nom	Type	Références	Description
opfmethod	oid	pg_am.oid	Méthode d'accès à l'index pour la famille d'opérateur
opfname	name		Nom de la famille d'opérateur
opfnamespace	oid	.o pg_namespaceid	*namespace* de la famille d'opérateur
opfowner	oid	pg_authid.oid	Propriétaire de la famille d'opérateur

La majorité des informations définissant une famille d'opérateur n'est pas dans la ligne correspondante de pg_opfamily mais dans les lignes associées de pg_amop, pg_amproc, et pg_opclass.

45.31. pg_pltemplate

Le catalogue pg_pltemplate stocke les informations squelettes (« template ») des langages procéduraux. Un squelette de langage permet la création de ce langage dans une base de données particulière à l'aide d'une simple commande **CREATE LANGUAGE**, sans qu'il soit nécessaire de spécifier les détails de l'implantation.

Contrairement à la plupart des catalogues système, pg_pltemplate est partagé par toutes les bases de données d'un cluster : il n'existe qu'une seule copie de pg_pltemplate par cluster, et non une par base de données. L'information est de ce fait accessible à toute base de données.

Tableau 45.31. Colonnes de pg_pltemplate

Nom	Type	Description
tmplname	name	Nom du langage auquel est associé le modèle
tmpltrusted	boolean	True s'il s'agit d'un langage de confiance
tmpldbacreate	boolean	True s'il s'agit d'un langage créé par le propriétaire de la base
tmplhandler	text	Nom de la fonction de gestion des appels
tmplinline	text	Nom de la fonction de gestion des blocs anonymes. NULL sinon

Nom	Type	Description
tmplvalidator	text	Nom de la fonction de validation, ou NULL si aucune
tmpllibrary	text	Chemin de la bibliothèque partagée qui code le langage
tmplacl	aclitem[]	Droits d'accès au modèle (actuellement inutilisé)

Il n'existe actuellement aucune commande de manipulation des modèles de langages procéduraux ; pour modifier l'information intégrée, un superutilisateur doit modifier la table en utilisant les commandes **INSERT**, **DELETE** ou **UPDATE** habituelles. Il est probable qu'une future version de PostgreSQL™ propose des commandes permettant de modifier les entrées de façon plus propre.

Lorsqu'il est renseigné, le champ *tmplacl* gère le contrôle des accès au modèle (c'est-à-dire le droit de créer un langage à partir de ce modèle), mais pas le contrôle des accès aux langages créés à partir de ce modèle.

45.32. pg_proc

Le catalogue pg_proc stocke les informations concernant les fonctions (ou procédures). Voir CREATE FUNCTION(7) et Section 35.3, « Fonctions utilisateur » pour plus d'informations.

Cette table contient des données sur les fonctions d'agrégat et les fonctions simples. Si *proisagg* est vrai, il doit y avoir une ligne correspondante dans *pg_aggregate*.

Tableau 45.32. Colonnes de pg_proc

Nom	Type	Références	Description
proname	name		Nom de la fonction
pronamespace	oid	`.o` `pg_namespaceid`	OID du *namespace* auquel appartient la fonction
proowner	oid	`pg_authid.oid`	Propriétaire de la fonction
prolang	oid	`.oi` `pg_languaged`	Langage de codage ou interface d'appel de la fonction
procost	float4		Coût d'exécution estimé (en unité de cpu_operator_cost) ; si *proretset*, il s'agit d'un coût par ligne
prorows	float4		Nombre estimé de ligne de résultat (zéro si *proretset* est faux)
provariadic	oid	`pg_type.oid`	Type des données des éléments du tableau de paramètres variadic, ou zéro si la fonction n'a pas de paramètres variadiques.
proisagg	bool		Si vrai, la fonction est une fonction d'agrégat
proiswindow	bool		La fonction est une fonction window
prosecdef	bool		Si vrai, la fonction définit la sécurité (c'est une fonction « setuid »)
proisstrict	bool		Si vrai, la fonction retourne NULL si l'un de ses arguments est NULL. Dans ce cas, la fonction n'est en fait pas appelée du tout. Les fonctions qui ne sont pas « strictes » doivent traiter les paramètres NULL.
proretset	bool		Si vrai, la fonction retourne un ensemble (c'est-à-dire des valeurs multiples du type défini)
provolatile	char		Indique si le résultat de la fonction dépend uniquement de ses arguments ou s'il est affecté par des facteurs externes. Il vaut `i` pour les fonctions « immuables », qui, pour un jeu de paramètres identique en entrée, donnent toujours le même résultat. Il vaut `s` pour les fonctions « stables », dont le résultat (pour les mêmes paramètres en entrée) ne change pas au cours du parcours (de table). Il vaut `v` pour les fonctions « volatiles », dont le ré-

Nom	Type	Références	Description
			sultat peut varier à tout instant. (`v` est également utilisé pour les fonctions qui ont des effets de bord, afin que les appels à ces fonctions ne soient pas optimisés.)
pronargs	int2		Nombre d'arguments en entrée
pronargdefaults	int2		Nombre d'arguments qui ont des valeurs par défaut
prorettype	oid	`pg_type.oid`	Type de données renvoyé
proargtypes	oidvector	`pg_type.oid`	Un tableau contenant les types de données des arguments de la fonction. Ceci n'inclut que les arguments en entrée (dont les arguments `INOUT` et `VARIADIC`) et représente, du coup, la signature d'appel de la fonction.
proallargtypes	oid[]	`pg_type.oid`	Un tableau contenant les types de données des arguments de la fonction. Ceci inclut tous les arguments (y compris les arguments `OUT` et `INOUT`) ; néanmoins, si tous les arguments sont `IN`, ce champ est NULL. L'indice commence à 1 alors que, pour des raisons historiques, *proargtypes* commence à 0.
proargmodes	char[]		Un tableau contenant les modes des arguments de la fonction, codés avec `i` pour les arguments `IN`, `o` pour les arguments `OUT`, `b` pour les arguments `INOUT`, `v` pour les arguments `VARIADIC`, `t` pour les arguments `TABLE`. Si tous les arguments sont des arguments `IN`, ce champ est NULL. Les indices correspondent aux positions de *proallargtypes*, et non à celles de *proargtypes*.
proargnames	text[]		Un tableau contenant les noms des arguments de la fonction. Les arguments sans nom sont initialisés à des chaînes vides dans le tableau. Si aucun des arguments n'a de nom, ce champ est NULL. Les indices correspondent aux positions de *proallargtypes*, et non à celles de *proargtypes*.
proargdefaults	text		Arbres d'expression (en représentation `nodeToString()`) pour les valeurs par défaut. C'est une liste avec *pronargdefaults* éléments, correspondant aux *N* derniers arguments d'*entrée* (c'est-à-dire, les *N* dernières positions *proargtypes*). Si aucun des arguments n'a de valeur par défaut, ce champ vaudra null.
prosrc	text		Ce champ indique au gestionnaire de fonctions la façon d'invoquer la fonction. Il peut s'agir du code source pour un langage interprété, d'un symbole de lien, d'un nom de fichier ou de toute autre chose, en fonction du langage ou de la convention d'appel.
probin	text		Information supplémentaire sur la façon d'invoquer la fonction. Encore une fois, l'interprétation dépend du langage.
proconfig	text[]		Configuration locale à la fonction pour les variables configurables à l'exécution
proacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour plus de détails.

Pour les fonctions compilées, intégrées ou chargées dynamiquement, *prosrc* contient le nom de la fonction en langage C (symbole de lien). Pour tous les autres types de langages, *prosrc* contient le code source de la fonction. *probin* est inutilisé,

sauf pour les fonctions C chargées dynamiquement, pour lesquelles il donne le nom de fichier de la bibliothèque partagée qui contient la fonction.

45.33. pg_rewrite

Le catalogue pg_rewrite stocke les règles de réécriture pour les tables et les vues.

Tableau 45.33. Colonnes de pg_rewrite

Nom	Type	Références	Description
rulename	name		Nom de la règle
ev_class	oid	`pg_class.oid`	Table sur laquelle porte la règle
ev_attr	int2		Colonne sur laquelle porte la règle. Actuellement, cette colonne vaut toujours -1 pour indiquer qu'il s'agit de la table entière.
ev_type	char		Type d'évènement associé à la règle : 1 = **SELECT**, 2 = **UPDATE**, 3 = **INSERT**, 4 = **DELETE**
ev_enabled	char		Contrôle l'exécution de la règle suivant le mode session_replication_role. `O` = la règle se déclenche dans les modes « origin » et « local », `D` = la règle est désactivée, `R` = la règle s'exécute en mode « replica », `A` = la règle s'exécute à chaque fois.
is_instead	bool		Vrai s'il s'agit d'une règle `INSTEAD` (à la place de).
ev_qual	text		Arbre d'expression (sous la forme d'une représentation `nodeToString()`) pour la condition qualifiant la règle.
ev_action	text		Arbre de requête (sous la forme d'une représentation `nodeToString()`) pour l'action de la règle.

> **Note**
>
> `pg_class.relhasrules` doit être vrai si une table possède une règle dans ce catalogue.

45.34. pg_shdepend

Le catalogue pg_shdepend enregistre les relations de dépendance entre les objets de la base de données et les objets partagés, comme les rôles. Cette information permet à PostgreSQL™ de s'assurer que tous ces objets sont déréférencés avant toute tentative de suppression.

Voir aussi pg_depend, qui réalise une fonction similaire pour les dépendances impliquant les objets contenus dans une seule base de données.

Contrairement à la plupart des catalogues système, pg_shdepend est partagé par toutes les bases de données d'un cluster : il n'existe qu'une seule copie de pg_shdepend par cluster, pas une par base de données.

Tableau 45.34. Colonnes de pg_shdepend

Nom	Type	Références	Description
dbid	oid	`.oi` `pg_databased`	L'OID de la base de données dont fait partie l'objet dépendant. 0 pour un objet partagé
classid	oid	`pg_class.oid`	L'OID du catalogue système dont fait partie l'objet dépendant
objid	oid	toute colonne OID	L'OID de l'objet dépendant
objsubid	int4		Pour une colonne de table, c'est le numéro de colonne (les *objid* et *classid* font référence à la

Nom	Type	Références	Description
			table elle-même). Pour tous les autres types d'objets, cette colonne vaut zéro
refclassid	oid	pg_class.oid	L'OID du catalogue système dont fait partie l'objet référencé (doit être un catalogue partagé)
refobjid	oid	toute colonne OID	L'OID de l'objet référencé
deptype	char		Un code définissant les sémantiques spécifiques des relations de cette dépendance ; voir le texte.

Dans tous les cas, une entrée pg_shdepend indique que l'objet référencé ne peut pas être supprimé sans supprimer aussi l'objet dépendant. Néanmoins, il existe quelques différences identifiées par le *deptype* :

SHARED_DEPENDENCY_OWNER (o)
: L'objet référencé (qui doit être un rôle) est le propriétaire de l'objet dépendant.

SHARED_DEPENDENCY_ACL (a)
: L'objet référencé (qui doit être un rôle) est mentionné dans la liste de contrôle des accès (ACL, acronyme de *access control list*) de l'objet dépendant. (Une entrée SHARED_DEPENDENCY_ACL n'est pas créée pour le propriétaire de l'objet car ce dernier a toujours une entrée SHARED_DEPENDENCY_OWNER.)

SHARED_DEPENDENCY_PIN (p)
: Il n'existe pas d'objet dépendant ; ce type d'entrée est un signal indiquant que le système lui-même dépend de l'objet référencé et que, cet objet ne doit donc jamais être supprimé. Les entrées de ce type ne sont créées que par **initdb**. Les colonnes pour l'objet dépendant contiennent des zéros.

D'autres types de dépendances peuvent s'avérer nécessaires dans le futur. La définition actuelle ne supporte que les rôles comme objets référencés.

45.35. pg_shdescription

Le catalogue pg_shdescription stocke les descriptions optionelles (commentaires) des objets partagés de la base. Les descriptions peuvent être manipulées avec la commande COMMENT(7) et visualisées avec les commandes `\d` de psql.

Voir aussi pg_description, qui assure les mêmes fonctions, mais pour les objets d'une seule base.

Contrairement à la plupart des catalogues systèmes, pg_shdescription est partagée par toutes les bases d'un cluster : il n'existe qu'une seule copie de pg_shdescription par cluster, et non une par base.

Tableau 45.35. Colonnes de pg_shdescription

Nom	Type	Références	Description
objoid	oid	toute colonne OID	L'OID de l'objet concerné par la description
classoid	oid	pg_class.oid	L'OID du catalogue système où cet objet apparaît
description	text		Texte arbitraire servant de description de l'objet

45.36. pg_statistic

Le catalogue pg_statistic stocke des données statistiques sur le contenu de la base de données. Les entrées sont créées par ANALYZE(7), puis utilisées par le planificateur de requêtes. Les données statistiques sont, par définition des approximations, même si elles sont à jour.

D'habitude, il existe une entrée, avec *staall* = `false`, pour chaque colonne de table qui a été analysée. Si la table a des enfants, une seconde entrée avec *staall* = `true` est aussi créé. Cette ligne représente les statistiques de la colonne sur l'arbre d'héritage, autrement dit les statistiques pour les données que vous voyez avec `SELECT` *colonne* `FROM` *table**, alors que la ligne *staall* = `false` représente le résultat de `SELECT` *column* `FROM ONLY` *table*.

pg_statistic stocke aussi les données statistiques des valeurs des expressions d'index. Elles sont décrites comme si elles étaient de vraies colonnes ; en particulier, *starelid* référence l'index. Néanmoins, aucune entrée n'est effectuée pour une colonne d'index

ordinaire sans expression car cela est redondant avec l'entrée correspondant à la colonne sous-jacente de la table. Actuellement, les entrées pour les expressions d'index ont toujours *staall* = false.

Comme des statistiques différentes peuvent être appropriées pour des types de données différents, pg_statistic ne fait qu'un minimum de suppositions sur les types de statistiques qu'il stocke. Seules des statistiques extrêmement générales (comme les valeurs NULL) ont des colonnes dédiées. Tout le reste est stocké dans des « connecteurs », groupes de colonnes associées dont le contenu est identifié par un numéro de code dans l'une des colonnes du connecteur. Pour plus d'information, voir src/include/catalog/pg_statistic.h.

pg_statistic ne doit pas être lisible par le public, car même les données statistiques sont sensibles. (Exemple : les valeurs maximales et minimales d'une colonne de salaire peuvent être intéressantes). pg_stats est une vue sur pg_statistic accessible à tous, qui n'expose que les informations sur les tables accessibles à l'utilisateur courant.

Tableau 45.36. Colonnes de pg_statistic

Nom	Type	Références	Description
starelid	oid	pg_class.oid	Table ou index à qui la colonne décrite appartient
staattnum	int2	.a tt nu pg_attributem	Numéro de la colonne décrite
staall	bool		Si vrai, les statistiques incluent les colonnes enfants de l'héritage, pas uniquement les valeurs de la relation spécifiée
stanullfrac	float4		Fraction des entrées de la colonne qui ont une valeur NULL
stawidth	int4		Taille moyenne, en octets, des entrées non NULL
stadistinct	float4		Nombre de valeurs distinctes non NULL dans la colonne. Une valeurs positive est le nombre réel de valeurs distinctes. Une valeur négative est le négatif d'un multiplieur pour le nombre de lignes dans la table ; par exemple, une colonne dans laquelle les valeurs apparaissent environ deux fois en moyenne peut être représentée par *stadistinct* = -0.5. 0 indique que le nombre de valeurs distinctes est inconnu.
stakindN	int2		Numéro de code indiquant le type de statistiques stockées dans « le connecteur » numéro *N* de la ligne de pg_statistic.
staopN	oid	.oi pg_operatord	Opérateur utilisé pour dériver les statistiques stockées dans « le connecteur » numéro *N*. Par exemple, un connecteur d'histogramme montre l'opérateur <, qui définit l'ordre de tri des données.
stanumbersN	float4[]		Statistiques numériques du type approprié pour « le connecteur » numéro *N* ou NULL si le type de connecteur n'implique pas de valeurs numériques.
stavaluesN	anyarray		Valeurs de données de la colonne du type approprié pour « le connecteur » numéro *N* ou NULL si le type de connecteur ne stocke aucune valeur de données. Chaque valeur d'élément du tableau est en fait du type de données de la colonne indiquée, si bien qu'il n'y a aucun moyen de définir le type de ces colonnes plus précisément qu'avec le type anyarray (tableau quelconque).

45.37. pg_tablespace

Le catalogue pg_tablespace enregistre les informations des *tablespaces* disponibles. Les tables peuvent être placées dans des *tablespaces* particuliers pour faciliter l'administration des espaces de stockage.

Contrairement à la plupart des catalogues système, pg_tablespace est partagée par toutes les bases de données du cluster : il n'y a donc qu'une copie de pg_tablespace par cluster, et non une par base.

Tableau 45.37. Colonnes de pg_tablespace

Nom	Type	Références	Description
spcname	name		Nom du *tablespace*
spcowner	oid	pg_authid.oid	Propriétaire du *tablespace*, habituellement l'utilisateur qui l'a créé
spclocation	text		Emplacement (chemin vers le répertoire) du *tablespace*
spcacl	aclitem[]		Droits d'accès ; voir GRANT(7) et REVOKE(7) pour les détails.
spcoptions	text[]		Options au niveau tablespace, sous la forme de chaînes « motclé=valeur »

45.38. pg_trigger

Le catalogue pg_trigger stocke les informations concernant les déclencheurs des tables. Voir la commande CREATE TRIGGER(7) pour plus d'informations.

Tableau 45.38. Colonnes de pg_trigger

Nom	Type	Références	Description
tgrelid	oid	pg_class.oid	Table sur laquelle porte le déclencheur
tgname	name		Nom du déclencheur (doit être unique parmi les déclencheurs d'une table)
tgfoid	oid	pg_proc.oid	Fonction à appeler
tgtype	int2		Masque de bits identifiant les conditions du déclencheur
tgenabled	char		Contrôle l'exécution du trigger suivant le mode session_replication_role. O = le trigger se déclenche dans les modes « origin » et « local », D = le trigger est désactivé, R = le trigger s'exécute en mode « replica », A = le trigger s'exécute à chaque fois.
tgisinternal	bool		Vrai si le trigger est généré en interne (habituellement pour forcer la contrainte identifiée par *tgconstraint*)
tgconstrrelid	oid	pg_class.oid	La table référencée par une contrainte d'intégrité référentielle
tgconstrindid	oid	pg_class.oid	L'index supportant une contrainte unique, clé primaire ou d'intégrité référentielle
tgconstraint	oid	. o i pg_constraintd	L'entrée pg_constraint associé au trigger, si elle existe
tgdeferrable	bool		Vrai si le déclencheur contrainte est retardable
tginitdeferred	bool		Vrai si le déclencheur de contrainte est initialement retardé
tgnargs	int2		Nombre de chaînes d'arguments passées à la fonction du déclencheur
tgattr	int2vector	.a tt	numéros de colonne, si le trigger est spécifique à la colonne ; sinon un tableau vide

Nom	Type	Références	Description
		num	
tgargs	bytea		Chaînes d'arguments à passer au déclencheur, chacune terminée par un NULL
tgqual	text		Arbre d'expression (d'après la représentation de nodeToString() pour la condition WHEN du trigger, ou NULL si aucune

Actuellement, les triggers spécifiques par colonne sont supportés seulement pour les événements UPDATE et, du coup, *tgattr* est valable seulement pour ce type d'événements. *tgtype* pourrait contenir des informations pour d'autres types d'événement mais ils sont supposés valides pour la table complète, quel que soit le contenu de *tgattr*.

> **Note**
>
> Quand *tgconstraint* est différent de zéro, *tgconstrrelid*, *tgconstrindid*, *tgdeferrable* et *tginitdeferred* sont grandement redondants avec l'entrée pg_constraint référencée. Néanmoins, il est possible qu'un trigger non déferrable soit associé à une contrainte déferrable : les contraintes de clé étrangère peuvent avoir quelques triggers déferrables et quelques triggers non déferrables.

> **Note**
>
> pg_class.relhastriggers doit valoir true si la table possède au moins un trigger dans ce catalogue.

45.39. pg_ts_config

Le catalogue pg_ts_config contient des entrées représentant les configurations de la recherche plein texte. Une configuration spécifie un analyseur et une liste de dictionnaires à utiliser pour chacun des types d'éléments en sortie de l'analyseur. L'analyseur est présenté dans l'entrée de pg_ts_config mais la correspondance élément/dictionnaire est définie par des entrées supplémentaires dans pg_ts_config_map.

Les fonctionnalités de recherche plein texte de PostgreSQL™ sont expliquées en détail dans Chapitre 12, Recherche plein texte.

Tableau 45.39. Colonnes de pg_ts_config

Nom	Type	Références	Description
cfgname	name		Nom de la configuration
cfgnamespace	oid	.o pg_namespaceid	OID du *namespace* qui contient la configuration
cfgowner	oid	pg_authid.oid	Propriétaire de la configuration
cfgparser	oid	.o pg_ts_parserid	OID de l'analyseur pour la configuration

45.40. pg_ts_config_map

Le catalogue pg_ts_config_map contient des entrées présentant les dictionnaires de recherche plein texte à consulter et l'ordre de consultation, pour chaque type de lexème en sortie de chaque analyseur de configuration.

Les fonctionnalités de la recherche plein texte de PostgreSQL™ sont expliquées en détail dans Chapitre 12, Recherche plein texte.

Tableau 45.40. Colonnes de pg_ts_config_map

Nom	Type	Références	Description
mapcfg	oid	.o pg_ts_configid	OID de l'entrée pg_ts_config qui possède l'entrée
maptokentype	integer		Un type de lexème émis par l'analyseur de configuration

Nom	Type	Références	Description
mapseqno	integer		Ordre dans lequel consulter l'entrée (les plus petits *mapseqno* en premier)
mapdict	oid	pg_ts_dict.oid	OID du dictionnaire de recherche plein texte à consulter

45.41. pg_ts_dict

Le catalogue pg_ts_dict contient des entrées définissant les dictionnaires de recherche plein texte. Un dictionnaire dépend d'un modèle de recherche plein texte qui spécifie toutes les fonctions d'implantation nécessaires ; le dictionnaire lui-même fournit des valeurs pour les paramètres utilisateur supportés par le modèle. Cette division du travail permet la création de dictionnaires par des utilisateurs non privilégiés. Les paramètres sont indiqués par une chaîne, *dictinitoption*, dont le format et la signification dépendent du modèle.

Les fonctionnalités de la recherche plein texte de PostgreSQL™ sont expliquées en détail dans Chapitre 12, Recherche plein texte.

Tableau 45.41. Colonnes de pg_ts_dict

Nom	Type	Références	Description
dictname	name		Nom du dictionnaire de recherche plein texte
dictnamespace	oid	pg_namespace.oid	OID du *namespace* contenant le dictionnaire
dictowner	oid	pg_authid.oid	Propriétaire du dictionnaire
dicttemplate	oid	pg_ts_template.oid	OID du modèle de recherche plein texte du dictionnaire
dictinitoption	text		Chaîne d'options d'initialisation du modèle

45.42. pg_ts_parser

Le catalogue pg_ts_parser contient des entrées définissant les analyseurs de la recherche plein texte. Un analyseur est responsable du découpage du texte en entrée en lexèmes et de l'assignation d'un type d'élément à chaque lexème. Puisqu'un analyseur doit être codé à l'aide de fonctions écrites en langage C, la création de nouveaux analyseurs est restreinte aux superutilisateurs des bases de données.

Les fonctionnalités de la recherche plein texte de PostgreSQL™ sont expliquées en détail dans Chapitre 12, Recherche plein texte.

Tableau 45.42. Colonnes de pg_ts_parser

Nom	Type	Références	Description
prsname	name		Nom de l'analyseur de recherche plein texte
prsnamespace	oid	.o pg_namespaceid	OID du *namespace* qui contient l'analyseur
prsstart	regproc	pg_proc.oid	OID de la fonction de démarrage de l'analyseur
prstoken	regproc	pg_proc.oid	OID de la fonction next-token de l'analyseur
prsend	regproc	pg_proc.oid	OID de la fonction d'arrêt de l'analyseur
prsheadline	regproc	pg_proc.oid	OID de la fonction headline de l'analyseur
prslextype	regproc	pg_proc.oid	OID de la fonction lextype de l'analyseur

45.43. pg_ts_template

Le catalogue pg_ts_template contient des entrées définissant les modèles de recherche plein texte. Un modèle est le squelette d'implantation d'une classe de dictionnaires de recherche plein texte. Puisqu'un modèle doit être codé à l'aide de fonctions codées en langage C, la création de nouveaux modèles est restreinte aux superutilisateurs des bases de données.

Les fonctionnalités de la recherche plein texte de PostgreSQL™ sont expliquées en détail dans Chapitre 12, Recherche plein texte.

Tableau 45.43. Colonnes de pg_ts_template

Nom	Type	Références	Description
tmplname	name		Nom du modèle de recherche plein texte
tmplnamespace	oid	.o pg_namespaceid	OID du *namespace* qui contient le modèle
tmplinit	regproc	pg_proc.oid	OID de la fonction d'initialisation du modèle
tmpllexize	regproc	pg_proc.oid	OID de la fonction lexize du modèle

45.44. pg_type

Le catalogue pg_type stocke les informations concernant les types de données. Les types basiques et d'énumération (types scalaires) sont créés avec la commande CREATE TYPE(7) et les domaines avec CREATE DOMAIN(7). Un type composite est créé automatiquement pour chaque table de la base pour représenter la structure des lignes de la table. Il est aussi possible de créer des types composites avec **CREATE TYPE AS**.

Tableau 45.44. Colonnes de pg_type

Nom	Type	Références	Description
typname	name		Nom du type
typnamespace	oid	.o pg_namespaceid	OID du *namespace* qui contient le type
typowner	oid	pg_authid.oid	Propriétaire du type
typlen	int2		Pour les types de taille fixe, *typlen* est le nombre d'octets de la représentation interne du type. Mais pour les types de longueur variable, *typlen* est négatif. -1 indique un type « varlena » (qui a un attribut de longueur), -2 indique une chaîne C terminée par le caractère NULL.
typbyval	bool		*typbyval* détermine si les routines internes passent une valeur de ce type par valeur ou par référence. *typbyval* doit être faux si *typlen* ne vaut pas 1, 2 ou 4 (ou 8 sur les machines dont le mot-machine est de 8 octets). Les types de longueur variable sont toujours passés par référence. *typbyval* peut être faux même si la longueur permet un passage par valeur.
typtype	char		*typtype* vaut b pour un type de base, c pour un type composite (le type d'une ligne de table, par exemple), d pour un domaine, e pour un enum ou p pour un pseudo-type. Voir aussi *typrelid* et *typbasetype*.
typcategory	char		*typcategory* est une classification arbitraire de types de données qui est utilisée par l'analyseur pour déterminer la conversion implicite devant être « préférée ». Voir Tableau 45.45, « Codes *typcategory* »
typispreferred	bool		Vrai si ce type est une cible de conversion préférée dans sa *typcategory*
typisdefined	bool		Vrai si le type est défini et faux s'il ne s'agit que d'un conteneur pour un type qui n'est pas encore défini. Lorsque *typisdefined* est faux, rien, à part le nom du type, le *namespace* et l'OID, n'est fiable.
typdelim	char		Caractère qui sépare deux valeurs de ce type

Nom	Type	Références	Description
			lorsque le programme lit les valeurs d'un tableau en entrée. Le délimiteur est associé au type d'élément du tableau, pas au type tableau.
typrelid	oid	pg_class.oid	S'il s'agit d'un type composite (voir *typtype*), alors cette colonne pointe vers la ligne de pg_class qui définit la table correspondante. Pour un type composite sans table, l'entrée dans pg_class ne représente pas vraiment une table, mais elle est néanmoins nécessaire pour trouver les lignes de pg_attribute liées au type. 0 pour les types autres que composites.
typelem	oid	pg_type.oid	Si *typelem* est différent de zéro, alors il identifie une autre ligne de pg_type. Le type courant peut alors être utilisé comme un tableau contenant des valeurs de type *typelem*. Un « vrai » type tableau a une longueur variable (*typlen* = -1), mais certains types de longueur fixe (*typlen* > 0) ont aussi un *typelem* non nul, par exemple name et point. Si un type de longueur fixe a un *typelem*, alors sa représentation interne est composée d'un certain nombre de valeurs du type *typelem*, sans autre donnée. Les types de données tableau de taille variable ont un en-tête défini par les sous-routines de tableau.
typarray	oid	pg_type.oid	Si *typarray* est différent de zéro, alors il identifie une autre ligne dans pg_type, qui est le type tableau « true » disposant de ce type en élément.
typinput	regproc	pg_proc.oid	Fonction de conversion en entrée (format texte)
typoutput	regproc	pg_proc.oid	Fonction de conversion en sortie (format texte)
typreceive	regproc	pg_proc.oid	Fonction de conversion en entrée (format binaire), ou 0 s'il n'y en a pas
typsend	regproc	pg_proc.oid	Fonction de conversion en sortie (format binaire), ou 0 s'il n'y en a pas
typmodin	regproc	pg_proc.oid	Fonction en entrée de modification du type ou 0 si le type ne supporte pas les modificateurs
typmodout	regproc	pg_proc.oid	Fonction en sortie de modification du type ou 0 pour utiliser le format standard
typanalyze	regproc	pg_proc.oid	Fonction **ANALYZE** personnalisée ou 0 pour utiliser la fonction standard
typalign	char		*typalign* est l'alignement requis pour stocker une valeur de ce type. Cela s'applique au stockage sur disque ainsi qu'à la plupart des représentations de cette valeur dans PostgreSQL™. Lorsque des valeurs multiples sont stockées consécutivement, comme dans la représentation d'une ligne complète sur disque, un remplissage est inséré avant la donnée de ce type pour qu'elle commence à l'alignement indiqué. La référence de l'alignement est le début de la première donnée de la séquence. Les valeurs possibles sont : • c = alignement char, aucun alignement n'est nécessaire ; • s = alignement short (deux octets sur la plupart des machines) ;

Nom	Type	Références	Description
			• `i` = alignement int (quatre octets sur la plupart des machines) ; • `d` = alignement double (huit octets sur la plupart des machines, mais pas sur toutes). **Note** Pour les types utilisés dans les tables systèmes il est indispensable que les tailles et alignements définis dans pg_type soient en accord avec la façon dont le compilateur dispose la colonne dans une structure représentant une ligne de table.
typstorage	char		*typstorage* indique, pour les types varlena (ceux pour lesquels *typlen* = -1), si le type accepte le TOASTage et la stratégie par défaut à utiliser pour les attributs de ce type. Les valeurs possibles sont : • `p` : la valeur doit être stockée normalement ; • `e` : la valeur peut être stockée dans une relation « secondaire » (si la relation en a une, voir `pg_class.reltoastrelid`) ; • `m` : la valeur peut être stockée compressée sur place ; • `x` : la valeur peut être stockée compressée sur place ou stockée dans une relation « secondaire ». Les colonnes `m` peuvent aussi être déplacées dans une table de stockage secondaire, mais seulement en dernier recours (les colonnes `e` et `x` sont déplacées les premières).
typnotnull	bool		Représente une contrainte non NULL pour le type. Ceci n'est utilisé que pour les domaines.
typbasetype	oid	`pg_type.oid`	S'il s'agit d'un domaine (voir *typtype*), alors *typbasetype* identifie le type sur lequel celui-ci est fondé. 0 s'il ne s'agit pas d'un domaine.
typtypmod	int4		Les domaines utilisent ce champ pour enregistrer le `typmod` à appliquer à leur type de base (-1 si le type de base n'utilise pas de `typmod`). -1 si ce type n'est pas un domaine.
typndims	int4		Le nombre de dimensions de tableau pour un domaine tableau (c'est-à-dire dont *typbasetype* est un type tableau ; le *typelem* du domaine correspond à celui du type de base). 0 pour les types autres que les domaines sur des types tableaux.
typdefaultbin	text		Si *typdefaultbin* n'est pas NULL, ce champ est la représentation `nodeToString()` d'une expression par défaut pour le type. Ceci n'est utilisé que pour les domaines.

Nom	Type	Références	Description
typdefault	text		NULL si le type n'a pas de valeur par défaut associée. Si *typdefaultbin* est non NULL, ce champ doit contenir une version lisible de l'expression par défaut représentée par *typdefaultbin*. Si *typdefaultbin* est NULL et si ce champ ne l'est pas, alors il stocke la représentation externe de la valeur par défaut du type, qui peut être passée à la fonction de conversion en entrée du type pour produire une constante.

Tableau 45.45, « Codes *typcategory* » liste les valeurs de *typcategory* définies par le système. Tout ajout futur à la liste sera aussi une lettre ASCII majuscule. Tous les autres caractères ASCII sont réservés pour les catégories définies par l'utilisateur.

Tableau 45.45. Codes *typcategory*

Code	Catégorie
A	Types tableaux
B	Types booléens
C	Types composites
D	Types date/time
E	Types enum
G	Types géometriques
I	Types adresses réseau
N	Types numériques
P	Pseudo-types
S	Types chaînes
T	Types 'Timespan' (étendue de temps, intervalle)
U	Types définis par l'utilisateur
V	Types Bit-string
X	Type unknown

45.45. pg_user_mapping

Le catalogue pg_user_mapping stocke les correspondances entre utilisateurs locaux et distants. L'accès à ce catalogue est interdite aux utilisateurs normaux, utilisez la vue pg_user_mappings à la place.

Tableau 45.46. Colonnes de pg_user_mapping

Nom	Type	Référence	Description
umuser	oid	pg_authid.oid	OID du rôle à faire correspondre, 0 si l'utilisateur à correspondre est public.
umserver	oid	pg_foreign_server.oid	L'OID du serveur distant qui contient cette correspondance
umoptions	text[]		Options spécifiques à la correspondance d'utilisateurs, sous forme de chaînes « motclé=valeur ».

45.46. Vues système

En plus des catalogues système, PostgreSQL™ fournit un certain nombre de vues internes. Certaines fournissent un moyen simple d'accéder à des requêtes habituellement utilisées dans les catalogues systèmes. D'autres vues donnent accès à l'état interne du serveur.

Le schéma d'information (Chapitre 34, Schéma d'information) fournit un autre ensemble de vues qui recouvrent les fonctionnalités des vues système. Comme le schéma d'information fait parti du standard SQL, alors que les vues décrites ici sont spécifiques à PostgreSQL™, il est généralement préférable d'utiliser le schéma d'information si celui-ci apporte toutes les informations nécessaires.

Tableau 45.47, « Vues système » liste les vues systèmes décrites plus en détails dans la suite du document. Il existe de plus des vues permettant d'accéder aux résultats du collecteur de statistiques elles sont décrites dans le Tableau 27.1, « Vues statistiques standards ».

Sauf lorsque c'est indiqué, toutes les vues décrites ici sont en lecture seule.

Tableau 45.47. Vues système

Nom de la vue	But
pg_cursors	curseurs ouverts
pg_group	groupe d'utilisateurs de la base de données
pg_indexes	index
pg_locks	verrous posés au moment de la consultation
pg_prepared_statements	instructions préparées
pg_prepared_xacts	transactions préparées
pg_roles	rôles des bases de données
pg_rules	règles
pg_settings	configuration
pg_shadow	utilisateurs des bases de données
pg_stats	statistiques du planificateur
pg_tables	tables
pg_timezone_abbrevs	abréviations des fuseaux horaires
pg_timezone_names	noms des fuseaux horaires
pg_user	utilisateurs des bases de données
pg_user_mappings	user mappings
pg_views	vues

45.47. pg_cursors

La vue pg_cursors liste les curseurs actuellement disponibles. Les curseurs peuvent être définis de plusieurs façons :

- via l'instruction SQL DECLARE(7) ;
- via le message Bind du protocole frontend/backend, décrit dans le Section 46.2.3, « Requête étendue » ;
- via l'interface de programmation du serveur (SPI), décrite dans le Section 43.1, « Fonctions d'interface ».

La vue pg_cursors affiche les curseurs créés par tout moyen précédent. Les curseurs n'existent que pour la durée de la transaction qui les définit, sauf s'ils ont été déclarés avec `WITH HOLD`. De ce fait, les curseurs volatils (*non-holdable*) ne sont présents dans la vue que jusqu'à la fin de la transaction qui les a créés.

Note

Les curseurs sont utilisés en interne pour coder certains composants de PostgreSQL™, comme les langages procéduraux. La vue pg_cursors peut ainsi inclure des curseurs qui n'ont pas été créés explicitement par l'utilisateur.

Tableau 45.48. Colonnes de pg_cursors

Nom	Type	Description
name	text	Le nom du curseur
statement	text	La chaîne utilisée comme requête pour créer le curseur
is_holdable	boolean	`true` si le curseur est persistant (*holdable*) (c'est-à-dire s'il peut être accédé après la validation de la transaction qui l'a déclaré) ; `false` sinon
is_binary	boolean	`true` si le curseur a été déclaré binaire (`BINARY`) ; `false` sinon
is_scrollable	boolean	`true` si le curseur autorise une récupération non séquentielle des lignes ; `false` sinon
creation_time	timestamptz	L'heure à laquelle le curseur a été déclaré

La vue pg_cursors est en lecture seule.

45.48. pg_group

La vue pg_group existe pour des raisons de compatibilité ascendante : elle émule un catalogue qui a existé avant la version 8.1 de PostgreSQL™. Elle affiche les noms et membres de tous les rôles dont l'attribut *rolcanlogin* est dévalidé, ce qui est une approximation des rôles utilisés comme groupes.

Tableau 45.49. Colonnes de pg_group

Nom	Type	Références	Description
groname	name	`.roln pg_authidame`	Nom du groupe
grosysid	oid	`pg_authid.oid`	Identifiant du groupe
grolist	oid[]	`pg_authid.oid`	Un tableau contenant les identifiants des rôles du groupe

45.49. pg_indexes

La vue pg_indexes fournit un accès aux informations utiles sur chaque index de la base de données.

Tableau 45.50. Colonnes de pg_indexes

Nom	Type	Références	Description
schemaname	name	`pg_namespace.nspname`	Nom du schéma contenant les tables et index
tablename	name	`pg_class.relname`	Nom de la table portant l'index
indexname	name	`pg_class.relname`	Nom de l'index
tablespace	name	`pg_tablespace.spcname`	Nom du *tablespace* contenant l'index (NULL s'il s'agit de celui par défaut pour la base de données)
indexdef	text		Définition de l'index (une commande **CREATE INDEX** reconstruite)

45.50. pg_locks

La vue pg_locks fournit un accès aux informations concernant les verrous détenus par les transactions ouvertes sur le serveur de bases de données. Voir le Chapitre 13, Contrôle d'accès simultané pour une discussion plus importante sur les verrous.

pg_locks contient une ligne par objet verrouillable actif, type de verrou demandé et transaction associée. Un même objet verrouillable peut apparaître plusieurs fois si plusieurs transactions ont posé ou attendent des verrous sur celui-ci. Toutefois, un objet

qui n'est pas actuellement verrouillé n'apparaît pas.

Il existe plusieurs types distincts d'objets verrouillables : les relations complètes (tables, par exemple), les pages individuelles de relations, des tuples individuels de relations, les identifiants de transaction (virtuels et permanents) et les objets généraux de la base de données (identifiés par l'OID de la classe et l'OID de l'objet, de la même façon que dans pg_description ou pg_depend). De plus, le droit d'étendre une relation est représenté comme un objet verrouillable distinct.

Tableau 45.51. Colonnes pg_locks

Nom	Type	Références	Description
locktype	text		Type de l'objet verrouillable : `relation`, `extend`, `page`, `tuple`, `transactionid`, `virtualxid`, `object`, `userlock` ou `advisory`
database	oid	`.oi` `pg_databased`	L'OID de la base de données dans laquelle existe l'objet, 0 si l'objet est partagé ou NULL si l'objet est un identifiant de transaction
relation	oid	`pg_class.oid`	L'OID de la relation ou NULL si l'objet n'est pas une relation ni une partie de relation
page	integer		Le numéro de page à l'intérieur de cette relation ou NULL si l'objet n'est pas un tuple ou une page de relation
tuple	smallint		Le numéro du tuple dans la page ou NULL si l'objet n'est pas un tuple
virtualxid	text		L'identifiant virtuel d'une transaction, ou NULL si l'objet n'est pas un identifiant virtuel de transaction
transactionid	xid		L'identifiant d'une transaction ou NULL si l'objet n'est pas un identifiant de transaction
classid	oid	`pg_class.oid`	L'OID du catalogue système contenant l'objet ou NULL si l'objet n'est pas un objet général de la base de données
objid	oid	n'importe quelle colonne OID	L'OID de l'objet dans son catalogue système ou NULL si l'objet n'est pas un objet général de la base de données.
objsubid	smallint		Pour une colonne de table, c'est le numéro de la colonne (*classid* et *objid* font référence à la table elle-même). Pour tous les autres types d'objets, cette colonne vaut 0. NULL si l'objet n'est pas un objet général de la base de données.
virtualtransaction	text		L'identifiant virtuel de la transaction qui détient ou attend le verrou.
pid	integer		L'identifiant du processus serveur qui détient ou attend le verrou. NULL si le verrou est possédé par une transaction préparée.
mode	text		Nom du type de verrou détenu ou attendu par ce processus (voir la Section 13.3.1, « Verrous de niveau table »)
granted	boolean		`True` si le verrou est détenu, `false` s'il est attendu

granted est `true` sur une ligne représentant un verrou tenu par la transaction indiquée. Une valeur `false` indique que cette transaction attend l'acquisition du verrou, ce qui implique qu'une autre transaction a choisi un mode de verrouillage conflictuel sur le même objet verrouillable. La transaction en attente dort jusqu'au relâchement du verrou (ou jusqu'à ce qu'une situation de blocage soit détectée). Une transaction unique peut attendre l'acquisition d'au plus un verrou à la fois.

Chaque transaction détient un verrou exclusif sur son identifiant virtuel de transaction pour toute sa durée. Si un identifiant permanent est affecté à la transaction (ce qui arrive habituellement si la transaction change l'état de la base de données), il détient aussi un verrou exclusif sur son identifiant de transaction permanent jusqu'à sa fin. Quand une transaction trouve nécessaire d'attendre

spécifiquement une autre transaction, elle le fait en essayant d'acquérir un verrou partagé sur l'identifiant de l'autre transaction (identifiant virtuel ou permanent selon la situation). Ceci n'est couronné de succès que lorsque l'autre transaction termine et relâche son verrou.

Bien que les lignes constituent un type d'objet verrouillable, les informations sur les verrous de niveau ligne sont stockées sur disque, et non en mémoire. Ainsi, les verrous de niveau ligne n'apparaissent normalement pas dans cette vue. Si une transaction attend un verrou de niveau ligne, elle apparaît sur la vue comme en attente de l'identifiant permanent de la transaction actuellement détentrice de ce verrou de niveau ligne.

Les verrous consultatifs peuvent être acquis par des clés constituées soit d'une seule valeur bigint, soit de deux valeurs integer. Une clé bigint est affichée avec sa moitié haute dans la colonne *classid*, sa partie basse dans la colonne *objid* et *objsubid* à 1. Les clés integer sont affichées avec la première clé dans la colonne *classid*, la deuxième clé dans la colonne *objid* et *objsubid* à 2. La signification réelle des clés est laissée à l'utilisateur. Les verrous consultatifs sont locaux à chaque base, la colonne *database* a donc un sens dans ce cas.

Lorsque la vue pg_locks est accédée, les structures de données du gestionnaire interne de verrous sont momentanément verrouillées et une copie est faite pour que la vue s'affiche. Ceci assure que la vue produit un ensemble consistant de résultats, tout en ne bloquant pas les opérations habituelles du gestionnaire de verrous plus longuement que nécessaire. Toutefois, si la vue est accédée trop fréquemment, cela peut avoir un impact sur les performances de la base de données.

pg_locks fournit une vue globale de tous les verrous du cluster, pas seulement de ceux de la base en cours d'utilisation. Bien que la colonne *relation* puisse être jointe avec pg_class.*oid* pour identifier les relations verrouillées, ceci ne fonctionne correctement qu'avec les relations de la base accédée (celles pour lesquelles la colonne *database* est l'OID de la base actuelle ou 0).

La colonne *pid* peut être jointe à la colonne *procpid* de la vue pg_stat_activity pour obtenir plus d'informations sur la session qui détient ou attend un verrou, par exemple :

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.procpid;
```

. De plus, si des transactions préparées sont utilisées, la colonne *virtualtransaction* peut être jointe à la colonne *transaction* de la vue pg_prepared_xacts pour obtenir plus d'informations sur les transactions préparées qui détiennent des verrous. (Une transaction préparée ne peut jamais être en attente d'un verrou mais elle continue à détenir les verrous qu'elle a acquis pendant son exécution.) Par exemple :

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

45.51. pg_prepared_statements

La vue pg_prepared_statements affiche toutes les instructions préparées disponibles pour la session en cours. Voir PREPARE(7) pour de plus amples informations sur les instructions préparées.

pg_prepared_statements contient une ligne pour chaque instruction préparée. Les lignes sont ajoutées à la vue quand une nouvelle instruction préparée est créée et supprimée quand une instruction préparée est abandonnée (par exemple, via la commande DEALLOCATE(7)).

Tableau 45.52. Colonnes de pg_prepared_statements

Nom	Type	Description	
name	text	L'identifiant de l'instruction préparée	
statement	text	La requête soumise par le client pour créer cette instruction préparée. Pour les instructions préparées créées en SQL, c'est l'instruction **PREPARE** soumise par le client. Pour les instructions préparées créées via le protocole frontend/backend, c'est le texte de l'intruction préparée elle-même.	
prepare_time	timestamptz	L'heure de création de l'instruction préparée	

Nom	Type	Description	
parameter_types	regtype[]	Les types des paramètres attendus par l'instruction préparée sous la forme d'un tableau de regtype. L'OID correspondant à un élément de ce tableau peut être obtenu en convertissant la valeur regtype en oid.	
from_sql	boolean	`true` si l'instruction préparée a été créée via l'instruction SQL **PREPARE** ; `false` si l'instruction a été préparée via le protocole frontend/backend	

La vue pg_prepared_statements est en lecture seule.

45.52. pg_prepared_xacts

La vue pg_prepared_xacts affiche les informations concernant les transactions actuellement préparées pour une validation en deux phases (voir PREPARE TRANSACTION(7) pour les détails).

pg_prepared_xacts contient une ligne par transaction préparée. L'entrée est supprimée quand la transaction est validée ou annulée.

Tableau 45.53. Colonnes de pg_prepared_xacts

Nom	Type	Références	Description
transaction	xid		L'identifiant numérique de la transaction préparée
gid	text		L'identifiant global de transaction assigné à la transaction
prepared	timestamp with time zone		L'heure de préparation de la transaction pour validation
owner	name	`pg_authid.rolname`	Le nom de l'utilisateur qui a exécuté la transaction
database	name	`pg_database.datname`	Nom de la base de données dans laquelle a été exécutée la transaction

Lors d'un accès à la vue pg_prepared_xacts, les structures de données du gestionnaire interne des transactions sont momentanément verrouillées et une copie de la vue est faite pour affichage. Ceci assure que la vue produit un ensemble cohérent de résultats tout en ne bloquant pas les opérations normales plus longtemps que nécessaire. Néanmoins, si la vue est accédée fréquemment, les performances de la base de données peuvent être impactées.

45.53. pg_roles

La vue pg_roles fournit un accès aux informations des rôles de base de données. C'est tout simplement une vue accessible de pg_authid qui n'affiche pas le champ du mot de passe.

Cette vue expose explicitement la colonne OID de la table sous-jacente car elle est nécessaire pour réaliser des jointures avec les autres catalogues.

Tableau 45.54. Colonnes de pg_roles

Nom	Type	Références	Description
rolname	name		Nom du rôle
rolsuper	bool		Le rôle est un superutilisateur
rolall	bool		Le rôle hérite automatiquement des droits des rôles dont il est membre
rolcreaterole	bool		Le rôle peut créer d'autres rôles
rolcreatedb	bool		Le rôle peut créer des bases de données
rolcatupdate	bool		Le rôle peut mettre à jour explicitement les catalogues

Nom	Type	Références	Description
			système. (Même un superutilisateur ne peut pas le faire si cette colonne n'est pas positionnée à `true`.)
rolcanlogin	bool		Le rôle peut se connecter, c'est-à-dire que ce rôle peut être indiqué comme identifiant initial d'autorisation de session.
rolconnlimit	int4		Pour les rôles autorisés à se connecter, ceci indique le nombre maximum de connexions concurrentes autorisées par rôle. -1 signifie qu'il n'y a pas de limite.
rolpassword	text		Ce n'est pas le mot de passe (toujours `********`)
rolvaliduntil	timestamptz		Estampille temporelle d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe) ; NULL s'il est indéfiniment valable
rolconfig	text[]		Configuration spécifique du rôle
oid	oid	`pg_authid.oid`	Identifiant du rôle

45.54. pg_rules

La vue pg_rules fournit un accès à des informations utiles sur les règles de réécriture des requêtes.

Tableau 45.55. Colonnes de pg_rules

Nom	Type	Références	Description
schemaname	name	`pg_namespace.nspname`	Nom du schéma contenant la table
tablename	name	`pg_class.relname`	Nom de la table pour laquelle est créée la règle
rulename	name	`pg_rewrite.rulename`	Nom de la règle
definition	text		Définition de la règle (une commande de création reconstruite)

La vue pg_rules exclut les règles `ON SELECT` des vues ; elles sont accessibles dans pg_views.

45.55. pg_settings

La vue pg_settings fournit un accès aux paramètres d'exécution du serveur. C'est essentiellement une interface alternative aux commandes SHOW(7) et SET(7). Elle fournit aussi un accès à certaines informations des paramètres qui ne sont pas directement accessibles avec **SHOW**, telles que les valeurs minimales et maximales.

Tableau 45.56. Colonnes de pg_settings

Nom	Type	Description
name	text	Nom du paramètre d'exécution
setting	text	Valeur actuelle du paramètre
unit	text	Unité implicite du paramètre
category	text	Groupe logique du paramètre
short_desc	text	Description brève du paramètre
extra_desc	text	Information supplémentaire, plus détaillée, sur le paramètre
context	text	Contexte requis pour positionner la valeur du paramètre
vartype	text	Type du paramètre (`bool`, `enum`, `integer`, `real` ou `string`)
source	text	Source de la valeur du paramètre actuel
min_val	text	Valeur minimale autorisée du paramètre (NULL pour les valeurs non numériques)
max_val	text	Valeur maximale autorisée du paramètre (NULL pour les valeurs non numé-

Nom	Type	Description
		riques)
enumvals	text[]	Valeurs autorisées pour un paramètre enum (NULL pour les valeurs non enum)
boot_val	text	Valeur de paramètre prise au démarrage du serveur si le paramètre n'est pas positionné d'une autre façon
reset_val	text	Valeur à laquelle **RESET** ramènerait le paramètre dans la session courante
sourcefile	text	Fichier de configuration dans lequel ce fichier a été positionné (NULL pour les valeurs positionnées ailleurs que dans un fichier de configuration, ou quand interrogé par un utilisateur standard). Pratique quand on utilise des directives d'inclusion de configuration
sourceline	integer	Numéro de ligne du fichier de configuration à laquelle cette valeur a été positionnée (NULL pour des valeurs positionnées ailleurs que dans un fichier de configuration, ou quand interrogé par un non-superutilisateur).

La vue pg_settings n'accepte ni insertion ni suppression mais peut être actualisée. Une requête **UPDATE** appliquée à une ligne de pg_settings est équivalente à exécuter la commande SET(7) sur ce paramètre. Le changement affecte uniquement la valeur utilisée par la session en cours. Si un **UPDATE** est lancé à l'intérieur d'une transaction annulée par la suite, les effets de la commande **UPDATE** disparaissent à l'annulation de la transaction. Lorsque la transaction est validée, les effets persistent jusqu'à la fin de la session, à moins qu'un autre **UPDATE** ou **SET** ne modifie la valeur.

45.56. pg_shadow

La vue pg_shadow existe pour des raisons de compatibilité ascendante : elle émule un catalogue qui a existé avant la version 8.1 de PostgreSQL™. Elle affiche les propriétés de tous les rôles marqués `rolcanlogin` dans pg_authid.

Cette table tire son nom de la nécessité de ne pas être publiquement lisible, car elle contient les mots de passe. pg_user est une vue sur pg_shadow, publiquement accessible, car elle masque le contenu du champ de mot de passe.

Tableau 45.57. Colonnes de pg_shadow

Nom	Type	Références	Description
usename	name	`pg_authid.rolname`	Nom de l'utilisateur
usesysid	oid	`pg_authid.oid`	Identifiant de l'utilisateur
usecreatedb	bool		L'utilisateur peut créer des bases de données
usesuper	bool		L'utilisateur est un superutilisateur
usecatupd	bool		L'utilisateur peut mettre à jour les catalogues système. (Même un superutilisateur ne peut pas le faire si cette colonne ne vaut pas `true`.)
passwd	text		Mot de passe (éventuellement chiffré) ; NULL si aucun. Voir pg_authid pour des détails sur le stockage des mots de passe chiffrés.
valuntil	abstime		Estampille temporelle d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe)
useconfig	text[]		Valeurs de session par défaut des variables de configuration

45.57. pg_stats

La vue pg_stats fournit un accès aux informations stockées dans la table système pg_statistic. Cette vue n'autorise l'accès qu'aux seules lignes de pg_statistic correspondant aux tables sur lesquelles l'utilisateur a un droit de lecture. Elle peut donc sans risque être publiquement accessible en lecture.

pg_stats est aussi conçue pour afficher l'information dans un format plus lisible que le catalogue sous-jacent -- au prix de l'extension du schéma lorsque de nouveaux types de connecteurs sont définis dans pg_statistic.

Tableau 45.58. Colonnes de pg_stats

Nom	Type	Références	Description
schemaname	name	pg_namespace.nspname	Nom du schéma contenant la table
tablename	name	pg_class.relname	Nom de la table
attname	name	pg_attribute.attname	Nom de la colonne décrite par la ligne
alled	bool		Si vrai, cette ligne inclut les colonnes enfant de l'héritage, pas seulement les valeurs de la table spécifiée
null_frac	real		Fraction d'entrées de colonnes qui sont NULL
avg_width	integer		Largeur moyenne en octets des entrées de la colonne
n_distinct	real		Si positif, nombre estimé de valeurs distinctes dans la colonne. Si négatif, nombre de valeurs distinctes divisé par le nombre de lignes, le tout muliplié par -1. (La forme négative est utilisée quand **ANALYZE** croit que le nombre de valeurs distinctes a tendance à grossir au fur et à mesure que la table grossit ; la forme positive est utilisée lorsque la commande semble avoir un nombre fixe de valeurs possibles.) Par exemple, -1 indique une colonne unique pour laquelle le nombre de valeurs distinctes est identique au nombre de lignes.
most_common_vals	anyarray		Liste de valeurs habituelles de la colonne. (NULL si aucune valeur ne semble identique aux autres.) Pour certains types de données comme tsvector, c'est une liste d'éléments les plus fréquents, plutôt que des valeurs du type lui-même.
most_common_freqs	real[]		Liste de fréquences des valeurs ou éléments les plus courants, c'est-à-dire le nombre d'occurrences de chacune divisé par le nombre total de lignes. (NULL lorsque *most_common_vals* l'est.) Pour certains types de données comme tsvector, il peut aussi stocker des informations supplémentaires, le rendant plus long que le tableau *most_common_vals*.
histogram_bounds	anyarray		Liste de valeurs qui divisent les valeurs de la colonne en groupes de population approximativement identiques. Les valeurs dans *most_common_vals*, s'il y en a, sont omises de ce calcul d'histogramme. (Cette colonne est NULL si le type de données de la colonne ne dispose pas de l'opérateur < ou si la liste *most_common_vals* compte la population complète.)
correlation	real		Corrélation statistique entre l'ordre physique des lignes et l'ordre logique des valeurs de la colonne. Ceci va de -1 à +1. Lorsque la valeur est proche de -1 ou +1, un parcours de l'index sur la colonne est estimé moins coûteux que si cette valeur tend vers 0, à cause de la réduction du nombre d'accès aléatoires au disque. (Cette colonne est NULL si le type de données de la colonne ne dispose pas de l'opérateur <.)

Le nombre maximum d'entrées dans *most_common_vals* et *histogram_bounds* est configurable colonne par colonne en utilisant la commande **ALTER TABLE SET STATISTICS** ou globalement avec le paramètre d'exécution default_statistics_target.

45.58. pg_tables

La vue pg_tables fournit un accès aux informations utiles de chaque table de la base de données.

Tableau 45.59. Colonnes de pg_tables

Nom	Type	Références	Description
schemaname	name	pg_namespace.nspname	Nom du schéma qui possède la table
tablename	name	pg_class.relname	Nom de la table
tableowner	name	pg_authid.rolname	Nom du propriétaire de la table
tablespace	name	pg_tablespace.spcname	Nom du *tablespace* qui contient la table (NULL s'il s'agit du *tablespace* par défaut de la base)
hasindexes	boolean	pg_class.relhasindex	Vrai si la table comporte (ou a récemment comporté) des index
hasrules	boolean	pg_class.relhasrules	Vrai si la table dispose (ou disposait) de règles
hastriggers	boolean	pg_class.reltriggers	Vrai si la table dispose (ou disposait) de déclencheurs

45.59. pg_timezone_abbrevs

La vue pg_timezone_abbrevs fournit la liste des abréviations de fuseaux horaires actuellement reconnues par les routines de saisie date/heure. Le contenu de cette vue change avec la modification du paramètre d'exécution timezone_abbreviations.

Tableau 45.60. Colonnes de pg_timezone_abbrevs

Nom	Type	Description
abbrev	text	Abréviation du fuseau horaire
utc_offset	interval	Décalage de l'UTC (positif signifiant à l'est de Greenwich)
is_dst	boolean	true s'il s'agit d'une abréviation de fuseau horaire soumis aux changements d'heure hiver/été

45.60. pg_timezone_names

La vue pg_timezone_names fournit la liste des noms de fuseaux horaires reconnus par **SET TIMEZONE**, avec les abréviations acceptées, les décalages UTC, et l'état du changement d'heure. Contrairement aux abréviations indiquées dans pg_timezone_abbrevs, la majorité des noms impliquent des règles concernant les dates de changement d'heure. De ce fait, l'information associée change en fonction des frontières de changement d'heure locales. L'information affichée est calculée suivant la valeur courante de CURRENT_TIMESTAMP.

Tableau 45.61. Colonnes de pg_timezone_names

Nom	Type	Description
name	text	Nom du fuseau horaire
abbrev	text	Abréviation du fuseau horaire
utc_offset	interval	Décalage à partir d'UTC (positif signifiant à l'est de Greenwich)
is_dst	boolean	true si les changements d'heure hiver/été sont suivis

45.61. pg_user

La vue pg_user fournit un accès aux informations concernant les utilisateurs de la base de données. C'est une simple vue publiquement lisible de pg_shadow qui masque la valeur du champ de mot de passe.

Tableau 45.62. Colonnes de pg_user

Nom	Type	Description
usename	name	Nom de l'utilisateur
usesysid	int4	Identifiant de l'utilisateur (un nombre arbitraire utilisé en référence à cet utilisateur)
usecreatedb	bool	L'utilisateur peut créer des bases de données
usesuper	bool	L'utilisateur est un superutilisateur
usecatupd	bool	L'utilisateur peut mettre à jour les tables systèmes. (Même un superutilisateur ne peut pas le faire si cette colonne n'est pas positionnée à `true`.)
passwd	text	Ce n'est pas le mot de passe (toujours `********`)
valuntil	abstime	Estampille temporelle d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe)
useconfig	text[]	Variables d'exécution par défaut de la session

45.62. pg_user_mappings

La vue pg_user_mappings donne accès aux informations sur les correspondances d'utilisateurs. C'est essentiellement une vue accessible à tous sur pg_user_mapping qui cache le champ d'options si l'utilisateur n'a pas le droit de l'utiliser.

Tableau 45.63. Colonnes de pg_user_mappings

Nom	Type	Réference
umid	oid	`pg_user_mapping.oid`
srvid	oid	`pg_foreign_server.oid`
srvname	text	
umuser	oid	`pg_authid.oid`
usename	name	
umoptions	text[]	

45.63. pg_views

La vue pg_views donne accès à des informations utiles à propos de chaque vue de la base.

Tableau 45.64. Colonnes de pg_views

Nom	Type	Références	Description
schemaname	name	`pg_namespace.nspname`	Nom du schéma contenant la vue
viewname	name	`pg_class.relname`	Nom de la vue
viewowner	name	`pg_authid.rolname`	Nom du propriétaire de la vue
definition	text		Définition de la vue (une requête **SELECT** reconstruite)

Chapitre 46. Protocole client/serveur

postgresql™ utilise un protocole messages pour la communication entre les clients et les serveurs (« frontend » et « backend »). le protocole est supporté par tcp/ip et par les sockets de domaine Unix. Le numéro de port 5432 a été enregistré par l'IANA comme numéro de port TCP personnalisé pour les serveurs supportant ce protocole mais en pratique tout numéro de port non privilégié peut être utilisé.

Ce document décrit la version 3.0 de ce protocole, telle qu'implantée dans postgresql™ depuis la version 7.4. pour obtenir la description des versions précédentes du protocole, il faudra se reporter aux versions antérieures de la documentation de postgresql™. un même serveur peut supporter plusieurs versions du protocole. Lors de l'établissement de la communication le client indique au serveur la version du protocole qu'il souhaite utiliser. Le serveur suivra ce protocole s'il en est capable.

Pour répondre efficacement à de multiples clients, le serveur lance un nouveau serveur (« backend ») pour chaque client. dans l'implémentation actuelle, un nouveau processus fils est créé immédiatement après la détection d'une connexion entrante. Et cela de façon transparente pour le protocole. Pour le protocole, les termes « backend » et « serveur » sont interchangeables ; comme « frontend », « interface » et « client ».

46.1. Aperçu

Le protocole utilise des phases distinctes pour le lancement et le fonctionnement habituel. Dans la phase de lancement, le client ouvre une connexion au serveur et s'authentifie (ce qui peut impliquer un message simple, ou plusieurs messages, en fonction de la méthode d'authentification utilisée). En cas de réussite, le serveur envoie une information de statut au client et entre dans le mode normal de fonctionnement. Exception faite du message initial de demande de lancement, cette partie du protocole est conduite par le serveur.

En mode de fonctionnement normal, le client envoie requêtes et commandes au serveur et celui-ci retourne les résultats de requêtes et autres réponses. Il existe quelques cas (comme **notify**) pour lesquels le serveur enverra des messages non sollicités. Mais dans l'ensemble, cette partie de la session est conduite par les requêtes du client.

En général, c'est le client qui décide de la clôture de la session. Il arrive, cependant, qu'elle soit forcée par le moteur. Dans tous les cas, lors de la fermeture de la connexion par le serveur, toute transaction ouverte (non terminée) sera annulée.

En mode opérationnel normal, les commandes SQL peuvent être exécutées via deux sous-protocoles. Dans le protocole des « requêtes simples », le client envoie juste une chaîne, la requête, qui est analysée et exécutée immédiatement par le serveur. Dans le protocole des « requêtes étendues », le traitement des requêtes est découpé en de nombreuses étapes : l'analyse, le lien avec les valeurs de paramètres et l'exécution. Ceci offre flexibilité et gains en performances au prix d'une complexité supplémentaire.

Le mode opérationnel normal offre des sous-protocoles supplémentaires pour certaines opérations comme **copy**.

46.1.1. Aperçu des messages

Toute la communication s'effectue au travers d'un flux de messages. Le premier octet d'un message identifie le type de message et les quatre octets suivants spécifient la longueur du reste du message (cette longueur inclut les 4 octets de longueur, mais pas l'octet du type de message). Le reste du contenu du message est déterminé par le type de message. Pour des raisons historiques, le tout premier message envoyé par le client (le message de lancement) n'a pas l'octet initial de type du message.

Pour éviter de perdre la synchronisation avec le flux de messages, le serveur et le client stocke le message complet dans un tampon (en utilisant le nombre d'octets) avant de tenter de traiter son contenu. Cela permet une récupération simple si une erreur est détectée lors du traitement du contenu. Dans les situations extrêmes (telles que de ne pas avoir assez de mémoire pour placer le message dans le tampon), le récepteur peut utiliser le nombre d'octets pour déterminer le nombre d'entrées à ignorer avant de continuer la lecture des messages.

En revanche, serveurs et clients doivent être attentifs à ne pas envoyer de message incomplet. Ceci est habituellement obtenu en plaçant le message complet dans un tampon avant de commencer l'envoi. Si un échec de communications survient pendant l'envoi ou la réception d'un message, la seule réponse plausible est l'abandon de la connexion. Il y a, en effet, peu d'espoir de resynchronisation des messages.

46.1.2. Aperçu des requêtes étendues

Dans le protocole des requêtes étendues, l'exécution de commandes SQL est scindée en plusieurs étapes. L'état retenu entre les étapes est représenté par deux types d'objets : les *instructions préparées* et les *portails*. une instruction préparée représente le résultat de l'analyse syntaxique, de l'analyse sémantique et de la planification d'une chaîne de requête textuelle. Une instruction préparée n'est pas nécessairement prête à être exécutée parce qu'il peut lui manquer certaines valeurs de *paramètres*. un portail représente une instruction prête à être exécutée ou déjà partiellement exécutée, dont toutes les valeurs de paramètres manquant

sont données (pour les instructions **select**, un portail est équivalent à un curseur ouvert. il est choisi d'utiliser un terme différent car les curseurs ne gèrent pas les instructions autres que **select**)

Le cycle d'exécution complet consiste en une étape d'*analyse syntaxique*, qui crée une instruction préparée à partir d'une chaîne de requête textuelle ; une étape de *liaison*, qui crée un portail à partir d'une instruction préparée et des valeurs pour les paramètres nécessaires ; et une étape d'*exécution* qui exécute une requête du portail. Dans le cas d'une requête qui renvoie des lignes (**select**, **show**, etc), il peut être signalé à l'étape d'exécution que seul un certain nombre de lignes doivent être retournées, de sorte que de multiples étapes d'exécution seront nécessaires pour terminer l'opération.

Le serveur peut garder la trace de multiples instructions préparées et portails (qui n'existent qu'à l'intérieur d'une session, et ne sont jamais partagés entre les sessions). Les instructions préparées et les portails sont référencés par les noms qui leur sont affectés à la création. De plus, il existe une instruction préparée et un portail « non nommés ». bien qu'ils se comportent comme des objets nommés, les opérations y sont optimisées en vue d'une exécution unique de la requête avant son annulation puis est annulée. En revanche, les opérations sur les objets nommés sont optimisées pour des utilisations multiples.

46.1.3. Formats et codes de format

Les données d'un type particulier pouvaient être transmises sous différents *formats*. depuis postgresql™ 7.4, les seuls formats supportés sont le « texte » et le « binaire » mais le protocole prévoit des extensions futures. Le format désiré pour toute valeur est spécifié par un *code de format*. les clients peuvent spécifier un code de format pour chaque valeur de paramètre transmise et pour chaque colonne du résultat d'une requête. Le texte a zéro pour code de format zéro, le binaire un. Tous les autres codes de format sont réservés pour des définitions futures.

La représentation au format texte des valeurs est toute chaîne produite et acceptée par les fonctions de conversion en entrée/sortie pour le type de données particulier. Dans la représentation transmise, il n'y a pas de caractère nul de terminaison de chaîne ; le client doit en ajouter un s'il souhaite traiter les valeurs comme des chaînes C (le format texte n'autorise pas les valeurs nulles intégrées).

Les représentations binaires des entiers utilisent l'ordre d'octet réseau (octet le plus significatif en premier). Pour les autres types de données, il faudra consulter la documentation ou le code source pour connaître la représentation binaire. Les représentations binaires des types de données complexes changent parfois entre les versions du serveur ; le format texte reste le choix le plus portable.

46.2. Flux de messages

Cette section décrit le flux des messages et la sémantique de chaque type de message (les détails concernant la représentation exacte de chaque message apparaissent dans Section 46.5, « Formats de message »). il existe différents sous-protocoles en fonction de l'état de la connexion : lancement, requête, appel de fonction, COPY et clôture. Il existe aussi des provisions spéciales pour les opérations asynchrones (incluant les réponses aux notifications et les annulations de commande), qui peuvent arriver à tout moment après la phase de lancement.

46.2.1. Lancement

Pour débuter une session, un client ouvre une connexion au serveur et envoie un message de démarrage. Ce message inclut les noms de l'utilisateur et de la base de données à laquelle le client souhaite se connecter ; il identifie aussi la version particulière du protocole à utiliser (optionnellement, le message de démarrage peut inclure des précisions supplémentaires pour les paramètres d'exécution). Le serveur utilise ces informations et le contenu des fichiers de configuration (tels que `pg_hba.conf`) pour déterminer si la connexion est acceptable et quelle éventuelle authentification supplémentaire est requise.

Le serveur envoie ensuite le message de demande d'authentification approprié, auquel le client doit répondre avec le message de réponse d'authentification adapté (tel un mot de passe). Pour toutes les méthodes d'authentification, sauf GSSAPI et SSPI, il y a au maximum une requête et une réponse. Avec certaines méthodes, aucune réponse du client n'est nécessaire aucune demande d'authentification n'est alors effectuée. Pour GSSAPI et SSPI, plusieurs échanges de paquets peuvent être nécessaire pour terminer l'authentification.

Le cycle d'authentification se termine lorsque le serveur rejette la tentative de connexion (ErrorResponse) ou l'accepte (AuthenticationOk).

Les messages possibles du serveur dans cette phase sont :

errorresponse
: La tentative de connexion a été rejetée. Le serveur ferme immédiatement la connexion.

authenticationok
: L'échange d'authentification s'est terminé avec succès.

authenticationkerberosv5

Le client doit alors prendre part à un dialogue d'authentification Kerberos V5 (spécification Kerberos, non décrite ici) avec le serveur. En cas de succès, le serveur répond AuthenticationOk, ErrorResponse sinon.

authenticationcleartextpassword
Le client doit alors envoyer un PasswordMessage contenant le mot de passe en clair. Si le mot de passe est correct, le serveur répond AuthenticationOk, ErrorResponse sinon.

authenticationmd5password
Le client doit alors envoyer un PasswordMessage contenant le mot de passe chiffré à l'aide de MD5, en utilisant le composant salt de quatre caractères spécifié dans le message AuthenticationMD5Password. Si le mot de passe est correct, le serveur répond AuthenticationOk, ErrorResponse sinon.

authenticationscmcredential
Cette réponse est possible uniquement pour les connexions locales de domaine Unix sur les plateformes qui supportent les messages de légitimation SCM. Le client doit fournir un message de légitimation SCM, puis envoyer une donnée d'un octet. Le contenu de cet octet importe peu ; il n'est utilisé que pour s'assurer que le serveur attend assez longtemps pour recevoir le message de légitimation. Si la légitimation est acceptable, le serveur répond AuthenticationOk, ErrorResponse sinon.

AuthenticationGSS
L'interface doit maintenant initier une négociation GSSAPI. L'interface doit envoyer un PasswordMessage avec la première partie du flux de données GSSAPI en réponse à ceci. Si plus de messages sont nécessaires, le serveur répondra avec AuthenticationGSSContinue.

AuthenticationSSPI
L'interface doit maintenant initier une négociation SSPI. L'interface doit envoyer un PasswordMessage avec la première partie du flux de données SSPI en réponse à ceci. Si plus de messages sont nécessaires, le serveur répondra avec AuthenticationGSSContinue.

AuthenticationGSSContinue
Ce message contient les données de la réponse de l'étape précédente pour la négociation GSSAPI ou SSPI (AuthenticationGSS ou un précédent AuthenticationGSSContinue). Si les données GSSAPI dans ce message indique que plus de données sont nécessaire pour terminer l'authentification, l'interface doit envoyer cette donnée dans un autre PasswordMessage. Si l'authentification GSSAPI ou SSPI est terminée par ce message, le serveur enverra ensuite AuthenticationOk pour indiquer une authentification réussie ou ErrorResponse pour indiquer l'échec.

Si le client ne supporte pas la méthode d'authentification demandée par le serveur, il doit immédiatement fermer la connexion.

Après la réception du message AuthenticationOk, le client attend d'autres messages du serveur. Au cours de cette phase, un processus serveur est lancé et le client est simplement en attente. Il est encore possible que la tentative de lancement échoue (ErrorResponse) mais, dans la plupart des cas, le serveur enverra les messages ParameterStatus, BackendKeyData et enfin ReadyForQuery.

Durant cette phase, le serveur tentera d'appliquer tous les paramètres d'exécution supplémentaires qui ont été fournis par le message de lancement. En cas de succès, ces valeurs deviennent les valeurs par défaut de la session. Une erreur engendre ErrorResponse et déclenche la sortie.

Les messages possibles du serveur dans cette phase sont :

backendkeydata
Ce message fournit une clé secrète que le client doit conserver s'il souhaite envoyer des annulations de requêtes par la suite. Le client ne devrait pas répondre à ce message, mais continuer à attendre un message ReadyForQuery.

parameterstatus
Ce message informe le client de la configuration actuelle (initiale) des paramètres du serveur, tels `client_encoding` ou `datestyle`. le client peut ignorer ce message ou enregistrer la configuration pour ses besoins futurs ; voir Section 46.2.6, « Opérations asynchrones » pour plus de détails. le client ne devrait pas répondre à ce message mais continuer à attendre un message ReadyForQuery.

readyforquery
Le lancement est terminé. Le client peut dès lors envoyer des commandes.

errorresponse
Le lancement a échoué. La connexion est fermée après l'envoi de ce message.

noticeresponse
Un message d'avertissement a été envoyé. Le client devrait afficher ce message mais continuer à attendre un ReadyForQuery ou un ErrorResponse.

Le même message ReadyForQuery est envoyé à chaque cycle de commande. En fonction des besoins de codage du client, il est possible de considérer ReadyForQuery comme le début d'un cycle de commande, ou de le considérer comme terminant la phase de lancement et chaque cycle de commande.

46.2.2. Requête simple

Un cycle de requête simple est initié par le client qui envoie un message Query au serveur. Le message inclut une commande SQL (ou plusieurs) exprimée comme une chaîne texte. Le serveur envoie, alors, un ou plusieurs messages de réponse dépendant du contenu de la chaîne représentant la requête et enfin un message ReadyForQuery. ReadyForQuery informe le client qu'il peut envoyer une nouvelle commande. Il n'est pas nécessaire que le client attende ReadyForQuery avant de lancer une autre commande mais le client prend alors la responsabilité de ce qui arrive si la commande précédente échoue et que les commandes suivantes, déjà lancées, réussissent.

Les messages de réponse du serveur sont :

commandcomplete
: Commande SQL terminée normalement.

copyinresponse
: Le serveur est prêt à copier des données du client vers une table voir Section 46.2.5, « Opérations copy ».

copyoutresponse
: Le serveur est prêt à copier des données d'une table vers le client ; voir Section 46.2.5, « Opérations copy ».

rowdescription
: Indique que des lignes vont être envoyées en réponse à une requête **select**, **fetch**... Le contenu de ce message décrit le placement des colonnes dans les lignes. Le contenu est suivi d'un message DataRow pour chaque ligne envoyée au client.

datarow
: Un des ensembles de lignes retournés par une requête **select**, **fetch**...

emptyqueryresponse
: Une chaîne de requête vide a été reconnue.

errorresponse
: Une erreur est survenue.

readyforquery
: Le traitement d'une requête est terminé. Un message séparé est envoyé pour l'indiquer parce qu'il se peut que la chaîne de la requête contienne plusieurs commandes SQL. CommandComplete marque la fin du traitement d'une commande SQL, pas de la chaîne complète. ReadyForQuery sera toujours envoyé que le traitement se termine avec succès ou non.

noticeresponse
: Un message d'avertissement concernant la requête a été envoyé. Les avertissements sont complémentaires des autres réponses, le serveur continuera à traiter la commande.

La réponse à une requête **select** (ou à d'autres requêtes, telles **explain** ou **show**, qui retournent des ensembles de données) consiste normalement en un RowDescription, plusieurs messages DataRow (ou aucun) et pour finir un CommandComplete. **copy** depuis ou vers le client utilise un protocole spécial décrit dans Section 46.2.5, « Opérations copy ». tous les autres types de requêtes produisent uniquement un message CommandComplete.

Puisqu'une chaîne de caractères peut contenir plusieurs requêtes (séparées par des points virgules), il peut y avoir plusieurs séquences de réponses avant que le serveur ne finisse de traiter la chaîne. ReadyForQuery est envoyé lorsque la chaîne complète a été traitée et que le serveur est prêt à accepter une nouvelle chaîne de requêtes.

Si une chaîne de requêtes complètement vide est reçue (aucun contenu autre que des espaces fines), la réponse sera EmptyQueryResponse suivie de ReadyForQuery.

En cas d'erreur, ErrorResponse est envoyé suivi de ReadyForQuery. Tous les traitements suivants de la chaîne sont annulés par ErrorResponse (quelque soit le nombre de requêtes restant à traiter). Ceci peut survenir au milieu de la séquence de messages engendrés par une requête individuelle.

En mode de requêtage simple, les valeurs récupérées sont toujours au format texte, sauf si la commande est un **fetch** sur un curseur déclaré avec l'option `binary`. dans ce cas, les valeurs récupérées sont au format binaire. Les codes de format donnés dans le message RowDescription indiquent le format utilisé.

La planification de requêtes pour des instructions préparées survient lorsque le message Parse est reçu. Si une requête sera exécuté de façon répété avec différents paramètres, il pourrait être bénéfique d'envoyer un seul message Parse contenant une requête avec paramètres, suivie de plusieurs messages Bind et Execute. Ceci évitera de planifier de nouveau la requête pour chaque exécution.

L'instruction préparée non nommée est planifiée lors du traitement de Parse si le message Parse ne définit aucun paramètre. Mais s'il existe des paramètres, la planification de la requête est repoussée jusqu'à ce que le premier message Bind de cette instruction est reçu. Le planificateur considérera les valeurs réelles des paramètres fournies dans le message Bind lors de la planification de la requête.

> **Note**
>
> Les plans de requêtes générés à partir d'une requête avec paramètres pourraient être moins efficaces que les plans de requêtes générés à partir d'une requête équivalente dont les valeurs de paramètres réelles ont été placées. Le planificateur de requêtes ne peut pas prendre les décisions suivant les valeurs réelles des paramètres (par exemple, la sélectivité de l'index) lors de la planification d'une requête avec paramètres affectée à un objet instruction préparée nommée. La pénalité possible est évitée lors de l'utilisation d'une instruction non nommée car elle n'est pas planifiée jusqu'à ce que des valeurs réelles de paramètres soient disponibles.
>
> Si un autre Bind référençant l'objet instruction préparée non nommée est reçu, la requête n'est pas de nouveau planifiée. Les valeurs de paramètres utilisées dans le premier message Bind pourrait produire un plan de requête qui est seulement efficace pour un sous-ensemble des valeurs de paramètres possibles. Pour forcer une nouvelle planification de la requête pour un ensemble nouveau de paramètres, envoyez un autre message Parse pour remplacer l'objet instruction préparée non nommée.

Un client doit être préparé à accepter des messages ErrorResponse et NoticeResponse quand bien même il s'attendrait à un autre type de message. Voir aussi Section 46.2.6, « Opérations asynchrones » concernant les messages que le client pourrait engendrer du fait d'événements extérieurs.

La bonne pratique consiste à coder les clients dans un style machine-état qui acceptera tout type de message à tout moment plutôt que de parier sur la séquence exacte des messages.

46.2.3. Requête étendue

Le protocole de requête étendu divise le protocole de requêtage simple décrit ci-dessus en plusieurs étapes. Les résultats des étapes de préparation peuvent être réutilisés plusieurs fois pour plus d'efficacité. De plus, des fonctionnalités supplémentaires sont disponibles, telles que la possibilité de fournir les valeurs des données comme des paramètres séparés au lieu d'avoir à les insérer directement dans une chaîne de requêtes.

Dans le protocole étendu, le client envoie tout d'abord un message Parse qui contient une chaîne de requête, optionnellement quelques informations sur les types de données aux emplacements des paramètres, et le nom de l'objet de destination d'une instruction préparée (une chaîne vide sélectionne l'instruction préparée sans nom). La réponse est soit ParseComplete soit ErrorResponse. Les types de données des paramètres peuvent être spécifiés par l'OID ; dans le cas contraire, l'analyseur tente d'inférer les types de données de la même façon qu'il le ferait pour les constantes chaînes littérales non typées.

> **Note**
>
> Un type de paramètre peut être laissé non spécifié en le positionnant à O, ou en créant un tableau d'OID de type plus court que le nombre de paramètres ($*n*) utilisés dans la chaîne de requête. Un autre cas spécial est d'utiliser void comme type de paramètre (c'est à dire l'OID du pseudo-type void). Cela permet d'utiliser des paramètres dans des fonctions en tant qu'argument OUT. Généralement, il n'y a pas de contexte dans lequel void peut être utilisé, mais si un tel paramètre apparaît dans les arguments d'une fonction, il sera simplement ignoré. Par exemple, un appel de fonction comme `foo($1,$2,$3,$4)` peu correspondre à une fonction avec 2 arguments IN et 2 autres OUT si `$3` et `$4` sont spécifiés avec le type void.

> **Note**
>
> La chaîne contenue dans un message Parse ne peut pas inclure plus d'une instruction SQL, sinon une erreur de syntaxe est rapportée. Cette restriction n'existe pas dans le protocole de requête simple, mais est présente dans le protocole étendu. En effet, permettre aux instructions préparées ou aux portails de contenir de multiples commandes compliquerait inutilement le protocole.

En cas de succès de sa création, une instruction préparée nommée dure jusqu'à la fin de la session courante, sauf si elle est détruite explicitement. Une instruction préparée non nommée ne dure que jusqu'à la prochaine instruction Parse spécifiant l'instruction non nommée comme destination. Un simple message Query détruit également l'instruction non nommée. Les instructions préparées nommées doivent être explicitement closes avant de pouvoir être redéfinies par un message Parse. Ce n'est pas obligatoire pour une instruction non nommée. Il est également possible de créer des instructions préparées nommées, et d'y accéder, en ligne de

commandes SQL à l'aide des instructions **prepare** et **execute**.

Dès lors qu'une instruction préparée existe, elle est déclarée exécutable par un message Bind. Le message Bind donne le nom de l'instruction préparée source (une chaîne vide désigne l'instruction préparée non nommée), le nom du portail destination (une chaîne vide désigne le portail non nommé) et les valeurs à utiliser pour tout emplacement de paramètres présent dans l'instruction préparée. L'ensemble des paramètres fournis doit correspondre à ceux nécessaires à l'instruction préparée. Bind spécifie aussi le format à utiliser pour toutes les données renvoyées par la requête ; le format peut être spécifié complètement ou par colonne. La réponse est, soit BindComplete, soit ErrorResponse.

> **Note**
>
> Le choix entre sortie texte et binaire est déterminé par les codes de format donnés dans Bind, quelque soit la commande SQL impliquée. L'attribut `BINARY` dans les déclarations du curseur n'est pas pertinent lors de l'utilisation du protocole de requête étendue.

La planification d'un requête préparée nommée se fait lorsque le message Parse est traité. Si une requête est exécutée plusieurs fois, avec différents paramètres, il peut être bénéfique d'envoyer un seul message Parse contenant la requête paramétrée, suivie de plusieurs messages Bind et Execute. Cette méthode permet d'éviter de replanifier la requête à chaque exécution.

Une requête préparée non nommée est aussi planifiée lors du traitement de Parse si ce dernier ne défini pas de paramètres. Si des paramètres sont utilisés, la planification se fera à chaque fois que les valeurs des paramètres seront transmis par le message Bind. Ce comportement permet au planificateur de prendre en compte les valeurs des paramètres du Bind au moment de créer son plan d'exécution plutôt que d'utiliser un plan générique estimé.

> **Note**
>
> Les plans d'exécution générés pour une requête paramétrée peuvent être moins efficaces que ceux générés pour une requête équivalente avec les paramètres directement substitués. Le planificateur de requête ne peut pas prendre de décision se basant sur les valeurs réelles des paramètres (par exemple, la sélectivité d'un index) lors de la planification d'une requête paramétrée nommée. Cette pénalité potentielle peut être évitée en utilisant des requêtes préparée non nommées, puisque'elles ne seront pas planifiées avant que les valeurs des paramètres soient disponibles. La contrepartie est que la planification doit alors être effectuée à chaque message Bind, même si la requête reste la même.

En cas de succès de sa création, un objet portail nommé dure jusqu'à la fin de la transaction courante sauf s'il est explicitement détruit. Un portail non nommé est détruit à la fin de la transaction ou dès la prochaine instruction Bind spécifiant le portail non nommé comme destination. Un simple message Query détruit également le portail non nommé. Les portails nommés doivent être explicitement fermés avant de pouvoir être redéfinis par un message Bind. Cela n'est pas obligatoire pour le portail non nommé. Il est également possible de créer des portails nommés, et d'y accéder, en ligne de commandes SQL à l'aide des instructions **declare cursor** et **fetch**.

Dès lors qu'un portail existe, il peut être exécuté à l'aide d'un message Execute. Ce message spécifie le nom du portail (une chaîne vide désigne le portail non nommé) et un nombre maximum de lignes de résultat (zéro signifiant la « récupération de toutes les lignes »). le nombre de lignes de résultat a seulement un sens pour les portails contenant des commandes qui renvoient des ensembles de lignes ; dans les autres cas, la commande est toujours exécutée jusqu'à la fin et le nombre de lignes est ignoré. Les réponses possibles d'Execute sont les même que celles décrites ci-dessus pour les requêtes lancées via le protocole de requête simple, si ce n'est qu'Execute ne cause pas l'envoi de ReadyForQuery ou de RowDescription.

Si Execute se termine avant la fin de l'exécution d'un portail (du fait d'un nombre de lignes de résultats différent de zéro), il enverra un message PortalSuspended ; la survenue de ce message indique au client qu'un autre Execute devrait être lancé sur le même portail pour terminer l'opération. Le message CommandComplete indiquant la fin de la commande SQL n'est pas envoyé avant l'exécution complète du portail. Une phase Execute est toujours terminée par la survenue d'un seul de ces messages : CommandComplete, EmptyQueryResponse (si le portail a été créé à partir d'une chaîne de requête vide), ErrorResponse ou PortalSuspended.

À la réalisation complète de chaque série de messages de requêtes étendues, le client doit lancer un message Sync. Ce message sans paramètre oblige le serveur à fermer la transaction courante si elle n'est pas à l'intérieur d'un bloc de transaction **begin**/**commit** (« fermer » signifiant valider en l'absence d'erreur ou annuler sinon). Une réponse ReadyForQuery est alors envoyée. Le but de Sync est de fournir un point de resynchronisation pour les récupérations d'erreurs. Quand une erreur est détectée lors du traitement d'un message de requête étendue, le serveur lance ErrorResponse, puis lit et annule les messages jusqu'à ce qu'un Sync soit atteint. Il envoie ensuite ReadyForQuery et retourne au traitement normal des messages. Aucun échappement n'est réalisé si une erreur est détectée *lors* du traitement de sync -- l'unicité du ReadyForQuery envoyé pour chaque Sync est ainsi assurée.

> **Note**
>
> Sync n'impose pas la fermeture d'un bloc de transactions ouvert avec **begin**. cette situation est détectable car le message ReadyForQuery inclut le statut de la transaction.

En plus de ces opérations fondamentales, requises, il y a plusieurs opérations optionnelles qui peuvent être utilisées avec le protocole de requête étendue.

Le message Describe (variante de portail) spécifie le nom d'un portail existant (ou une chaîne vide pour le portail non nommé). La réponse est un message RowDescription décrivant les lignes qui seront renvoyées par l'exécution du portail ; ou un message NoData si le portail ne contient pas de requête renvoyant des lignes ; ou ErrorResponse le portail n'existe pas.

Le message Describe (variante d'instruction) spécifie le nom d'une instruction préparée existante (ou une chaîne vide pour l'instruction préparée non nommée). La réponse est un message ParameterDescription décrivant les paramètres nécessaires à l'instruction, suivi d'un message RowDescription décrivant les lignes qui seront renvoyées lors de l'éventuelle exécution de l'instruction (ou un message NoData si l'instruction ne renvoie pas de lignes). ErrorResponse est retourné si l'instruction préparée n'existe pas. Comme Bind n'a pas encore été exécuté, les formats à utiliser pour les lignes retournées ne sont pas encore connues du serveur ; dans ce cas, les champs du code de format dans le message RowDescription seront composés de zéros.

> **Astuce**
>
> Dans la plupart des scénarios, le client devra exécuter une des variantes de Describe avant de lancer Execute pour s'assurer qu'il sait interpréter les résultats reçus.

Le message Close ferme une instruction préparée ou un portail et libère les ressources. L'exécution de Close sur une instruction ou un portail inexistant ne constitue pas une erreur. La réponse est en général CloseComplete mais peut être ErrorResponse si une difficulté quelconque est rencontrée lors de la libération des ressources. Clore une instruction préparée ferme implicitement tout autre portail ouvert construit à partir de cette instruction.

Le message Flush n'engendre pas de sortie spécifique, mais force le serveur à délivrer toute donnée restante dans les tampons de sortie. Un Flush doit être envoyé après toute commande de requête étendue, à l'exception de Sync, si le client souhaite examiner le résultat de cette commande avant de lancer d'autres commandes. Sans Flush, les messages retournés par le serveur seront combinés en un nombre minimum de paquets pour minimiser la charge réseau.

> **Note**
>
> Le message Query simple est approximativement équivalent aux séries Parse, Bind, Describe sur un portail, Execute, Close, Sync utilisant les objets de l'instruction préparée ou du portail, non nommés et sans paramètres. Une différence est l'acceptation de plusieurs instructions SQL dans la chaîne de requêtes, la séquence bind/describe/execute étant automatiquement réalisée pour chacune, successivement. Il en diffère également en ne retournant pas les messages ParseComplete, BindComplete, CloseComplete ou NoData.

46.2.4. Appel de fonction

Le sous-protocole d'appel de fonction (NDT : Function Call dans la version originale) permet au client d'effectuer un appel direct à toute fonction du catalogue système pg_proc de la base de données. Le client doit avoir le droit d'exécution de la fonction.

> **Note**
>
> Le sous-protocole d'appel de fonction est une fonctionnalité qu'il vaudrait probablement mieux éviter dans tout nouveau code. Des résultats similaires peuvent être obtenus en initialisant une instruction préparée qui lance `select function($1, ...)`. le cycle de l'appel de fonction peut alors être remplacé par Bind/Execute.

Un cycle d'appel de fonction est initié par le client envoyant un message FunctionCall au serveur. Le serveur envoie alors un ou plusieurs messages de réponse en fonction des résultats de l'appel de la fonction et finalement un message de réponse ReadyForQuery. ReadyForQuery informe le client qu'il peut envoyer en toute sécurité une nouvelle requête ou un nouvel appel de fonction.

Les messages de réponse possibles du serveur sont :

errorresponse
: Une erreur est survenue.

functioncallresponse

L'appel de la fonction est terminé et a retourné le résultat donné dans le message. Le protocole d'appel de fonction ne peut gérer qu'un résultat scalaire simple, pas un type ligne ou un ensemble de résultats.

readyforquery
: Le traitement de l'appel de fonction est terminé. ReadyForQuery sera toujours envoyé, que le traitement se termine avec succès ou avec une erreur.

noticeresponse
: Un message d'avertissement relatif à l'appel de fonction a été retourné. Les avertissements sont complémentaires des autres réponses, c'est-à-dire que le serveur continuera à traiter la commande.

46.2.5. Opérations copy

La commande **copy** permet des transferts rapides de données en lot vers ou à partir du serveur. Les opérations Copy-in et Copy-out basculent chacune la connexion dans un sous-protocole distinct qui existe jusqu'à la fin de l'opération.

Le mode Copy-in (transfert de données vers le serveur) est initié quand le serveur exécute une instruction SQL **copy from stdin**. le serveur envoie une message CopyInResponse au client. Le client peut alors envoyer zéro (ou plusieurs) message(s) CopyData, formant un flux de données en entrée (il n'est pas nécessaire que les limites du message aient un rapport avec les limites de la ligne, mais cela est souvent un choix raisonnable). Le client peut terminer le mode Copy-in en envoyant un message CopyDone (permettant une fin avec succès) ou un message CopyFail (qui causera l'échec de l'instruction SQL **copy** avec une erreur). Le serveur retourne alors au mode de traitement de la commande précédant le début de **copy**, protocole de requête simple ou étendu. il enverra enfin CommandComplete (en cas de succès) ou ErrorResponse (sinon).

Si le serveur détecte un erreur en mode copy-in (ce qui inclut la réception d'un message CopyFail), il enverra un message ErrorResponse. Si la commande **copy** a été lancée à l'aide d'un message de requête étendue, le serveur annulera les messages du client jusqu'à ce qu'un message Sync soit reçu. Il enverra alors un message ReadyForQuery et retournera dans le mode de fonctionnement normal. Si la commande **copy** a été lancée dans un message simple Query, le reste de ce message est annulé et ReadyForQuery est envoyé. Dans tous les cas, les messages CopyData, CopyDone ou CopyFail suivants envoyés par l'interface seront simplement annulés.

Le serveur ignorera les messages Flush et Sync reçus en mode copy-in. La réception de tout autre type de messages hors-copie constitue une erreur qui annulera l'état Copy-in, comme cela est décrit plus haut. L'exception pour Flush et Sync est faite pour les bibliothèques clientes qui envoient systématiquement Flush ou Sync après un message Execute sans vérifier si la commande à exécuter est **copy from stdin**.

Le mode Copy-out (transfert de données à partir du serveur) est initié lorsque le serveur exécute une instruction SQL **copy to stdout**. Le moteur envoie un message CopyOutResponse au client suivi de zéro (ou plusieurs) message(s) CopyData (un par ligne), suivi de CopyDone. Le serveur retourne ensuite au mode de traitement de commande dans lequel il se trouvait avant le lancement de **copy** et envoie commandcomplete. Le client ne peut pas annuler le transfert (sauf en fermant la connexion ou en lançant une requête d'annulation, Cancel), mais il peut ignorer les messages CopyData et CopyDone non souhaités.

Si le serveur détecte une erreur en mode Copy-out, il enverra un message ErrorResponse et retournera dans le mode de traitement normal. Le client devrait traiter la réception d'un message ErrorResponse comme terminant le mode « copy-out ».

Il est possible que les messages NoticeResponse et ParameterStatus soient entremêlés avec des messages CopyData ; les interfaces doivent gérer ce cas, et devraient être aussi préparées à d'autres types de messages asynchrones (voir Section 46.2.6, « Opérations asynchrones »). Sinon, tout type de message autre que CopyData et CopyDone pourrait être traité comme terminant le mode copy-out.

Les messages CopyInResponse et CopyOutResponse incluent les champs qui informent le client du nombre de colonnes par ligne et des codes de format de chaque colonne. Dans l'implantation actuelle, toutes les colonnes d'une opération **copy** donnée utilisent le même format, mais la conception du message n'en tient pas compte.

46.2.6. Opérations asynchrones

Il existe plusieurs cas pour lesquels le serveur enverra des messages qui ne sont pas spécifiquement demandés par le flux de commande du client. Les clients doivent être préparés à gérer ces messages à tout moment même si aucune requête n'est en cours. Vérifier ces cas avant de commencer à lire la réponse d'une requête est un minimum.

Il est possible que des messages NoticeResponse soient engendrés en dehors de toute activité ; par exemple, si l'administrateur de la base de données commande un arrêt « rapide » de la base de données, le serveur enverra un NoticeResponse l'indiquant avant de fermer la connexion. Les clients devraient toujours être prêts à accepter et afficher les messages NoticeResponse, même si la connexion est inactive.

Des messages ParameterStatus seront engendrés à chaque fois que la valeur active d'un paramètre est modifiée, et cela pour tout paramètre que le serveur pense utile au client. Cela survient plus généralement en réponse à une commande SQL **set** exécutée par le client. ce cas est en fait synchrone -- mais il est possible aussi que le changement de statut d'un paramètre survienne à la suite

d'une modification par l'administrateur des fichiers de configuration ; changements suivis de l'envoi du signal `sighup` au postmaster. de plus, si une commande set est annulée, un message ParameterStatus approprié sera engendré pour rapporter la valeur effective.

À ce jour, il existe un certain nombre de paramètres codés en dur pour lesquels des messages ParameterStatus seront engendrés : on trouve `server_version`, `server_encoding`, `client_encoding`, `application_name`, `is_superuser`, `session_authorization` et `session_authorization`, `DateStyle`, `IntervalStyle`, `TimeZone` et `integer_datetimes` (`server_encoding`, `timezone` et `integer_datetimes` n'ont pas été reportés par les sorties avant la 8.0 ; `standard_conforming_strings` n'a pas été reporté par les sorties avant la 8.1 ; `IntervalStyle` n'a pas été reporté par les sorties avant la 8.4; `application_name` n'a pas été reporté par les sorties avant la 9.0.). Notez que `server_version`, `server_encoding` et `integer_datetimes` sont des pseudo-paramètres qui ne peuvent pas changer après le lancement. Cet ensemble pourrait changer dans le futur, voire devenir configurable. De toute façon, un client peut ignorer un message ParameterStatus pour les paramètres qu'il ne comprend pas ou qui ne le concernent pas.

Si un client lance une commande **listen**, alors le serveur enverra un message NotificationResponse (à ne pas confondre avec NoticeResponse !) à chaque fois qu'une commande **notify** est exécutée pour le canal de même nom.

> **Note**
>
> Actuellement, NotificationResponse ne peut être envoyé qu'à l'extérieur d'une transaction. Il ne surviendra donc pas au milieu d'une réponse à une commande, mais il peut survenir juste avant ReadyForQuery. Il est toutefois déconseillé de concevoir un client en partant de ce principe. La bonne pratique est d'être capable d'accepter NotificationResponse à tout moment du protocole.

46.2.7. Annulation de requêtes en cours

Pendant le traitement d'une requête, le client peut demander l'annulation de la requête. La demande d'annulation n'est pas envoyée directement au serveur par la connexion ouverte pour des raisons d'efficacité de l'implémentation : il n'est pas admissible que le serveur vérifie constamment les messages émanant du client lors du traitement des requêtes. Les demandes d'annulation sont relativement inhabituelles ; c'est pourquoi elles sont traitées de manière relativement simple afin d'éviter que ce traitement ne pénalise le fonctionnement normal.

Pour effectuer une demande d'annulation, le client ouvre une nouvelle connexion au serveur et envoie un message CancelRequest à la place du message StartupMessage envoyé habituellement à l'ouverture d'une connexion. Le serveur traitera cette requête et fermera la connexion. Pour des raisons de sécurité, aucune réponse directe n'est faite au message de requête d'annulation.

Un message CancelRequest sera ignoré sauf s'il contient la même donnée clé (PID et clé secrète) que celle passée au client lors du démarrage de la connexion. Si la donnée clé correspond, le traitement de la requête en cours est annulé (dans l'implantation existante, ceci est obtenu en envoyant un signal spécial au processus serveur qui traite la requête).

Le signal d'annulation peut ou non être suivi d'effet -- par exemple, s'il arrive après la fin du traitement de la requête par le serveur, il n'aura alors aucun effet. Si l'annulation est effective, il en résulte la fin précoce de la commande accompagnée d'un message d'erreur.

De tout ceci, il ressort que, pour des raisons de sécurité et d'efficacité, le client n'a aucun moyen de savoir si la demande d'annulation a abouti. Il continuera d'attendre que le serveur réponde à la requête. Effectuer une annulation permet simplement d'augmenter la probabilité de voir la requête en cours finir rapidement et échouer accompagnée d'un message d'erreur plutôt que réussir.

Comme la requête d'annulation est envoyée via une nouvelle connexion au serveur et non pas au travers du lien de communication client/serveur établi, il est possible que la requête d'annulation soit lancée par un processus quelconque, pas forcément celui du client pour lequel la requête doit être annulée. Cela peut fournir une flexibilité supplémentaire dans la construction d'applications multi-processus ; mais également une faille de sécurité puisque des personnes non autorisées pourraient tenter d'annuler des requêtes. La faille de sécurité est comblée par l'exigence d'une clé secrète, engendrée dynamiquement, pour toute requête d'annulation.

46.2.8. Fin

Lors de la procédure normale de fin le client envoie un message Terminate et ferme immédiatement la connexion. À la réception de ce message, le serveur ferme la connexion et s'arrête.

Dans de rares cas (tel un arrêt de la base de données par l'administrateur), le serveur peut se déconnecter sans demande du client. Dans de tels cas, le serveur tentera d'envoyer un message d'erreur ou d'avertissement en donnant la raison de la déconnexion avant de fermer la connexion.

D'autres scénarios de fin peuvent être dus à différents cas d'échecs, tels qu'un « core dump » côté client ou serveur, la perte du lien

de communications, la perte de synchronisation des limites du message, etc. Que le client ou le serveur s'aperçoive d'une fermeture de la connexion, le buffer sera vidé et le processus terminé. Le client a la possibilité de lancer un nouveau processus serveur en recontactant le serveur s'il ne souhaite pas se finir. Il peut également envisager de clore la connexion si un type de message non reconnu est reçu ; en effet, ceci est probablement le résultat de la perte de synchronisation des limite de messages.

Que la fin soit normale ou non, toute transaction ouverte est annulée, non pas validée. Si un client se déconnecte alors qu'une requête autre que **select** est en cours de traitement, le serveur terminera probablement la requête avant de prendre connaissance de la déconnexion. Si la requête est en dehors d'un bloc de transaction (séquence **begin** ... **commit**), il se peut que les résultats soient validés avant que la connexion ne soit reconnue.

46.2.9. Chiffrement ssl de session

Si postgresql™ a été construit avec le support de ssl, les communications client/serveur peuvent être chiffrées en l'utilisant. Ce chiffrement assure la sécurité de la communication dans les environnements où des agresseurs pourraient capturer le trafic de la session. Pour plus d'informations sur le cryptage des sessions postgresql™ avec ssl, voir Section 17.8, « Connexions tcp/ip sécurisées avec ssl ».

Pour initier une connexion chiffrée par ssl, le client envoie initialement un message SSLRequest à la place d'un StartupMessage. Le serveur répond avec un seul octet contenant `s` ou `n` indiquant respectivement s'il souhaite ou non utiliser le ssl. Le client peut alors clore la connexion s'il n'est pas satisfait de la réponse. Pour continuer après un `s`, il faut échanger une poignée de main ssl (handshake) (non décrite ici car faisant partie de la spécification ssl) avec le serveur. en cas de succès, le StartupMessage habituel est envoyé. Dans ce cas, StartupMessage et toutes les données suivantes seront chiffrées avec ssl. pour continuer après un `n`, il suffit d'envoyer le startupmessage habituel et de continuer sans chiffrage.

Le client doit être préparé à gérer une réponse ErrorMessage à un SSLRequest émanant du serveur. Ceci ne peut survenir que si le serveur ne dispose pas du support de ssl. dans ce cas, la connexion doit être fermée, mais le client peut choisir d'ouvrir une nouvelle connexion et procéder sans ssl.

Un SSLRequest initial peut également être utilisé dans une connexion en cours d'ouverture pour envoyer un message CancelRequest.

Alors que le protocole lui-même ne fournit pas au serveur de moyen de forcer le chiffrage ssl, l'administrateur peut configurer le serveur pour rejeter les sessions non chiffrées, ce qui est une autre façon de vérifier l'authentification.

46.3. Types de données des message

Cette section décrit les types de données basiques utilisés dans les messages.

Int*n*(*i*)
: Un entier sur *n* bits dans l'ordre des octets réseau (octet le plus significatif en premier). Si *i* est spécifié, c'est exactement la valeur qui apparaîtra, sinon la valeur est variable, par exemple Int16, Int32(42).

Int*n*[*k*]
: Un tableau de *k* entiers sur *n* bits, tous dans l'ordre des octets réseau. La longueur *k* du tableau est toujours déterminée par un champ précédent du message, par exemple, Int16[M].

String(*s*)
: Une chaîne terminée par un octet nul (chaîne style C). Il n'y a pas de limitation sur la longueur des chaînes. Si *s* est spécifié, c'est la valeur exacte qui apparaîtra, sinon la valeur est variable. Par exemple, String("utilisateur").

> **Note**
>
> *il n'y a aucune limite prédéfinie* à la longueur d'une chaîne retournée par le serveur. Une bonne stratégie de codage de client consiste à utiliser un tampon dont la taille peut croître pour que tout ce qui tient en mémoire puisse être accepté. Si cela n'est pas faisable, il faudra lire la chaîne complète et supprimer les caractères qui ne tiennent pas dans le tampon de taille fixe.

Byte*n*(*c*)
: Exactement *n* octets. si la largeur *n* du champ n'est pas une constante, elle peut toujours être déterminée à partir d'un champ précédent du message. Si *c* est spécifié, c'est la valeur exacte. Par exemple, Byte2, Byte1('\n').

46.4. Protocole de réplication en continu

Pour initier la réplication en flux continu, le client envoi le paramètre `replication` dans son message d'ouverture. Il indique au serveur de se placer en mode walsender dans lequel en petit ensemble de commandes de réplication peuvent être utilisées à la

place d'ordres SQL. Dans le mode walsender, seul le protocole de requête simple est disponible. Les commandes acceptées en mode walsender sont:

IDENTIFY_SYSTEM
Demande au serveur de s'identifier. Le serveur répond avec un set de résultat d'une seule ligne contenant deux champs:

systemid
L'identifiant système unique du cluster. Il peut être utilisé pour vérifier que la base de sauvegarde utilisée pour initialiser le serveur en attente provient du même cluster.

timeline
TimelineID courant. Tout aussi utile pour vérifier que le serveur en attente est consistant avec le maître.

START_REPLICATION *XXX*/*XXX*
Demande au serveur de débuter l'envoi de WAL en continu, en commençant à la position *XXX*/*XXX* dans le WAL. Le serveur peu répondre avec une erreur, par exemple si la section de WAL demandée a déjà été recyclée. En cas de succès, le serveur répond avec un message CopyOutResponse et débute l'envoi en continu de WAL au client. Les WAL seront envoyé en continu jusqu'à ce que la connexion soit interrompue ; aucune autre commande ne sera acceptée.

Les données des WAL sont envoyées en une série de messages CopyData (ce qui permet d'envoyer d'autre informations dans les intervalles ; en particulier un serveur peut envoyer un message ErrorResponse s'il rencontre une erreur après la début de l'envoi en continu des données). Le contenu de chaque message CopyData suit le format suivant:

XLogData (B)

Byte1('w')
Identifie le message comme une donnée de WAL.

Byte8
Le point de départ de la donnée du WAL dans ce message, donné au format XLogRecPtr.

Byte8
Le fin courante du WAL sur le serveur, donné au format XLogRecPtr

Byte8
L'horloge système du serveur à l'heure de la transmission, donné au format TimestampTz.

Byte*n*
Une section de donnée du flux de WAL.

Une entrée de WAL n'est jamais découpée dans deux messages CopyData. Cependant, lorsqu'une entrée bute sur la fin d'une page de WAL, et est par conséquent divisée en utilisant des enregistrements de suite, elle peut être divisée sur la fin de la page. En d'autre terme, le premier enregistrement principal et les autres de suite peuvent êtres envoyés dans différents messages CopyData.

À noter que tous les champs à l'intérieur d'une donnée de WAL et les entêtes décrites précédemment seront envoyés au format natif du serveur. L'endianisme et le format de timestamp ne sont pas prévsible à moins que le receveur ait vérifié que l'identifiant système de l'émeteur corresponde au contenu de son propre `pg_control`.

Si le processus d'émission de WAL se termine correctement (pendant l'arrêt du postmaster), il enverra un message CommandComplete avant de s'arrêter. Ce message pourrait évidement ne pas être envoyé en cas d'arrêt brutal.

46.5. Formats de message

Cette section décrit le format détaillé de chaque message. Chaque message est marqué pour indiquer s'il peut être envoyé par un client (F pour *frontend*), un serveur (B pour *backend*) ou les deux (F & B). bien que chaque message commence par son nombre d'octets, le format du message est défini de telle sorte que la fin du message puisse être trouvée sans ce nombre. Cela contribue à la vérification de la validité. Le message CopyData est une exception car il constitue une partie du flux de données ; le contenu d'un message CopyData individuel n'est, en soi, pas interprétable.

AuthenticationOk (B)

Byte1('R')
Marqueur de demande d'authentification.

Int32(8)
Taille du message en octets, y compris la taille elle-même.

Int32(0)
L'authentification a réussi.

AuthenticationKerberosV5 (B)

Byte1('R')
Marqueur de demande d'authentification.

Int32(8)
Taille du message en octets, y compris la taille elle-même.

Int32(2)
Une authentification Kerberos V5 est requise.

AuthenticationCleartextPassword (B)

Byte1('R')
Marqueur de demande d'authentification.

Int32(8)
Taille du message en octets, y compris la taille elle-même.

Int32(3)
Un mot de passe en clair est requis.

AuthenticationMD5Password (B)

Byte1('R')
Marqueur de demande d'authentification.

Int32(12)
Taille du message en octets, y compris la taille elle-même.

Int32(5)
Un mot de passe chiffré par MD5 est requis.

Byte4
Composant (salt) à utiliser lors du chiffrement du mot de passe.

AuthenticationSCMCredential (B)

Byte1('R')
Marqueur de demande d'authentification.

Int32(8)
Taille du message en octets, y compris la taille elle-même.

Int32(6)
Un message d'accréditation SCM est requis.

AuthenticationGSS (B)

Byte1('R')
Identifie le message en tant que requête d'authentification.

Int32(8)
Longueur du contenu du message en octets, lui-même inclus.

Int32(7)
Spécifie qu'une authentification GSSAPI est requise.

AuthenticationSSPI (B)

Byte1('R')
Identifie le message en tant que requête d'authentification.

Int32(8)
Longueur du message en octet, incluant la longueur.

Int32(9)
Specifie que l'authentification SSPI est requise.

AuthenticationGSSContinue (B)

Byte1('R')

Identifie le message comme une requête d'authentification.

Int32
Longueur du message en octet, incluant la longueur.

Int32(8)
Spécifie que ce message contient des données GSSAPI ou SSPI.

Byte*n*
Données d'authentification GSSAPI ou SSPI.

BackendKeyData (B)

Byte1('K')
Marqueur de clé d'annulation. Le client doit sauvegarder ces valeurs s'il souhaite initier des messages CancelRequest par la suite.

Int32(12)
Taille du message en octets, y compris la taille elle-même.

Int32
ID du processus du serveur concerné.

Int32
Clé secrète du serveur concerné.

Bind (F)

Byte1('B')
Marqueur de commande Bind.

Int32
Taille du message en octets, y compris la taille elle-même.

String
Nom du portail de destination (une chaîne vide sélectionne le portail non-nommé).

String
Nom de l'instruction source préparée (une chaîne vide sélectionne l'instruction préparée non-nommée).

Int16
Nombre de codes de format de paramètres qui suivent (notés *c* ci-dessous). peut valoir zéro pour indiquer qu'il n'y a aucun paramètre ou que tous les paramètres utilisent le format par défaut (texte) ; ou un, auquel cas le code de format spécifié est appliqué à tous les paramètres ; il peut aussi être égal au nombre courant de paramètres.

Int16[*c*]
Codes de format des paramètres. Tous doivent valoir zéro (texte) ou un (binaire).

Int16
Nombre de valeurs de paramètres qui suivent (peut valoir zéro). Cela doit correspondre au nombre de paramètres nécessaires à la requête.
Puis, le couple de champs suivant apparaît pour chaque paramètre : paramètre :

Int32
Taille de la valeur du paramètre, en octets (ce nombre n'inclut pas la longueur elle-même). Peut valoir zéro. Traité comme un cas spécial, -1 indique une valeur de paramètre NULL. Aucun octet de valeur ne suit le cas NULL.

Byte*n*
Valeur du paramètre, dans le format indiqué par le code de format associé. *n* est la longueur ci-dessus.
Après le dernier paramètre, les champs suivants apparaissent :

Int16
Nombre de codes de format des colonnes de résultat qui suivent (noté *r* ci-dessous). peut valoir zéro pour indiquer qu'il n'y a pas de colonnes de résultat ou que les colonnes de résultat utilisent le format par défaut (texte) ; ou une, auquel cas le code de format spécifié est appliqué à toutes les colonnes de résultat (s'il y en a) ; il peut aussi être égal au nombre de colonnes de résultat de la requête.

Int16[*r*]
Codes de format des colonnes de résultat. Tous doivent valoir zéro (texte) ou un (binaire).

BindComplete (B)

Byte1('2')

Indicateur de Bind complet.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

CancelRequest (F)

Int32(16)
Taille du message en octets, y compris la taille elle-même.

Int32(80877102)
Code d'annulation de la requête. La valeur est choisie pour contenir `1234` dans les 16 bits les plus significatifs et `5678` dans les 16 bits les moins significatifs (pour éviter toute confusion, ce code ne doit pas être le même qu'un numéro de version de protocole).

Int32
ID du processus du serveur cible.

Int32
Clé secrète du serveur cible.

Close (F)

Byte1('C')
Marqueur de commande Close.

Int32
Taille du message en octets, y compris la taille elle-même.

Byte1
'`s`' pour fermer une instruction préparée ; ou '`p`' pour fermer un portail.

String
Nom de l'instruction préparée ou du portail à fermer (une chaîne vide sélectionne l'instruction préparée ou le portail non-nommé(e)).

CloseComplete (B)

Byte1('3')
Indicateur de complétude de Close.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

CommandComplete (B)

Byte1('C')
Marqueur de réponse de complétude de commande.

Int32
Taille du message en octets, y compris la taille elle-même.

String
Balise de la commande. Mot simple identifiant la commande SQL terminée.

Pour une commande **insert**, la balise est `insert` *`oid`* *`lignes`* où *`lignes`* est le nombre de lignes insérées. *`oid`* est l'id de l'objet de la ligne insérée si *`lignes`* vaut 1 et que la table cible a des OID ; sinon *`oid`* vaut 0.

Pour une commande **delete**, la balise est `delete` *`lignes`* où *`lignes`* est le nombre de lignes supprimées.

Pour une commande **update**, la balise est `update` *`lignes`* où *`lignes`* est le nombre de lignes mises à jour.

Pour les commandes **SELECT** ou **CREATE TABLE AS**, la balise est `SELECT` *`lignes`* où *`lignes`* est le nombre de ligne récupérées.

Pour une commande **move**, la balise est `move` *`lignes`* où *`lignes`* est le nombre de lignes de déplacement du curseur.

Pour une commande **fetch**, la balise est `fetch` *`lignes`* où *`lignes`* est le nombre de lignes récupérées à partir du curseur.

CopyData (F & B)

Byte1('d')
Marqueur de données de COPY.

Int32

Taille du message en octets, y compris la taille elle-même.

Byte*n*
Données formant une partie d'un flux de données **copy**. les messages envoyés depuis le serveur correspondront toujours à des lignes uniques de données, mais les messages envoyés par les clients peuvent diviser le flux de données de façon arbitraire.

CopyDone (F & B)

Byte1('c')
Indicateur de fin de COPY.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

CopyFail (F)

Byte1('f')
Indicateur d'échec de COPY.

Int32
Taille du message en octets, y compris la taille elle-même.

String
Message d'erreur rapportant la cause d'un échec.

CopyInResponse (B)

Byte1('G')
Marqueur de réponse de Start Copy In. Le client doit alors envoyer des données de copie (s'il n'est pas à cela, il enverra un message CopyFail).

Int32
Taille du message en octets, y compris la taille elle-même.

Int8
0 indique que le format de copie complet est textuel (lignes séparées par des retours chariot, colonnes séparées par des caractères de séparation, etc). 1 indique que le format de copie complet est binaire (similaire au format DataRow). Voir COPY(7) pour plus d'informations.

Int16
Nombre de colonnes dans les données à copier (noté *n* ci-dessous).

Int16[*n*]
Codes de format à utiliser pour chaque colonne. Chacun doit valoir zéro (texte) ou un (binaire). Tous doivent valoir zéro si le format de copie complet est de type texte.

CopyOutResponse (B)

Byte1('H')
Marqueur de réponse Start Copy Out. Ce message sera suivi de données copy-out.

Int32
Taille du message en octets, y compris la taille elle-même.

Int8
0 indique que le format de copie complet est textuel (lignes séparées par des retours chariots, colonnes séparées par des caractères séparateur, etc). 1 indique que le format de copie complet est binaire (similaire au format DataRow). Voir COPY(7) pour plus d'informations.

Int16
Nombre de colonnes de données à copier (noté *n* ci-dessous).

Int16[*n*]
Codes de format à utiliser pour chaque colonne. Chaque code doit valoir zéro (texte) ou un (binaire). Tous doivent valoir zéro si le format de copie complet est de type texte.

DataRow (B)

Byte1('D')
Marqueur de ligne de données.

Int32
Taille du message en octets, y compris la taille elle-même.

Int16
Nombre de valeurs de colonnes qui suivent (peut valoir zéro).
Apparaît ensuite le couple de champs suivant, pour chaque colonne :

Int32
Longueur de la valeur de la colonne, en octets (ce nombre n'inclut pas la longueur elle-même). Peut valoir zéro. Traité comme un cas spécial, -1 indique une valeur NULL de colonne. Aucun octet de valeur ne suit le cas NULL.

Byte*n*
Valeur de la colonne dans le format indiqué par le code de format associé. *n* est la longueur ci-dessus.

Describe (F)

Byte1('D')
Marqueur de commande Describe.

Int32
Taille du message en octets, y compris la taille elle-même.

Byte1
'S' pour décrire une instruction préparée ; ou 'P' pour décrire un portail.

String
Nom de l'instruction préparée ou du portail à décrire (une chaîne vide sélectionne l'instruction préparée ou le portail non-nommé(e)).

EmptyQueryResponse (B)

Byte1('I')
Marqueur de réponse à une chaîne de requête vide (c'est un substitut de CommandComplete).

Int32(4)
Taille du message en octets, y compris la taille elle-même.

ErrorResponse (B)

Byte1('E')
Marqueur d'erreur.

Int32
Taille du message en octets, y compris la taille elle-même.
Le corps du message est constitué d'un ou plusieurs champs identifié(s), suivi(s) d'un octet nul comme délimiteur de fin. L'ordre des champs n'est pas fixé. Pour chaque champ, on trouve les informations suivantes :

Byte1
Code identifiant le type de champ ; s'il vaut zéro, c'est la fin du message et aucune chaîne ne suit. Les types de champs définis sont listés dans Section 46.6, « Champs des messages d'erreur et d'avertissement ». de nouveaux types de champs pourraient être ajoutés dans le futur, les clients doivent donc ignorer silencieusement les types non reconnus.

String
Valeur du champ.

Execute (F)

Byte1('E')
Marqueur de commande Execute.

Int32
Taille du message en octets, y compris la taille elle-même.

String
Nom du portail à exécuter (une chaîne vide sélectionne le portail non-nommé).

Int32
Nombre maximum de lignes à retourner si le portail contient une requête retournant des lignes (ignoré sinon). Zéro signifie « aucune limite ».

Flush (F)

Byte1('H')
Marqueur de commande Flush.

Int32(4)

Taille du message en octets, y compris la taille elle-même.

FunctionCall (F)

Byte1('F')
Marqueur d'appel de fonction.

Int32
Taille du message en octets, y compris la taille elle-même.

Int32
Spécifie l'ID de l'objet représentant la fonction à appeler.

Int16
Nombre de codes de format de l'argument qui suivent (noté *c* ci-dessous). cela peut être zéro pour indiquer qu'il n'y a pas d'arguments ou que tous les arguments utilisent le format par défaut (texte) ; un, auquel cas le code de format est appliqué à tous les arguments ; il peut aussi être égal au nombre réel d'arguments.

Int16[*c*]
Les codes de format d'argument. Chacun doit valoir zéro (texte) ou un (binaire).

Int16
Nombre d'arguments fournis à la fonction.
Apparaît ensuite, pour chaque argument, le couple de champs suivant :

Int32
Longueur de la valeur de l'argument, en octets (ce nombre n'inclut pas la longueur elle-même). Peut valoir zéro. Traité comme un cas spécial, -1 indique une valeur NULL de l'argument. Aucun octet de valeur ne suit le cas NULL.

Byte*n*
Valeur de l'argument dans le format indiqué par le code de format associé. *n* est la longueur ci-dessus.
Après le dernier argument, le champ suivant apparaît :

Int16
Code du format du résultat de la fonction. Doit valoir zéro (texte) ou un (binaire).

FunctionCallResponse (B)

Byte1('V')
Marqueur de résultat d'un appel de fonction.

Int32
Taille du message en octets, y compris la taille elle-même.

Int32
Longueur de la valeur du résultat de la fonction, en octets (ce nombre n'inclut pas la longueur elle-même). Peut valoir zéro. Traité comme un cas spécial, -1 indique un résultat de fonction NULL. Aucun octet de valeur ne suit le cas NULL.

Byte*n*
Valeur du résultat de la fonction, dans le format indiqué par le code de format associé. *n* est la longueur ci-dessus.

NoData (B)

Byte1('n')
Indicateur d'absence de données.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

NoticeResponse (B)

Byte1('N')
Marqueur d'avertissement.

Int32
Taille du message en octets, y compris la taille elle-même.
Le corps du message est constitué d'un ou plusieurs champs identifié(s), suivi(s) d'un octet zéro comme délimiteur de fin. L'ordre des champs n'est pas fixé. Pour chaque champ, on trouve les informations suivantes :

Byte1
Code identifiant le type de champ ; s'il vaut zéro, c'est la fin du message et aucune chaîne ne suit. Les types de champs déjà définis sont listés dans Section 46.6, « Champs des messages d'erreur et d'avertissement ». de nouveaux types de champs pourraient être ajoutés dans le futur, les clients doivent donc ignorer silencieusement les champs de type non reconnu.

String
Valeur du champ.

NotificationResponse (B)

Byte1('A')
Marqueur de réponse de notification.

Int32
Taille du message en octets, y compris la taille elle-même.

Int32
ID du processus serveur ayant procédé à la notification.

String
Nom du canal à l'origine de la notification.

String
La chaîne « embarquée » passée lors de la notification

ParameterDescription (B)

Byte1('t')
Marqueur de description de paramètre.

Int32
Taille du message en octets, y compris la taille elle-même.

Int16
Nombre de paramètres utilisé par l'instruction (peut valoir zéro).
Pour chaque paramètre, suivent :

Int32
ID de l'objet du type de données du paramètre.

ParameterStatus (B)

Byte1('S')
Marqueur de rapport d'état de paramètre d'exécution.

Int32
Taille du message en octets, y compris la taille elle-même.

String
Nom du paramètre d'exécution dont le rapport est en cours.

String
Valeur actuelle du paramètre.

Parse (F)

Byte1('P')
Marqueur de commande Parse.

Int32
Taille du message en octets, y compris la taille elle-même.

String
Nom de l'instruction préparée de destination (une chaîne vide sélectionne l'instruction préparée non-nommée).

String
Chaîne de requête à analyser.

Int16
Nombre de types de données de paramètre spécifiés (peut valoir zéro). Ce n'est pas une indication du nombre de paramètres pouvant apparaître dans la chaîne de requête, mais simplement le nombre de paramètres pour lesquels le client veut pré-spécifier les types.
Pour chaque paramètre, on trouve ensuite :

Int32
ID de l'objet du type de données du paramètre. la valeur zéro équivaut à ne pas spécifier le type.

ParseComplete (B)

Byte1('1')
Indicateur de fin de Parse.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

PasswordMessage (F)

Byte1('p')
Identifie le message comme une réponse à un mot de passe. Notez que c'est aussi utilisé par les messages de réponse GSSAPI et SSPI (qui est vraiment une erreur de conception car les données contenues ne sont pas une chaîne terminée par un octet nul dans ce cas, mais peut être une donnée binaire arbitraire).

Int32
Taille du message en octets, y compris la taille elle-même.

String
Mot de passe (chiffré à la demande).

PortalSuspended (B)

Byte1('s')
Indicateur de suspension du portail. Apparaît seulement si la limite du nombre de lignes d'un message Execute a été atteint.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

Query (F)

Byte1('Q')
Marqueur de requête simple.

Int32
Taille du message en octets, y compris la taille elle-même.

String
La chaîne de requête elle-même.

ReadyForQuery (B)

Byte1('Z')
Identifie le type du message. ReadyForQuery est envoyé à chaque fois que le serveur est prêt pour un nouveau cycle de requêtes.

Int32(5)
Taille du message en octets, y compris la taille elle-même.

Byte1
Indicateur de l'état transactionnel du serveur. Les valeurs possibles sont '`i`' s'il est en pause (en dehors d'un bloc de transaction) ; '`t`' s'il est dans un bloc de transaction ; ou '`e`' s'il est dans un bloc de transaction échouée (les requêtes seront rejetées jusqu'à la fin du bloc).

RowDescription (B)

Byte1('T')
Marqueur de description de ligne.

Int32
Taille du message en octets, y compris la taille elle-même.

Int16
Nombre de champs dans une ligne (peut valoir zéro).
On trouve, ensuite, pour chaque champ :

String
Nom du champ.

Int32
Si le champ peut être identifié comme colonne d'une table spécifique, l'ID de l'objet de la table ; sinon zéro.

Int16
Si le champ peut être identifié comme colonne d'une table spécifique, le numéro d'attribut de la colonne ; sinon zéro.

Int32
ID de l'objet du type de données du champ.

Int16
Taille du type de données (voir `pg_type.typlen`). Les valeurs négatives indiquent des types de largeur variable.

Int32
Modificateur de type (voir `pg_attribute.atttypmod`). La signification du modificateur est spécifique au type.

Int16
Code de format utilisé pour le champ. Zéro (texte) ou un (binaire), à l'heure actuelle. Dans un RowDescription retourné par la variante de l'instruction de Describe, le code du format n'est pas encore connu et vaudra toujours zéro.

SSLRequest (F)

Int32(8)
Taille du message en octets, y compris la taille elle-même.

Int32(80877103)
Code de requête ssl. la valeur est choisie pour contenir `1234` dans les 16 bits les plus significatifs, et `5679` dans les 16 bits les moins significatifs (pour éviter toute confusion, ce code ne doit pas être le même que celui d'un numéro de version de protocole).

StartupMessage (F)

Int32
Taille du message en octets, y compris la taille elle-même.

Int32(196608)
Numéro de version du protocole. Les 16 bits les plus significatifs représentent le numéro de version majeure (3 pour le protocole décrit ici). Les 16 bits les moins significatifs représentent le numéro de version mineure (0 pour le protocole décrit ici).
Le numéro de version du protocole est suivi par un ou plusieurs couple(s) nom de paramètre et chaîne de valeur. Un octet zéro est requis comme délimiteur de fin après le dernier couple nom/valeur. L'ordre des paramètres n'est pas fixé. Le paramètre `user` est requis, les autres sont optionnels. Chaque paramètre est spécifié de la façon suivante :

String
Nom du paramètre. Les noms actuellement reconnus sont :

`user`
Nom de l'utilisateur de base de données sous lequel se connecter. Requis ; il n'y a pas de valeur par défaut.

`database`
Base de données à laquelle se connecter. Par défaut le nom de l'utilisateur.

`options`
Arguments en ligne de commande pour le serveur (rendu obsolète par l'utilisation de paramètres individuels d'exécution).
En plus de ce qui précède, tout paramètre d'exécution pouvant être initialisé au démarrage du serveur peut être listé. Ces paramètres seront appliqués au démarrage du serveur (après analyse des options en ligne de commande, s'il y en a). Leurs valeurs agiront comme valeurs de session par défaut.

String
Valeur du paramètre.

Sync (F)

Byte1('S')
Marqueur de commande Sync.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

Terminate (F)

Byte1('X')
Marqueur de fin.

Int32(4)
Taille du message en octets, y compris la taille elle-même.

46.6. Champs des messages d'erreur et d'avertissement

Cette section décrit les champs qui peuvent apparaître dans les messages ErrorResponse et NoticeResponse. Chaque type de champ a un motif d'identification codé sur un octet. Tout type de champ donné doit apparaître au plus une fois par message.

`s`
: Gravité (Severity) : le contenu du champ peut être `error`, `fatal` ou `panic` dans un message d'erreur, `warning`, `notice`, `debug`, `info` ou `log` dans un message d'avertissement, ou la traduction régionale de l'un d'eux. Toujours présent.

`c`
: Code : code SQLSTATE de l'erreur (voir Annexe A, Codes d'erreurs de PostgreSQL™). non internationalisable. Toujours présent.

`m`
: Message : premier message d'erreur, en clair. Doit être court et précis (typiquement une ligne). Toujours présent.

`d`
: Détail : deuxième message d'erreur, optionnel, apportant des informations supplémentaires sur le problème. Peut être sur plusieurs lignes.

`h`
: Astuce (Hint) : suggestion optionnelle de résolution du problème. Différent de Détail parce qu'il offre un conseil (potentiellement inapproprié) plutôt que des faits réels. Peut être sur plusieurs lignes.

`p`
: Position : valeur du champ, entier décimal ASCII indiquant un curseur sur la position de l'erreur dans la chaîne de requête originale. Le premier caractère a l'index 1. Les positions sont mesurées en caractères, non pas en octets.

`p`
: Position interne : ceci est défini de la même façon que le champ `p` mais c'est utilisé quand la position du curseur se réfère à une commande générée en interne plutôt qu'une soumise par le client. Le champ `q` apparaîtra toujours quand ce champ apparaît.

`q`
: Requête interne : le texte d'une commande générée en interne et qui a échoué. Ceci pourrait être, par exemple, une requête SQL lancée par une fonction PL/pgSQL.

`w`
: Où (Where) : indication du contexte dans lequel l'erreur est survenue. Inclut, actuellement, une trace de la pile des appels des fonctions PL actives. Cette trace comprend une entrée par ligne, la plus récente en premier.

`f`
: Fichier (File) : nom du fichier de code source comportant l'erreur.

`l`
: Ligne (Line) : numéro de ligne dans le fichier de code source comportant l'erreur.

`r`
: Routine : nom de la routine dans le code source comportant l'erreur.

Le client est responsable du formatage adapté à ses besoins des informations affichées ; en particulier par l'ajout de retours chariots sur les lignes longues, si cela s'avérait nécessaire. Les caractères de retour chariot apparaissant dans les champs de messages d'erreur devraient être traités comme des changements de paragraphes, non comme des changements de lignes.

46.7. Résumé des modifications depuis le protocole 2.0

Cette section fournit une liste rapide des modifications à l'attention des développeurs essayant d'adapter au protocole 3.0 des bibliothèques clientes existantes.

Le paquet de démarrage initial utilise un format flexible de liste de chaînes au lieu d'un format fixe. Les valeurs de session par défaut des paramètres d'exécution peuvent même être spécifiées directement dans le paquet de démarrage (en fait, cela était déjà possible en utilisant le champ `options` ; mais étant donné la largeur limitée d'`options` et l'impossibilité de mettre entre guillemets les espaces fines dans les valeurs, ce n'était pas une technique très sûre).

Tous les messages possèdent désormais une indication de longueur qui suit immédiatement l'octet du type de message (sauf pour les paquets de démarrage qui n'ont pas d'octet de type). PasswordMessage possède à présent un octet de type.

Les messages ErrorResponse et NoticeResponse ('`e`' et '`n`') contiennent maintenant plusieurs champs, à partir desquels le code client peut assembler un message d'erreur fonction du niveau de verbiage désiré. Des champs individuels ne se termineront plus par un retour chariot alors que la chaîne seule envoyée dans l'ancien protocole le faisait systématiquement.

Le message ReadyForQuery ('z') inclut un indicateur d'état de la transaction.

La distinction entre les types de messages BinaryRow et DataRow est supprimée ; le type de message DataRow seul sert à retourner les données dans tous les formats. La disposition de DataRow a changé pour faciliter son analyse. La représentation des valeurs binaires a également été modifiée : elle n'est plus liée directement à la représentation interne du serveur.

Il existe un nouveau sous-protocole pour les « requêtes étendues » qui ajoute les types de messages client Parse, Bind, Execute, Describe, Close, Flush et Sync et les types de messages serveur ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData et CloseComplete. Les clients existants ne sont pas directement concernés par ce sous-protocole, mais son utilisation apportera des améliorations en terme de performances et de fonctionnalités.

Les données de **copy** sont désormais encapsulées dans des messages CopyData et CopyDone. Il y a une façon bien définie de réparer les erreurs lors du **copy**. la dernière ligne spéciale « `\.` » n'est plus nécessaire et n'est pas envoyée lors de **copy out** (elle est toujours reconnue comme un indicateur de fin lors du **copy in** mais son utilisation est obsolète. Elle sera éventuellement supprimée). Le **copy** binaire est supporté. les messages copyinresponse et CopyOutResponse incluent les champs indiquant le nombre de colonnes et le format de chaque colonne.

La disposition des messages FunctionCall et FunctionCallResponse a changé. FunctionCall supporte à présent le passage aux fonctions d'arguments NULL. Il peut aussi gérer le passage de paramètres et la récupération de résultats aux formats texte et binaire. Il n'y a plus aucune raison de considérer FunctionCall comme une faille potentielle de sécurité car il n'offre plus d'accès direct aux représentations internes des données du serveur.

Le serveur envoie des messages ParameterStatus ('s') lors de l'initialisation de la connexion pour tous les paramètres qu'il considère intéressants pour la bibliothèque client. En conséquence, un message ParameterStatus est envoyé à chaque fois que la valeur active d'un de ces paramètres change.

Le message RowDescription ('t') contient les nouveaux champs oid de table et de numéro de colonne pour chaque colonne de la ligne décrite. Il affiche aussi le code de format pour chaque colonne.

Le message CursorResponse ('p') n'est plus engendré par le serveur.

Le message NotificationResponse ('a') a un champ de type chaîne supplémentaire qui peu « embarquer » une chaîne passée par l'émetteur de l'événement **notify**.

Le message EmptyQueryResponse ('i') nécessitait un paramètre chaîne vide ; ce n'est plus le cas.

Chapitre 47. Conventions de codage pour PostgreSQL

47.1. Formatage

Le formatage du code source utilise un espacement de quatre colonnes pour les tabulations, avec préservation de celles-ci (c'est-à-dire que les tabulations ne sont pas converties en espaces). Chaque niveau logique d'indentation est une tabulation supplémentaire.

Les règles de disposition (positionnement des parenthèses, etc) suivent les conventions BSD. En particulier, les accolades pour les blocs de contrôle `if`, `while`, `switch`, etc ont leur propre ligne.

Ne pas utiliser les commentaires style C++ (`//`). Les compilateurs C ANSI stricts ne les acceptent pas. Pour la même raison, ne pas utiliser les extensions C++ comme la déclaration de nouvelles variables à l'intérieur d'un bloc.

Le style préféré pour les blocs multilignes de commentaires est :

```
/*
 * le commentaire commence ici
 * et continue ici
 */
```

Notez que les blocs de commentaire commençant en colonne 1 seront préservés par pgindent, mais qu'il déplacera (au niveau de la colonne) les blocs de commentaires indentés comme tout autre texte. Si vous voulez préserver les retours à la ligne dans un bloc indenté, ajoutez des tirets comme ceci :

```
    /*----------
     * le commentaire commence ici
     * et continue ici
     *----------
     */
```

Bien que les correctifs (patchs) soumis ne soient absolument pas tenus de suivre ces règles de formatage, il est recommandé de le faire. Le code est passé dans pgindent avant la sortie de la prochaine version, donc il n'y a pas de raison de l'écrire avec une autre convention de formatage.

Le répertoire `src/tools` contient des fichiers d'exemples de configuration qui peuvent être employés avec les éditeurs emacs™, xemacs™ ou vim™ pour valider que le format du code écrit respecte ces conventions.

Les outils de parcours de texte more et less peuvent être appelés de la manière suivante :

```
more -x4
less -x4
```

pour qu'ils affichent correctement les tabulations.

47.2. Reporter les erreurs dans le serveur

Les messages d'erreurs, d'alertes et de traces produites dans le code du serveur doivent être créés avec `ereport` ou son ancien cousin `elog`. L'utilisation de cette fonction est suffisamment complexe pour nécessiter quelques explications.

Il y a deux éléments requis pour chaque message : un niveau de sévérité (allant de `DEBUG` à `PANIC`) et un message texte primaire. De plus, il y a des éléments optionnels, le plus commun d'entre eux est le code identifiant de l'erreur qui suit les conventions SQLSTATE des spécifications SQL. `ereport` en elle-même n'est qu'une fonction shell qui existe principalement pour des convenances syntaxiques faisant ressembler la génération de messages à l'appel d'une fonction dans un code source C. Le seul paramètre directement accepté par `ereport` est le niveau de sévérité. Le message texte primaire et les autres éléments de messages optionnels sont produits par appel de fonctions auxiliaires, comme `errmsg`, dans l'appel à `ereport`.

Un appel typique à `ereport` peut ressembler à :

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

Le niveau de sévérité de l'erreur est ainsi positionné à `ERROR` (une erreur banale). L'appel à `errcode` précise l'erreur SQL-

STATE en utilisant une macro définie dans `src/include/utils/errcodes.h`. L'appel à `errmsg` fournit le message texte primaire. L'ensemble supplémentaire de parenthèses entourant les appels aux fonctions auxiliaires est ennuyeux mais syntaxiquement nécessaire.

Exemple plus complexe :

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      NIL, actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You might need to add explicit typecasts.")));
```

Cela illustre l'utilisation des codes de formatage pour intégrer des valeurs d'exécution dans un message texte. Un message « conseil », optionnel, est également fourni.

Les routines auxiliaires disponibles pour `ereport` sont :

- `errcode(sqlerrcode)` précise le code SQLSTATE de l'identifiant erreur pour la condition. Si cette routine n'est pas appelée, l'identifiant l'erreur est, par défaut, `ERRCODE_INTERNAL_ERROR` quand le niveau de sévérité de l'erreur est `ERROR` ou plus haut, `ERRCODE_WARNING` quand le niveau d'erreur est `WARNING` et `ERRCODE_SUCCESSFUL_COMPLETION` pour `NOTICE` et inférieur. Bien que ces valeurs par défaut soient souvent commodes, il faut se demander si elles sont appropriées avant d'omettre l'appel à `errcode()`.
- `errmsg(const char *msg, ...)` indique le message texte primaire de l'erreur et les possibles valeurs d'exécutions à y insérer. Les insertions sont précisées par les codes de formatage dans le style `sprintf`. En plus des codes de formatage standard acceptés par `sprintf`, le code `%m` peut être utilisé pour insérer le message d'erreur retourné par `strerror` pour la valeur courante de `errno`. [1] `%m` ne nécessite aucune entrée correspondante dans la liste de paramètres pour `errmsg`. Notez que la chaîne de caractères du message sera passée à travers `gettext` pour une possible adaptation linguistique avant que les codes de formatage ne soient exécutés.
- `errmsg_internal(const char *msg, ...)` fait la même chose que `errmsg` à l'exception que la chaîne de caractères du message ne sera ni traduite ni incluse dans le dictionnaire de messages d'internationalisation. Cela devrait être utilisé pour les cas qui « ne peuvent pas arriver » et pour lesquels il n'est probablement pas intéressant de déployer un effort de traduction.
- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` est identique à `errmsg` mais avec le support pour plusieurs formes de pluriel du message. *`fmt_singular`* est le format singulier de l'anglais, *`fmt_plural`* est le format pluriel en anglais, *`n`* est la valeur entière qui détermine la forme utilisée. Les arguments restants sont formatés suivant le chaîne de format sélectionnée. Pour plus d'informations, voir Section 48.2.2, « Guide d'écriture des messages ».
- `errdetail(const char *msg, ...)` fournit un message « détail » optionnel ; cela est utilisé quand il y a des informations supplémentaires qu'il semble inadéquat de mettre dans le message primaire. La chaîne de caractères du message est traitée de la même manière que celle de `errmsg`.
- `errdetail_log(const char *msg, ...)` est identique à `errdetail` sauf que cette chaîne ne va que dans les traces du serveur. Elle n'est jamais envoyée au client. Si `errdetail` et `errdetail_log` sont utilisées ensemble, alors une chaîne est envoyés au client et l'autre dans les traces du serveur. C'est utile pour les détails d'erreur qui concernent la sécurité ou qui sont trop techniques pour être inclus dans le rapport envoyé au client.
- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` est identique à `errdetail` mais avec le support de plusieurs formes de pluriel pour le message. Pour plus d'information, voir Section 48.2.2, « Guide d'écriture des messages ».
- `errhint(const char *msg, ...)` fournit un message « conseil » optionnel ; cela est utilisé pour offrir des suggestions sur la façon de régler un problème, par opposition aux détails effectifs au sujet de ce qui a mal tourné. La chaîne de caractères du message est traitée de la même manière que celle de `errmsg`.
- `errcontext(const char *msg, ...)` n'est normalement pas appelée directement depuis un site de message de `ereport` mais plutôt elle est utilisée dans les fonctions de rappels `error_context_stack` pour fournir des informations à propos du contexte dans lequel une erreur s'est produite, comme les endroits courants dans la fonction PL. La chaîne de caractères du message est traitée de la même manière que celle de `errmsg`. À l'inverse des autres fonctions auxiliaires, celle-ci peut être appelée plus d'une fois dans un appel de `ereport` ; les chaînes successives ainsi fournies sont concaténées et séparées pas des caractères d'interlignes (NL).
- `errposition(int cursorpos)` spécifie l'endroit textuel d'une erreur dans la chaîne de caractères de la requête. Actuellement, c'est seulement utile pour les erreurs détectées dans les phases d'analyses lexicales et syntaxiques du traitement de la

[1] C'est-à-dire la valeur qui était courante quand l'appel à `ereport` a été atteinte ; les changements d'`errno` dans les routines auxiliaires de rapports ne l'affecteront pas. Cela ne sera pas vrai si vous devez écrire explicitement `strerror(errno)` dans la liste de paramètres de `errmsg` ; en conséquence ne faites pas comme ça.

requête.

- `errcode_for_file_access()` est une fonction commode qui sélectionne l'identifiant d'erreur SQLSTATE approprié pour une défaillance dans l'appel système relatif à l'accès d'un fichier. Elle utilise le `errno` sauvegardé pour déterminer quel code d'erreur générer. Habituellement cela devrait être utilisé en combinaison avec `%m` dans le texte du message d'erreur primaire.
- `errcode_for_socket_access()` est une fonction commode qui sélectionne l'identifiant d'erreur SQLSTATE approprié pour une défaillance dans l'appel système relatif à une socket.
- `errhidestmt(bool hide_stmt)` peut être appelé pour indiquer la suppression de la portion `STATEMENT:` d'un message dans le journal applicatif de postmaster. Habituellement, c'est approprié si le texte du message contient déjà l'instruction en cours.

Il y a une plus ancienne fonction nommée `elog`, qui est toujours largement utilisée. Un appel à `elog` :

```
elog(niveau, "chaine format", ...);
```

est strictement équivalent à :

```
ereport(level, (errmsg_internal("chaine format", ...)));
```

Le code d'erreur SQLSTATE est toujours celui par défaut, la chaîne de caractères du message n'est pas sujette à traduction. Par conséquent, `elog` ne devrait être utilisé que pour les erreurs internes et l'enregistrement de trace de débogage de bas niveau. N'importe quel message susceptible d'intéresser les utilisateurs ordinaires devrait passer par `ereport`. Néanmoins, il y a suffisamment de contrôles des erreurs internes qui « ne peuvent pas arriver » dans le système, pour que `elog` soit toujours largement utilisée ; elle est préférée pour ces messages du fait de sa simplicité d'écriture.

Des conseils sur l'écriture de bons messages d'erreurs peuvent être trouvés dans la Section 47.3, « Guide de style des messages d'erreurs ».

47.3. Guide de style des messages d'erreurs

Ce guide de style est fournit dans l'espoir de maintenir une cohérence et un style facile à comprendre dans tous les messages générés par PostgreSQL™.

Ce qui va où

Le message primaire devrait être court, factuel et éviter les références aux détails d'exécution comme le nom de fonction spécifique. « Court » veut dire « devrait tenir sur une ligne dans des conditions normales ». Utilisez un message détail si nécessaire pour garder le message primaire court ou si vous sentez le besoin de mentionner les détails de l'implémentation comme un appel système particulier qui échoue. Les messages primaires et détails doivent être factuels. Utilisez un message conseil pour les suggestions à propos de quoi faire pour fixer le problème, spécialement si la suggestion ne pourrait pas toujours être applicable.

Par exemple, au lieu de :

```
IpcMemoryCreate: shmget(clé=%d, taille=%u, 0%o) a échoué : %m
(plus un long supplément qui est basiquement un conseil)
```

écrivez :

```
Primaire:    Ne peut pas créer un ségment en mémoire partagée : %m
Détail:      L'appel système qui a échoué était shmget(key=%d, size=%u, 0%o).
Astuce:     Le supplément
```

Raisonnement : garder le message primaire court aide à le garder au point et laisse les clients présenter un espace à l'écran sur la supposition qu'une ligne est suffisante pour les messages d'erreurs. Les messages détails et conseils peuvent être relégués à un mode verbeux ou peut-être dans une fenêtre pop-up détaillant l'erreur. De plus, les détails et les conseils devront normalement être supprimés des traces du serveur pour gagner de l'espace. La référence aux détails d'implémentation est à éviter puisque les utilisateurs n'en connaissent de toute façon pas les détails.

Formatage

N'émettez pas d'hypothèses spécifiques à propos du formatage dans les messages textes. Attendez-vous à ce que les clients et les traces du serveur enveloppent les lignes pour correspondre à leurs propres besoins. Dans les messages longs, les caractères d'interlignes (\n) peuvent être utilisés pour indiquer les coupures suggérées d'un paragraphe. Ne terminez pas un message avec un caractère d'interlignes. N'utilisez pas des tabulations ou d'autres caractères de formatage (dans les affichages des contextes d'erreurs, les caractères d'interlignes sont automatiquement ajoutés pour séparer les niveaux d'un contexte comme dans les appels

aux fonctions).

Raisonnement : les messages ne sont pas nécessairement affichés dans un affichage de type terminal. Dans les interfaces graphiques ou les navigateurs, ces instructions de formatage sont, au mieux, ignorées.

Guillemets

Les textes en anglais devraient utiliser des guillemets doubles quand la mise entre guillemets est appropriée. Les textes dans les autres langues devraient uniformément employer un genre de guillemets qui est conforme aux coutumes de publication et à la sortie visuelle des autres programmes.

Raisonnement : le choix des guillemets doubles sur celui des guillemets simples est quelque peu arbitraire mais tend à être l'utilisation préférée. Certains ont suggéré de choisir le type de guillemets en fonction du type d'objets des conventions SQL (notamment, les chaînes de caractères entre guillemets simples, les identifiants entre guillemets doubles). Mais ceci est un point technique à l'intérieur du langage avec lequel beaucoup d'utilisateurs ne sont pas familiers ; les conventions SQL ne prennent pas en compte les autres genres de termes entre guillemets, ne sont pas traduites dans d'autres langues et manquent un peu de sens aussi.

Utilisation des guillemets

Utilisez toujours les guillemets pour délimiter les noms de fichiers, les identifiants fournis par les utilisateurs et les autres variables qui peuvent contenir des mots. Ne les utilisez pas pour marquer des variables qui ne contiennent pas de mots (par exemple, les noms d'opérateurs).

Il y a des fonctions au niveau du serveur qui vont, au besoin, mettre entre guillemets leur propre flux de sortie (par exemple, `format_type_be()`). Ne mettez pas de guillemets supplémentaires autour du flux de sortie de ce genre de fonctions.

Raisonnement : les objets peuvent avoir un nom qui crée une ambiguïté une fois incorporé dans un message. Soyez prudent en indiquant où un nom commence et fini. Mais n'encombrez pas les messages avec des guillemets qui ne sont pas nécessaires ou qui sont dupliqués.

Grammaire et ponctuation

Les règles sont différentes pour les messages d'erreurs primaires et pour les messages détails/conseils :

Messages d'erreurs primaires : ne mettez pas en majuscule la première lettre. Ne terminez pas un message avec un point. Ne pensez même pas à finir un message avec un point d'exclamation.

Messages détails et conseils : utilisez des phrases complètes et toutes terminées par des points. Mettez en majuscule le premier mot des phrases. Placez deux espaces après le point si une autre phrase suit (pour un texte en anglais... cela pourrait être différent dans une autre langue).

Chaînes de contexte d'erreur: Ne mettez pas en majuscule la première lettre et ne terminer pas la chaîne avec un point. Les chaînes de contexte ne sont normalement pas des phrases complètes.

Raisonnement : éviter la ponctuation rend plus facile, pour les applications clientes, l'intégration du message dans des contextes grammaticaux variés. Souvent, les messages primaires ne sont de toute façon pas des phrases complètes (et s'ils sont assez longs pour être sur plusieurs phrases, ils devraient être divisés en une partie primaire et une partie détail). Cependant, les messages détails et conseils sont longs et peuvent avoir besoin d'inclure de nombreuses phrases. Pour la cohérence, ils devraient suivre le style des phrases complètes même s'il y a seulement une phrase.

Majuscule contre minuscule

Utilisez les minuscules pour les mots d'un message, inclus la première lettre d'un message d'erreur primaire. Utilisez les majuscules pour les commandes et les mots-clé SQL s'ils apparaissent dans le message.

Raisonnement : il est plus facile de rendre toutes les choses plus cohérentes au regard de cette façon, puisque certains messages sont des phrases complètes et d'autres non.

Éviter la voix passive

Utilisez la voix active. Utilisez des phrases complètes quand il y a un sujet (« A ne peut pas faire B »). Utilisez le style télégramme, sans sujet, si le sujet est le programme lui-même ; n'utilisez pas « Je » pour le programme.

Raisonnement : le programme n'est pas humain. Ne prétendez pas autre chose.

Présent contre passé

Utilisez le passé si une tentative de faire quelque chose échouait, mais pourrait peut-être réussir la prochaine fois (peut-être après avoir corriger certains problèmes). Utilisez le présent si l'échec est sans doute permanent.

Il y a une différence sémantique non triviale entre les phrases de la forme :

```
n'a pas pu ouvrir le fichier "%s": %m
```

et :

```
ne peut pas ouvrir le dossier "%s"
```

La première forme signifie que la tentative d'ouverture du fichier a échoué. Le message devrait donner une raison comme « disque plein » ou « le fichier n'existe pas ». Le passé est approprié parce que la prochaine fois le disque peut ne plus être plein ou le fichier en question peut exister.

La seconde forme indique que la fonctionnalité d'ouvrir le fichier nommé n'existe pas du tout dans le programme ou que c'est conceptuellement impossible. Le présent est approprié car la condition persistera indéfiniment.

Raisonnement : d'accord, l'utilisateur moyen ne sera pas capable de tirer de grandes conclusions simplement à partir du temps du message mais, puisque la langue nous fournit une grammaire, nous devons l'utiliser correctement.

Type de l'objet

En citant le nom d'un objet, spécifiez quel genre d'objet c'est.

Raisonnement : sinon personne ne saura ce qu'est « foo.bar.baz ».

Crochets

Les crochets sont uniquement utilisés (1) dans les synopsis des commandes pour indiquer des arguments optionnels ou (2) pour indiquer l'indice inférieur d'un tableau.

Raisonnement : rien de ce qui ne correspond pas à l'utilisation habituelle, largement connue troublera les gens.

Assembler les messages d'erreurs

Quand un message inclut du texte produit ailleurs, il est intégré dans ce style :

```
n'a pas pu ouvrir le fichier %s: %m
```

Raisonnement : il serait difficile d'expliquer tous les codes d'erreurs possibles pour coller ceci dans une unique phrase douce, ainsi une certaine forme de ponctuation est nécessaire. Mettre le texte inclus entre parenthèses a été également suggéré, mais ce n'est pas naturel si le texte inclus est susceptible d'être la partie la plus importante du message, comme c'est souvent le cas.

Raisons pour les erreurs

Les messages devraient toujours indiquer la raison pour laquelle une erreur s'est produite. Par exemple :

```
MAUVAIS :  n'a pas pu ouvrir le fichier %s
MEILLEUR : n'a pas pu ouvrir le fichier %s (échec E/S)
```

Si aucune raison n'est connue, vous feriez mieux de corriger le code.

Nom des fonctions

N'incluez pas le nom de la routine de rapport dans le texte de l'erreur. Nous avons d'autres mécanismes pour trouver cela quand c'est nécessaire et, pour la plupart des utilisateurs, ce n'est pas une information utile. Si le texte de l'erreur n'a plus beaucoup de sens sans le nom de la fonction, reformulez-le.

```
MAUVAIS :  pg_atoi: erreur dans "z": ne peut pas analyser "z"
MEILLEUR : syntaxe en entrée invalide pour l'entier : "z"
```

Évitez de mentionner le nom des fonctions appelées, au lieu de cela dites ce que le code essayait de faire :

```
MAUVAIS :  ouvrir() a échoué : %m
MEILLEUR : n'a pas pu ouvrir le fichier %s: %m
```

Si cela semble vraiment nécessaire, mentionnez l'appel système dans le message détail (dans certains cas, fournir les valeurs réelles passées à l'appel système pourrait être une information appropriée pour le message détail).

Raisonnement : les utilisateurs ne savent pas tout ce que ces fonctions font.

Mots délicats à éviter

Incapable. « Incapable » est presque la voix passive. Une meilleure utilisation est « ne pouvait pas » ou « ne pourrait pas » selon les cas.

Mauvais. Les messages d'erreurs comme « mauvais résultat » sont vraiment difficile à interpréter intelligemment. Cela est mieux d'écrire pourquoi le résultat est « mauvais », par exemple, « format invalide ».

Illégal. « Illégal » représente une violation de la loi, le reste est « invalide ». Meilleur encore, dites pourquoi cela est invalide.

Inconnu. Essayez d'éviter « inconnu ». Considérez « erreur : réponse inconnue ». Si vous ne savez pas qu'elle est la réponse, comment savez-vous que cela est incorrect ? « Non reconnu » est souvent un meilleur choix. En outre, assurez-vous d'inclure la valeur pour laquelle il y a un problème.

```
MAUVAIS :  type de nœud inconnu
MEILLEUR : type de nœud non reconnu : 42
```

Trouver contre Exister. Si le programme emploie un algorithme non trivial pour localiser une ressource (par exemple, une recherche de chemin) et que l'algorithme échoue, il est juste de dire que le programme n'a pas pû « trouver » la ressource. D'un autre côté, si l'endroit prévu pour la ressource est connu mais que le programme ne peut pas accéder à celle-ci, alors dites que la ressource n'« existe » pas. Utilisez « trouvez » dans ce cas là semble faible et embrouille le problème.

May vs. Can vs. Might. « May » suggère un droit (par exemple *You may borrow my rake.*) et a peu d'utilité dans la documentation et dans les messages d'erreur. « Can » suggère une capacité (par exemple *I can lift that log.*), et « might » suggère une possibilité (par exemple *It might rain today.*). Utiliser le bon mot clarifie la signification et aide les traducteurs.

Contractions. Éviter les contractions comme « can't » ; utilisez « cannot » à la place.

Orthographe appropriée

Orthographiez les mots en entier. Par exemple, évitez :

- spec (NdT : spécification)
- stats (NdT : statistiques)
- params (NdT : paramètres)
- auth (NdT : authentification)
- xact (NdT : transaction)

Raisonnement : cela améliore la cohérence.

Adaptation linguistique

Gardez à l'esprit que les textes des messages d'erreurs ont besoin d'être traduit en d'autres langues. Suivez les directives dans la Section 48.2.2, « Guide d'écriture des messages » pour éviter de rendre la vie difficile aux traducteurs.

Chapitre 48. Support natif des langues

Peter Eisentraut

48.1. Pour le traducteur

Les programmes PostgreSQL™ (serveur et client) peuvent afficher leur message dans la langue préférée de l'utilisateur -- si les messages ont été traduits. Créer et maintenir les ensembles de messages traduits nécessite l'aide de personnes parlant leur propre langue et souhaitant contribuer à PostgreSQL™. Il n'est nul besoin d'être un développeur pour cela. Cette section explique comment apporter son aide.

48.1.1. Prérequis

Les compétences dans sa langue d'un traducteur ne seront pas jugées -- cette section concerne uniquement les outils logiciels. Théoriquement, seul un éditeur de texte est nécessaire. Mais ceci n'est vrai que dans le cas très improbable où un traducteur ne souhaiterait pas tester ses traductions des messages. Lors de la configuration des sources, il faudra s'assurer d'utiliser l'option `--enable-nls`. Ceci assurera également la présence de la bibliothèque libintl et du programme `msgfmt` dont tous les utilisateurs finaux ont indéniablement besoin. Pour tester son travail, il sera utile de suivre les parties pertinentes des instructions d'installation.

Pour commencer un nouvel effort de traduction ou pour faire un assemblage de catalogues de messages (décrit ci-après), il faudra installer respectivement les programmes `xgettext` et `msgmerge` dans une implémentation compatible GNU. Il est prévu dans le futur que `xgettext` ne soit plus nécessaire lorsqu'une distribution empaquetée des sources est utilisée (en travaillant à partir du Git, il sera toujours utile). GNU Gettext 0.10.36 ou ultérieure est actuellement recommandé.

Toute implémentation locale de gettext devrait être disponible avec sa propre documentation. Une partie en est certainement dupliquée dans ce qui suit mais des détails complémentaires y sont certainement disponibles.

48.1.2. Concepts

Les couples de messages originaux (anglais) et de leurs (possibles) traductions sont conservés dans les *catalogues de messages*, un pour chaque programme (bien que des programmes liés puissent partager un catalogue de messages) et pour chaque langue cible. Il existe deux formats de fichiers pour les catalogues de messages : le premier est le fichier « PO » (pour "Portable Object" ou Objet Portable), qui est un fichier texte muni d'une syntaxe spéciale et que les traducteurs éditent. Le second est un fichier « MO » (pour "Machine Object" ou Objet Machine), qui est un fichier binaire engendré à partir du fichier PO respectif et qui est utilisé lorsque le programme internationalisé est exécuté. Les traducteurs ne s'occupent pas des fichiers MO ; en fait, quasiment personne ne s'en occupe.

L'extension du fichier de catalogue de messages est, sans surprise, soit `.po`, soit `.mo`. Le nom de base est soit le nom du programme qu'il accompagne soit la langue utilisée dans le fichier, suivant la situation. Ceci peut s'avérer être une source de confusion. Des exemples sont `psql.po` (fichier PO pour psql) ou `fr.mo` (fichier MO en français).

Le format du fichier PO est illustré ici :

```
# commentaire

msgid "chaîne originale"
msgstr "chaîne traduite"

msgid "encore une originale"
msgstr "encore une de traduite"
"les chaînes peuvent être sur plusieurs lignes, comme ceci"

...
```

Les chaînes msgid sont extraites des sources du programme. (Elles n'ont pas besoin de l'être mais c'est le moyen le plus commun). Les lignes msgstr sont initialement vides puis complétées avec les chaînes traduites. Les chaînes peuvent contenir des caractères d'échappement de style C et peuvent être sur plusieurs lignes comme le montre l'exemple ci-dessus (la ligne suivante doit démarrer au début de la ligne).

Le caractère # introduit un commentaire. Si une espace fine suit immédiatement le caractère #, c'est qu'il s'agit là d'un commentaire maintenu par le traducteur. On trouve aussi des commentaires automatiques qui n'ont pas d'espace fine suivant immédiatement le caractère #. Ils sont maintenus par les différents outils qui opèrent sur les fichiers PO et ont pour but d'aider le traducteur.

```
#. commentaire automatique
#: fichier.c:1023
#, drapeau, drapeau
```

Les commentaires du style #. sont extraits du fichier source où le message est utilisé. Il est possible que le développeur ait ajouté des informations pour le traducteur, telles que l'alignement attendu. Le commentaire #: indique l'emplacement exact où le message est utilisé dans le source. Le traducteur n'a pas besoin de regarder le source du programme, mais il peut le faire s'il subsiste un doute sur l'exactitude d'une traduction. Le commentaire #, contient des drapeaux décrivant le message d'une certaine façon. Il existe actuellement deux drapeaux : `fuzzy` est positionné si le message risque d'être rendu obsolète par des changements dans les sources. Le traducteur peut alors vérifier ceci et supprimer ce drapeau. Notez que les messages « fuzzy » ne sont pas accessibles à l'utilisateur final. L'autre drapeau est `c-format` indiquant que le message utilise le format de la fonction C `printf`. Ceci signifie que la traduction devrait aussi être de ce format avec le même nombre et le même type de paramètres fictifs. Il existe des outils qui vérifient que le message est une chaîne au format printf et valident le drapeau c-format en conséquence.

48.1.3. Créer et maintenir des catalogues de messages

OK, alors comment faire pour créer un catalogue de messages « vide » ? Tout d'abord, se placer dans le répertoire contenant le programme dont on souhaite traduire les messages. S'il existe un fichier `nls.mk`, alors ce programme est préparé pour la traduction.

S'il y a déjà des fichiers `.po`, alors quelqu'un a déjà réalisé des travaux de traduction. Les fichiers sont nommés *`langue`*`.po`, où *`langue`* est le code de langue sur deux caractères (en minuscules) tel que défini par l' *ISO 639-1, le code du pays composé de deux lettres en minuscule*, c'est-à-dire `fr.po` pour le français. S'il existe réellement un besoin pour plus d'une traduction par langue, alors les fichiers peuvent être renommés *`langue_region`*`.po` où *`region`* est le code de langue sur deux caractères (en majuscules), tel que défini par l'*ISO 3166-1, le code du payes sur deux lettres en majuscule*, c'est-à-dire `pt_BR.po` pour le portugais du Brésil. Si vous trouvez la langue que vous souhaitez, vous pouvez commencer à travailler sur ce fichier.

Pour commencer une nouvelle traduction, il faudra préalablement exécuter la commande :

```
gmake init-po
```

Ceci créera un fichier *`nomprog`*`.pot`. (`.pot` pour le distinguer des fichiers PO qui sont « en production ». Le `T` signifie « template » (NdT : modèle en anglais). On copiera ce fichier sous le nom *`langue`*`.po`. On peut alors l'éditer. Pour faire savoir qu'une nouvelle langue est disponible, il faut également éditer le fichier `nls.mk` et y ajouter le code de la langue (ou de la langue et du pays) avec une ligne ressemblant à ceci :

```
AVAIL_LANGUAGES := de fr
```

(d'autres langues peuvent apparaître, bien entendu).

À mesure que le programme ou la bibliothèque change, des messages peuvent être modifiés ou ajoutés par les développeurs. Dans ce cas, il n'est pas nécessaire de tout recommencer depuis le début. À la place, on lancera la commande :

```
gmake update-po
```

qui créera un nouveau catalogue de messages vides (le fichier pot avec lequel la traduction a été initiée) et le fusionnera avec les fichiers PO existants. Si l'algorithme de fusion a une incertitude sur un message particulier, il le marquera « fuzzy » comme expliqué ci-dessus. Le nouveau fichier PO est sauvegardé avec l'extension `.po.new`.

48.1.4. Éditer les fichiers PO

Les fichiers PO sont éditables avec un éditeur de texte standard. Le traducteur doit seulement modifier l'emplacement entre les guillemets après la directive msgstr, peut ajouter des commentaires et modifier le drapeau fuzzy (NdA : Il existe, ce qui n'est pas surprenant, un mode PO pour Emacs, que je trouve assez utile).

Les fichiers PO n'ont pas besoin d'être entièrement remplis. Le logiciel retournera automatiquement à la chaîne originale si une traduction n'est pas disponible ou est laissée vide. Soumettre des traductions incomplètes pour les inclure dans l'arborescence des sources n'est pas un problème ; cela permet à d'autres personnes de récupérer le travail commencé pour le continuer. Néanmoins, les traducteurs sont encouragés à donner une haute priorité à la suppression des entrées fuzzy après avoir fait une fusion. Les entrées fuzzy ne seront pas installées ; elles servent seulement de référence à ce qui pourrait être une bonne traduction.

Certaines choses sont à garder à l'esprit lors de l'édition des traductions :

- S'assurer que si la chaîne originale se termine par un retour chariot, la traduction le fasse bien aussi. De même pour les tabulations, etc.
- Si la chaîne originale est une chaîne au format `printf`, la traduction doit l'être aussi. La traduction doit également avoir les même spécificateurs de format et dans le même ordre. Quelques fois, les règles naturelles de la langue rendent cela impossible ou tout au moins difficile. Dans ce cas, il est possible de modifier les spécificateurs de format de la façon suivante :

```
msgstr "Die Datei %2$s hat %1$u Zeichen."
```

Le premier paramètre fictif sera alors utilisé par le deuxième argument de la liste. Le *chiffre*$ a besoin de suivre immédiatement le %, avant tout autre manipulateur de format (cette fonctionnalité existe réellement dans la famille des fonctions `printf`, mais elle est peu connue, n'ayant que peu d'utilité en dehors de l'internationalisation des messages).

- Si la chaîne originale contient une erreur linguistique, on pourra la rapporter (ou la corriger soi-même dans le source du programme) et la traduire normalement. La chaîne corrigée peut être fusionnée lorsque les programmes sources auront été mis à jour. Si la chaîne originale contient une erreur factuelle, on la rapportera (ou la corrigera soi-même) mais on ne la traduira pas. À la place, on marquera la chaîne avec un commentaire dans le fichier PO.
- Maintenir le style et le ton de la chaîne originale. En particulier, les messages qui ne sont pas des phrases (`cannot open file %s`, soit `ne peut pas ouvrir le fichier %s`) ne devraient probablement pas commencer avec une lettre capitale (si votre langue distingue la casse des lettres) ou finir avec un point (si votre langue utilise des marques de ponctuation). Lire Section 47.3, « Guide de style des messages d'erreurs » peut aider.
- Lorsque la signification d'un message n'est pas connue ou s'il est ambigü, on pourra demander sa signification sur la liste de diffusion des développeurs. Il est possible qu'un anglophone puisse aussi ne pas le comprendre ou le trouver ambigü. Il est alors préférable d'améliorer le message.

48.2. Pour le développeur

48.2.1. Mécaniques

Cette section explique comment implémenter le support natif d'une langue dans un programme ou dans une bibliothèque qui fait partie de la distribution PostgreSQL™. Actuellement, cela s'applique uniquement aux programmes C.

Procédure 48.1. Ajouter le support NLS à un programme

1. Le code suivant est inséré dans la séquence initiale du programme :

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   #endif

   ...

   #ifdef ENABLE_NLS
   setlocale(LC_ALL, "");
   bindtextdomain("nomprog", LOCALEDIR);
   textdomain("nomprog");
   #endif
   ```

   (*nomprog* peut être choisi tout à fait librement).

2. Partout où un message candidat à la traduction est trouvé, un appel à `gettext()` doit être inséré. Par exemple :

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   devra être changé avec :

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   (gettext est défini comme une opération nulle si NLS n'est pas configuré).

   Cela peut engendrer du fouillis. Un raccourci habituel consiste à utiliser :

   ```
   #define _(x) gettext(x)
   ```

   Une autre solution est envisageable si le programme effectue la plupart de ses communications via une fonction ou un nombre restreint de fonctions, telle `ereport()` pour le moteur. Le fonctionnement interne de cette fonction peut alors être modifiée pour qu'elle appelle `gettext` pour toutes les chaînes en entrée.

3. Un fichier `nls.mk` est ajouté dans le répertoire des sources du programme. Ce fichier sera lu comme un makefile. Les affectations des variables suivantes doivent être réalisées ici :

   `CATALOG_NAME`
   Le nom du programme tel que fourni lors de l'appel à `textdomain()`.

`AVAIL_LANGUAGES`
Liste des traductions fournies -- initialement vide.

`GETTEXT_FILES`
Liste des fichiers contenant les chaînes traduisibles, c'est-à-dire celles marquées avec `gettext` ou avec une solution alternative. Il se peut que cette liste inclut pratiquement tous les fichiers sources du programme. Si cette liste est trop longue, le premier « fichier » peut être remplacé par un `+` et le deuxième mot représenter un fichier contenant un nom de fichier par ligne.

`GETTEXT_TRIGGERS`
Les outils qui engendrent des catalogues de messages pour les traducteurs ont besoin de connaître les appels de fonction contenant des chaînes à traduire. Par défaut, seuls les appels à `gettext()` sont reconnus. Si _ ou d'autres identifiants sont utilisés, il est nécessaire de les lister ici. Si la chaîne traduisible n'est pas le premier argument, l'élément a besoin d'être de la forme `func:2` (pour le second argument). Si vous avez une fonction qui supporte les messages au format pluriel, l'élément ressemblera à `func:1,2` (identifiant les arguments singulier et pluriel du message).

Le système de construction s'occupera automatiquement de construire et installer les catalogues de messages.

48.2.2. Guide d'écriture des messages

Voici quelques lignes de conduite pour l'écriture de messages facilement traduisibles.

- Ne pas construire de phrases à l'exécution, telles que :

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  L'ordre des mots d'une phrase peut être différent dans d'autres langues. De plus, même si `gettext()` est correctement appelé sur chaque fragment, il pourrait être difficile de traduire séparément les fragments. Il est préférable de dupliquer un peu de code de façon à ce que chaque message à traduire forme un tout cohérent. Seuls les nombres, noms de fichiers et autres variables d'exécution devraient être insérés au moment de l'exécution dans le texte d'un message.

- Pour des raisons similaires, ceci ne fonctionnera pas :

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  parce que cette forme présume de la façon dont la forme plurielle est obtenue. L'idée de résoudre ce cas de la façon suivante :

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

  sera source de déception. Certaines langues ont plus de deux formes avec des règles particulières. Il est souvent préférable de concevoir le message de façon à éviter le problème, par exemple ainsi :

  ```
  printf("number of copied files: %d", n);
  ```

  Si vous voulez vraiment construire un message correctement pluralisé, il existe un support pour cela, mais il est un peu étrange. Quand vous générez un message d'erreur primaire ou détaillé dans `ereport()`, vous pouvez écrire quelque-chose comme ceci :

  ```
  errmsg_plural("copied %d file",
                "copied %d files",
                n,
                n)
  ```

  Le premier argument est le chaîne dans le format approprié pour la forme au singulier en anglais, le second est le format de chaîne approprié pour la forme plurielle en anglais, et le troisième est la valeur entière déterminant la forme à utiliser. Des arguments additionnels sont formatés suivant la chaîne de formatage comme d'habitude. (Habituellement, la valeur de contrôle de la pluralisation sera aussi une des valeurs à formater, donc elle sera écrite deux fois.) En anglais, cela n'importe que si *n* est égale à 1 ou est différent de 1, mais dans d'autres langues, il pourrait y avoir plusieurs formes de pluriel. Le traducteur voit les deux formes anglaises comme un groupe et a l'opportunité de fournir des chaînes de substitution supplémentaires, la bonne étant sélectionnée suivant la valeur à l'exécution de *n*.

  Si vous avez besoin de pluraliser un message qui ne va pas directement à `errmsg` ou `errdetail`, vous devez utiliser la fonction sous-jacente `ngettext`. Voir la documentation gettext.

- Lorsque quelque chose doit être communiqué au traducteur, telle que la façon dont un message doit être aligné avec quelque

autre sortie, on pourra faire précéder l'occurrence de la chaîne d'un commentaire commençant par `translator`, par exemple :

```
/* translator: This message is not what it seems to be. */
```

Ces commentaires sont copiés dans les catalogues de messages de façon à ce que les traducteurs les voient.

Chapitre 49. Écrire un gestionnaire de langage procédural

Tous les appels de fonctions écrites dans un langage autre que celui de l'interface « version 1 » pour les langages compilés (ce qui inclut les fonctions dans les langages procéduraux utilisateur, les fonctions SQL et les fonctions utilisant l'interface de langage compilé version 0), passent par une fonction spécifique au langage du *gestionnaire d'appels*. Le gestionnaire d'appels exécute la fonction de manière appropriée, par exemple en interprétant le code source fourni. Ce chapitre décrit l'écriture du gestionnaire d'appels d'un nouveau langage procédural.

Le gestionnaire d'appel d'un langage procédural est une fonction « normale » qui doit être écrite dans un langage compilé tel que le C, en utilisant l'interface version-1, et enregistrée sous PostgreSQL™ comme une fonction sans argument et retournant le type language_handler. Ce pseudo-type spécial identifie la fonction comme gestionnaire d'appel et empêche son appel à partir des commandes SQL. Pour plus de détails sur les conventions d'appels et le chargement dynamique en langage C, voir Section 35.9, « Fonctions en langage C ».

L'appel du gestionnaire d'appels est identique à celui de toute autre fonction : il reçoit un pointeur de structure FunctionCallInfoData qui contient les valeurs des arguments et d'autres informations de la fonction appelée. Il retourne un résultat Datum (et, initialise le champ `isnull` de la structure FunctionCallInfoData si un résultat SQL NULL doit être retourné). La différence entre un gestionnaire d'appels et une fonction ordinaire se situe au niveau du champ `flinfo->fn_oid` de la structure FunctionCallInfoData. Dans le cas du gestionnaire d'appels, il contiendra l'OID de la fonction à appeler, et non pas celui du gestionnaire d'appels lui-même. Le gestionnaire d'appels utilise ce champ pour déterminer la fonction à exécuter. De plus, la liste d'arguments passée a été dressée à partir de la déclaration de la fonction cible, et non pas en fonction du gestionnaire d'appels.

C'est le gestionnaire d'appels qui récupère l'entrée de la fonction dans la table système `pg_proc` et analyse les types des arguments et de la valeur de retour de la fonction appelée. La clause AS de la commande **CREATE FUNCTION** se situe dans la colonne `prosrc` de `pg_proc`. Il s'agit généralement du texte source du langage procédural lui-même (comme pour PL/Tcl) mais, en théorie, cela peut être un chemin vers un fichier ou tout ce qui indique au gestionnaire d'appels les détails des actions à effectuer.

Souvent, la même fonction est appelée plusieurs fois dans la même instruction SQL. L'utilisation du champ `flinfo->fn_extra` évite au gestionnaire d'appels de répéter la recherche des informations concernant la fonction appelée. Ce champ, initialement NULL, peut être configuré par le gestionnaire d'appels pour pointer sur l'information concernant la fonction appelée. Lors des appels suivants, si `flinfo->fn_extra` est différent de NULL, alors il peut être utilisé et l'étape de recherche d'information évitée. Le gestionnaire d'appels doit s'assurer que `flinfo->fn_extra` pointe sur une zone mémoire qui restera allouée au moins jusqu'à la fin de la requête en cours, car une structure de données FmgrInfo peut être conservée aussi longtemps. Cela peut-être obtenu par l'allocation des données supplémentaires dans le contexte mémoire spécifié par `flinfo->fn_mcxt` ; de telles données ont la même espérance de vie que FmgrInfo. Le gestionnaire peut également choisir d'utiliser un contexte mémoire de plus longue espérance de vie de façon à mettre en cache sur plusieurs requêtes les informations concernant les définitions des fonctions.

Lorsqu'une fonction en langage procédural est appelée via un déclencheur, aucun argument ne lui est passé de façon traditionnelle mais le champ `context` de FunctionCallInfoData pointe sur une structure TriggerData. Il n'est pas NULL comme c'est le cas dans les appels de fonctions standard. Un gestionnaire de langage doit fournir les mécanismes pour que les fonctions de langages procéduraux obtiennent les informations du déclencheur.

Voici un modèle de gestionnaire de langage procédural écrit en C :

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
```

```
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Appelé comme procédure de déclencheur
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Appelé en tant que fonction
         */

        retval = ...
    }

    return retval;
}
```

Il suffit de remplacer les points de suspension par quelques milliers de lignes de codes pour compléter ce modèle.

Lorsque la fonction du gestionnaire est compilée dans un module chargeable (voir Section 35.9.6, « Compiler et lier des fonctions chargées dynamiquement »), les commandes suivantes enregistrent le langage procédural défini dans l'exemple :

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'nomfichier'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Bien que fournir un gestionnaire d'appels est suffisant pour créer un langage de procédures minimal, il existe deux autres fonctions qui peuvent être fournies pour faciliter l'utilisation du langage. Ce sont les fonctions de validation (*validator*) et de traitement en ligne (*inline handler*). Une fonction de validation peut être fournie pour activer une vérification spécifique au langage lors du CREATE FUNCTION(7). Une fonction de traitement en ligne sera utilisé pour supporter les blocs de code anonymes exécutés via la commande DO(7).

Si une fonction de validation est fournie par un langage de procédures, elle doit être déclarée comme une fonction prenant un seul paramètre, de type oid. Le résultat de la validation est ignoré, donc elle peut renvoyer le type void. La fonction de validation sera appelée à la fin de la commande **CREATE FUNCTION** qui a créé ou mis à jour une fonction écrite dans ce langage. L'OID passé en argument est l'OID de la fonction, disponible dans le catalogue `pg_proc`. La fonction de validation doit récupérer cette ligne de la façon habituelle et réaliser les vérifications appropriées. Tout d'abord, elle appelle `CheckFunctionValidatorAccess()` pour diagnostiquer les appels explicites au validateur que l'utilisateur ne peut pas réaliser via **CREATE FUNCTION**. Les vérifications typiques incluent la vérification du support des types en arguments et en sortie, ainsi que la vérification syntaxique du corps de la requête pour ce langage. Si la fonction de validation est satisfait par la fonction, elle quitte sans erreur. Si, par contre, elle trouve une erreur, elle doit rapporter cette erreur au travers du mécanisme `ereport()` standard. Renvoyer une erreur forcera une annulation de la transaction et empêchera du même coup l'enregistrement de la fonction dont la définition est erronée.

Les fonctions de validation devraient typiquement accepter le paramètre check_function_bodies : s'il est désactivé, alors tout vérification coûteuse ou spécifique au contexte devrait être abandonnée. Si le langage permet l'exécution de code à la compilation, le validateur doit supprimer les vérifications qui impliquerait une telle exécution. En particulier, ce paramètre est désactivé par pg_dump, pour qu'il puisse charger le langage de procédures sans avoir à s'inquiéter des effets de bord et des dépendances possibles dans le corps des procédures stockées avec d'autres objets de la base de données. (À cause de cela, le gestionnaire d'appels doit éviter de supposer que la fonction de validation a vérifié complètement la fonction. Le but d'avoir une fonction de validation n'est pas d'éviter au gestionnaire d'appels de faire des vérifications, mais plutôt de notifier immédiatement à l'utilisateur si des erreurs évidentes apparaissent dans la commande **CREATE FUNCTION**.) Bien que le choix de ce qui est à vérifier est laissé à la discrétion de la fonction de validation, il faut noter que le code de **CREATE FUNCTION** exécute seulement les clauses `SET` attachées à la fonction quand le paramètre `check_function_bodies` est activé. Du coup, les vérifications dont les résultats pourraient être affectés par les paramètres en question doivent être ignorées quand `check_function_bodies` est désactivé pour éviter de échecs erronés lors du chargement d'une sauvegarde.

Si une fonction de traitement en ligne est fournie au langage de procédures, elle doit être déclarée comme une fonction acceptant un seul paramètre de type internal. Le résultat de la fonction de traitement en ligne est ignoré, donc elle peut renvoyer le type void. Elle sera appelée quand une instruction **DO** est exécutée pour ce langage. Le paramètre qui lui est fourni est un pointeur vers une

structure InlineCodeBlock, structure contenant des informations sur les paramètres de l'instruction **DO**, en particulier le texte du bloc de code anonyme à exécuter. La fonction doit exécuter ce code.

Les langages procéduraux inclus dans la distribution standard sont de bons points de départ à l'écriture de son propre gestionnaire de langage. Les sources se trouvent dans le répertoire `src/pl`. La page de référence de CREATE LANGUAGE(7) contient aussi certains détails utiles.

Chapitre 50. Optimiseur génétique de requêtes (*Genetic Query Optimizer*)

Martin Utesch, University of Mining and Technology

Auteur

Écrit par Martin Utesch (<utesch@aut.tu-freiberg.de>) de l'Institut de Contrôle Automatique à l'Université des Mines et de Technologie de Freiberg, Allemagne.

50.1. Gérer les requêtes, un problème d'optimisation complexe

De tous les opérateurs relationnels, le plus difficile à exécuter et à optimiser est la jointure (*join*). Le nombre de plans de requêtes possibles croît exponentiellement avec le nombre de jointures de la requête. Un effort supplémentaire d'optimisation est nécessité par le support de différentes *méthodes de jointure* (boucles imbriquées, jointures de hachage, jointures de fusion...) pour exécuter des jointures individuelles et différents *index* (B-tree, hash, GiST et GIN...) pour accéder aux relations.

L'optimiseur standard de requêtes pour PostgreSQL™ réalise une *recherche quasi-exhaustive* sur l'ensemble des stratégies alternatives. Cet algorithme, introduit à l'origine dans la base de données System R d'IBM, produit un ordre de jointure quasi-optimal mais peut occuper beaucoup de temps et de mémoire à mesure que le nombre de jointures d'une requête augmente. L'optimiseur ordinaire de requêtes de PostgreSQL™ devient donc inapproprié pour les requêtes qui joignent un grand nombre de tables.

L'Institut de Contrôle Automatique de l'Université des Mines et de Technologie basé à Freiberg, Allemagne, a rencontré des difficultés lorsqu'il s'est agi d'utiliser PostgreSQL™ comme moteur d'un système d'aide à la décision reposant sur une base de connaissance utilisé pour la maintenance d'une grille de courant électrique. Le SGBD devait gérer des requêtes à nombreuses jointures pour la machine d'inférence de la base de connaissances. Le nombre de jointures de ces requêtes empêchait l'utilisation de l'optimiseur de requête standard.

La suite du document décrit le codage d'un *algorithme génétique* de résolution de l'ordonnancement des jointures qui soit efficace pour les requêtes à jointures nombreuses.

50.2. Algorithmes génétiques

L'algorithme génétique (GA) est une méthode d'optimisation heuristique qui opère par recherches aléatoires. L'ensemble des solutions possibles au problème d'optimisation est considéré comme une *population* d'*individus*. Le degré d'adaptation d'un individu à son environnement est indiqué par sa *valeur d'adaptation* (*fitness*).

Les coordonnées d'un individu dans l'espace de recherche sont représentées par des *chromosomes*, en fait un ensemble de chaînes de caractères. Un *gène* est une sous-section de chromosome qui code la valeur d'un paramètre simple en cours d'optimisation. Les codages habituels d'un gène sont *binary* ou *integer*.

La simulation des opérations d'évolution (*recombinaison*, *mutation* et *sélection*) permet de trouver de nouvelles générations de points de recherche qui présentent une meilleure adaptation moyenne que leurs ancêtres.

Selon la FAQ de `comp.ai.genetic`, on ne peut pas réellement affirmer qu'un GA n'est pas purement une recherche aléatoire. Un GA utilise des processus stochastiques, mais le résultat est assurément non-aléatoire (il est mieux qu'aléatoire).

Figure 50.1. Diagramme structuré d'un algorithme génétique

P(t)	génération des ancêtres au temps t
P''(t)	génération des descendants au temps t

```
+=========================================+
|>>>>>>>>>>>  Algorithme GA  <<<<<<<<<<<<<<|
+=========================================+
| INITIALISE t := 0                       |
+=========================================+
| INITIALISE P(t)                         |
```

```
+=========================================+
| évalue ADAPTATION de P(t)               |
+=========================================+
| tant que pas CRITERE ARRET faire        |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINAISON{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | évalue ADAPTATION de P''(t)         |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

50.3. Optimisation génétique des requêtes (GEQO) dans PostgreSQL

Le module GEQO utilise une approche du problème d'optimisation des requêtes similaire à celui du voyageur de commerce (TSP). Les plans de requêtes possibles sont codés comme des chaînes d'entiers. Chaque chaîne représente l'ordre de jointure d'une relation de la requête à une autre. Par exemple, l'arbre de jointure

```
   /\
  /\ 2
 /\ 3
4  1
```

est codé avec la chaîne d'entiers '4-1-3-2', ce qui signifie : première jointure entre les relations '4' et '1', puis '3' et enfin '2', avec 1, 2, 3, 4 les identifiants des relations pour l'optimiseur de PostgreSQL™.

Les caractéristiques spécifiques de l'implantation du GEQO sont :

- l'utilisation d'un algorithme génétique monostable (ou à état stable) (remplacement des individus les moins ciblés au lieu d'un remplacement global de génération) permet une convergence rapide vers des plans de requêtes améliorés ; c'est indispensable au traitement des requêtes dans un temps raisonnable ;
- l'utilisation de croisements (recombinaisons) aux limites est particulièrement adapté pour la restriction des pertes aux limites lors de la résolution du problème du voyageur de commerce par un algorithme génétique ;
- la mutation en tant qu'opérateur génétique est rendue obsolète afin d'éviter la nécessité de mécanismes de réparation lors de la génération de tournées valides du problème du voyageur de commerce.

Diverses parties du module GEQO sont adaptées de l'algorithme Genitor de D. Whitley.

Le module GEQO permet à l'optimiseur de requêtes de PostgreSQL™ de supporter les requêtes disposant de jointures importantes de manière efficace via une recherche non exhaustive.

50.3.1. Génération par le GEQO des plans envisageables

Le processus de planification du GEQO utilise le code standard du planificateur pour créer les plans de parcours des relations individuelles. Les plans de jointure sont alors développés à l'aide de l'approche génétique. Comme décrit plus bas, chaque plan de jointure candidat est représenté par une séquence à laquelle joindre les relations de base. Lors de l'étape initiale, l'algorithme produit simplement quelques séquences de jointure aléatoirement. Pour chaque séquence considérée, le code du planificateur standard est invoqué pour estimer le coût de la requête à l'aide de cette séquence. (Pour chaque étape de la séquence, les trois stratégies de jointure sont considérées ; et tous les plans de parcours initiaux sont disponibles. Le coût estimé est le moins coûteux.) Les séquences dont le coût est moindre sont considérées « plus adaptée » que celle de coût plus élevé. L'algorithme génétique élimine les candidats les moins adaptés. De nouveaux candidats sont alors engendrés par combinaison de gènes de candidats à forte valeur d'adaptation -- par l'utilisation de portions aléatoires de plans peu coûteux pour créer de nouvelles séquences. Ce processus est répété jusqu'à ce qu'un nombre prédéterminé de séquences aient été considérées ; la meilleure séquence rencontrée pendant la recherche est utilisée pour produire le plan final.

Ce processus est intrinsèquement non-déterministe, du fait des choix aléatoires effectués lors de la sélection initiale de la population et lors des « mutations » des meilleurs candidats qui s'en suivent. Pour éviter des modifications surprenantes du plan sélectionné, chaque exécution de l'algorithme relance son générateur aléatoire de numéros avec le paramètre geqo_seed. Tant que `geqo_seed` et les autres paramètres GEQO sont fixes, le même plan sera généré pour une même requête (ainsi que pour certaines

informations du planificateur comme les statistiques). Pour expérimenter différents chemins de recherche, modifiez `geqo_seed`.

50.3.2. Tâches à réaliser pour la future implantation du GEQO

Un gros travail est toujours nécessaire pour améliorer les paramètres de l'algorithme génétique. Dans le fichier `src/backend/optimizer/geqo/geqo_main.c`, pour les routines `gimme_pool_size` et `gimme_number_generations`, il faut trouver un compromis pour que les paramètres satisfassent deux besoins concurrents :

- l'optimisation du plan de requête ;
- le temps de calcul.

Dans l'implantation courante, l'adaptation de chaque séquence de jointure candidate est estimée par l'exécution ab-initio du code standard de sélection de jointure et d'estimation de coût utilisé par le planificateur. Avec l'hypothèse que différents candidats utilisent des sous-séquences de jointure similaires, une grande partie du travail est répétée. Ce processus peut être grandement accéléré en retenant les estimations de coût des sous-jointures. Le problème consiste à éviter d'étendre inutilement la mémoire en mémorisant ces états.

À un niveau plus basique, il n'est pas certain qu'optimiser une requête avec un algorithme génétique conçu pour le problème du voyageur de commerce soit approprié. Dans le cas du voyageur de commerce, le coût associé à une sous-chaîne quelconque (tour partiel) est indépendant du reste du tour, mais cela n'est certainement plus vrai dans le cas de l'optimisation de requêtes. Du coup, la question reste posée quant au fait que la recombinaison soit la procédure de mutation la plus efficace.

50.4. Lectures supplémentaires

Les ressources suivantes contiennent des informations supplémentaires sur les algorithmes génétiques :

- *The Hitch-Hiker's Guide to Evolutionary Computation* (FAQ de *news://comp.ai.genetic*) ;
- *Evolutionary Computation and its application to art and design*, par Craig Reynolds ;
- elma04
- fong

Chapitre 51. Définition de l'interface des méthodes d'accès aux index

Ce chapitre définit l'interface entre le système PostgreSQL™ et les *méthodes d'accès aux index*, qui gérent les types d'index individuels. Le système principal ne sait rien des index en dehors de ce qui est spécifié ici. Il est donc possible de développer de nouveaux types d'index en écrivant du code supplémentaire.

Tous les index de PostgreSQL™ sont connus techniquement en tant qu'*index secondaires* ; c'est-à-dire que l'index est séparé physiquement du fichier de table qu'il décrit. Chaque index est stocké dans sa propre *relation* physique et donc décrit par une entrée dans le catalogue pg_class. Le contenu d'un index est entièrement contrôlé par la méthode d'accès à l'index. En pratique, toutes les méthodes d'accès aux index les divisent en pages de taille standard de façon à utiliser le gestionnaire de stockage et le gestionnaire de tampon pour accéder au contenu de l'index. De plus, toutes les méthodes existantes d'accès aux index utilisent la disposition de page standard décrite dans Section 54.5, « Emplacement des pages de la base de données » et le même format pour les en-têtes de ligne de l'index ; mais ces décisions ne sont pas contraintes sur une méthode d'index.

Dans les faits, un index est une correspondance entre les valeurs des clés de données et les identifiants de lignes (*tuple identifiers*, ou TIDs) des versions de lignes dans la table parent de l'index. Un TID consiste en un numéro de bloc et un numéro d'élément à l'intérieur de ce bloc (voir Section 54.5, « Emplacement des pages de la base de données »). C'est une information suffisante pour récupérer une version de ligne particulière à partir de la table. Les index n'ont pas directement la connaissance de l'existence éventuelle, sous MVCC, de plusieurs versions de la même ligne logique ; pour un index, chaque ligne est un objet indépendant qui a besoin de sa propre entrée dans l'index. Du coup, la mise à jour d'une ligne crée toujours toutes les nouvelles entrées d'index pour la ligne, même si les valeurs de la clé ne changent pas. (Les lignes HOT sont une exception ; mais les index ne sont pas concernés.) Les entrées d'index pour les lignes mortes sont réclamées (par le VACUUM) lorsque les lignes mortes elles-même sont réclamées.

51.1. Entrées du catalogue pour les index

Chaque méthode d'accès à l'index est décrite par une ligne dans le catalogue système pg_am (voir Section 45.3, « pg_am »). Le contenu principal d'une ligne de pg_am est constitué de références à des entrées de pg_proc qui identifient les fonctions d'accès à l'index fournies par la méthode d'accès. Les API de ces fonctions sont définies plus loin dans ce chapitre. De plus, la ligne de pg_am spécifie quelques propriétés fixes de la méthode d'accès, comme le support des index multi-colonnes. Il n'existe pas de support spécial pour la création ou la suppression d'entrées dans pg_am ; toute personne capable d'écrire une nouvelle méthode d'accès est supposée assez compétente pour insérer la ligne appropriée.

Pour être utile, une méthode d'accès à l'index doit aussi avoir une ou plusieurs *familles d'opérateur* et *classes d'opérateur* définies dans pg_opfamily, pg_opclass, pg_amop et pg_amproc. Ces entrées permettent au planificateur de déterminer les types de qualification des requêtes qui peuvent être utilisés avec les index de cette méthode d'accès. Les familles et classes d'opérateurs sont décrites dans Section 35.14, « Interfacer des extensions d'index », qui est un élément requis pour comprendre ce chapitre.

Un index individuel est défini par une entrée dans pg_class le définissant comme une relation physique, et une entrée dans pg_index affichant le contenu logique de l'index -- c'est-à-dire ses colonnes d'index et la sémantique de ces colonnes, telles que récupérées par les classes d'opérateur associées. Les colonnes de l'index (valeurs clés) peuvent être des colonnes simples de la table sous-jacente ou des expressions sur les lignes de la table. Habituellement, la méthode d'accès à l'index ne s'intérese pas à la provenance des valeurs clés de l'index (ce sont toujours des valeurs clés pré-traitées), mais trouve beaucoup d'intérêt aux informations de la classe d'opérateur dans pg_index. Les entrées de ces deux catalogues peuvent être accédées comme partie de la structure de données de Relation passée à toute opération sur l'index.

Certaines colonnes d'options de pg_am ont des implications peu évidentes. Les besoins de `amcanunique` sont discutés dans Section 51.5, « Vérification de l'unicité de l'index ». L'option `amcanmulticol` indique que la méthode d'accès supporte les index multi-colonnes alors que `amoptionalkey` indique que des parcours sont autorisés lorsqu'aucune clause de restriction indexable n'est donnée pour la première colonne de l'index. Quand `amcanmulticol` est faux, `amoptionalkey` indique essentiellement si la méthode d'accès autorise les parcours complets de l'index sans clause de restriction. Les méthodes d'accès qui supportent les colonnes d'index multiples *doivent* supporter les parcours qui omettent des restrictions sur une ou toutes les colonnes après la première ; néanmoins, elles peuvent imposées qu'une restriction apparaisse pour la première colonne de l'index, ce qui est signalé par l'initialisation de `amoptionalkey` à faux. `amindexnulls` indique que des entrées d'index sont créées pour les valeurs de clés NULL. Comme la plupart des opérateurs indexables sont stricts et, du coup, ne peuvent pas renvoyer TRUE pour des entrées NULL, il est à première vue attractif de ne pas stocker les entrées d'index pour les valeurs NULL : un parcours d'index ne peut, de toute façon, pas les retourner. Néanmoins, cet argument tombe lorsqu'un parcours d'index n'a pas de clause de restriction pour une colonne d'index donnée. En pratique, cela signifie que les index dont `amoptionalkey` vaut true doivent indexer les valeurs NULL, car le planificateur peut décider d'utiliser un tel index sans aucune clé de parcours. Une restriction en découle : une méthode d'accès qui supporte des colonnes d'index multiples *doit* supporter l'indexage des valeurs NULL dans les colonnes qui suivent la première, car le planificateur suppose que l'index peut être utilisé pour les requêtes qui

ne restreignent pas ces colonnes. Par exemple, si l'on considère un index sur (a,b) et une requête avec `WHERE a = 4`, le système suppose que l'index peut être utilisé pour rechercher les lignes pour lesquelles `a = 4`, ce qui est faux si l'index omet les lignes où `b` est null. Néanmoins, il est correct d'omettre les lignes où la première colonne indexée est NULL. Du coup, *`amindexnulls`* doit valoir true seulement si la méthode d'accès à l'index indexe toutes les lignes, en incluant les combinaisons arbitraires de valeurs NULL. Une méthode d'accès d'index qui initialise *`amindexnulls`* peut aussi initialiser *`amsearchnulls`*, indiquant ainsi qu'elle supporte les clauses `IS NULL` et `IS NOT NULL` dans les conditions de recherche.

51.2. Fonctions de la méthode d'accès aux index

Les fonctions de construction et de maintenance d'index que doit fournir une méthode d'accès aux index sont :

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

Construire un nouvel index. La relation de l'index a été créée physiquement mais elle est vide. Elle doit être remplie avec toute donnée fixe nécessaire à la méthode d'accès, ainsi que les entrées pour toutes les lignes existant déjà dans la table. Habituellement, la fonction `ambuild` appelle `IndexBuildHeapScan()` pour parcourir la table à la recherche des lignes qui existent déjà et calculer les clés à insérer dans l'index. La fonction doit renvoyer une structure allouée par palloc contenant les statistiques du nouvel index.

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique);
```

Insérer une nouvelle ligne dans un index existant. Les tableaux `values` et `isnull` donnent les valeurs de clés à indexer. `heap_tid` est le TID à indexer. Si la méthode d'accès supporte les index uniques (son drapeau pg_am.*`amcanunique`* vaut true), alors `checkUnique` indique le type de vérification unique à réaliser. Cela varie si la contrainte unique est déferrable ou non ; voir Section 51.5, « Vérification de l'unicité de l'index » pour les détails. Habituellement, la méthode d'accès a seulement besoin du paramètre `heapRelation` lors de la vérification de l'unicité (car après, elle doit regarder la table pour vérifier la visibilité de la ligne).

La valeur résultat, de type booléen, de la fonction est significative seulement quand `checkUnique` vaut `UNIQUE_CHECK_PARTIAL`. Dans ce cas, un résultat TRUE signifie que la nouvelle entrée est reconnue comme unique alors que FALSE indique qu'elle pourrait ne pas être unique (et une vérification d'unicité déferrée doit être planifiée). Dans les autres cas, un résultat FALSE constant est recommendé.

Certains index pourraient ne pas indexer toutes les lignes. Si la ligne ne doit pas être indexée, `aminsert` devrait terminer sans rien faire.

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

Supprimer un(des) tuple(s) de l'index. Il s'agit d'une opération de « suppression massive » à implanter par le parcours complet de l'index et la vérification de chaque entrée pour déterminer si elle doit être supprimée. La fonction `callback` en argument doit être appelée, sous la forme `callback(`*`TID`*`, callback_state) returns bool`, pour déterminer si une entrée d'index particulière, identifiée par son TID, est à supprimer. Cette fonction Doit renvoyer NULL ou une structure issue d'un palloc qui contient des statistiques sur les effets de l'opération de suppression. La fonction peut retourner NULL si aucune information ne doit être envoyée à `amvacuumcleanup`.

En cas de limitation de `maintenance_work_mem`, la suppression de nombreux tuples impose d'appeler `ambulkdelete` à plusieurs reprises. L'argument `stats` est le résultat du dernier appel pour cet index (il est NULL au premier appel dans une opération **VACUUM**). Ceci permet à l'AM d'accumuler des statistiques sur l'opération dans son intégralité. Typiquement, `ambulkdelete` modifie et renvoie la même structure si le `stats` fourni n'est pas NULL.

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

Nettoyage après une opération **VACUUM** (zéro à plusieurs appels à `ambulkdelete`). La fonction n'a pas d'autre but que de retourner des statistiques concernant les index, mais elle peut réaliser un nettoyage en masse (réclamer les pages d'index vides, par exemple). `stats` est le retour de l'appel à `ambulkdelete`, ou NULL si `ambulkdelete` n'a pas été appelée car aucune ligne n'avait besoin d'être supprimée. Si le résultat n'est pas NULL, il s'agit d'une structure allouée par palloc. Les statistiques qu'elle contient sont utilisées pour mettre à jour pg_class, et sont rapportées par **VACUUM** si `VERBOSE` est indiqué. La fonction peut retourner NULL si l'index n'a pas été modifié lors de l'opération de **VACUUM** mais, dans le cas contraire, il faut retourner des statistiques correctes.

À partir de PostgreSQL™ 8.4, `amvacuumcleanup` sera aussi appelé à la fin d'une opération **ANALYZE** operation. Dans ce cas, `stats` vaut toujours NULL et toute valeur de retour sera ignorée. Ce cas peut être distingué en vérifiant `info->analyze_only`. Il est recommandé que la méthode d'accès ne fasse rien en dehors du nettoyage après insertion pour ce type d'appel, et cel aseulement dans un processus de travail autovacuum.

```
void
amcostestimate (PlannerInfo *root,
                IndexOptInfo *index,
                List *indexQuals,
                RelOptInfo *outer_rel,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

Estimer les coûts d'un parcours d'index. Cette fonction est décrite complètement dans Section 51.6, « Fonctions d'estimation des coûts d'index », ci-dessous.

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

Analyser et valider le tableau reloptions pour un index. Cette fonction n'est appelée que lorsqu'il existe un tableau reloptions non NULL pour l'index. *`reloptions`* est un tableau de type text contenant des entrées de la forme *`nom=valeur`*. La fonction construit une valeur de type bytea à copier dans le champ *`rd_options`* de l'entrée relcache de l'index. Les données contenues dans la valeur bytea sont définies par la méthode d'accès. La plupart des méthodes d'accès standard utilisent la structure StdRdOptions. Lorsque *`validate`* est true, la fonction remonte un message d'erreur clair si une option n'est pas reconnue ou a des valeurs invalides ; quand *`validate`* est false, les entrées invalides sont ignorées silencieusement. (*`validate`* est faux lors du chargement d'options déjà stockées dans pg_catalog ; une entrée invalide ne peut être trouvée que si la méthode d'accès a modifié ses règles pour les options et, dans ce cas, ignorer les entrées obsolètes est approprié.) Pour obtenir le comportement par défaut, il suffit de retourner NULL.

Le but d'un index est de supporter les parcours de lignes qui correspondent à une condition `WHERE` indexable, souvent appelée *qualificateur* (*qualifier*) ou *clé de parcours* (*scan key*). La sémantique du parcours d'index est décrite plus complètement dans Section 51.3, « Parcours d'index », ci-dessous. Une méthode d'accès à l'index peut supporter les parcours d'accès standards (« plain »), les parcours d'index « bitmap » ou les deux. Les fonctions liées au parcours qu'une méthode d'accès à l'index doit ou devrait fournir sont :

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             ScanKey key);
```

Débuter un nouveau parcours. Le tableau `key` (de longueur `nkeys`) décrit les clés de parcours pour le parcours de l'index. Le résultat doit être une structure allouée par palloc. Pour des raisons de codage, la méthode d'accès à l'index *doit* créer cette structure en appelant `RelationGetIndexScan()`. Dans la plupart des cas, `ambeginscan` se contente de réaliser l'appel ; les parties intéressantes de début de parcours d'index sont dans `amrescan`.

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

Récupérer la prochaine ligne d'un parcours donné, dans la direction donnée (vers l'avant ou l'arrière de l'index). Renvoie TRUE si une ligne a été obtenue, FALSE s'il ne reste aucune ligne. Dans le cas TRUE, le TID de la ligne est stocké dans la structure `scan`. « success » signifie uniquement que l'index contient une entrée qui correspond aux clés de parcours, pas que la ligne existe toujours dans la pile ou qu'elle peut réussir le test d'instantané de l'appelant. En cas de succès, `amgettuple` doit aussi configuré `scan->xs_recheck` à TRUE ou FALSE. FALSE signifie qu'il est certain que l'entrée de l'index correspond aux clés de par-

cours. TRUE signifie que ce n'est pas certain et que les conditions représentées par les clés de parcours doivent être de nouveau vérifiées sur la ligne de la table après l'avoir récupéré. Cette différence permet de supporter les opérateurs d'index « à perte ». Notez que la nouvelle vérification s'étendra seulement aux conditions de parcours ; un prédicat partiel d'index n'est jamais vérifié par les appelants à `amgettuple`.

La fonction `amgettuple` a seulement besoin d'exister si la méthode d'accès supporte les parcours d'index standards. Si ce n'est pas le cas, le champ *`amgettuple`* de la ligne de pg_am doit valoir zéro.

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

Récupère toutes les lignes du parcours sélectionné et les ajoute au TIDBitmap fournit par l'appelant (c'est-à-dire un OU de l'ensemble des identifiants de ligne dans l'ensemble où se trouve déjà le bitmap). Le nombre de lignes récupérées est renvoyé (cela peut n'être qu'une estimation car certaines méthodes d'accès ne détectent pas les duplicats). Lors de l'insertion d'identifiants de ligne dans le bitmap, `amgetbitmap` peut indiquer que la vérification des conditions du parcours est requis pour des identifiants précis de transactions. C'est identique au paramètre de sortie `xs_recheck` de `amgettuple`. Note : dans l'implantation actuelle, le support de cette fonctionnalité peut beaucoup ressembler au support du stockage à perte du bitmap lui-même, et du coup les appelants vérifient les conditions du parcours et le prédicat de l'index partiel (si disponible) pour les lignes vérifiables à nouveau. Cela pourrait ne pas toujours être vrai. `amgetbitmap` et `amgettuple` ne peuvent pas être utilisés dans le même parcours d'index ; il existe d'autres restrictions lors de l'utilisation de `amgetbitmap`, comme expliquées dans Section 51.3, « Parcours d'index ».

La fonction `amgetbitmap` doit seulement exister si la méthode d'accès supporte les parcours d'index « bitmap ». Dans le cas contraire, le champ *`amgetbitmap`* de la ligne correspondante dans pg_am doit être à zéro.

```
void
amrescan (IndexScanDesc scan,
          ScanKey key);
```

Recommencer le parcours donné, si possible avec de nouvelles clés de parcours (pour continuer à utiliser les anciennes clés, passer NULL pour `key`). Il n'est pas possible de changer le nombre de clés. En pratique, la fonctionnalité de relance est utilisée quand une nouvelle ligne externe est sélectionnée par une jointure imbriquée en boucle, ce qui impose une nouvelle valeur de comparaison de clés, mais la structure de la clé de parcours reste inchangée. Cette fonction est aussi appelée par `RelationGetIndexScan()`. Elle est de ce fait à la fois utilisée pour initialiser un parcours d'index et pour le re-parcourir.

```
void
amendscan (IndexScanDesc scan);
```

Terminer un parcours et libérer les ressources. La structure `scan` elle-même ne doit pas être libérée, mais tout verrou pris en interne par la méthode d'accès doit être libéré.

```
void
ammarkpos (IndexScanDesc scan);
```

Marquer la position courante du parcours. La méthode d'accès ne mémorise qu'une seule position par parcours.

```
void
amrestrpos (IndexScanDesc scan);
```

Restaurer le parcours à sa plus récente position marquée.

Par convention, l'entrée `pg_proc` de toute fonction de méthode d'accès aux index affiche le bon nombre d'arguments, mais les déclare tous du type internal (la plupart des arguments ont des types qui ne sont pas connus en SQL, et il n'est pas souhaitable que les utilisateurs appelent les fonctions directement). Le type de retour est déclaré void, internal ou boolean suivant le cas. La seule exception est `amoptions`, qui doit être correctement déclarée prenant text[] et bool et retournant bytea. Cette protection autorise le code client à exécuter `amoptions` pour tester la validité des paramètres.

51.3. Parcours d'index

Dans un parcours d'index, la méthode d'accès à l'index retourne les TID de toutes les lignes annoncées correspondre aux *clés de parcours*. La méthode d'accès n'est impliquée *ni* dans la récupération de ces lignes dans la table parent de l'index, ni dans les tests de qualification temporelle ou autre.

Une clé de parcours est une représentation interne d'une clause `WHERE` de la forme *`clé_index opérateur constante`*, où la clé d'index est une des colonnes de l'index et l'opérateur est un des membres de la famille d'opérateur associée avec cette colonne d'index. Un parcours d'index contient entre aucune et plusieurs clés de parcours qui sont assemblées implicitement avec des

AND -- les lignes renvoyées doivent satisfaire toutes les conditions indiquées.

La méthode d'accès peut indiquer que l'index est *à perte* ou nécessite une vérification pour une requête particulière ; ceci implique que le parcours d'index renvoie toutes les entrées qui correspondent à la clé de parcours, avec éventuellement des entrées supplémentaires qui ne correspondent pas. La machinerie du parcours d'index du système principal applique alors les conditions de l'index au tuple pour vérifier s'il doit bien effectivement être retenu. Si l'option de vérification n'est pas indiquée, le parcours d'index doit renvoyer exactement l'ensemble d'entrées correspondantes.

La méthode d'accès doit s'assurer qu'elle trouve correctement toutes les entrées correspondantes aux clés de parcours données, et seulement celles-ci. De plus, le système principal transfert toutes les clauses `WHERE` qui correspondent aux clés d'index et aux familles d'opérateurs, sans analyse sémantique permettant de déterminer si elles sont redondantes ou contradictoires. Par exemple, étant donné `WHERE x > 4 AND x > 14` où `x` est une colonne indexée B-tree, c'est à la fonction B-tree `amrescan` de déterminer que la première clé de parcours est redondante et peut être annulée. Le supplément de pré-traitement nécessaire lors de `amrescan` dépend du niveau de réduction des clés de parcours en une forme « normalisée » nécessaire à la méthode d'accès à l'index.

Certaines méthodes d'accès renvoient des entrées d'index dans un ordre bien défini, d'autres non. Si les entrées sont renvoyées triées, la méthode d'accès doit initialiser pg_am.*`amcanorder`* à true pour indiquer qu'elle supporte les parcours triés. Toutes les méthodes d'accès doivent utiliser des numéros de stratégie compatibles btree pour les opérateurs d'égalité et de tri.

La fonction `amgettuple` dispose d'un argument `direction`, qui peut être soit `ForwardScanDirection` (le cas normal) soit `BackwardScanDirection`. Si le premier appel après `amrescan` précise `BackwardScanDirection`, alors l'ensemble des entrées d'index correspondantes est à parcourir de l'arrière vers l'avant plutôt que dans la direction normale (d'avant en arrière). `amgettuple` doit donc renvoyer la dernière ligne correspondante dans l'index, plutôt que la première, comme cela se fait normalement. (Cela ne survient que pour les méthodes d'accès qui indiquent qu'elles supportent les parcours ordonnés.) Après le premier appel, `amgettuple` doit être préparé pour continuer le parcours dans la direction adaptée à partir de l'entrée la plus récemment renvoyée. (Mais si pg_am.*`amcanbackward`* vaut false, tous les appels suivants auront la même direction que le premier.)

Les méthodes d'accès qui supportent les parcours ordonnés doivent supporter le « marquage » d'une position dans un parcours pour retourner plus tard à la position marquée. La même position pourrait être restaurée plusieurs fois. Néanmoins, seule une position par parcours a besoin d'être conservée en mémoire ; un nouvel appel à `ammarkpos` surcharge la position anciennement marquée. Une méthode d'accès qui ne supporte pas les parcours ordonnés doit quand même fournir les fonctions de marquage et de restauration dans pg_am, mais il est suffisant de leur faire renvoyer des erreurs si les fonctions sont appelées.

Les positions du parcours et du marquage doivent être conservées de façon cohérente dans le cas d'insertions et de suppressions concurrentes dans l'index. Il est tout à fait correct qu'une entrée tout juste insérée ne soit pas retournée par un parcours, qui si l'entrée avait existé au démarrage du parcours, aurait été retournée. De même est-il correct qu'un parcours retourne une telle entrée lors d'un re-parcours ou d'un retour arrière, alors même qu'il ne l'a pas retourné lors du parcours initial. À l'identique, une suppression concurrente peut être, ou non, visible dans les résultats d'un parcours. Il est primordial qu'insertions et suppressions ne conduisent pas le parcours à oublier ou dupliquer des entrées qui ne sont pas elles-même insérées ou supprimées.

`amgetbitmap` peut être utilisé à la place de `amgettuple` pour un parcours d'index. Cela permet de récupérer toutes les lignes en un appel. Cette méthode peut s'avérer notablement plus efficace que `amgettuple` parce qu'elle permet d'éviter les cycles de verrouillage/déverrouillage à l'intérieur de la méthode d'accès. En principe, `amgetbitmap` a les mêmes effets que des appels répétés à `amgettuple`, mais plusieurs restrictions ont été imposées pour simplifier la procédure. En premier lieu, `amgetbitmap` renvoie toutes les lignes en une fois et le marquage ou la restauration des positions de parcours n'est pas supporté. Ensuite, les lignes sont renvoyées dans un bitmap qui n'a pas d'ordre spécifique, ce qui explique pourquoi `amgetbitmap` ne prend pas de `direction` en argument. Enfin, `amgetmulti` ne garantit pas le verrouillage des lignes renvoyées, avec les implications précisées dans Section 51.4, « Considérations sur le verrouillage d'index ».

Notez qu'il est permis à une méthode d'accès d'implanter seulement `amgetbitmap` et pas `amgettuple`, ou vice versa, si son implantation interne ne convient qu'à une seule des API.

51.4. Considérations sur le verrouillage d'index

Les méthodes d'accès aux index doivent gérer des mises à jour concurrentes de l'index par plusieurs processus. Le système principal PostgreSQL™ obtient `AccessShareLock` sur l'index lors d'un parcours d'index et `RowExclusiveLock` lors de sa mise à jour (ce qui inclut le **VACUUM** simple). Comme ces types de verrous ne sont pas conflictuels, la méthode d'accès est responsable de la finesse du verrouillage dont elle a besoin. Un verrou exclusif sur l'intégralité de l'index entier n'est pris qu'à la création de l'index, sa destruction ou dans une opération `REINDEX`.

Construire un type d'index qui supporte les mises à jour concurrentes requiert une analyse complète et subtile du comportement requis. Pour les types d'index B-tree et hash, on peut lire les implication sur les décisions de conception dans `src/backend/access/nbtree/README` et `src/backend/access/hash/README`.

En plus des besoins de cohérence interne de l'index, les mises à jour concurrentes créent des problèmes de cohérence entre la table

parent (l'*en-tête*, ou *heap*) et l'index. Comme PostgreSQL™ sépare les accès et les mises à jour de l'en-tête de ceux de l'index, il existe des fenêtres temporelles pendant lesquelles l'index et l'en-tête peuvent être incohérents. Ce problème est géré avec les règles suivantes :

- une nouvelle entrée dans l'en-tête est effectuée avant son entrée dans l'index. (Un parcours d'index concurrent peut alors ne pas voir l'entrée dans l'en-tête. Ce n'est pas gênant dans la mesure où un lecteur de l'index ne s'intéresse pas à une ligne non validée. Voir Section 51.5, « Vérification de l'unicité de l'index »);
- Lorsqu'entrée de l'en-tête va être supprimée (par **VACUUM**), toutes les entrées de l'index doivent d'abord être supprimées ;
- un parcours d'index doit maintenir un lien sur la page d'index contenant le dernier élément renvoyé par `amgettuple`, et `ambulkdelete` ne peut pas supprimer des entrées de pages liées à d'autres processus. La raison de cette règle est expliquée plus bas.

Sans la troisième règle, il est possible qu'un lecteur d'index voit une entrée dans l'index juste avant qu'elle ne soit supprimée par un **VACUUM**, et arrive à l'entrée correspondante de l'en-tête après sa suppression par le **VACUUM**. Cela ne pose aucun problème sérieux si ce numéro d'élément est toujours inutilisé quand le lecteur l'atteint, car tout emplacement d'élément vide est ignoré par `heap_fetch()`. Mais que se passe-t-il si un troisième moteur a déjà ré-utilisé l'emplacement de l'élément pour quelque chose d'autre ? Lors de l'utilisation d'un instantané compatible MVCC, il n'y a pas de problème car le nouvel occupant de l'emplacement est certain d'être trop récent pour réussir le test de l'insstantané. En revanche, avec un instantané non-compatible MVCC (tel que `SnapshotNow`), une ligne qui ne correspond pas aux clés de parcours peut être acceptée ou retournée. Ce scénario peut être évité en imposant que les clés de parcours soient re-confrontées à la ligne d'en-tête dans tous les cas, mais cela est trop coûteux. À la place, un lien sur une page d'index est utilisé comme *proxy* pour indiquer que le lecteur peut être « en parcours » entre l'entrée de l'index et l'entrée de l'en-tête correspondante. Bloquer `ambulkdelete` sur un tel lien assure que **VACUUM** ne peut pas supprimer l'entrée de l'en-tête avant que le lecteur n'en ait terminé avec elle. Cette solution est peu coûteuse en temps d'exécution, et n'ajoute de surcharge du fait du blocage que dans de rares cas réellement un conflictuels.

Cette solution requiert que les parcours d'index soient « synchrones » : chaque ligne d'en-tête doit être récupérée immédiatement après le parcours de l'entrée d'index correspondante. C'est coûteux pour plusieurs raisons. Un parcours « asynchrone » dans lequel de nombreux TID sont récupérés de l'index, et pour lequel les en-têtes de lignes ne sont visités que plus tard, requiert moins de surcharge de verrouillage d'index et autorise un modèle d'accès à l'en-tête plus efficace. D'après l'analyse ci-dessus, l'approche synchronisée doit être utilisée pour les instantanés non compatibles avec MVCC, mais un parcours asynchrone est possible pour une requête utilisant une instantané MVCC.

Dans un parcours d'index `amgetbitmap`, la méthode d'accès ne conserve pas de lien à l'index sur quelque ligne renvoyée. C'est pourquoi, il est préférable d'utiliser de tels parcours avec les instantanés compatibles MVCC.

51.5. Vérification de l'unicité de l'index

PostgreSQL™ assure les contraintes d'unicité SQL en utilisant des *index d'unicité*, qui sont des index qui refusent les entrées multiples à clés identiques. Une méthode d'accès qui supporte cette fonctionnalité initialise pg_am.*`amcanunique`* à true. (À ce jour, seul B-tree le supporte).

Du fait de MVCC, il est toujours nécessaire de permettre à des entrées dupliquées d'exister physiquement dans un index : les entrées peuvent faire référence à des versions successives d'une même ligne logique. Le comportement qu'il est réellement souhaitable d'assurer est qu'aucune image MVCC n'inclut deux lignes avec les mêmes clés d'index. Cela se résume aux cas suivants, qu'il est nécessaire de vérifier à l'insertion d'une nouvelle ligne dans un index d'unicité :

- si une ligne valide conflictuelle a été supprimée par la transaction courante, pas de problème. (En particulier, comme un UPDATE supprime toujours l'ancienne version de la ligne avant d'insérer la nouvelle version, cela permet un UPDATE sur une ligne sans changer la clé) ;
- si une ligne conflictuelle a été insérée par une transaction non encore validée, l'inséreur potentiel doit attendre de voir si la transaction est validée. Si la transaction est annulée, alors il n'y a pas de conflit. Si la transaction est validée sans que la ligne conflictuelle soit supprimée, il y a violation de la contrainte d'unicité. (En pratique, on attend que l'autre transaction finisse et le contrôle de visibilité est effectué à nouveau dans son intégralité) ;
- de façon similaire, si une ligne valide conflictuelle est supprimée par une transaction non encore validée, l'inserant potentiel doit attendre la validation ou l'annulation de cette transaction et recommencer le test.

De plus, immédiatement avant de lever une violation d'unicité en fonction des règles ci-dessus, la méthode d'accès doit revérifier l'état de la ligne en cours d'insertion. Si elle est validée tout en étant morte, alors aucune erreur ne survient. (Ce cas ne peut pas survenir lors du scénario ordinaire d'insertion d'une ligne tout juste créée par la transaction en cours. Cela peut néanmoins arriver lors d'un **CREATE UNIQUE INDEX CONCURRENTLY**.)

La méthode d'accès à l'index doit appliquer elle-même ces tests, ce qui signifie qu'elle doit accéder à l'en-tête pour vérifier le statut

de validation de toute ligne présentée avec une clé dupliquée au regard du contenu de l'index. C'est sans aucun doute moche et non modulaire, mais cela permet d'éviter un travail redondant : si un test séparé est effectué, alors la recherche d'une ligne conflictuelle dans l'index est en grande partie répétée lors de la recherche d'une place pour insérer l'entrée d'index de la nouvelle ligne. Qui plus, est, il n'y a pas de façon triviale d'éviter les conflits, sauf si la recherche de conflit est partie intégrante de l'insertion de la nouvelle entrée d'index.

Si la contrainte unique est déferrable, il y a une complication supplémentaire : nous devons être capable d'insérer une entrée d'index pour une nouvelle ligne mais de déferrer toute erreur de violation de l'unicité jusqu'à la fin de l'instruction, voire même après. Pour éviter des recherches répétées et inutiles de l'index, la méthode d'accès de l'index doit faire une vérification préliminaire d'unicité lors de l'insertion initiale. Si cela montre qu'il n'y a pas de conflit avec une ligne visible, nous avons terminé. Sinon, nous devons planifier une nouvelle vérification quand il sera temps de forcer la contrainte. Si, au moment de la nouvelle vérification, la ligne insérée et d'autres lignes de la même clé sont vivantes, alors l'erreur doit être reportée. (Notez que, dans ce contexte, « vivant » signifie réellement « toute ligne dans la chaine HOT de l'entrée de l'index est vivante ».) Pour implanter ceci, la fonction `aminsert` reçoit un paramètre `checkUnique` qui peut avoir une des valeurs suivantes :

- `UNIQUE_CHECK_NO` indicates that no uniqueness checking should be done (this is not a unique index).
- `UNIQUE_CHECK_YES` indique qu'il s'agit d'un index unique non déferrable et la vérification de l'unicité doit se faire immédiatement, comme décrit ci-dessus.
- `UNIQUE_CHECK_PARTIAL` indique que la contrainte unique est déferrable. PostgreSQL™ utilisera ce mode pour insérer l'entrée d'index de chaque ligne. La méthode d'accès doit autoriser les entrées dupliquées dans l'index et rapporter tout dupliquat potentiel en renvoyant FALSE à partir de `aminsert`. Pour chaque ligne pour laquelle FALSE est renvoyé, une revérification déferrée sera planifiée.

  La méthode d'accès doit identifier toute ligne qui pourrait violer la contrainte unique, mais rapporter des faux positifs n'est pas une erreur. Cela permet de faire la vérification sans attendre la fin des autres transactions ; les conflits rapportés ici ne sont pas traités comme des erreurs, et seront revérifiés plus tard, à un moment où ils ne seront peut-être plus en conflit.
- `UNIQUE_CHECK_EXISTING` indique que c'est une revérification déferrée d'une ligne qui a été rapportée comme en violation potentielle d'unicité. Bien que cela soit implanté par un appel à `aminsert`, la méthode d'accès ne doit *pas* insérer une nouvelle entrée d'index dans ce cas. L'entrée d'index est déjà présente. À la place, la méthode d'accès doit vérifier s'il existe une autre entrée d'index vivante. Si c'est le cas et que la ligne cible est toujours vivante, elle doit rapporter une erreur.

  Il est recommendé que, dans un appel à `UNIQUE_CHECK_EXISTING`, la méthode d'accès vérifie en plus que la ligne cible ait réellement une entrée existante dans l'index et de rapporter une erreur si ce n'est pas le cas. C'est une bonne idée car les valeurs de la ligne d'index passées à `aminsert` auront été recalculées. Si la définition de l'index implique des fonctions qui ne sont pas vraiment immutables, nous pourrions vérifier la mauvaise aire de l'index. Vérifier que la ligne cible est trouvée dans la revérification permet de s'assurer que nous recherchons les mêmes valeurs de la ligne comme elles ont été utilisées lors de l'insertion originale.

51.6. Fonctions d'estimation des coûts d'index

La fonction `amcostestimate` reçoit une liste de clauses WHERE qui ont été déterminées pour être utilisables avec l'index. Elle doit renvoyer une estimation du coût de l'accès à l'index et de la sélectivité des clauses WHERE (c'est-à-dire la fraction des lignes de la table parent qui seront récupérées lors du parcours de l'index). Pour les cas simples, pratiquement tout le travail de l'estimateur de coût peut être effectué en appelant des routines standard dans l'optimiseur ; la raison d'avoir une fonction `amcostestimate` est d'autoriser les méthodes d'accès aux index à fournir une connaissance spécifique au type d'index, au cas où il serait possible d'améliorer les estimations standard.

Chaque fonction `amcostestimate` doit avoir la signature :

```
void
amcostestimate (PlannerInfo *root,
                IndexOptInfo *index,
                List *indexQuals,
                RelOptInfo *outer_rel,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

Les quatre premiers paramètres sont des entrées :

`root`
: Information du planificateur sur la requête en cours de traitement.

index
: Index considéré.

indexQuals
: Liste des clauses de qualifications (*qual clauses*) d'index (implicitement assemblées avec des AND) ; une liste NIL indique qu'aucun qualificateur n'est disponible. La liste contient des arbres d'expression, pas des ScanKeys.

outer_rel
: Si l'utilisation de l'index est considérée dans un parcours d'index pour une jointure interne, c'est l'information du planificateur concernant le côté externe de la jointure. Sinon NULL. Si non NULL, certaines clauses de qualifications sont des clauses de jointure avec cette relation plutôt que de simples clauses de restriction. De plus, le processus d'estimation du coût doit s'attendre à ce que le parcours d'index soit répété pour chaque ligne de la relation externe.

Les quatre derniers paramètres sont les sorties passées par référence :

**indexStartupCost*
: Initialisé au coût du lancement du traitement de l'index.

**indexTotalCost*
: Initialisé au coût du traitement total de l'index.

**indexSelectivity*
: Initialisé à la sélectivité de l'index.

**indexCorrelation*
: Initialisé au coefficient de corrélation entre l'ordre du parcours de l'index et l'ordre sous-jacent de la table.

Les fonctions d'estimation de coûts doivent être écrites en C, pas en SQL ou dans tout autre langage de procédure, parce qu'elles doivent accéder aux structures de données internes du planificateur/optimiseur.

Les coûts d'accès aux index doivent être calculés en utilisant les paramètres utilisés par `src/backend/optimizer/path/costsize.c` : la récupération d'un bloc disque séquentiel a un coût de `seq_page_cost`, une récupération non séquentielle a un coût de `random_page_cost`, et le coût de traitement d'une ligne d'index doit habituellement être considéré comme `cpu_index_tuple_cost`. De plus, un multiple approprié de `cpu_operator_cost` doit être chargé pour tous les opérateurs de comparaison impliqués lors du traitement de l'index (spécialement l'évaluation des `indexQuals`).

Les coûts d'accès doivent inclure tous les coûts dûs aux disques et aux CPU associés au parcours d'index lui-même, mais *pas* les coûts de récupération ou de traitement des lignes de la table parent qui sont identifiées par l'index.

Le « coût de lancement » est la partie du coût total de parcours à dépenser avant de commencer à récupérer la première ligne. Pour la plupart des index, cela s'évalue à zéro, mais un type d'index avec un grand coût de lancement peut vouloir le configurer à une autre valeur que zéro.

indexSelectivity doit être initialisé à la fraction estimée des lignes de la table parent qui sera récupérée lors du parcours d'index. Dans le cas d'une requête à perte, c'est typiquement plus élevé que la fraction des lignes qui satisfont les conditions de qualification données.

indexCorrelation doit être initialisé à la corrélation (valeur entre -1.0 et 1.0) entre l'ordre de l'index et celui de la table. Cela permet d'ajuster l'estimation du coût de récupération des lignes de la table parent.

Dans le cas de la jointure, les nombres renvoyés doivent être les moyennes attendues pour tout parcours de l'index.

Procédure 51.1. Estimation du coût

Un estimateur typique de coût exécute le traitement ainsi :

1. Estime et renvoie la fraction des lignes de la table parent visitées d'après les conditions de qualification données. En l'absence de toute connaissance spécifique sur le type de l'index, on utilise la fonction de l'optimiseur standard `clauselist_selectivity()`:

```
*indexSelectivity = clauselist_selectivity(root, indexQuals,
                                           index->rel->relid,
                                           JOIN_INNER, NULL);
```

2. Estime le nombre de lignes d'index visitées lors du parcours. Pour de nombreux types d'index, il s'agit de *indexSelectivity* multiplié par le nombre de lignes dans l'index, mais cela peut valoir plus (la taille de l'index en pages et lignes est disponible à partir de la structure IndexOptInfo).

3. Estime le nombre de pages d'index récupérées pendant le parcours. Ceci peutt être simplement *`indexSelectivity`* multiplié par la taille de l'index en pages.
4. Calcule le coût d'accès à l'index. Un estimateur générique peut le faire ainsi :

```
/*
 * Our generic assumption is that the index pages will be read
 * sequentially, so they have cost seq_page_cost each, not random_page_cost.
 * Also, we charge for evaluation of the indexquals at each index row.
 * All the costs are assumed to be paid incrementally during the scan.
 */
cost_qual_eval(&index_qual_cost, indexQuals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

   Néanmoins, le calcul ci-dessus ne prend pas en compte l'amortissement des lectures des index à travers les parcours répétés d'index dans le cas de la jointure.
5. Estime la corrélation de l'index. Pour un index ordonné sur un seul champ, cela peut s'extraire de pg_statistic. Si la corrélation est inconnue, l'estimation conservative est zéro (pas de corrélation).

Des exemples de fonctions d'estimation du coût sont disponibles dans `src/backend/utils/adt/selfuncs.c`.

Chapitre 52. Index GiST

52.1. Introduction

GiST est un acronyme de *Generalized Search Tree*, c'est-à-dire arbre de recherche généralisé. C'est une méthode d'accès balancée à structure de type arbre, qui agit comme un modèle de base dans lequel il est possible d'implanter des schémas d'indexage arbitraires. B-trees, R-trees et de nombreux autres schémas d'indexage peuvent être implantés en GiST.

GiST a pour avantage d'autoriser le développement de types de données personnalisés avec les méthodes d'accès appropriées, par un expert en types de données, plutôt que par un expert en bases de données.

Quelques informations disponibles ici sont dérivées du *site web* du projet d'indexage GiST de l'université de Californie à Berkeley et de la *thèse de Marcel Kornacker, Méthodes d'accès pour les systèmes de bases de données de la prochaine génération*. L'implantation GiST de PostgreSQL™ est principalement maintenu par Teodor Sigaev et Oleg Bartunov. Leur *site web* fournit de plus amples informations.

52.2. Extensibilité

L'implantation d'une nouvelle méthode d'accès à un index a toujours été un travail complexe. Il est, en effet, nécessaire de comprendre le fonctionnement interne de la base de données, tel que le gestionnaire de verrous ou le WAL.

L'interface GiST dispose d'un haut niveau d'abstraction, ce qui autorise le codeur de la méthode d'accès à ne coder que la sémantique du type de données accédé. La couche GiST se charge elle-même de la gestion des accès concurrents, des traces et de la recherche dans la structure en arbre.

Cette extensibilité n'est pas comparable à celle des autres arbres de recherche standard en termes de données gérées. Par exemple, PostgreSQL™ supporte les B-trees et les index de hachage extensibles. Cela signifie qu'il est possible d'utiliser PostgreSQL™ pour construire un B-tree ou un hachage sur tout type de données. Mais, les B-trees ne supportent que les prédicats d'échelle (<, =, >), les index de hachage que les requêtes d'égalité.

Donc, lors de l'indexation d'une collection d'images, par exemple, avec un B-tree PostgreSQL™, seules peuvent être lancées des requêtes de type « est-ce que imagex est égale à imagey », « est-ce que imagex est plus petite que imagey » et « est-ce que imagex est plus grande que imagey ». En fonction de la définition donnée à « égale à », « inférieure à » ou « supérieure à », cela peut avoir une utilité. Néanmoins, l'utilisation d'un index basé sur GiST permet de créer de nombreuses possibilités de poser des questions spécifiques au domaine, telles que « trouver toutes les images de chevaux » ou « trouver toutes les images surexposées ».

Pour obtenir une méthode d'accès GiST fonctionnelle, il suffit de coder sept méthodes utilisateur définissant le comportement des clés dans l'arbre. Ces méthodes doivent être suffisamment élaborées pour supporter des requêtes avancées, mais pour toutes les requêtes standard (B-trees, R-trees, etc.) elles sont relativement simples. En bref, GiST combine extensibilité, généralité, réutilisation de code et interface claire.

52.3. Implantation

Une classe d'opérateur d'index GiST doit fournir sept méthodes. La précision de l'index est assurée par l'implantation des méthodes `same`, `consistent` et `union` alors que l'efficacité (taille et rapidité) de l'index dépendra des méthodes `penalty` et `picksplit`. Les deux fonctions restantes sont `compress` et `decompress`, qui permettent à un index d'avoir des données internes de l'arbre d'un type différent de ceux des données qu'il indexe. Les feuilles doivent être du type des données indexées alors que les autres nœuds peuvent être de n'importe quelle structure C (mais vous devez toujours suivre les règles des types de données de PostgreSQL™ dans ce cas, voir ce qui concerne `varlena` pour les données de taille variable). Si le type de données interne de l'arbre existe au niveau SQL, l'option `STORAGE` de la commande **CREATE OPERATOR CLASS** peut être utilisée.

`consistent`

Étant donné une entrée d'index `p` et une valeur de requête `q`, cette fonction détermine si l'entrée de l'index est cohérente (« consistent » en anglais) avec la requête ; c'est-à-dire, est-ce que le prédicat « *`colonne_indexée opérateur_indexable`* `q` » soit vrai pour toute ligne représentée par l'entrée de l'index ? Pour une entrée de l'index de type feuille, c'est l'équivalent pour tester la condition indexable, alors que pour un nœud interne de l'arbre, ceci détermine s'il est nécessaire de parcourir le sous-arbre de l'index représenté par le nœud. Quand le résultat est `true`, un drapeau `recheck` doit aussi être renvoyé. Ceci indique si le prédicat est vrai à coup sûr ou seulement peut-être vrai. Si `recheck` = `false`, alors l'index a testé exactement la condition du prédicat, alors que si `recheck` = `true`, la ligne est seulement un correspondance de candidat. Dans ce cas, le système évaluera automatiquement l'*`opérateur_indexable`* avec la valeur actuelle de la ligne pour voir s'il s'agit réellement d'une correspondance. Cette convention permet à GiST de supporter à la fois

les structures sans pertes et celles avec perte de l'index.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid,
internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_consistent(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * determine return value as a function of strategy, key and query.
     *
     * Use GIST_LEAF(entry) to know where you're called in the index tree,
     * which comes handy when supporting the = operator for example (you could
     * check for non empty union() in non-leaf nodes and equality in leaf
     * nodes).
     */

    *recheck = true;        /* or false if check is exact */

    PG_RETURN_BOOL(retval);
}
```

Ici, `key` est un élément dans l'index et `query` la valeur la recherchée dans l'index. Le paramètre `StrategyNumber` indique l'opérateur appliqué de votre classe d'opérateur. Il correspond à un des nombres d'opérateurs dans la commande **CREATE OPERATOR CLASS**. Suivant les opérateurs que vous avez inclus dans la classe, le type de données de `query` pourrait varier avec l'opérateur, mais le squelette ci-dessus suppose que ce n'est pas le cas.

`union`

Cette méthode consolide l'information dans l'arbre. Suivant un ensemble d'entrées, cette fonction génère une nouvelle entrée d'index qui représente toutes les entrées données.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_union(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
```

```
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

Comme vous pouvez le voir dans ce quelette, nous gérons un type de données où `union(X, Y, Z) = union(union(X, Y), Z)`. C'est assez simple pour supporter les types de données où ce n'est pas le cas, en implantant un autre algorithme d'union dans cette méthode de support GiST.

La fonction d'implantation de `union` doit renvoyer un pointeur vers la mémoire qui vient d'être allouée via la fonction `palloc()`. Vous ne pouvez pas tout simplement renvoyer l'entrée.

`compress`

Convertit l'élément de données dans un format compatible avec le stockage physique dans une page d'index.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_compress(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* replace entry->key with a compressed version */
        compressed_data_type *compressed_data =
palloc(sizeof(compressed_data_type));

        /* fill *compressed_data from entry->key ... */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* typically we needn't do anything with non-leaf entries */
        retval = entry;
    }
```

```
    PG_RETURN_POINTER(retval);
}
```

Vous devez adapter *compressed_data_type* au type spécifique que vous essayez d'obtenir pour compresser les nœuds finaux.

Vous pourriez aussi avoir besoin de faire attention à la compression des valeurs `NULL`, en enregistrant par exemple `(Datum) 0` comme le fait `gist_circle_compress`.

`decompress`

L'inverse de la fonction `compress`. Convertit la représentation de l'élément de donnée en un format manipulable par la base de données.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_decompress(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

Le squelette ci-dessus est convenable dans le cas iù aucune décompression n'est nécessaire.

`penalty`

Renvoie une valeur indiquant le « coût » d'insertion d'une nouvelle entrée dans une branche particulière de l'arbre. Les éléments seront insérés dans l'ordre des pénalités moindres (`penalty`) de l'arbre. Les valeurs renvoyées par la fonction `penalty` doivent être positives ou nulles. Si une valeur négative est renvoyée, elle sera traitée comme une valeur zéro.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- in some cases penalty functions need not be strict
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_penalty(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

La fonction `penalty` est crucial pour de bonnes performances de l'index. Elle sera utilisée lors de l'insertion pour déterminer la branche à suivre pour savoir où ajoter la nouvelle entrée dans l'arbre. Lors de l'exécution de la requête, plus l'arbre sera bien balancé, plus l'exécution sera rapide.

`picksplit`

Quand une division de page est nécessaire pour un index, cette fonction décide des entrées de la page qui resteront sur l'ancienne page et de celles qui seront déplacées sur la nouvelle page.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_picksplit(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    int         i,
                nbytes;
    OffsetNumber *left,
                *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* Initialize the raw entry vector. */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * Choose where to put the index entries and update unionL and unionR
         * accordingly. Append the entries to either v_spl_left or
         * v_spl_right, and care about the counters.
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
```

```
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
        }
        else
        {
            /*
             * Same on the right
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

Comme `penalty`, la fonction `picksplit` est cruciale pour de bonnes performances de l'index. Concevoir des implantations convenables des fonctions `penalty` et `picksplit` est le challenge d'un index GiST performant.

`same`
: Renvoit true si les deux entrées de l'index sont identiques, faux sinon.

La déclaration SQL de la fonction ressemble à ceci :

```
CREATE OR REPLACE FUNCTION my_same(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
Datum       my_same(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

Pour des raisons historiques, la fonction `same` ne renvoie pas seulement un résultat booléen ; à la place, il doit enregistrer le drapeau à l'emplacement indiqué par le troisième argument.

52.4. Exemples

La distribution source de PostgreSQL™ inclut plusieurs exemples de méthodes d'indexation implantées selon GiST. Le système principal fournit des fonctionnalités de recherche plein texte (indexation des tsvector et tsquery) ainsi que des fonctionnalités équivalentes aux R-Tree pour certains types de données géométriques (voir `src/backend/access/gist/gistproc.c`). Les modules `contrib` suivants contiennent aussi des classes d'opérateur GiST :

`btree_gist`
: Fonctionnalités équivalentes aux B-Tree pour plusieurs types de données

`cube`
: Indexation de cubes multi-dimensionnels

`hstore`
: Module pour le stockage des paires (clé, valeur)

`intarray`
: RD-Tree pour tableaux uni-dimensionnels de valeurs int4

`ltree`
: Indexation des structures de type arbre

`pg_trgm`
: Similarité textuelle par correspondance de trigrammes

`seg`
: Indexation pour les « nombres flottants »

52.5. Récupération après un arrêt brutal

Il est généralement suffisant de rejouer les traces WAL pour restaurer l'intégrité d'un index GiST après un arrêt brutal de la base de données. Néanmoins, il existe quelques cas particuliers pour lesquels l'état de l'index n'est pas entièrement reconstruit. L'index reste fonctionnellement correct mais les performances peuvent se trouver dégradées. Lorsque cela arrive, l'index peut être réparé par **VACUUM** de sa table, ou par reconstruction avec **REINDEX**. Dans certains cas, un simple **VACUUM** ne suffit pas et un **VACUUM FULL** ou un **REINDEX** est nécessaire. Ce besoin est indiqué par la survenue d'un tel message :

```
LOG:  index NNN/NNN/NNN needs VACUUM or REINDEX to finish crash recovery
```

ou le message suivant lors d'insertions dans l'index :

```
LOG:  index "FOO" needs VACUUM or REINDEX to finish crash recovery
```

Si un simple **VACUUM** est incapable de réaliser complètement la récupération, l'avertissement suivant est envoyé :

```
NOTICE:  index "FOO" needs VACUUM FULL or REINDEX to finish crash recovery
```

Chapitre 53. Index GIN

53.1. Introduction

GIN est l'acronyme de *Generalized Inverted Index* (ou index générique inverse). C'est une structure d'index stockant des ensembles de paires (clé, liste de lignes) où le terme « liste de lignes » est un ensemble de lignes dans lesquelles la clé est valable. Chaque valeur indexée peut contenir plusieurs clés, le même identifiant de ligne peut ainsi apparaître dans plusieurs listes.

Il est généralisé dans le sens où un index GIN n'a pas à connaître l'opération qu'il accélère. À la place, il utilise des stratégies adaptées définies pour des types de données particuliers.

Un des avantages de GIN est la possibilité qu'il offre que des types de données personnalisés et les méthodes d'accès appropriées soient développés par un expert du domaine du type de données, plutôt que par un expert en bases de données. L'utilisation de GiST offre le même avantage.

L'implantation de GIN dans PostgreSQL™ est principalement l'oeuvre de Teodor Sigaev et Oleg Bartunov. Plus d'informations sur GIN sont disponibles sur leur *site web*.

53.2. Extensibilité

L'interface GIN a un haut niveau d'abstraction. De ce fait, la personne qui code la méthode d'accès n'a besoin d'implanter que les sémantiques du type de données accédé. La couche GIN prend en charge la gestion de la concurrence, des traces et des recherches dans la structure de l'arbre.

Pour obtenir une méthode d'accès GIN fonctionnelle, il suffit d'implanter quatre (ou cinq) méthodes utilisateur. Celles-ci définissent le comportement des clés dans l'arbre et les relations entre clés, valeurs indexées et requêtes indexables. En résumé, GIN combine extensibilité, généralisation, ré-utilisation du code à une interface claire.

Les quatre méthodes qu'une classe d'opérateur GIN doit fournir sont :

`int compare(Datum a, Datum b)`
: Compare les clés (et non les valeurs indexées !) et renvoie un entier négatif, zéro ou un entier positif, qui indique si la première clé est inférieure, égale à ou supérieure à la seconde.

`Datum *extractValue(Datum inputValue, int32 *nkeys)`
: Renvoie un tableau de clés à partir d'une valeur à indexer. Le nombre de clés renvoyées doit être stocké dans `*nkeys`.

`Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data)`
: Renvoie un tableau de clés en fonction de la valeur à requêter ; c'est-à-dire que `query` est la valeur du côté droit d'un opérateur indexable dont le côté gauche est la colonne indexée. `n` est le numéro de stratégie de l'opérateur dans la classe d'opérateur (voir Section 35.14.2, « Stratégies des méthode d'indexation »). Souvent, `extractQuery` doit consulter `n` pour déterminer le type de données de `query` et les valeurs de clés à extraire. Le nombre de clés renvoyées doit être stocké dans `*nkeys`. Si `query` ne contient aucune clé alors `extractQuery` devra retourner 0 ou -1 dans `*nkeys`, en fonction des sémantiques de l'opérateur. 0 signifie que toutes les valeurs correspondent à `query` et qu'un parcours d'index complet doit être effectué (mais voir Section 53.5, « Limitations » pour plus de détails). -1 signifie que rien ne peut correspondre à `query`, ce qui entraîne que le parcours d'index peut être totalement évité. `pmatch` est un paramètre de sortie à utiliser quand une correspondance partielle est permise. Pour l'utiliser, `extractQuery` doit allouer un tableau de booléens `*nkeys` et stocker son adresse dans `*pmatch`. Chaque élément du tableau devrait être positionné à TRUE si la clé correspondante a besoin d'une correspondance partielle, FALSE sinon. Si `*pmatch` est positionné à NULL alors GIN suppose qu'une mise en correspondance partielle n'est pas nécessaire. La variable est initialisée à NULL avant l'appel, et peut donc être simplement ignorée par les classes d'opérateurs qui ne supportent pas les correspondances partielles. `extra_data` est un paramètre de sortie qui autorise `extractQuery` à passer des données supplémentaires aux méthodes `consistent` et `comparePartial`. Pour l'utiliser, `extractQuery` doit allouer un tableau de pointeurs `*nkeys` et stocker son adresse à `*extra_data`, puis stocker ce qu'il souhaite dans les pointeurs individuels. La variable est initialisée à NULL avant l'appel, afin que ce paramètre soit simplement ignoré par une classe d'opérateurs qui n'a pas besoin de données supplémentaires. Si `*extra_data` est positionné, le tableau dans son ensemble est passé à la méthode `consistent` method, et l'élément approprié à la méthode `comparePartial`.

`bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck)`
: Retourne TRUE si la valeur indexée satisfait à l'opérateur de requête avec le numéro de stratégie `n` (ou pourrait satisfaire, si l'indication recheck est retournée). Le tableau `check` a la longueur `nkeys`, qui est la même que le nombre de clés retournées précédemment par `extractQuery` pour ce datum, `query`. Chaque élément du tableau `check` est TRUE si la va-

leur indexée contient la clé correspondante de la requête, c'est-à-dire que si (check[i] == TRUE), la i-ème clé du tableau résultant d'`extractQuery` est présente dans la valeur indexée. Le datum `query` d'origine (pas le tableau de clés extrait !) est passé au cas où la méthode `consistent` a besoin de le consulter. `extra_data` est le tableau de données supplémentaires retourné par `extractQuery`, ou NULL si aucun. En cas de succès, `*recheck` devrait être positionné à TRUE si l'enregistrement du heap a besoin d'être revérifié vis-à-vis de l'opérateur de la requête, ou FALSE si le test sur l'index est exact.

En option, une classe d'opérateurs pour GIN peut fournir une cinquième méthode :

`int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)`
: Compare une requête de correspondance partielle à une clé d'index. Renvoie un entier dont le signe indique le résultat : inférieur à zéro signifie que la clé d'index ne correspond pas à la requête mais que le parcours d'index va continuer ; zéro signifie que la clé d'index ne correspond pas à la requête ; supérieur à zéro indique que le parcours d'index doit s'arrêter car il n'existe pas d'autres correspondances. Le numéro de stratégie `n` de l'opérateur qui a généré la requête de correspondance partielle est fourni au cas où sa sémantique est nécessaire pour déterminer la fin du parcours. De plus, `extra_data` est l'élément correspondant du tableau extra-data fait par `extractQuery`, ou NULL sinon.

Pour supporter des requêtes à « correspondance partielle », une classe d'opérateur doit fournir la méthode `comparePartial`, et sa méthode `extractQuery` doit positionner le paramètre `pmatch` quand une requête à correspondance partielle est rencontrée. Voir Section 53.3.2, « Algorithme de mise en correspondance partielle » pour les détails.

53.3. Implantation

En interne, un index GIN contient un index B-tree construit sur des clés, où chaque clé est un élément de la valeur indexée (un membre d'un tableau par exemple) et où chaque ligne d'une page feuille est, soit un pointeur vers un B-tree de pointeurs heap (PT, posting tree), soit une liste de pointeurs heap (PL, posting list) si la liste est suffisamment petite.

53.3.1. Technique GIN de mise à jour rapide

Mettre à jour un index GIN a tendance à être lent en raison de la nature intrinsèque des index inversés : insérer ou mettre à jour un enregistrement de la table peut causer de nombreuses insertions dans l'index (une pour chaque clé extraite de la valeur indexée). À partir de PostgreSQL™ 8.4, GIN est capable de reporter à plus tard la plupart de ce travail en insérant les nouveaux enregistrements dans une liste temporaire et non triée des entrées en attente. Quand un vacuum est déclenché sur la table, ou si la liste en attente devient trop grosse (plus grande que work_mem), les entrées sont déplacées vers la structure de données GIN principale en utilisant la même technique d'insertion de masse que durant la création de l'index. Ceci améliore grandement la vitesse de mise à jour de l'index GIN, même en prenant en compte le surcoût engendré au niveau du vacuum. De plus, ce surcoût peut être attribué à un processus d'arrière-plan plutôt qu'à la requête en avant-plan.

Le principal défaut de cette approche est que les recherches doivent parcourir la liste d'entrées en attente en plus de l'index habituel, et que par conséquent une grande liste d'entrées en attente ralentira les recherches de façon significative. Un autre défaut est que, bien que la majorité des mises à jour seront rapides, une mise à jour qui rend la liste d'attente « trop grande » déclenchera un cycle de nettoyage immédiat et sera donc bien plus lente que les autres mises à jour. Une utilisation appropriée d'autovacuum peut minimiser ces deux problèmes.

Si la cohérence des temps de réponse est plus importante que la vitesse de mise à jour, l'utilisation de liste d'entrées en attente peut être désactivée en désactivant le paramètre de stockage `FASTUPDATE` pour un index GIN. Voir CREATE INDEX(7) pour plus de détails.

53.3.2. Algorithme de mise en correspondance partielle

GIN peut supporter des requêtes de « correspondances partielles », dans lesquelles la requête ne détermine pas une correspondance parfaite pour une ou plusieurs clés, mais que la correspondance tombe à une distance suffisamment faible des valeurs de clé (dans l'ordre de tri des clés déterminé par la méthode de support `compare`). La méthode `extractQuery`, au lieu de retourner une valeur de clé à mettre en correspondance de façon exacte, retourne une valeur de clé qui est la limite inférieure de la plage à rechercher, et retourne l'indicateur `pmatch` positionné à true. La plage de clé est alors parcourue en utilisant la méthode `comparePartial`. `comparePartial` doit retourner 0 pour une vraie correspondance, une valeur négative pour une non-correspondance qui est toujours dans la plage de recherche, et une valeur positive si la clé d'index est sortie de la plage qui pourrait correspondre.

53.4. Conseils et astuces GIN

Création vs insertion
: L'insertion dans un index GIN peut être lente du fait de la probabilité d'insertion de nombreuses clés pour chaque valeur. C'est

pourquoi, pour les chargements massifs dans une table, il est conseillé de supprimer l'index GIN et de le re-créer après le chargement.

À partir de PostgreSQL™ 8.4, ce conseil est moins important puisqu'une technique de mise à jour retardée est utilisée (voir Section 53.3.1, « Technique GIN de mise à jour rapide » pour plus de détails). Mais pour les très grosses mises à jour, il peut toujours être plus efficace de détruire et recréer l'index.

maintenance_work_mem
: Le temps de construction d'un index GIN dépend grandement du paramètre `maintenance_work_mem` ; il est contre-productif de limiter la mémoire de travail lors de la création d'un index.

work_mem
: Durant une série d'insertions dans un index GIN existant qui a `FASTUPDATE` activé, le système nettoiera la liste d'entrées en attente dès qu'elle deviendra plus grosse que `work_mem`. Afin d'éviter des fluctuations mesurables de temps de réponse, il est souhaitable d'avoir un nettoyage de la liste d'attente en arrière-plan (c'est-à-dire via autovacuum). Les opérations de nettoyage en avant-plan peuvent être évitées en augmentant `work_mem` ou en rendant autovacuum plus aggressif. Toutefois, augmenter `work_mem` implique que si un nettoyage en avant-plan se produit, il prendra encore plus longtemps.

gin_fuzzy_search_limit
: La raison principale qui a poussé le développement des index GIN a été la volonté de supporter les recherches plein texte dans PostgreSQL™ et il arrive fréquemment qu'une recherche renvoie un ensemble volumineux de résultats. Cela arrive d'autant plus fréquemment que la requête contient des mots très fréquents, auquel cas l'ensemble de résultats n'est même pas utile. Puisque la lecture des lignes sur disque et leur tri prend beaucoup de temps, cette situation est inacceptable en production. (La recherche dans l'index est, elle, très rapide.)

Pour faciliter l'exécution contrôlée de telles requêtes, GIN dispose d'une limite supérieure souple configurable du nombre de lignes renvoyées, le paramètre de configuration `gin_fuzzy_search_limit`. Par défaut, il est positionné à 0 (c'est-à-dire sans limite). Si une limite différente de 0 est choisie, alors l'ensemble renvoyé est un sous-ensemble du résultat complet, choisi aléatoirement.

« Souple » signifie que le nombre réel de résultats renvoyés peut différer légèrement de la limite indiquée, en fonction de la requête et de la qualité du générateur de nombres aléatoires du système.

53.5. Limitations

GIN ne supporte pas les parcours d'index complets. La raison est que `extractValue` est autorisée à retourner zéro clés, ce qui pourrait par exemple arriver avec une chaîne vide ou un tableau vide. Dans ce cas la valeur indexée ne sera pas présente dans l'index. Il est donc impossible à GIN de garantir que le parcours d'un index trouvera tous les enregistrements de la table.

En raison de cette limitation, quand `extractQuery` retourne `nkeys = 0` pour indiquer que toutes les valeurs correspondent à la requête, GIN émettra une erreur. (Si il y a plusieurs opérateurs indexables liés par des ET logiques, cela n'arrivera que si ils retournent tous `nkeys = 0`.)

Il est possible pour un opérateur de contourner cette restriction sur les scans d'index. Pour cela, `extractValue` doit conserver au moins une (éventuellement factice) clé pour toutes les valeurs indexées, et `extractQuery` doit convertir la recherche sans limitations en une requête de correspondance partielle qui parcourera l'ensemble de l'index. C'est inefficace, mais peut être nécessaire pour éviter des cas particuliers d'échec avec certains opértateurs comme `LIKE` ou l'inclusion de sous-ensemble.

GIN suppose que les opérateurs indexables sont stricts. Cela signifie que `extractValue` ne sera pas appelé du tout sur une valeur NULL (et que donc la valeur ne sera pas indexée), et que `extractQuery` ne sera pas appelée sur une valeur de comparaison NULL non plus (à la place, la requête est supposée ne pouvant pas correspondre).

Une limitation qui peut être plus sérieuse est que GIN ne peut pas pas gérer des clés NULL -- par exemple, un tableau contenant une valeur NULL ne peut pas être gérée, excepté en ignorant la valeur NULL.

53.6. Exemples

Les sources de PostgreSQL™ incluent des classes d'opérateur GIN pour tsvector et pour les tableaux unidimensionnels de tous les types internes. La recherche de préfixe dans tsvector est implémentée en utilisant les correspondances partielles de GIN. Les modules `contrib` suivants contiennent aussi des classes d'opérateurs GIN :

`btree-gin`
: Fonctionnalité équivalente à B-tree pour plusieurs types de données

`hstore`
: Module pour le stockage des paires (clé, valeur)

intarray
: Support amélioré pour le type int[]

`pg_trgm`
: Similarité de texte par correspondance de trigramme

Chapitre 54. Stockage physique de la base de données

Ce chapitre fournit un aperçu du format de stockage physique utilisé par les bases de données PostgreSQL™.

54.1. Emplacement des fichiers de la base de données

Cette section décrit le format de stockage au niveau des fichiers et répertoires.

Toutes les données nécessaires à un groupe de bases de données sont stockées dans le répertoire data du groupe, habituellement référencé en tant que `PGDATA` (d'après le nom de la variable d'environnement qui peut être utilisé pour le définir). Un emplacement courant pour `PGDATA` est `/var/lib/pgsql/data`. Plusieurs groupes, gérés par différentes instances du serveur, peuvent exister sur la même machine.

Le répertoire `PGDATA` contient plusieurs sous-répertoires et fichiers de contrôle, comme indiqué dans le Tableau 54.1, « Contenu de `PGDATA` ». En plus de ces éléments requis, les fichiers de configuration du groupe, `postgresql.conf`, `pg_hba.conf` et `pg_ident.conf` sont traditionnellement stockés dans `PGDATA` (bien qu'il soit possible de les conserver ailleurs à partir de la version 8.0 de PostgreSQL™).

Tableau 54.1. Contenu de `PGDATA`

Élément	Description
`PG_VERSION`	Un fichier contenant le numéro de version majeur de PostgreSQL™
`base`	Sous-répertoire contenant les sous-répertoires par base de données
`global`	Sous-répertoire contenant les tables communes au groupe, telles que pg_database
`pg_clog`	Sous-répertoire contenant les données d'état de validation des transactions
`pg_multixact`	Sous-répertoire contenant des données sur l'état des multi-transactions (utilisé pour les verrous de lignes partagées)
`pg_notify`	Sous-répertoire contenant les données de statut de LISTEN/NOTIFY
`pg_stat_tmp`	Sous-répertoire contenant les fichiers temporaires pour le sous-système des statistiques
`pg_subtrans`	Sous-répertoire contenant les données d'états des sous-transaction
`pg_tblspc`	Sous-répertoire contenant les liens symboliques vers les espaces logiques
`pg_twophase`	Sous-répertoire contenant les fichiers d'état pour les transactions préparées
`pg_xlog`	Sous-répertoire contenant les fichiers WAL (Write Ahead Log)
`postmaster.opts`	Un fichier enregistrant les options en ligne de commande avec lesquelles le serveur a été lancé la dernière fois
`postmaster.pid`	Un fichier verrou enregistrant le PID courant du serveur et l'identifiant du segment de mémoire partagé (absent après l'arrêt du serveur)

Pour chaque base de données dans le groupe, il existe un sous-répertoire dans `PGDATA/base`, nommé d'après l'OID de la base de données dans pg_database. Ce sous-répertoire est l'emplacement par défaut pour les fichiers de la base de données; en particulier, ses catalogues système sont stockés ici.

Chaque table et index sont stockés dans un fichier séparé, nommé d'après le numéro *filenode* de la table ou de l'index, lequel se trouve dans pg_class.*`relfilenode`*. En plus du fichier principal (appelé aussi « main fork »), chaque table et index a une *carte des espaces libres* (voir Section 54.3, « Carte des espaces libres »), qui stocke des informations sur les emplacements libres dans la relation. La carte des espace libre est stockée dans un fichier nommé par le numéro relfilenode à qui est ajouté le suffixe `_fsm`. Les tables ont aussi un fichier pour la *carte de visibilité*, de suffixe `_vm`, pour tracer les pages connues pour n'avoir aucune ligne morte. La carte de visibilité est décrite plus bas dans Section 54.4, « Carte de visibilité ».

> **Attention**
>
> Notez que, bien que le filenode de la table correspond souvent à son OID, cela n'est *pas* nécessairement le cas; certaines opérations, comme **TRUNCATE**, **REINDEX**, **CLUSTER** et quelques formes d'**ALTER TABLE**, peuvent modifier le filenode tout en préservant l'OID. Évitez de supposer que filenode et OID sont identiques. De

plus, pour certains catalogues système incluant pg_class lui-même, pg_class.*relfilenode* contient zéro. Le numéro filenode en cours est stocké dans une structure de données de bas niveau, et peut être obtenu avec la fonction `pg_relation_filenode()`.

Quand une table ou un index dépasse 1 Go, il est divisé en *segments* d'un Go. Le nom du fichier du premier segment est identique au filenode ; les segments suivants sont nommés filenode.1, filenode.2, etc. Cette disposition évite des problèmes sur les plateformes qui ont des limitations sur les tailles des fichiers. (Actuellement, 1 Go est la taille du segment par défaut. Cette taille est ajustable en utilisant l'option `--with-segsize` pour configure avant de construire PostgreSQL™.) En principe, les fichiers de la carte des espaces libres et de la carte de visibilité pourraient aussi nécessiter plusieurs segments, bien qu'il y a peu de chance que cela arrive réellement.

Une table contenant des colonnes avec des entrées potentiellement volumineuses aura une table *TOAST* associée, qui est utilisée pour le stockage de valeurs de champs trop importantes pour conserver des lignes adéquates. pg_class.*reltoastrelid* établit un lien entre une table et sa table TOAST, si elle existe. Voir Section 54.2, « TOAST » pour plus d'informations.

Le contenu des tables et des index est discuté plus en détails dans Section 54.5, « Emplacement des pages de la base de données ».

Les tablespaces rendent ce scénario plus compliqués. Chaque espace logique défini par l'utilisateur contient un lien symbolique dans le répertoire `PGDATA/pg_tblspc`, pointant vers le répertoire physique du tablespace (celui spécifié dans la commande **CREATE TABLESPACE**). Ce lien symbolique est nommé d'après l'OID du tablespace. À l'intérieur du répertoire du tablespace, il existe un sous-répertoire avec un nom qui dépend de la version du serveur PostgreSQL™, comme par exemple `PG_9.0_201008051`. (La raison de l'utilisation de ce sous-répertoire est que des versions successives de la base de données puissent utiliser le même emplacement indiqué par **CREATE TABLESPACE** sans que cela provoque des conflits.) À l'intérieur de ce répertoire spécifique à la version, il existe un sous-répertoire pour chacune des bases de données contenant des éléments dans ce tablespace. Ce sous-répertoire est nommé d'après l'OID de la base. Les tables et les index sont enregistrés dans ce répertoire et suivent le schéma de nommage des filenodes. Le tablespace `pg_default` n'est pas accédé via `pg_tblspc` mais correspond à `PGDATA/base`. De façon similaire, le tablespace `pg_global` n'est pas accédé via `pg_tblspc` mais correspond à `PGDATA/global`.

La fonction `pg_relation_filepath()` affiche le chemin entier (relatif à `PGDATA`) de toute relation. Il est souvent utile pour ne pas avoir à se rappeler toutes les différentes règles ci-dessus. Gardez néanmoins en tête que cette fonction donne seulement le nom du premier segment du fichier principal de la relation -- vous pourriez avoir besoin d'ajouter le numéro de segment et/ou les extensions `_fsm` ou `_vm` pour trouver tous les fichiers associés avec la relation.

Les fichiers temporaires (pour des opérations comme le tri de plus de données que ce que la mémoire peut contenir) sont créés à l'intérieur de `PGDATA/base/pgsql_tmp`, ou dans un sous-répertoire `pgsql_tmp` du répertoire du tablespace si un tablespace autre que `pg_default` est indiqué pour eux. Le nom du fichier temporaire est de la forme `pgsql_tmp`*PPP*`.`*NNN*, où *PPP* est le PID du serveur propriétaire et *NNN* distingue les différents fichiers temporaires de ce serveur.

54.2. TOAST

Cette section fournit un aperçu de TOAST (*The Oversized-Attribute Storage Technique*, la technique de stockage des attributs trop grands).

Puisque PostgreSQL™ utilise une taille de page fixe (habituellement 8 Ko) et n'autorise pas qu'une ligne s'étende sur plusieurs pages. Du coup, il n'est pas possible de stocker de grandes valeurs directement dans les champs. Pour dépasser cette limitation, les valeurs de champ volumineuses sont compressées et/ou divisées en plusieurs lignes physiques. Ceci survient de façon transparente pour l'utilisateur, avec seulement un petit impact sur le code du serveur. Cette technique est connu sous l'acronyme affectueux de TOAST (ou « the best thing since sliced bread »).

Seuls certains types de données supportent TOAST -- il n'est pas nécessaire d'imposer cette surcharge sur les types de données qui ne produisent pas de gros volumes. Pour supporter TOAST, un type de données doit avoir une représentation (*varlena*) à longueur variable, dans laquelle les 32 premiers bits contiennent la longueur totale de la valeur en octets (ceci incluant la longueur elle-même). TOAST n'a aucune contrainte supplémentaire sur la représentation. Toutes les fonctions niveau C qui gèrent un type données supportant TOAST doivent faire attention à gérer les valeurs en entrée TOASTées. (Ceci se fait normalement en appelant `PG_DETOAST_DATUM` avant de faire quoi que ce soit avec une valeur en entrée; mais dans certains cas, des approches plus efficaces sont possibles.)

TOAST récupère deux bits du mot contenant la longueur d'un varlena (ceux de poids fort sur les machines big-endian, ceux de poids faible sur les machines little-endian), limitant du coup la taille logique de toute valeur d'un type de données TOAST à 1 Go (2^{30} - 1 octets). Quand les deux bits sont à zéro, la valeur est une valeur non TOASTé du type de données et les bits restants dans le mot contenant la longueur indiquent la taille total du datum (incluant ce mot) en octets. Quand le bit de poids fort (ou de poids faible) est à un, la valeur a un en-tête de seulement un octet alors qu'un en-tête normal en fait quatre. Les bits restants donnent la taille total du datum (incluant ce mot) en octets. Il reste un cas spécial : si les bits restants sont tous à zéro (ce qui est impossible étant donné que le mot indiquant la longueur est inclus dans la taille), la valeur est un pointeur vers une donnée stockée dans une table TOAST séparée (la taille d'un pointeur TOAST est indiquée dans le second octet du datum). Les valeurs dont l'en-tête fait un

seul octet ne sont pas alignées sur une limite particulière. Enfin, quand le bit de poids fort (ou de poids faible) est supprimé mais que le bit adjacent vaut un, le contenu du datum est compressé et doit être décompressé avant utilisation. Dans ce cas, les bits restants du mot contenant la longueur indiquent la taille totale du datum compressé, pas celles des données au départ. Notez que la compression est aussi possible pour les données de la table TOAST mais l'en-tête varlena n'indique pas si c'est le cas -- le contenu du pointeur TOAST le précise.

Si une des colonnes d'une table est TOAST-able, la table disposera d'une table TOAST associé, dont l'OID est stockée dans l'entrée pg_class.*reltoastrelid* de la table. Les valeurs TOASTées hors-ligne sont conservées dans la table TOAST comme décrit avec plus de détails ci-dessous.

La technique de compression utilisée est un simple et rapide membre de la famille des techniques de compression LZ. Voir `src/backend/utils/adt/pg_lzcompress.c` pour les détails.

Les valeurs hors-ligne sont divisées (après compression si nécessaire) en morceaux d'au plus `TOAST_MAX_CHUNK_SIZE` octets (par défaut, cette valeur est choisie pour que quatre morceaux de ligne tiennent sur une page, d'où les 2000 octets). Chaque morceau est stocké comme une ligne séparée dans la table TOAST de la table propriétaire. Chaque table TOAST contient les colonnes *chunk_id* (un OID identifiant la valeur TOASTée particulière), *chunk_seq* (un numéro de séquence pour le morceau de la valeur) et *chunk_data* (la donnée réelle du morceau). Un index unique sur *chunk_id* et *chunk_seq* offre une récupération rapide des valeurs. Un pointeur datum représentant une valeur TOASTée hors-ligne a par conséquent besoin de stocker l'OID de la table TOAST dans laquelle chercher et l'OID de la valeur spécifique (son *chunk_id*). Par commodité, les pointeurs datums stockent aussi la taille logique du datum (taille de la donnée originale non compressée) et la taille stockée réelle (différente si la compression a été appliquée). À partir des octets d'en-tête varlena, la taille totale d'un pointeur datum TOAST est par conséquent de 18 octets quelque soit la taille réelle de la valeur représentée.

Le code TOAST est déclenché seulement quand une valeur de ligne à stocker dans une table est plus grande que TOAST_TUPLE_THRESHOLD octets (habituellement 2 Ko). Le code TOAST compressera et/ou déplacera les valeurs de champ hors la ligne jusqu'à ce que la valeur de la ligne soit plus petite que TOAST_TUPLE_TARGET octets (habituellement là-aussi 2 Ko) ou que plus aucun gain ne puisse être réalisé. Lors d'une opération UPDATE, les valeurs des champs non modifiées sont habituellement préservées telles quelles ; donc un UPDATE sur une ligne avec des valeurs hors ligne n'induit pas de coûts à cause de TOAST si aucune des valeurs hors-ligne n'est modifiée.

Le code TOAST connaît quatre stratégies différentes pour stocker les colonnes TOAST-ables :

- `PLAIN` empêche soit la compression soit le stockage hors-ligne ; de plus, il désactive l'utilisation d'en-tête sur un octet pour les types varlena. Ceci est la seule stratégie possible pour les colonnes des types de données non TOAST-ables.
- `EXTENDED` permet à la fois la compression et le stockage hors-ligne. Ceci est la valeur par défaut de la plupart des types de données TOAST-ables. La compression sera tentée en premier, ensuite le stockage hors-ligne si la ligne est toujours trop grande.
- `EXTERNAL` autorise le stockage hors-ligne mais pas la compression. L'utilisation d'`EXTERNAL` rendra plus rapides les opérations sur des sous-chaînes d'importantes colonnes de type text et bytea (au dépens d'un espace de stockage accrus) car ces opérations sont optimisées pour récupérer seulement les parties requises de la valeur hors-ligne lorsqu'elle n'est pas compressée.
- `MAIN` autorise la compression mais pas le stockage hors-ligne. (En réalité le stockage hors-ligne sera toujours réalisé pour de telles colonnes mais seulement en dernier ressort s'il n'existe aucune autre solution pour diminuer suffisamment la taille de la ligne pour qu'elle tienne sur une page.)

Chaque type de données TOAST-able spécifie une stratégie par défaut pour les colonnes de ce type de donnée, mais la stratégie pour une colonne d'une table donnée peut être modifiée avec **ALTER TABLE SET STORAGE**.

Cette combinaison a de nombreux avantages comparés à une approche plus directe comme autoriser le stockage des valeurs de lignes sur plusieurs pages. En supposant que les requêtes sont habituellement qualifiées par comparaison avec des valeurs de clé relativement petites, la grosse partie du travail de l'exécuteur sera réalisée en utilisant l'entrée principale de la ligne. Les grandes valeurs des attributs TOASTés seront seulement récupérées (si elles sont sélectionnées) au moment où l'ensemble de résultats est envoyé au client. Ainsi, la table principale est bien plus petite et un plus grand nombre de ses lignes tiennent dans le cache du tampon partagé, ce qui ne serait pas le cas sans aucun stockage hors-ligne. Le tri l'utilise aussi, et les tris seront plus souvent réalisés entièrement en mémoire. Un petit test a montré qu'une table contenant des pages HTML typiques ainsi que leurs URL étaient stockées en à peu près la moitié de la taille des données brutes en incluant la table TOAST et que la table principale contenait moins de 10 % de la totalité des données (les URL et quelques petites pages HTML). Il n'y avait pas de différence à l'exécution en comparaison avec une table non TOASTée, dans laquelle toutes les pages HTLM avaient été coupées à 7 Ko pour tenir.

54.3. Carte des espaces libres

Chaque table et index, en dehors des index hash, a une carte des espaces libres (appelée aussi FSM, acronyme de *Free Space Map*) pour conserver le trace des emplacements disponibles dans la relation. Elle est stockée dans un fichier séparé du fichier des données. Le nom de fichier est le numéro relfilenode suivi du suffixe `_fsm`. Par exemple, si le relfilenode d'une relation est 12345, la

FSM est stockée dans un fichier appelé `12345_fsm`, dans même répertoire que celui utilisé pour le fichier des données.

La carte des espaces libres est organisée comme un arbre de pages FSM. Les pages FSM de niveau bas stockent l'espace libre disponible dans chaque page de la relation. Les niveaux suppérieurs agrégent l'information des niveaux bas.

À l'intérieur de chaque page FSM se trouve un arbre binaire stocké dans un tableau avec un octet par nœud. Chaque nœud final représente une page de la relation, ou une page FSM de niveau bas. Dans chaque nœud non final, la valeur la plus haute des valeurs enfants est stockée. Du coup, la valeur maximum de tous les nœuds se trouve à la racine.

Voir `src/backend/storage/freespace/README` pour plus de détails sur la façon dont la FSM est structurée, et comment elle est mise à jour et recherchée. Le module contrib `contrib/pg_freespacemap` peut être utilisé pour examiner l'information stockée dans les cartes d'espace libre (voir Section F.26, « pg_freespacemap »).

54.4. Carte de visibilité

Chaque relation a une carte de visibilité (VM acronyme de *Visibility Map*) pour garder trace des pages contenant seulement des lignes connues pour être visibles par toutes les transactions actives. Elle est stockée en dehors du fichier de données dans un fichier séparé nommé suivant le numéro relfilenode de la relation, auquel est ajouté le suffixe `_vm`. Par exemple, si le relfilenode de la relation est 12345, la VM est stockée dans un fichier appelé `12345_vm`, dans le même répertoire que celui du fichier de données. Notez que les index n'ont pas de VM.

La carte de visibilité enregistre un bit par page. Un bit à 1 signifie que toutes les lignes de la page sont visibles par toutes les transactions. Cela signifie que le page ne contient pas de lignes nécessitant un VACUUM ; dans le futur, cela pourra aussi être utilisé pour éviter de visiter la page lors de vérifications de visibilité. Chaque fois qu'un bit est à 1, la condition est vraie à coup sûr. Par contre, dans le cas contraire, la condition peut être vraie comme fausse.

54.5. Emplacement des pages de la base de données

Cette section fournit un aperçu du format des pages utilisées par les tables et index de PostgreSQL™.[1] Les séquences et les tables TOAST tables sont formatées comme des tables standards.

Dans l'explication qui suit, un *octet* contient huit bits. De plus, le terme *élément* fait référence à une valeur de données individuelle qui est stockée dans une page. Dans une table, un élément est une ligne ; dans un index, un élément est une entrée d'index.

Chaque table et index est stocké comme un tableau de *pages* d'une taille fixe (habituellement 8 Ko, bien qu'une taille de page différente peut être sélectionnée lors de la compilation du serveur). Dans une table, toutes les pages sont logiquement équivalentes pour qu'un élément (ligne) particulier puisse être stocké dans n'importe quelle page. Dans les index, la première page est généralement réservée comme *métapage* contenant des informations de contrôle, et il peut exister différents types de pages à l'intérieur de l'index, suivant la méthode d'accès à l'index. Les tables ont aussi une carte de visibilité dans un fichier de suffixe `_vm`, pour tracer les pages dont on sait qu'elles ne contiennent pas de lignes mortes et qui n'ont pas du coup besoin de VACUUM.

Tableau 54.2, « Disposition d'une page » affiche le contenu complet d'une page. Il existe cinq parties pour chaque page.

Tableau 54.2. Disposition générale d'une page

Élément	Description
PageHeaderData	Longueur de 24 octets. Contient des informations générales sur la page y compris des pointeurs sur les espaces libres.
ItemIdData	Tableau de paires (décalage,longueur) pointant sur les éléments réels. Quatre octets par élément.
Free space	L'espace non alloué. Les pointeurs de nouveaux éléments sont alloués à partir du début de cette région, les nouveaux éléments à partir de la fin.
Items	Les éléments eux-mêmes.
Special space	Données spécifiques des méthodes d'accès aux index. Différentes méthodes stockent différentes données. Vide pour les tables ordinaires.

Les 24 premiers octets de chaque page consistent en un en-tête de page (PageHeaderData). Son format est détaillé dans Tableau 54.3, « Disposition de PageHeaderData ». Les deux premiers champs traquent l'entrée WAL la plus récente relative à cette page. Ensuite se trouve un champ de deux octets contenant des drapeaux. Ils sont suivis par trois champs d'entiers sur deux octets (*`pd_lower`*, *`pd_upper`* et *`pd_special`*). Ils contiennent des décalages d'octets à partir du début de la page jusqu'au début de l'espace non alloué, jusqu'à la fin de l'espace non alloué, et jusqu'au début de l'espace spécial. Les deux octets suivants de l'en-tête de page, *`pd_pagesize_version`*, stockent à la fois la taille de la page et un indicateur de versoin. À partir de la version 8.3 de

[1] En réalité, les méthodes d'accès par index n'ont pas besoin d'utiliser ce format de page. Toutes les méthodes d'indexage existantes utilisent ce format de base mais les données conservées dans les métapages des index ne suivent habituellement pas les règles d'emplacement des éléments.

PostgreSQL™, le numéro de version est 4 ; PostgreSQL™ 8.1 et 8.2 ont utilisé le numéro de version 3 ; PostgreSQL™ 8.0 a utilisé le numéro de version 2 ; PostgreSQL™ 7.3 et 7.4 ont utilisé le numéro de version 1 ; les versions précédentes utilisaient le numéro de version 0. (La disposition fondamentale de la page et le format de l'en-tête n'ont pas changé dans la plupart de ces versions mais la disposition de l'en-tête des lignes de tête a changé.) La taille de la page est seulement présente comme vérification croisée ; il n'existe pas de support pour avoir plus d'une taille de page dans une installation. Le dernier champ est une aide indiquant si traiter la page serait profitable : il garde l'information sur le plus vieux XMAX non traité de la page.

Tableau 54.3. Disposition de PageHeaderData

Champ	Type	Longueur	Description
pd_lsn	XLogRecPtr	8 octets	LSN : octet suivant le dernier octet de l'enregistrement xlog pour la dernière modification de cette page
pd_tli	uint16	2 octets	TimeLineID de la dernière modification (seulement les 16 bits de poids faible)
pd_flags	uint16	2 octets	Bits d'état
pd_lower	LocationIndex	2 octets	Décalage jusqu'au début de l'espace libre
pd_upper	LocationIndex	2 octets	Décalage jusqu'à la fin de l'espace libre
pd_special	LocationIndex	2 octets	Décalage jusqu'au début de l'espace spécial
pd_pagesize_version	uint16	2 octets	Taille de la page et disposition de l'information du numéro de version
pd_prune_xid	TransactionId	4 bytes	Plus vieux XMAX non traité sur la page, ou zéro si aucun

Tous les détails se trouvent dans `src/include/storage/bufpage.h`.

Après l'en-tête de la page se trouvent les identificateurs d'élément (ItemIdData), chacun nécessitant quatre octets. Un identificateur d'élément contient un décalage d'octet vers le début d'un élément, sa longueur en octets, et quelques bits d'attributs qui affectent son interprétation. Les nouveaux identificateurs d'éléments sont alloués si nécessaire à partir du début de l'espace non alloué. Le nombre d'identificateurs d'éléments présents peut être déterminé en regardant *`pd_lower`*, qui est augmenté pour allouer un nouvel identificateur. Comme un identificateur d'élément n'est jamais déplacé tant qu'il n'est pas libéré, son index pourrait être utilisé sur une base à long terme pour référencer un élément, même quand l'élément lui-même est déplacé le long de la page pour compresser l'espace libre. En fait, chaque pointeur vers un élément (ItemPointer, aussi connu sous le nom de CTID), créé par PostgreSQL™ consiste en un numéro de page et l'index de l'identificateur d'élément.

Les éléments eux-mêmes sont stockés dans l'espace alloué en marche arrière, à partir de la fin de l'espace non alloué. La structure exacte varie suivant le contenu de la table. Les tables et les séquences utilisent toutes les deux une structure nommée HeapTupleHeaderData, décrite ci-dessous.

La section finale est la « section spéciale » qui pourrait contenir tout ce que les méthodes d'accès souhaitent stocker. Par exemple, les index b-tree stockent des liens vers les enfants gauche et droit de la page ainsi que quelques autres données sur la structure de l'index. Les tables ordinaires n'utilisent pas du tout de section spéciale (indiquée en configurant *`pd_special`* à la taille de la page).

Toutes les lignes de la table sont structurées de la même façon. Il existe un en-tête à taille fixe (occupant 23 octets sur la plupart des machines), suivi par un bitmap NULL optionnel, un champ ID de l'objet optionnel et les données de l'utilisateur. L'en-tête est détaillé dans Tableau 54.4, « Disposition de HeapTupleHeaderData ». Les données réelles de l'utilisateur (les colonnes de la ligne) commencent àu décalage indiqué par *`t_hoff`*, qui doit toujours être un multiple de la distance MAXALIGN pour la plateforme. Le bitmap NULL est seulement présent si le bit *HEAP_HASNULL* est initialisé dans *`t_infomask`*. S'il est présent, il commence juste après l'en-tête fixe et occupe suffisamment d'octets pour avoir un bit par colonne de données (c'est-à-dire *`t_natts`* bits ensemble). Dans cette liste de bits, un bit 1 indique une valeur non NULL, un bit 0 une valeur NULL. Quand le bitmap n'est pas présent, toutes les colonnes sont supposées non NULL. L'ID de l'objet est seulement présent si le bit *HEAP_HASOID* est initialisé dans *`t_infomask`*. S'il est présent, il apparaît juste avant la limite *`t_hoff`*. Tout ajout nécessaire pour faire de *`t_hoff`* un multiple de MAXALIGN apparaîtra entre le bitmap NULL et l'ID de l'objet. (Ceci nous assure en retour que l'ID de l'objet est convenablement aligné.)

Tableau 54.4. Disposition de HeapTupleHeaderData

Champ	Type	Longueur	Description
t_xmin	TransactionId	4 octets	XID d'insertion
t_xmax	TransactionId	4 octets	XID de suppression
t_cid	CommandId	4 octets	CID d'insertion et de suppression (surcharge avec t_xvac)

Champ	Type	Longueur	Description
t_xvac	TransactionId	4 octets	XID pour l'opération VACUUM déplaçant une version de ligne
t_ctid	ItemPointer-Data	6 octets	TID en cours pour cette version de ligne ou pour une version plus récente
t_infomask2	int16	2 octets	nombre d'attributs et quelques bits d'état
t_infomask	uint16	2 octets	différents bits d'options (flag bits)
t_hoff	uint8	1 octet	décalage vers les données utilisateur

Tous les détails sont disponibles dans `src/include/access/htup.h`.

Interpréter les données réelles peut seulement se faire avec des informations obtenues à partir d'autres tables, principalement pg_attribute. Les valeurs clés nécessaires pour identifier les emplacements des champs sont `attlen` et `attalign`. Il n'existe aucun moyen pour obtenir directement un attribut particulier, sauf quand il n'y a que des champs de largeur fixe et aucune colonne NULL. Tout ceci est emballé dans les fonctions *heap_getattr*, *fastgetattr* et *heap_getsysattr*.

Pour lire les données, vous avez besoin d'examinez chaque attribut à son tour. Commencez par vérifier si le champ est NULL en fonction du bitmap NULL. S'il l'est, allez au suivant. Puis, assurez-vous que vous avez le bon alignement. Si le champ est un champ à taille fixe, alors tous les octets sont placés simplement. S'il s'agit d'un champ à taille variable (attlen = -1), alors c'est un peu plus compliqué. Tous les types de données à longueur variable partagent la même structure commune d'en-tête, struct varlena, qui inclut la longueur totale de la valeur stockée et quelques bits d'option. Suivant les options, les données pourraient être soit dans la table de base soit dans une table TOAST ; elles pourraient aussi être compressées (voir Section 54.2, « TOAST »).

Chapitre 55. Interface du moteur, BKI

Les fichiers d'interface du moteur (BKI pour *Backend Interface*) sont des scripts écrits dans un langage spécial, compris par le serveur PostgreSQL™ lorsqu'il est exécuté en mode « bootstrap ». Ce mode autorise la création et le remplissage des catalogues systèmes *ab initio*, là où les commandes SQL exigent leur existence préalable. Les fichiers BKI peuvent donc être utilisés en premier lieu pour créer le système de base de données. (Ils n'ont probablement pas d'autre utilité.)

initdb utilise un fichier BKI pour réaliser une partie de son travail lors de la création d'un nouveau cluster de bases de données. Le fichier d'entrée utilisé par initdb est créé, lors de la construction et de l'installation de PostgreSQL™, par un programme nommé `genbki.pl` qui lit différents fichiers d'en-têtes C spécialement formatés à partir du répertoire `src/include/catalog` des sources. Le fichier BKI créé est appelé `postgres.bki` et est normalement installé dans le sous-répertoire `share` du répertoire d'installation.

D'autres informations sont disponibles dans la documentation d'initdb.

55.1. Format des fichiers BKI

Cette section décrit l'interprétation des fichiers BKI par le moteur de PostgreSQL™. Cette description est plus facile à comprendre si le fichier `postgres.bki` est utilisé comme exemple.

L'entrée de BKI représente une séquence de commandes. Les commandes sont constituées de lexèmes (*tokens*) dont le nombre dépend de la syntaxe de la commande. Les lexèmes sont habituellement séparés par des espaces fines, mais en l'absence d'ambiguïté ce n'est pas nécessaire. Il n'y a pas de séparateur spécial pour les commandes ; le prochain lexème qui ne peut syntaxiquement pas appartenir à la commande qui précède en lance une autre. (En général, il est préférable, pour des raisons de clarté, de placer toute nouvelle commande sur une nouvelle ligne.) Les lexèmes peuvent être des mots clés, des caractères spéciaux (parenthèses, virgules, etc.), nombres ou chaînes de caractères entre guillemets doubles. Tous sont sensibles à la casse.

Les lignes qui débutent par # sont ignorées.

55.2. Commandes BKI

`create` *`tablename tableoid`* [`bootstrap`] [`shared_relation`] [`without_oids`] [`rowtype_oid` *`oid`*] (*`name1`* = *`type1`* [, *`name2`* = *`type2`*, ...])

Crée une table nommée *`nomtable`*, possédant l'OID *`tableoid`* et composée des colonnes données entre parenthèses.

Les types de colonnes suivants sont supportés directement par `bootstrap.c`: bool, bytea, char (1 byte), name, int2, int4, regproc, regclass, regtype, text, oid, tid, xid, cid, int2vector, oidvector, _int4 (array), _text (array), _oid (array), _char (array), _aclitem (array). Bien qu'il soit possible de créer des tables contenant des colonnes d'autres types, cela ne peut pas être réalisé avant que pg_type ne soit créé et rempli avec les entrées appropriées. (Ce qui signifie en fait que seuls ces types de colonnes peuvent être utilisés dans les tables utilisant le « bootstrap » mais que les catalogues ne l'utilisant pas peuvent contenir tout type interne.)

Quand `bootstrap` est précisé, la table est uniquement construite sur disque ; rien n'est entré dans pg_class, pg_attribute, etc, pour cette table. Du coup, la table n'est pas accessible par les opérations SQL standard tant que ces entrées ne sont pas réalisées en dur (à l'aide de commandes `insert`). Cette option est utilisée pour créer pg_class, etc.

La table est créée partagée si `shared_relation` est indiqué. Elle possède des OID à moins que `without_oids` ne soit précisé. L'OID du type de ligne de la table (OID de pg_type) peut en option être indiquée via la clause `rowtype_oid` ; dans le cas contraire, un OID est automatiquement généré pour lui. (La clause `rowtype_oid` est inutile si `bootstrap` est spécifié, mais il peut néanmoins être fourni pour documentation.)

`open` *`nomtable`*

Ouvre la table nommée *`nomtable`* pour l'ajout de données. Toute table alors ouverte est fermée.

`close` [*`nomtable`*]

Ferme la table ouverte. Le nom de la table peut-être indiqué pour vérification mais ce n'est pas nécessaire.

`insert` [`OID =` *`valeur_oid`*] (*`valeur1 valeur2`* ...)

Insère une nouvelle ligne dans la table ouverte en utilisant *`valeur1`*, *`valeur2`*, etc., comme valeurs de colonnes et *`valeur_oid`* comme OID. Si *`valeur_oid`* vaut zéro (0) ou si la clause est omise, et que la table a des OID, alors le prochain OID disponible est utilisé.

La valeur NULL peut être indiquée en utilisant le mot clé spécial `_null_`. Les valeurs contenant des espaces doivent être placées entre guillemets doubles.

`declare` [`unique`] `index` *`nomindex oidindex`* `on` *`nomtable`* `using` *`nomam`* (*`classeop1 nom1`* [, ...])

Crée un index nommé *nomindex*, d'OID *indexoid*, sur la table nommée *nomtable* en utilisant la méthode d'accès nommée *nomam*. Les champs à indexer sont appelés *nom1*, *nom2* etc., et les classes d'opérateur à utiliser sont respectivement *classeop1*, *classeop2* etc. Le fichier index est créé et les entrées appropriées du catalogue sont ajoutées pour lui, mais le contenu de l'index n'est pas initialisé par cette commande.

`declare toast` *toasttableoid toastindexoid* `on` *nomtable*
: Crée une table TOAST pour la table nommée *nomtable*. La table TOAST se voit affecter l'OID *toasttableoid* et son index l'OID *toastindexoid*. Comme avec `declare index`, le remplissage de l'index est reporté.

`build indices`
: Remplit les index précédemment déclarés.

55.3. Structure du fichier BKI de « bootstrap »

La commande `open` ne peut pas être utilisée avant que les tables qu'elle utilise n'existent et n'aient des entrées pour la table à ouvrir. (Ces tables minimales sont pg_class, pg_attribute, pg_proc et pg_type.) Pour permettre le remplissage de ces tables elles-mêmes, `create` utilisé avec l'option `bootstrap` ouvre implicitement la table créée pour l'insertion de données.

De la même façon, les commandes `declare index` et `declare toast` ne peuvent pas être utilisées tant que les catalogues systèmes dont elles ont besoin n'ont pas été créés et remplis.

Du coup, la structure du fichier `postgres.bki` doit être :

1. `create bootstrap` une des tables critiques
2. `insert` les données décrivant au moins les tables critiques
3. `close`
4. À répéter pour les autres tables critiques.
5. `create` (sans `bootstrap`) une table non critique
6. `open`
7. `insert` les données souhaitées
8. `close`
9. À répéter pour les autres tables non critiques.
10. Définir les index et les tables TOAST.
11. `build indices`

.

Il existe, sans doute, d'autres dépendances d'ordre non documentées.

55.4. Exemple

La séquence de commandes suivante crée la table `test_table` avec l'OID 420, deux colonnes `cola` et `colb` de types respectifs int4 et text et insère deux lignes dans la table :

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

Chapitre 56. Comment le planificateur utilise les statistiques

Ce chapitre est construit sur les informations fournies dans Section 14.1, « Utiliser **EXPLAIN** » et Section 14.2, « Statistiques utilisées par le planificateur » pour montrer certains détails supplémentaires sur la façon dont le planificateur utilise les statistiques système pour estimer le nombre de lignes que chaque partie d'une requête pourrait renvoyer. C'est une partie importante du processus de planification, fournissant une bonne partie des informations pour le calcul des coûts.

Le but de ce chapitre n'est pas de documenter le code en détail mais plutôt de présenter un aperçu du fonctionnement. Ceci aidera peut-être la phase d'apprentissage pour quelqu'un souhaitant lire le code.

56.1. Exemples d'estimation des lignes

Les exemples montrés ci-dessous utilisent les tables de la base de tests de régression de PostgreSQL™. Les affichages indiqués sont pris depuis la version 8.3. Le comportement des versions précédentes (ou ultérieures) pourraient varier. Notez aussi que, comme **ANALYZE** utilise un échantillonage statistique lors de la réalisation des statistiques, les résultats peuvent changer légèrement après toute exécution d'**ANALYZE**.

Commençons avec une requête simple :

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Comment le planificateur détermine la cardinalité de tenk1 est couvert dans Section 14.2, « Statistiques utilisées par le planificateur » mais est répété ici pour être complet. Le nombre de pages et de lignes est trouvé dans pg_class :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';


 relpages | reltuples
----------+-----------
      358 |     10000
```

Ces nombres sont corrects à partir du dernier **VACUUM** ou **ANALYZE** sur la table. Le planificateur récupère ensuite le nombre de pages actuel dans la table (c'est une opération peu coûteuse, ne nécessitant pas un parcours de table). Si c'est différent de *`relpages`*, alors *`reltuples`* est modifié en accord pour arriver à une estimation actuelle du nombre de lignes. Dans ce cas, les valeurs sont correctes donc l'estimation du nombre de lignes est identique à *`reltuples`*.

Passons à un exemple avec une condition dans sa clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Le planificateur examine la condition de la clause `WHERE` et cherche la fonction de sélectivité à partir de l'opérateur `<` dans pg_operator. C'est contenu dans la colonne *`oprrest`* et le résultat, dans ce cas, est `scalarltsel`. La fonction `scalarltsel` récupère l'histogramme pour *`unique1`* à partir de `pg_statistics`. Pour les requêtes manuelles, il est plus simple de regarder dans la vue pg_stats :

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                   histogram_bounds
-----------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Ensuite, la fraction de l'histogramme occupée par « < 1000 » est traitée. C'est la sélectivité. L'histogramme divise l'ensemble en plus petites parties d'égales fréquences, donc tout ce que nous devons faire est de localiser la partie où se trouve notre valeur et compter une *partie* d'elle et *toutes* celles qui la précèdent. La valeur 1000 est clairement dans la seconde partie (993-1997), donc en supposant une distribution linéaire des valeurs à l'intérieur de chaque partie, nous pouvons calculer la sélectivité comme

étant :

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

c'est-à-dire une partie complète plus une fraction linéaire de la seconde, divisée par le nombre de parties. Le nombre de lignes estimées peut maintenant être calculé comme le produit de la sélectivité et de la cardinalité de tenk1 :

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

Maintenant, considérons un exemple avec une condition d'égalité dans sa clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

De nouveau, le planificateur examine la condition de la clause `WHERE` et cherche la fonction de sélectivité pour `=`, qui est `eqsel`. Pour une estimation d'égalité, l'histogramme n'est pas utile ; à la place, la liste des valeurs les plus communes (*most common values*, d'où l'acronyme MCV fréquemment utilisé) est utilisé pour déterminer la sélectivité. Regardons-les avec quelques colonnes supplémentaires qui nous serons utiles plus tard :

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  |
{EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Comme `CRAAAA` apparaît dans la liste des MCV, la sélectivité est tout simplement l'entrée correspondante dans la liste des fréquences les plus courantes (MCF, acronyme de *Most Common Frequencies*) :

```
selectivity = mcf[3]
            = 0.003
```

Comme auparavant, le nombre estimé de lignes est seulement le produit de ceci avec la cardinalité de tenk1 comme précédemment :

```
rows = 10000 * 0.003
     = 30
```

Maintenant, considérez la même requête mais avec une constante qui n'est pas dans la liste MCV :

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

C'est un problème assez différent, comment estimer la sélectivité quand la valeur n'est *pas* dans la liste MCV. L'approche est d'utiliser le fait que la valeur n'est pas dans la liste, combinée avec la connaissance des fréquences pour tout les MCV :

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

C'est-à-dire ajouter toutes les fréquences pour les MCV et les soustraire d'un, puis les diviser par le nombre des *autres* valeurs distinctes. Notez qu'il n'y a pas de valeurs NULL, donc vous n'avez pas à vous en inquiéter (sinon nous pourrions soustraire la fraction NULL à partir du numérateur). Le nombre estimé de lignes est ensuite calculé comme d'habitude :

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

L'exemple précédent avec `unique1 < 1000` était une sur-simplification de ce que `scalarltsel` faisait réellement ; maintenant que nous avons vu un exemple de l'utilisation des MCV, nous pouvons ajouter quelques détails supplémentaires. L'exemple était correct aussi loin qu'il a été car, comme *`unique1`* est une colonne unique, elle n'a pas de MCV (évidemment, n'avoir aucune valeur n'est pas plus courant que toute autre valeur). Pour une colonne non unique, il y a normalement un histogramme et une liste MCV, et *l'histogramme n'inclut pas la portion de la population de colonne représentée par les MCV*. Nous le faisons ainsi parce que cela permet une estimation plus précise. Dans cette situation, `scalarltsel` s'applique directement à la condition (c'est-à-dire « < 1000 ») pour chaque valeur de la liste MCV, et ajoute les fréquence des MCV pour lesquelles la condition est vérifiée. Ceci donne une estimation exacte de la sélectivité dans la portion de la table qui est MCV. L'histogramme est ensuite utilisée de la même façon que ci-dessus pour estimer la sélectivité dans la portion de la table qui n'est pas MCV, et ensuite les deux nombres sont combinées pour estimer la sélectivité. Par exemple, considérez

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

Nous voyons déjà l'information MCV pour *`stringu1`*, et voici son histogramme :

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
--------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

En vérifiant la liste MCV, nous trouvons que la condition `stringu1 < 'IAAAAA'` est satisfaite par les six premières entrées et non pas les quatre dernières, donc la sélectivité dans la partie MCV de la population est :

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

Additionner toutes les MFC nous indique aussi que la fraction totale de la population représentée par les MCV est de 0.03033333, et du coup la fraction représentée par l'histogramme est de 0.96966667 (encore une fois, il n'y a pas de NULL, sinon nous devrions les exclure ici). Nous pouvons voir que la valeur `IAAAAA` tombe près de la fin du troisième jeton d'histogramme. En utilisant un peu de suggestions sur la fréquence des caractères différents, le planificateur arrive à l'estimation 0.298387 pour la portion de la population de l'histogramme qui est moindre que `IAAAAA`. Ensuite nous combinons les estimations pour les populations MCV et non MCV :

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

Dans cet exemple particulier, la correction à partir de la liste MCV est très petit car la distribution de la colonne est réellement assez plat (les statistiques affichant ces valeurs particulières comme étant plus communes que les autres sont principalement dûes à une erreur d'échantillonage). Dans un cas plus typique où certaines valeurs sont significativement plus communes que les autres, ce processus compliqué donne une amélioration utile dans la précision car la sélectivité pour les valeurs les plus communes est trouvée exactement.

Maintenant, considérons un cas avec plus d'une condition dans la clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
```

```
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Le planificateur suppose que les deux conditions sont indépendantes, pour que les sélectivités individuelles des clauses puissent être multipliées ensemble :

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

Notez que le nombre de lignes estimé être renvoyées à partir bitmap index scan reflète seulement la condition utilisée avec l'index ; c'est important car cela affecte l'estimation du coût pour les récupérations suivantes sur la table.

Enfin, nous examinerons une requête qui implique une jointure :

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                       QUERY PLAN
-----------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
         Index Cond: (t2.unique2 = t1.unique2)
```

La restriction sur tenk1, `unique1 < 50`, est évaluée avant la jointure de boucle imbriquée. Ceci est géré de façon analogue à l'exemple précédent. Cette fois, la valeur 50 est dans la première partie de l'histogramme *`unique1`* :

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

La restriction pour la jointure est `t2.unique2 = t1.unique2`. L'opérateur est tout simplement le `=`, néanmoins la fonction de sélectivité est obtenue à partir de la colonne *`oprjoin`* de pg_operator, et est `eqjoinsel`. `eqjoinsel` recherche l'information statistique de tenk2 et tenk1 :

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

Dans ce cas, il n'y a pas d'information MCV pour *`unique2`* parce que toutes les valeurs semblent être unique, donc nous utilisons un algorithme qui relie seulement le nombre de valeurs distinctes pour les deux relations ensembles avec leur fractions NULL :

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1,
1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

C'est-à-dire, soustraire la fraction NULL pour chacune des relations, et divisez par le maximum of the numbers of distinct values. Le nombre de lignes que la jointure pourrait émettre est calculé comme la cardinalité du produit cartésien de deux inputs, multiplié par la sélectivité :

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
```

```
      = 50
```

S'il y avait eu des listes MCV pour les deux colonnes, `eqjoinsel` aurait utilisé une comparaison directe des listes MCV pour déterminer la sélectivité de jointure à l'intérieur de la aprtie des populations de colonne représentées par les MCV. L'estimation pour le reste des populations suit la même approche affichée ici.

Notez que nous montrons `inner_cardinality` comme 10000, c'est-à-dire la taille non modifiée de tenk2. Il pourrait apparaître en inspectant l'affichage **EXPLAIN** que l'estimation des lignes jointes vient de 50 * 1, c'est-à-dire que le nombre de lignes externes multiplié par le nombre estimé de lignes obtenu par chaque parcours d'index interne sur tenk2. Mais ce n'est pas le cas : la taille de la relation jointe est estimée avant tout plan de jointure particulier considéré. Si tout fonctionne si bien, alors les deux façons d'estimer la taille de la jointure produiront la même réponse mais, à cause de l'erreur d'arrondi et d'autres facteurs, ils divergent quelque fois significativement.

Pour les personnes intéressées par plus de détails, l'estimation de la taille d'une table (avant toute clause `WHERE`) se fait dans `src/backend/optimizer/util/plancat.c`. La logique générique pour les sélectivités de clause est dans `src/backend/optimizer/path/clausesel.c`. Les fonctions de sélectivité spécifiques aux opérateurs se trouvent principalement dans `src/backend/utils/adt/selfuncs.c`.

Partie VIII. Annexes

Annexe A. Codes d'erreurs de PostgreSQL™

Tous les messages émis par le serveur PostgreSQL™ se voient affectés des codes d'erreur sur cinq caractères. Ces codes suivent les conventions du standard SQL pour les codes « SQLSTATE ».

Les applications qui souhaitent connaître la condition d'erreur survenue peuvent tester le code d'erreur plutôt que récupérer le message d'erreur textuel. Les codes d'erreurs sont moins sujets à changement au fil des versions de PostgreSQL™ et ne dépendent pas de la localisation des messages d'erreur. Seuls certains codes d'erreur produits par PostgreSQL™ sont définis par le standard SQL ; divers codes d'erreur supplémentaires, pour des conditions non définies par le standard, ont été inventés ou empruntés à d'autres bases de données.

Comme le préconise le standard, les deux premiers caractères d'un code d'erreur définissent la classe d'erreurs, les trois derniers indiquent la condition spécifique à l'intérieur de cette classe. Ainsi, une application qui ne reconnaît pas le code d'erreur spécifique peut toujours agir en fonction de la classe de l'erreur.

Tableau A.1, « Codes d'erreur de PostgreSQL™ » liste tous les codes d'erreurs définis dans PostgreSQL™ 9.0.18. (Certains ne sont pas réellement utilisés mais sont définis par le standard SQL.) Les classes d'erreurs sont aussi affichées. Pour chaque classe d'erreur, il y a un code d'erreur « standard » dont les trois derniers caractères sont `000`. Ce code n'est utilisé que pour les conditions d'erreurs de cette classe qui ne possèdent pas de code plus spécifique.

Le nom de la constante PL/pgSQL de chaque code erreur est le même que la phrase affichée dans le tableau, les tirets bas étant substitués aux espaces. Par exemple, le code `22012`, DIVISION BY ZERO, a comme nom de constante `DIVISION_BY_ZERO`. Les noms de constante peuvent être écrits en majuscule ou en minuscule. (PL/pgSQL ne fait pas la distinction entre avertissement et erreur au niveau des noms des conditions ; il s'agit des classes 00, 01 et 02.)

Tableau A.1. Codes d'erreur de PostgreSQL™

Code erreur	Signification	Nom de condition
Class 00 -- Succès de l'opération		
`00000`	SUCCESSFUL COMPLETION	successful_completion
Class 01 -- Avertissement		
`01000`	WARNING	warning
`0100C`	DYNAMIC RESULT SETS RETURNED	dynamic_result_sets_returned
`01008`	IMPLICIT ZERO BIT PADDING	implicit_zero_bit_padding
`01003`	NULL VALUE ELIMINATED IN SET FUNCTION	null_value_eliminated_in_set_function
`01007`	PRIVILEGE NOT GRANTED	privilege_not_granted
`01006`	PRIVILEGE NOT REVOKED	privilege_not_revoked
`01004`	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
`01P01`	DEPRECATED FEATURE	deprecated_feature
Class 02 -- Pas de données (également une classe d'averissement selon le standard SQL)		
`02000`	NO DATA	no_data
`02001`	NO ADDITIONAL DYNAMIC RESULT SETS RETURNED	no_additional_dynamic_result_sets_returned
Class 03 -- Instruction SQL pas encore terminée		
`03000`	SQL STATEMENT NOT YET COMPLETE	sql_statement_not_yet_complete
Class 08 -- Problème de connexion		
`08000`	CONNECTION EXCEPTION	connection_exception
`08003`	CONNECTION DOES NOT EXIST	connection_does_not_exist
`08006`	CONNECTION FAILURE	connection_failure
`08001`	SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION	sqlclient_unable_to_establish_sqlconnection
`08004`	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION	sqlserver_rejected_establishment_of_sqlconnection

Code erreur	Signification	Nom de condition
08007	TRANSACTION RESOLUTION UNKNOWN	transaction_resolution_unknown
08P01	PROTOCOL VIOLATION	protocol_violation
Class 09 -- Problème d'action déclenchée		
09000	TRIGGERED ACTION EXCEPTION	triggered_action_exception
Class 0A -- Fonctionnalité non supportée		
0A000	FEATURE NOT SUPPORTED	feature_not_supported
Class 0B -- Initialisation de transaction invalide		
0B000	INVALID TRANSACTION INITIATION	invalid_transaction_initiation
Classe 0F -- Problème de pointeur (*Locator*)		
0F000	LOCATOR EXCEPTION	locator_exception
0F001	INVALID LOCATOR SPECIFICATION	invalid_locator_specification
Classe 0L -- Granteur invalide		
0L000	INVALID GRANTOR	invalid_grantor
0LP01	INVALID GRANT OPERATION	invalid_grant_operation
Classe 0P -- Spécification de rôle invalide		
0P000	INVALID ROLE SPECIFICATION	invalid_role_specification
Class 20 -- Case Not Found		
20000	CASE NOT FOUND	case_not_found
Class 21 -- Violation de cardinalité		
21000	CARDINALITY VIOLATION	cardinality_violation
Class 22 -- Problème de données		
22000	DATA EXCEPTION	data_exception
2202E	ARRAY SUBSCRIPT ERROR	array_subscript_error
22021	CHARACTER NOT IN REPERTOIRE	character_not_in_repertoire
22008	DATETIME FIELD OVERFLOW	datetime_field_overflow
22012	DIVISION BY ZERO	division_by_zero
22005	ERROR IN ASSIGNMENT	error_in_assignment
2200B	ESCAPE CHARACTER CONFLICT	escape_character_conflict
22022	INDICATOR OVERFLOW	indicator_overflow
22015	INTERVAL FIELD OVERFLOW	interval_field_overflow
2201E	INVALID ARGUMENT FOR LOGARITHM	invalid_argument_for_logarithm
22014	INVALID ARGUMENT FOR NTILE FUNCTION	invalid_argument_for_ntile_function
22016	INVALID ARGUMENT FOR NTH_VALUE FUNCTION	invalid_argument_for_nth_value_function
2201F	INVALID ARGUMENT FOR POWER FUNCTION	invalid_argument_for_power_function
2201G	INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION	invalid_argument_for_width_bucket_function
22018	INVALID CHARACTER VALUE FOR CAST	invalid_character_value_for_cast
22007	INVALID DATETIME FORMAT	invalid_datetime_format
22019	INVALID ESCAPE CHARACTER	invalid_escape_character
2200D	INVALID ESCAPE OCTET	invalid_escape_octet
22025	INVALID ESCAPE SEQUENCE	invalid_escape_sequence
22P06	NONSTANDARD USE OF ESCAPE CHARACTER	nonstandard_use_of_escape_character

Code erreur	Signification	Nom de condition
22010	INVALID INDICATOR PARAMETER VALUE	invalid_indicator_parameter_value
22023	INVALID PARAMETER VALUE	invalid_parameter_value
2201B	INVALID REGULAR EXPRESSION	invalid_regular_expression
2201W	INVALID ROW COUNT IN LIMIT CLAUSE	invalid_row_count_in_limit_clause
2201X	INVALID ROW COUNT IN RESULT OFFSET CLAUSE	invalid_row_count_in_result_offset_clause
22009	INVALID TIME ZONE DISPLACEMENT VALUE	invalid_time_zone_displacement_value
2200C	INVALID USE OF ESCAPE CHARACTER	invalid_use_of_escape_character
2200G	MOST SPECIFIC TYPE MISMATCH	most_specific_type_mismatch
22004	NULL VALUE NOT ALLOWED	null_value_not_allowed
22002	NULL VALUE NO INDICATOR PARAMETER	null_value_no_indicator_parameter
22003	NUMERIC VALUE OUT OF RANGE	numeric_value_out_of_range
22026	STRING DATA LENGTH MISMATCH	string_data_length_mismatch
22001	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
22011	SUBSTRING ERROR	substring_error
22027	TRIM ERROR	trim_error
22024	UNTERMINATED C STRING	unterminated_c_string
2200F	ZERO LENGTH CHARACTER STRING	zero_length_character_string
22P01	FLOATING POINT EXCEPTION	floating_point_exception
22P02	INVALID TEXT REPRESENTATION	invalid_text_representation
22P03	INVALID BINARY REPRESENTATION	invalid_binary_representation
22P04	BAD COPY FILE FORMAT	bad_copy_file_format
22P05	UNTRANSLATABLE CHARACTER	untranslatable_character
2200L	NOT AN XML DOCUMENT	not_an_xml_document
2200M	INVALID XML DOCUMENT	invalid_xml_document
2200N	INVALID XML CONTENT	invalid_xml_content
2200S	INVALID XML COMMENT	invalid_xml_comment
2200T	INVALID XML PROCESSING INSTRUCTION	invalid_xml_processing_instruction
Class 23 -- Violation de contrainte d'intégrité		
23000	INTEGRITY CONSTRAINT VIOLATION	integrity_constraint_violation
23001	RESTRICT VIOLATION	restrict_violation
23502	NOT NULL VIOLATION	not_null_violation
23503	FOREIGN KEY VIOLATION	foreign_key_violation
23505	UNIQUE VIOLATION	unique_violation
23514	CHECK VIOLATION	check_violation
23P01	EXCLUSION VIOLATION	exclusion_violation
Classe 24 -- État de curseur invalide		
24000	INVALID CURSOR STATE	invalid_cursor_state
Class 25 -- État de transaction invalide		
25000	INVALID TRANSACTION STATE	invalid_transaction_state
25001	ACTIVE SQL TRANSACTION	active_sql_transaction
25002	BRANCH TRANSACTION ALREADY ACTIVE	branch_transaction_already_active
25008	HELD CURSOR REQUIRES SAME ISOLATION LEVEL	held_cursor_requires_same_isolation_level

Code erreur	Signification	Nom de condition
25003	INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION	inappropriate_access_mode_for_branch_transaction
25004	INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION	inappropriate_isolation_level_for_branch_transaction
25005	NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION	no_active_sql_transaction_for_branch_transaction
25006	READ ONLY SQL TRANSACTION	read_only_sql_transaction
25007	SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED	schema_and_data_statement_mixing_not_supported
25P01	NO ACTIVE SQL TRANSACTION	no_active_sql_transaction
25P02	IN FAILED SQL TRANSACTION	in_failed_sql_transaction
Classe 26 -- Nom d'instruction SQL invalide		
26000	INVALID SQL STATEMENT NAME	invalid_sql_statement_name
Classe 27 -- Violation de modification de donnée déclenchée		
27000	TRIGGERED DATA CHANGE VIOLATION	triggered_data_change_violation
Classe 28 -- Spécification d'autorisation invalide		
28000	INVALID AUTHORIZATION SPECIFICATION	invalid_authorization_specification
28P01	INVALID PASSWORD	invalid_password
Classe 2B -- Descripteurs de privilège dépendant toujours existant		
2B000	DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST	dependent_privilege_descriptors_still_exist
2BP01	DEPENDENT OBJECTS STILL EXIST	dependent_objects_still_exist
Classe 2D -- Fin de transaction invalide		
2D000	INVALID TRANSACTION TERMINATION	invalid_transaction_termination
Classe 2F -- Exception dans une routine SQL		
2F000	SQL ROUTINE EXCEPTION	sql_routine_exception
2F005	FUNCTION EXECUTED NO RETURN STATEMENT	function_executed_no_return_statement
2F002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
2F003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
2F004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Classe 34 -- Nom de curseur invalide		
34000	INVALID CURSOR NAME	invalid_cursor_name
Classe 38 -- Exception de routine externe		
38000	EXTERNAL ROUTINE EXCEPTION	external_routine_exception
38001	CONTAINING SQL NOT PERMITTED	containing_sql_not_permitted
38002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
38003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
38004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Classe 39 -- Exception dans l'appel d'une routine externe		
39000	EXTERNAL ROUTINE INVOCATION EXCEPTION	external_routine_invocation_exception
39001	INVALID SQLSTATE RETURNED	invalid_sqlstate_returned
39004	NULL VALUE NOT ALLOWED	null_value_not_allowed
39P01	TRIGGER PROTOCOL VIOLATED	trigger_protocol_violated
39P02	SRF PROTOCOL VIOLATED	srf_protocol_violated

Code erreur	Signification	Nom de condition
Classe 3B -- Exception dans un point de retournement		
3B000	SAVEPOINT EXCEPTION	savepoint_exception
3B001	INVALID SAVEPOINT SPECIFICATION	invalid_savepoint_specification
Classe 3D -- Nom de catalogue invalide		
3D000	INVALID CATALOG NAME	invalid_catalog_name
Classe 3F -- Nom de schéma invalide		
3F000	INVALID SCHEMA NAME	invalid_schema_name
Classe 40 -- Annulation de transaction		
40000	TRANSACTION ROLLBACK	transaction_rollback
40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION	transaction_integrity_constraint_violation
40001	SERIALIZATION FAILURE	serialization_failure
40003	STATEMENT COMPLETION UNKNOWN	statement_completion_unknown
40P01	DEADLOCK DETECTED	deadlock_detected
Classe 42 -- Erreur de syntaxe ou violation de rêgle d'accès		
42000	SYNTAX ERROR OR ACCESS RULE VIOLATION	syntax_error_or_access_rule_violation
42601	SYNTAX ERROR	syntax_error
42501	INSUFFICIENT PRIVILEGE	insufficient_privilege
42846	CANNOT COERCE	cannot_coerce
42803	GROUPING ERROR	grouping_error
42P20	WINDOWING ERROR	windowing_error
42P19	INVALID RECURSION	invalid_recursion
42830	INVALID FOREIGN KEY	invalid_foreign_key
42602	INVALID NAME	invalid_name
42622	NAME TOO LONG	name_too_long
42939	RESERVED NAME	reserved_name
42804	DATATYPE MISMATCH	datatype_mismatch
42P18	INDETERMINATE DATATYPE	indeterminate_datatype
42809	WRONG OBJECT TYPE	wrong_object_type
42703	UNDEFINED COLUMN	undefined_column
42883	UNDEFINED FUNCTION	undefined_function
42P01	UNDEFINED TABLE	undefined_table
42P02	UNDEFINED PARAMETER	undefined_parameter
42704	UNDEFINED OBJECT	undefined_object
42701	DUPLICATE COLUMN	duplicate_column
42P03	DUPLICATE CURSOR	duplicate_cursor
42P04	DUPLICATE DATABASE	duplicate_database
42723	DUPLICATE FUNCTION	duplicate_function
42P05	DUPLICATE PREPARED STATEMENT	duplicate_prepared_statement
42P06	DUPLICATE SCHEMA	duplicate_schema
42P07	DUPLICATE TABLE	duplicate_table
42712	DUPLICATE ALIAS	duplicate_alias
42710	DUPLICATE OBJECT	duplicate_object
42702	AMBIGUOUS COLUMN	ambiguous_column

Code erreur	Signification	Nom de condition
42725	AMBIGUOUS FUNCTION	ambiguous_function
42P08	AMBIGUOUS PARAMETER	ambiguous_parameter
42P09	AMBIGUOUS ALIAS	ambiguous_alias
42P10	INVALID COLUMN REFERENCE	invalid_column_reference
42611	INVALID COLUMN DEFINITION	invalid_column_definition
42P11	INVALID CURSOR DEFINITION	invalid_cursor_definition
42P12	INVALID DATABASE DEFINITION	invalid_database_definition
42P13	INVALID FUNCTION DEFINITION	invalid_function_definition
42P14	INVALID PREPARED STATEMENT DEFINITION	invalid_prepared_statement_definition
42P15	INVALID SCHEMA DEFINITION	invalid_schema_definition
42P16	INVALID TABLE DEFINITION	invalid_table_definition
42P17	INVALID OBJECT DEFINITION	invalid_object_definition
Classe 44 -- Violation de WITH CHECK OPTION		
44000	WITH CHECK OPTION VIOLATION	with_check_option_violation
Classe 53 -- Ressources insuffisantes		
53000	INSUFFICIENT RESOURCES	insufficient_resources
53100	DISK FULL	disk_full
53200	OUT OF MEMORY	out_of_memory
53300	TOO MANY CONNECTIONS	too_many_connections
Classe 54 -- Limite du programme dépassée		
54000	PROGRAM LIMIT EXCEEDED	program_limit_exceeded
54001	STATEMENT TOO COMPLEX	statement_too_complex
54011	TOO MANY COLUMNS	too_many_columns
54023	TOO MANY ARGUMENTS	too_many_arguments
Classe 55 -- L'objet n'est pas l'état prérequis		
55000	OBJECT NOT IN PREREQUISITE STATE	object_not_in_prerequisite_state
55006	OBJECT IN USE	object_in_use
55P02	CANT CHANGE RUNTIME PARAM	cant_change_runtime_param
55P03	LOCK NOT AVAILABLE	lock_not_available
Classe 57 -- Intervention d'un opérateur		
57000	OPERATOR INTERVENTION	operator_intervention
57014	QUERY CANCELED	query_canceled
57P01	ADMIN SHUTDOWN	admin_shutdown
57P02	CRASH SHUTDOWN	crash_shutdown
57P03	CANNOT CONNECT NOW	cannot_connect_now
57P04	DATABASE DROPPED	database_dropped
Class 58 -- Erreur système (erreurs externes à PostgreSQL™)		
58030	IO ERROR	io_error
58P01	UNDEFINED FILE	undefined_file
58P02	DUPLICATE FILE	duplicate_file
Classe F0 -- Erreur dans le fichier de configuration		
F0000	CONFIG FILE ERROR	config_file_error
F0001	LOCK FILE EXISTS	lock_file_exists
Classe P0 -- Erreur PL/pgSQL		

Code erreur	Signification	Nom de condition
P0000	PLPGSQL ERROR	plpgsql_error
P0001	RAISE EXCEPTION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows
Classe XX -- Erreur interne		
XX000	INTERNAL ERROR	internal_error
XX001	DATA CORRUPTED	data_corrupted
XX002	INDEX CORRUPTED	index_corrupted

Annexe B. Support de date/heure

PostgreSQL™ utilise un analyseur heuristique interne pour le support des dates/heures saisies. Les dates et heures, saisies sous la forme de chaînes de caractères, sont découpées en champs distincts après détermination du type d'information contenue dans chaque champ. Chaque champ est interpreté ; une valeur peut lui être affectée, il peut être ignoré ou encore être rejeté. Le parseur contient des tables de recherche internes pour tous les champs textuels y compris les mois, les jours de la semaine et les fuseaux horaires.

Cette annexe décrit le contenu des tables de correspondance et les méthodes utilisées par le parseur pour décoder les dates et heures.

B.1. Interprétation des Date/Heure saisies

Les entrées de type date/heure sont toutes décodées en utilisant le processus suivant.

1. Diviser la chaîne saisie en lexèmes et catégoriser les lexèmes en chaînes, heures, fuseaux horaires et nombres.
   a. Si le lexème numérique contient un double-point (`:`), c'est une chaîne de type heure. On inclut tous les chiffres et double-points qui suivent.
   b. Si le lexème numérique contient un tiret (`-`), une barre oblique (`/`) ou au moins deux points (`.`), c'est une chaîne de type date qui contient peut-être un mois sous forme textuelle. Si un lexème de date a déjà été reconnu, il est alors interprété comme un nom de fuseau horaire (par exemple `America/New_York`).
   c. Si le lexème n'est que numérique alors il s'agit soit d'un champ simple soit d'une date concaténée ISO 8601 (`19990113` pour le 13 janvier 1999, par exemple) ou d'une heure concaténée ISO 8601 (`141516` pour 14:15:16, par exemple).
   d. Si le lexème débute par le signe plus (`+`) ou le signe moins (`-`), alors il s'agit soit d'un fuseau horaire numérique, soit d'un champ spécial.
2. Si le lexème est une chaîne texte, le comparer avec les différentes chaînes possibles :
   a. Faire une recherche binaire dans la table pour vérifier si le lexème est une abréviation de fuseau horaire.
   b. S'il n'est pas trouvé, une recherche binaire est effectuée dans la table pour vérifier si le lexème est une chaîne spéciale (`today`, par exemple), un jour (`Thursday`, par exemple), un mois (`January`, par exemple), ou du bruit (`at`, `on`, par exemple).
   c. Si le lexème n'est toujours pas trouvé, une erreur est levée.
3. Lorsque le lexème est un nombre ou un champ de nombre :
   a. S'il y a huit ou six chiffres, et qu'aucun autre champ date n'a été lu, alors il est interprété comme une « date concaténée » (`19990118` ou `990118`, par exemple). L'interprétation est `AAAAMMJJ` ou `AAMMJJ`.
   b. Si le lexème est composé de trois chiffres et qu'une année est déjà lue, alors il est interprété comme un jour de l'année.
   c. Si quatre ou six chiffres et une année sont déjà lus, alors il est interprété comme une heure (`HHMM` ou `HHMMSS`).
   d. Si le lexème est composé de trois chiffres ou plus et qu'aucun champ date n'a été trouvé, il est interprété comme une année (cela impose l'ordre aa-mm-jj des champs dates restants).
   e. Dans tous les autres cas, le champ date est supposé suivre l'ordre imposé par le paramêtre `datestyle` : mm-jj-aa, jj-mm-aa, ou aa-mm-jj. Si un champ jour ou mois est en dehors des limites, une erreur est levée.
4. Si BC est indiqué, le signe de l'année est inversé et un est ajouté pour le stockage interne. (Il n'y a pas d'année zéro dans le calendrier Grégorien, alors numériquement 1 BC devient l'année zéro.)
5. Si BC n'est pas indiqué et que le champ année est composé de deux chiffres, alors l'année est ajustée à quatre chiffres. Si le champ vaut moins que 70, alors on ajoute 2000, sinon 1900.

Astuce

Les années du calendrier Grégorien AD 1-99 peuvent être saisie avec 4 chiffres, deux zéros en tête (`0099`

pour AD 99, par exemple).

B.2. Mots clés Date/Heure

Tableau B.1, « Noms de mois » présente les lexèmes reconnus commme des noms de mois.

Tableau B.1. Noms de mois

Mois	Abréviations
January (Janvier)	Jan
February (Février)	Feb
March (Mars)	Mar
April (Avril)	Apr
May (Mai)	
June (Juin)	Jun
July (Juillet)	Jul
August (Août)	Aug
September (Septembre)	Sep, Sept
October (Octobre)	Oct
November (Novembre)	Nov
December (Décembre)	Dec

Tableau B.2, « Noms des jours de la semaine » présente les lexèmes reconnus comme des noms de jours de la semaine.

Tableau B.2. Noms des jours de la semaine

Jour	Abréviation
Sunday (Dimanche)	Sun
Monday (Lundi)	Mon
Tuesday (Mardi)	Tue, Tues
Wednesday (Mercredi)	Wed, Weds
Thursday (Jeudi)	Thu, Thur, Thurs
Friday (Vendredi)	Fri
Saturday (Samedi)	Sat

Tableau B.3, « Modificateurs de Champs Date/Heure » présente les lexèmes utilisés par divers modificateurs.

Tableau B.3. Modificateurs de Champs Date/Heure

Identifiant	Description
`AM`	L'heure précède 12:00
`AT`	Ignoré
`JULIAN`, `JD`, `J`	Le champ suivant est un jour du calendrier Julien
`ON`	Ignoré
`PM`	L'heure suit 12:00
`T`	Le champ suivant est une heure

B.3. Fichiers de configuration date/heure

Comme il n'existe pas de réel standard des abréviations de fuseaux horaire, PostgreSQL™ permet de personnaliser l'ensemble des abréviations acceptées par le serveur. Le paramètre d'exécution timezone_abbreviations détermine l'ensemble des abréviations actives. Bien que tout utilisateur de la base puisse modifier ce paramètre, les valeurs possibles sont sous le contrôle de l'administrateur de bases de données -- ce sont en fait les noms des fichiers de configuration stockés dans `.../share/timezonesets/` du répertoire d'installation. En ajoutant ou en modifiant les fichiers de ce répertoire, l'administrateur peut définir les règles d'abréviation des fuseaux horaires.

`timezone_abbreviations` peut prendre tout nom de fichier situé dans `.../share/timezonesets/`, sous réserve que ce nom soit purement alphabétique. (L'interdiction de caractères non alphabétique dans `timezone_abbreviations` empêche la lecture de fichiers en dehors du répertoire prévu et celle de fichiers de sauvegarde ou autre.)

Un fichier d'abréviation de zones horaires peut contenir des lignes blanches et des commentaires (commençant avec un #). Les autres lignes doivent suivre l'un des formats suivants :

```
nom_fuseau_horaire décalage
nom_fuseau_horaire décalage D
@INCLUDE nom_fichier
@OVERRIDE
```

Un *`nom_fuseau_horaire`* n'est que l'abréviation définie. Le *`décalage`* est le décalage en secondes à partir d'UTC, une valeur positive signifiant à l'est de Greenwich, une valeur négative à l'ouest. Ainsi, -18000 représente cinq heures à l'ouest de Greenwich, soit l'heure standard de la côte ouest nord américaine. `D` indique que le nom du fuseau représente une heure soumise à des règles de changement d'heure plutôt que l'heure standard. Comme tous les décalages de fuseau horaire ont des limites de 15 minutes, le nombre de secondes doit être un multiple de 900.

La syntaxe `@INCLUDE` autorise l'inclusion d'autres fichiers du répertoire `.../share/timezonesets/`. Les inclusions peuvent être imbriquées jusqu'à une certaine profondeur.

La syntaxe `@OVERRIDE` indique que les entrées suivantes du fichier peuvent surcharger les entrées précédentes (c'est-à-dire des entrées obtenues à partir de fichiers inclus). Sans cela, les définitions en conflit au sein d'une même abréviation lèvent une erreur.

Dans une installation non modifiée, le fichier `Default` contient toutes les abréviations de fuseaux horaire, sans conflit, pour la quasi-totalité du monde. Les fichiers supplémentaires `Australia` et `India` sont fournis pour ces régions : ces fichiers incluent le fichier `Default` puis ajoutent ou modifient les fuseaux horaires si nécessaire.

Pour des raisons de référence, une installation standard contient aussi des fichiers `Africa.txt`, `America.txt`, etc. qui contiennent des informations sur les abréviations connues et utilisées en accord avec la base de données de fuseaux horaires `zoneinfo`. Les définitions des noms de zone trouvées dans ces fichiers peuvent être copiées et collées dans un fichier de configuration personnalisé si nécessaire. Il ne peut pas être fait directement référence à ces fichiers dans le paramètre `timezone_abbreviations` à cause du point dans leur nom.

Note

Si une erreur survient lors de la lecture des jeux de données de fuseaux horaires, aucune nouvelle valeur n'est acceptée mais les anciennes sont conservées. Si l'erreur survient au démarrage de la base, celui-ci échoue.

Attention

Les abréviations de fuseau horaire définies dans le fichier de configuration surchargent les informations sans fuseau définies nativement dans PostgreSQL™. Par exemple, le fichier de configuration `Australia` définit `SAT` (*South Australian Standard Time*, soit l'heure standard pour l'Australie du sud). Si ce fichier est actif, `SAT` n'est plus reconnu comme abréviation de samedi (*Saturday*).

Attention

Si les fichiers de `.../share/timezonesets/` sont modifiés, il revient à l'utilisateur de procéder à leur sauvegarde -- une sauvegarde normale de base n'inclut pas ce répertoire.

B.4. Histoire des unités

Le calendrier Julien a été introduit par Julius Caesar en -45. Il était couramment utilisé dans le monde occidental jusqu'en l'an

1582, date à laquelle des pays ont commencé à se convertir au calendrier Grégorien. Dans le calendrier Julien, l'année tropicale est arrondie à 365 jours 1/4, soit 365,25 jours. Cela conduit à une erreur de l'ordre d'un jour tous les 128 ans.

L'erreur grandissante du calendrier poussa le Pape Gregoire XIII a réformé le calendrier en accord avec les instructions du Concile de Trent. Dans le calendrier Grégorien, l'année tropicale est arrondie à 365 + 97/400 jours, soit 365,2425 jours. Il faut donc à peu prés 3300 ans pour que l'année tropicale subissent un décalage d'un an dans le calendrier Grégorien.

L'arrondi 365+97/400 est obtenu à l'aide de 97 années bissextiles tous les 400 ans. Les règles suivantes sont utilisées :

toute année divisible par 4 est bissextile ;
cependant, toute année divisible par 100 n'est pas bissextile ;
cependant, toute années divisible par 400 est bissextile.
1700, 1800, 1900, 2100 et 2200 ne sont donc pas des années bissextiles. 1600, 2000 et 2400 si. Par opposition, dans l'ancien calendrier Julien, toutes les années divisibles par 4 sont bissextiles.

En février 1582, le pape décrèta que 10 jours devaient étre supprimés du mois d'octobre 1582, le 15 octobre devant ainsi arriver après le 4 octobre. Cela a été appliqué en Italie, Pologne, Portugal et Espagne. Les autres pays catholiques ont suivi peu après, mais les pays protestants ont été plus rétifs et les contrées orthodoxes grèques n'ont pas effctué le changement avant le début du 20ème siècle. La réforme a été appliquée par la Grande Bretagne et ses colonies (y compris les actuels Etats-Unis) en 1752. Donc le 2 septembre 1752 a été suivi du 14 septembre 1752. C'est pour cela que la commande **cal** produit la sortie suivante :

```
$ cal 9 1752
   septembre 1752
di lu ma me je ve sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Le standard SQL stipule que « dans la définition d'un “libellé date/heure” (*datetime literal*), les “valeurs date/heure” sont contraintes par les règles naturelles des dates et heures imposées par le calendrier Grégorien ». Les dates comprises entre le 5 octobre 1582 et le 14 octobre 1582, bien qu'éliminées dans plusieurs pays par ordre du Pape, sont conformes aux « règles naturelles » et sont donc des dates valables. PostgreSQL™ suit le standard SQL en comptant les dates exclusivement dans le calendrier grégorien, même pour les années où ce calendrier n'existait pas encore.

Divers calendriers ont été developés dans différentes parties du monde, la plupart précède le système Grégorien. Par exemple, les débuts du calendrier chinois peuvent être évalués aux alentours du 14ème siécle avant J.-C. La légende veut que l'empereur Huangdi inventa le calendrier en 2637 avant J-C. La République de Chine utilise le calendrier Grégorien pour les besoins civils. Le calendrier chinois est utilisé pour déterminer les festivals.

La « date Julien » n'a pas de relation avec le « calendrier Julien ». Le système de date Julien a été inventé par le précepteur français Joseph Justus Scaliger (1540-1609) et tient probablement son nom du père de Scaliger, le précepteur italien Julius Caesar Scaliger (1484-1558). Dans le système de date Julien, chaque jour est un nombre séquentiel, commençant à partir de JD 0, appelé quelque fois *la* date Julien. JD 0 correspond au 1er janvier 4713 avant JC dans le calendrier Julien, ou au 24 novembre 4714 avant JC dans le calendrier grégorien. Le comptage de la date Julien est le plus souvent utilisé par les astronomes pour donner un nom à leurs observations, et du coup une date part de midi UTC jusqu'au prochain midi UTC, plutôt que de minuit à minuit : JD 0 désigne les 24 heures de midi UTC le 1er janvier 4713 avant JC jusqu'au midi UTC du 2 janvier 4713 avant JC.

Bien que PostgreSQL™ accepte la saisie et l'affichage des dates en notation de date Julien (et les utilise aussi pour quelques calculs internes de date et heure), il n'utilise pas le coup des dates de midi à midi. PostgreSQL™ traite une date Julien comme allant de minuit à minuit.

Annexe C. Mots-clé SQL

La Tableau C.1, « Mots-clé SQL » liste tous les éléments qui sont des mots-clé dans le standard SQL et dans PostgreSQL™ 9.0.18. Des informations sous-jacentes peuvent être trouvées dans Section 4.1.1, « identificateurs et mots clés ».

SQL distingue les mots-clé *réservés* et *non réservés*. Selon le standard, les mots-clé réservés sont réellement les seuls mots-clé ; ils ne sont jamais autorisés comme identifiants. Les mots-clé non réservés ont seulement un sens spécial dans certains contextes et peuvent être utilisés comme identifiants dans d'autres contextes. La plupart des mots-clé non réservés sont en fait les noms des tables et des fonctions prédéfinies spécifiés par SQL. Le concept de mots-clé non réservés existe seulement pour indiquer que certains sens prédéfinis sont attachés à un mot dans certains contextes.

Dans l'analyseur de PostgreSQL™, la vie est un peu plus compliquée. Il y a différentes classes d'éléments allant de ceux que l'on ne peut jamais utiliser comme identifiants à ceux qui n'ont absolument aucun statut spécial dans l'analyseur par rapport à un identifiant ordinaire (c'est généralement le cas pour les fonctions spécifiées par SQL). Même les mots-clé réservés ne sont pas complètement réservés dans PostgreSQL™ et peuvent être utilisés comme noms des colonnes (par exemple, `SELECT 55 AS CHECK`, même si CHECK est un mot-clé).

Dans Tableau C.1, « Mots-clé SQL », dans la colonne pour PostgreSQL™, nous classons comme « non réservé » les mots-clé qui sont explicitement connus par l'analyseur mais qui sont autorisés en tant que noms de colonnes ou de tables. Certains mots-clé qui sont non réservés et qui ne peuvent pas être utilisés comme un nom de fonction ou un type de données sont marqués en conséquence. (La plupart des mots représentent des fonctions prédéfinies ou des types de données avec une syntaxe spéciale. La fonction ou le type est toujours disponible mais il ne peut pas être redéfini par un utilisateur.) Les « réservés » sont des éléments qui ne sont pas autorisés en tant que noms de colonne ou de table. Certains mots-clé réservés sont autorisés comme noms pour les fonctions et les types de données ; cela est également montré dans le tableau. Dans le cas contraire, un mot clé réservé est seulement autorisé dans un nom de label « AS » d'une colonne.

En règle générale, si vous avez des erreurs de la part de l'analyseur pour des commandes qui contiennent un des mots-clés listés comme identifiants, vous devriez essayer de mettre entre guillemets l'identifiant pour voir si le problème disparait.

Il est important de comprendre avant d'étudier la Tableau C.1, « Mots-clé SQL » que le fait qu'un mot-clé ne soit pas réservé dans PostgreSQL™ ne signifie pas que la fonctionnalité en rapport avec ce mot n'est pas implémentée. Réciproquement, la présence d'un mot-clé n'indique pas l'existance d'une fonctionnalité.

Tableau C.1. Mots-clé SQL

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
A		non réservé	non réservé		
ABORT	non réservé				
ABS		réservé	réservé	non réservé	
ABSENT		non réservé			
ABSOLUTE	non réservé	non réservé	non réservé	réservé	réservé
ACCESS	non réservé				
ACCORDING		non réservé			
ACTION	non réservé	non réservé	non réservé	réservé	réservé
ADA		non réservé	non réservé	non réservé	non réservé
ADD	non réservé	non réservé	non réservé	réservé	réservé
ADMIN	non réservé	non réservé	non réservé	réservé	
AFTER	non réservé	non réservé	non réservé	réservé	
AGGREGATE	non réservé			réservé	
ALIAS				réservé	
ALL	réservé	réservé	réservé	réservé	réservé
ALLOCATE		réservé	réservé	réservé	réservé
ALSO	non réservé				
ALTER	non réservé	réservé	réservé	réservé	réservé
ALWAYS	non réservé	non réservé	non réservé		
ANALYSE	réservé				

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
ANALYZE	réservé				
AND	réservé	réservé	réservé	réservé	réservé
ANY	réservé	réservé	réservé	réservé	réservé
ARE		réservé	réservé	réservé	réservé
ARRAY	réservé	réservé	réservé	réservé	
ARRAY_AGG		réservé			
AS	réservé	réservé	réservé	réservé	réservé
ASC	réservé	non réservé	non réservé	réservé	réservé
ASENSITIVE		réservé	réservé	non réservé	
ASSERTION	non réservé	non réservé	non réservé	réservé	réservé
ASSIGNMENT	non réservé	non réservé	non réservé	non réservé	
ASYMMETRIC	réservé	réservé	réservé	non réservé	
AT	non réservé	réservé	réservé	réservé	réservé
ATOMIC		réservé	réservé	non réservé	
ATTRIBUTE		non réservé	non réservé		
ATTRIBUTES		non réservé	non réservé		
AUTHORIZATION	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
AVG		réservé	réservé	non réservé	réservé
BACKWARD	non réservé				
BASE64		non réservé	non réservé		
BEFORE	non réservé	non réservé	non réservé	réservé	
BEGIN	non réservé	réservé	réservé	réservé	réservé
BERNOULLI		non réservé	non réservé		
BETWEEN	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
BIGINT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
BINARY	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	
BIT	non réservé (ne peut pas être une fonction ou un type)			réservé	réservé
BITVAR				non réservé	
BIT_LENGTH				non réservé	réservé
BLOB		réservé	réservé	réservé	
BLOCKED		non réservé	non réservé		
BOM		non réservé			
BOOLEAN	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	
BOTH	réservé	réservé	réservé	réservé	réservé
BREADTH		non réservé	non réservé	réservé	
BY	non réservé	réservé	réservé	réservé	réservé
C		non réservé	non réservé	non réservé	non réservé

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
CACHE	non réservé				
CALL		réservé	réservé	réservé	
CALLED	non réservé	réservé	réservé	non réservé	
CARDINALITY		réservé	réservé	non réservé	
CASCADE	non réservé	non réservé	non réservé	réservé	réservé
CASCADED	non réservé	réservé	réservé	réservé	réservé
CASE	réservé	réservé	réservé	réservé	réservé
CAST	réservé	réservé	réservé	réservé	réservé
CATALOG	non réservé	non réservé	non réservé	réservé	réservé
CATALOG_NAME		non réservé	non réservé	non réservé	non réservé
CEIL		réservé	réservé		
CEILING		réservé	réservé		
CHAIN	non réservé	non réservé	non réservé	non réservé	
CHAR	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
CHARACTER	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
CHARACTERISTICS	non réservé	non réservé	non réservé		
CHARACTERS		non réservé	non réservé		
CHARACTER_LENGTH		réservé	réservé	non réservé	réservé
CHARACTER_SET_CATALOG		non réservé	non réservé	non réservé	non réservé
CHARACTER_SET_NAME		non réservé	non réservé	non réservé	non réservé
CHARACTER_SET_SCHEMA		non réservé	non réservé	non réservé	non réservé
CHAR_LENGTH		réservé	réservé	non réservé	réservé
CHECK	réservé	réservé	réservé	réservé	réservé
CHECKED				non réservé	
CHECKPOINT	non réservé				
CLASS	non réservé			réservé	
CLASS_ORIGIN		non réservé	non réservé	non réservé	non réservé
CLOB		réservé	réservé	réservé	
CLOSE	non réservé	réservé	réservé	réservé	réservé
CLUSTER	non réservé				
COALESCE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
COBOL		non réservé	non réservé	non réservé	non réservé
COLLATE	réservé	réservé	réservé	réservé	réservé
COLLATION		non réservé	non réservé	réservé	réservé
COLLATION_CATALOG		non réservé	non réservé	non réservé	non réservé
COLLATION_NAME		non réservé	non réservé	non réservé	non réservé
COLLATION_SCHEMA		non réservé	non réservé	non réservé	non réservé
COLLECT		réservé	réservé		
COLUMN	réservé	réservé	réservé	réservé	réservé
COLUMNS		non réservé			

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
COLUMN_NAME		non réservé	non réservé	non réservé	non réservé
COMMAND_FUNCTION		non réservé	non réservé	non réservé	non réservé
COMMAND_FUNCTION_CODE		non réservé	non réservé	non réservé	
COMMENT	non réservé				
COMMENTS	non réservé				
COMMIT	non réservé	réservé	réservé	réservé	réservé
COMMITTED	non réservé	non réservé	non réservé	non réservé	non réservé
COMPLETION				réservé	
CONCURRENTLY	non réservé (peut être une fonction ou un type)				
CONDITION		réservé	réservé		
CONDITION_NUMBER		non réservé	non réservé	non réservé	non réservé
CONFIGURATION	non réservé				
CONNECT		réservé	réservé	réservé	réservé
CONNECTION	non réservé	non réservé	non réservé	réservé	réservé
CONNECTION_NAME		non réservé	non réservé	non réservé	non réservé
CONSTRAINT	réservé	réservé	réservé	réservé	réservé
CONSTRAINTS	non réservé	non réservé	non réservé	réservé	réservé
CONSTRAINT_CATALOG		non réservé	non réservé	non réservé	non réservé
CONSTRAINT_NAME		non réservé	non réservé	non réservé	non réservé
CONSTRAINT_SCHEMA		non réservé	non réservé	non réservé	non réservé
CONSTRUCTOR		non réservé	non réservé	réservé	
CONTAINS		non réservé	non réservé	non réservé	
CONTENT	non réservé	non réservé	non réservé		
CONTINUE	non réservé	non réservé	non réservé	réservé	réservé
CONTROL		non réservé	non réservé		
CONVERSION	non réservé				
CONVERT		réservé	réservé	non réservé	réservé
COPY	non réservé				
CORR		réservé	réservé		
CORRESPONDING		réservé	réservé	réservé	réservé
COST	non réservé				
COUNT		réservé	réservé	non réservé	réservé
COVAR_POP		réservé	réservé		
COVAR_SAMP		réservé	réservé		
CREATE	réservé	réservé	réservé	réservé	réservé
CREATEDB	non réservé				
CREATEROLE	non réservé				
CREATEUSER	non réservé				
CROSS	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
CSV	non réservé				
CUBE		réservé	réservé	réservé	
CUME_DIST		réservé	réservé		

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
CURRENT	non réservé	réservé	réservé	réservé	réservé
CURRENT_CATALOG	réservé	réservé			
CURRENT_DATE	réservé	réservé	réservé	réservé	réservé
CURRENT_DEFAULT_TRANSFORM_GROUP		réservé	réservé		
CURRENT_PATH		réservé	réservé	réservé	
CURRENT_ROLE	réservé	réservé	réservé	réservé	
CURRENT_SCHEMA	réservé (peut être une fonction ou un type)	réservé			
CURRENT_TIME	réservé	réservé	réservé	réservé	réservé
CURRENT_TIMESTAMP	réservé	réservé	réservé	réservé	réservé
CURRENT_TRANSFORM_GROUP_FOR_TYPE		réservé	réservé		
CURRENT_USER	réservé	réservé	réservé	réservé	réservé
CURSOR	non réservé	réservé	réservé	réservé	réservé
CURSOR_NAME		non réservé	non réservé	non réservé	non réservé
CYCLE	non réservé	réservé	réservé	réservé	
DATA	non réservé	non réservé	non réservé	réservé	non réservé
DATABASE	non réservé				
DATALINK		réservé	réservé		
DATE		réservé	réservé	réservé	réservé
DATETIME_INTERVAL_CODE		non réservé	non réservé	non réservé	non réservé
DATETIME_INTERVAL_PRECISION		non réservé	non réservé	non réservé	non réservé
DAY	non réservé	réservé	réservé	réservé	réservé
DB		non réservé	non réservé		
DEALLOCATE	non réservé	réservé	réservé	réservé	réservé
DEC	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
DECIMAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
DECLARE	non réservé	réservé	réservé	réservé	réservé
DEFAULT	réservé	réservé	réservé	réservé	réservé
DEFAULTS	non réservé	non réservé	non réservé		
DEFERRABLE	réservé	non réservé	non réservé	réservé	réservé
DEFERRED	non réservé	non réservé	non réservé	réservé	réservé
DEFINED		non réservé	non réservé	non réservé	
DEFINER	non réservé	non réservé	non réservé	non réservé	
DEGREE		non réservé	non réservé		
DELETE	non réservé	réservé	réservé	réservé	réservé
DELIMITER	non réservé				
DELIMITERS	non réservé				
DENSE_RANK		réservé	réservé		
DEPTH		non réservé	non réservé	réservé	
DEREF		réservé	réservé	réservé	

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
DERIVED		non réservé	non réservé		
DESC	réservé	non réservé	non réservé	réservé	réservé
DESCRIBE		réservé	réservé	réservé	réservé
DESCRIPTOR		non réservé	non réservé	réservé	réservé
DESTROY				réservé	
DESTRUCTOR				réservé	
DETERMINISTIC		réservé	réservé	réservé	
DIAGNOSTICS		non réservé	non réservé	réservé	réservé
DICTIONARY	non réservé			réservé	
DISABLE	non réservé				
DISCARD	non réservé				
DISCONNECT		réservé	réservé	réservé	réservé
DISPATCH		non réservé	non réservé	non réservé	
DISTINCT	réservé	réservé	réservé	réservé	réservé
DLNEWCOPY		réservé	réservé		
DLPREVIOUSCOPY		réservé	réservé		
DLURLCOMPLETE		réservé	réservé		
DLURLCOMPLETEONLY		réservé	réservé		
DLURLCOMPLETEWRITE		réservé	réservé		
DLURLPATH		réservé	réservé		
DLURLPATHONLY		réservé	réservé		
DLURLPATHWRITE		réservé	réservé		
DLURLSCHEME		réservé	réservé		
DLURLSERVER		réservé	réservé		
DLVALUE		réservé	réservé		
DO	réservé				
DOCUMENT	non réservé	non réservé	non réservé		
DOMAIN	non réservé	non réservé	non réservé	réservé	réservé
DOUBLE	non réservé	réservé	réservé	réservé	réservé
DROP	non réservé	réservé	réservé	réservé	réservé
DYNAMIC		réservé	réservé	réservé	
DYNAMIC_FUNCTION		non réservé	non réservé	non réservé	non réservé
DYNAMIC_FUNCTION_CODE		non réservé	non réservé	non réservé	
EACH	non réservé	réservé	réservé	réservé	
ELEMENT		réservé	réservé		
ELSE	réservé	réservé	réservé	réservé	réservé
EMPTY		non réservé			
ENABLE	non réservé				
ENCODING	non réservé	non réservé			
ENCRYPTED	non réservé				
END	réservé	réservé	réservé	réservé	réservé
END-EXEC		réservé	réservé	réservé	réservé
ENUM	non réservé				
EQUALS		non réservé	non réservé	réservé	

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
ESCAPE	non réservé	réservé	réservé	réservé	réservé
EVERY		réservé	réservé	réservé	
EXCEPT	réservé	réservé	réservé	réservé	réservé
EXCEPTION			non réservé	réservé	réservé
EXCLUDE	non réservé	non réservé	non réservé		
EXCLUDING	non réservé	non réservé	non réservé		
EXCLUSIVE	non réservé				
EXEC		réservé	réservé	réservé	réservé
EXECUTE	non réservé	réservé	réservé	réservé	réservé
EXISTING				non réservé	
EXISTS	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
EXP		réservé	réservé		
EXPLAIN	non réservé				
EXTERNAL	non réservé	réservé	réservé	réservé	réservé
EXTRACT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
FALSE	réservé	réservé	réservé	réservé	réservé
FAMILY	non réservé				
FETCH	réservé	réservé	réservé	réservé	réservé
FILE		non réservé	non réservé		
FILTER		réservé	réservé		
FINAL		non réservé	non réservé	non réservé	
FIRST	non réservé	non réservé	non réservé	réservé	réservé
FIRST_VALUE		réservé			
FLAG		non réservé			
FLOAT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
FLOOR		réservé	réservé		
FOLLOWING	non réservé	non réservé	non réservé		
FOR	réservé	réservé	réservé	réservé	réservé
FORCE	non réservé				
FOREIGN	réservé	réservé	réservé	réservé	réservé
FORTRAN		non réservé	non réservé	non réservé	non réservé
FORWARD	non réservé				
FOUND		non réservé	non réservé	réservé	réservé
FREE		réservé	réservé	réservé	
FREEZE	réservé (peut être une fonction ou un type)				
FROM	réservé	réservé	réservé	réservé	réservé
FS		non réservé	non réservé		
FULL	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
FUNCTION	non réservé	réservé	réservé	réservé	

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
FUNCTIONS	non réservé				
FUSION		réservé	réservé		
G		non réservé	non réservé	non réservé	
GENERAL		non réservé	non réservé	réservé	
GENERATED		non réservé	non réservé	non réservé	
GET		réservé	réservé	réservé	réservé
GLOBAL	non réservé	réservé	réservé	réservé	réservé
GO		non réservé	non réservé	réservé	réservé
GOTO		non réservé	non réservé	réservé	réservé
GRANT	réservé	réservé	réservé	réservé	réservé
GRANTED	non réservé	non réservé	non réservé	non réservé	
GREATEST	non réservé (ne peut pas être une fonction ou un type)				
GROUP	réservé	réservé	réservé	réservé	réservé
GROUPING		réservé	réservé	réservé	
HANDLER	non réservé				
HAVING	réservé	réservé	réservé	réservé	réservé
HEADER	non réservé				
HEX		non réservé	non réservé		
HIERARCHY		non réservé	non réservé	non réservé	
HOLD	non réservé	réservé	réservé	non réservé	
HOST				réservé	
HOUR	non réservé	réservé	réservé	réservé	réservé
ID		non réservé			
IDENTITY	non réservé	réservé	réservé	réservé	réservé
IF	non réservé				
IGNORE		non réservé		réservé	
ILIKE	réservé (peut être une fonction ou un type)				
IMMEDIATE	non réservé	non réservé	non réservé	réservé	réservé
IMMUTABLE	non réservé				
IMPLEMENTATION		non réservé	non réservé	non réservé	
IMPLICIT	non réservé				
IMPORT		réservé	réservé		
IN	réservé	réservé	réservé	réservé	réservé
INCLUDING	non réservé	non réservé	non réservé		
INCREMENT	non réservé	non réservé	non réservé		
INDENT		non réservé			
INDEX	non réservé				
INDEXES	non réservé				
INDICATOR		réservé	réservé	réservé	réservé
INFIX				non réservé	
INHERIT	non réservé				
INHERITS	non réservé				

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
INITIALIZE				réservé	
INITIALLY	réservé	non réservé	non réservé	réservé	réservé
INLINE	non réservé				
INNER	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
INOUT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	
INPUT	non réservé	non réservé	non réservé	réservé	réservé
INSENSITIVE	non réservé	réservé	réservé	non réservé	réservé
INSERT	non réservé	réservé	réservé	réservé	réservé
INSTANCE		non réservé	non réservé	non réservé	
INSTANTIABLE		non réservé	non réservé	non réservé	
INSTEAD	non réservé	non réservé			
INT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
INTEGER	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
INTEGRITY		non réservé	non réservé		
INTERSECT	réservé	réservé	réservé	réservé	réservé
INTERSECTION		réservé	réservé		
INTERVAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
INTO	réservé	réservé	réservé	réservé	réservé
INVOKER	non réservé	non réservé	non réservé	non réservé	
IS	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
ISNULL	réservé (peut être une fonction ou un type)				
ISOLATION	non réservé	non réservé	non réservé	réservé	réservé
ITERATE				réservé	
JOIN	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
K		non réservé	non réservé	non réservé	
KEY	non réservé	non réservé	non réservé	réservé	réservé
KEY_MEMBER		non réservé	non réservé	non réservé	
KEY_TYPE		non réservé	non réservé	non réservé	
LAG		réservé			
LANGUAGE	non réservé	réservé	réservé	réservé	réservé
LARGE	non réservé	réservé	réservé	réservé	
LAST	non réservé	non réservé	non réservé	réservé	réservé
LAST_VALUE		réservé			
LATERAL		réservé	réservé	réservé	
LC_COLLATE	non réservé				
LC_CTYPE	non réservé				

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
LEAD		réservé			
LEADING	réservé	réservé	réservé	réservé	réservé
LEAST	non réservé (ne peut pas être une fonction ou un type)				
LEFT	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
LENGTH		non réservé	non réservé	non réservé	non réservé
LESS				réservé	
LEVEL	non réservé	non réservé	non réservé	réservé	réservé
LIBRARY		non réservé	non réservé		
LIKE	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
LIKE_REGEX		réservé			
LIMIT	réservé	non réservé	non réservé	réservé	
LINK		non réservé	non réservé		
LISTEN	non réservé				
LN		réservé	réservé		
LOAD	non réservé				
LOCAL	non réservé	réservé	réservé	réservé	réservé
LOCALTIME	réservé	réservé	réservé	réservé	
LOCALTIMESTAMP	réservé	réservé	réservé	réservé	
LOCATION	non réservé	non réservé			
LOCATOR		non réservé	non réservé	réservé	
LOCK	non réservé				
LOGIN	non réservé				
LOWER		réservé	réservé	non réservé	réservé
M		non réservé	non réservé	non réservé	
MAP		non réservé	non réservé	réservé	
MAPPING	non réservé	non réservé	non réservé		
MATCH	non réservé	réservé	réservé	réservé	réservé
MATCHED		non réservé	non réservé		
MAX		réservé	réservé	non réservé	réservé
MAXVALUE	non réservé	non réservé	non réservé		
MAX_CARDINALITY		réservé			
MEMBER		réservé	réservé		
MERGE		réservé	réservé		
MESSAGE_LENGTH		non réservé	non réservé	non réservé	non réservé
MESSAGE_OCTET_LENGTH		non réservé	non réservé	non réservé	non réservé
MESSAGE_TEXT		non réservé	non réservé	non réservé	non réservé
METHOD		réservé	réservé	non réservé	
MIN		réservé	réservé	non réservé	réservé
MINUTE	non réservé	réservé	réservé	réservé	réservé
MINVALUE	non réservé	non réservé	non réservé		
MOD		réservé	réservé	non réservé	
MODE	non réservé				

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
MODIFIES		réservé	réservé	réservé	
MODIFY				réservé	
MODULE		réservé	réservé	réservé	réservé
MONTH	non réservé	réservé	réservé	réservé	réservé
MORE		non réservé	non réservé	non réservé	non réservé
MOVE	non réservé				
MULTISET		réservé	réservé		
MUMPS		non réservé	non réservé	non réservé	non réservé
NAME	non réservé	non réservé	non réservé	non réservé	non réservé
NAMES	non réservé	non réservé	non réservé	réservé	réservé
NAMESPACE		non réservé			
NATIONAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
NATURAL	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
NCHAR	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
NCLOB		réservé	réservé	réservé	
NESTING		non réservé	non réservé		
NEW		réservé	réservé	réservé	
NEXT	non réservé	non réservé	non réservé	réservé	réservé
NFC		non réservé			
NFD		non réservé			
NFKC		non réservé			
NFKD		non réservé			
NIL		non réservé			
NO	non réservé	réservé	réservé	réservé	réservé
NOCREATEDB	non réservé				
NOCREATEROLE	non réservé				
NOCREATEUSER	non réservé				
NOINHERIT	non réservé				
NOLOGIN	non réservé				
NONE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	
NORMALIZE		réservé	réservé		
NORMALIZED		non réservé	non réservé		
NOSUPERUSER	non réservé				
NOT	réservé	réservé	réservé	réservé	réservé
NOTHING	non réservé				
NOTIFY	non réservé				
NOTNULL	réservé (peut être une fonction ou un type)				
NOWAIT	non réservé				
NTH_VALUE		réservé			

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
NTILE		réservé			
NULL	réservé	réservé	réservé	réservé	réservé
NULLABLE		non réservé	non réservé	non réservé	non réservé
NULLIF	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
NULLS	non réservé	non réservé	non réservé		
NUMBER		non réservé	non réservé	non réservé	non réservé
NUMERIC	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
OBJECT	non réservé	non réservé	non réservé	réservé	
OCCURRENCES_REGEX		réservé			
OCTETS		non réservé	non réservé		
OCTET_LENGTH		réservé	réservé	non réservé	réservé
OF	non réservé	réservé	réservé	réservé	réservé
OFF	non réservé	non réservé	non réservé	réservé	
OFFSET	réservé	réservé			
OIDS	non réservé				
OLD		réservé	réservé	réservé	
ON	réservé	réservé	réservé	réservé	réservé
ONLY	réservé	réservé	réservé	réservé	réservé
OPEN		réservé	réservé	réservé	réservé
OPERATION				réservé	
OPERATOR	non réservé				
OPTION	non réservé	non réservé	non réservé	réservé	réservé
OPTIONS	non réservé	non réservé	non réservé	non réservé	
OR	réservé	réservé	réservé	réservé	réservé
ORDER	réservé	réservé	réservé	réservé	réservé
ORDERING		non réservé	non réservé		
ORDINALITY		non réservé	non réservé	réservé	
OTHERS		non réservé	non réservé		
OUT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	
OUTER	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
OUTPUT		non réservé	non réservé	réservé	réservé
OVER	réservé (peut être une fonction ou un type)	réservé	réservé		
OVERLAPS	réservé (peut être une fonction ou un type)	réservé	réservé	non réservé	réservé
OVERLAY	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	
OVERRIDING		non réservé	non réservé	non réservé	
OWNED	non réservé				
OWNER	non réservé				

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
P		non réservé			
PAD		non réservé	non réservé	réservé	réservé
PARAMETER		réservé	réservé	réservé	
PARAMETERS				réservé	
PARAMETER_MODE		non réservé	non réservé	non réservé	
PARAMETER_NAME		non réservé	non réservé	non réservé	
PARAMETER_ORDINAL_POSITION		non réservé	non réservé	non réservé	
PARAMETER_SPECIFIC_CATALOG		non réservé	non réservé	non réservé	
PARAMETER_SPECIFIC_NAME		non réservé	non réservé	non réservé	
PARAMETER_SPECIFIC_SCHEMA		non réservé	non réservé	non réservé	
PARSER	non réservé				
PARTIAL	non réservé	non réservé	non réservé	réservé	réservé
PARTITION	non réservé	réservé	réservé		
PASCAL		non réservé	non réservé	non réservé	non réservé
PASSING		non réservé			
PASSTHROUGH		non réservé	non réservé		
PASSWORD	non réservé				
PATH		non réservé	non réservé	réservé	
PERCENTILE_CONT		réservé	réservé		
PERCENTILE_DISC		réservé	réservé		
PERCENT_RANK		réservé	réservé		
PERMISSION		non réservé	non réservé		
PLACING	réservé	non réservé	non réservé		
PLANS	non réservé				
PLI		non réservé	non réservé	non réservé	non réservé
POSITION	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
POSITION_REGEX		réservé			
POSTFIX				réservé	
POWER		réservé	réservé		
PRECEDING	non réservé	non réservé	non réservé		
PRECISION	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
PREFIX				réservé	
PREORDER				réservé	
PREPARE	non réservé	réservé	réservé	réservé	réservé
PREPARED	non réservé				
PRESERVE	non réservé	non réservé	non réservé	réservé	réservé
PRIMARY	réservé	réservé	réservé	réservé	réservé
PRIOR	non réservé	non réservé	non réservé	réservé	réservé
PRIVILEGES	non réservé	non réservé	non réservé	réservé	réservé
PROCEDURAL	non réservé				
PROCEDURE	non réservé	réservé	réservé	réservé	réservé
PUBLIC		non réservé	non réservé	réservé	réservé

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
QUOTE	non réservé				
RANGE	non réservé	réservé	réservé		
RANK		réservé	réservé		
READ	non réservé	non réservé	non réservé	réservé	réservé
READS		réservé	réservé	réservé	
REAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
REASSIGN	non réservé				
RECHECK	non réservé				
RECOVERY		non réservé	non réservé		
RECURSIVE	non réservé	réservé	réservé	réservé	
REF		réservé	réservé	réservé	
REFERENCES	réservé	réservé	réservé	réservé	réservé
REFERENCING		réservé	réservé	réservé	
REGR_AVGX		réservé	réservé		
REGR_AVGY		réservé	réservé		
REGR_COUNT		réservé	réservé		
REGR_INTERCEPT		réservé	réservé		
REGR_R2		réservé	réservé		
REGR_SLOPE		réservé	réservé		
REGR_SXX		réservé	réservé		
REGR_SXY		réservé	réservé		
REGR_SYY		réservé	réservé		
REINDEX	non réservé				
RELATIVE	non réservé	non réservé	non réservé	réservé	réservé
RELEASE	non réservé	réservé	réservé		
RENAME	non réservé				
REPEATABLE	non réservé	non réservé	non réservé	non réservé	non réservé
REPLACE	non réservé				
REPLICA	non réservé				
REQUIRING		non réservé	non réservé		
RESET	non réservé				
RESPECT		non réservé			
RESTART	non réservé	non réservé	non réservé		
RESTORE		non réservé	non réservé		
RESTRICT	non réservé	non réservé	non réservé	réservé	réservé
RESULT		réservé	réservé	réservé	
RETURN		réservé	réservé	réservé	
RETURNED_CARDINALITY		non réservé	non réservé		
RETURNED_LENGTH		non réservé	non réservé	non réservé	non réservé
RETURNED_OCTET_LENGTH		non réservé	non réservé	non réservé	non réservé
RETURNED_SQLSTATE		non réservé	non réservé	non réservé	non réservé
RETURNING	réservé	non réservé			
RETURNS	non réservé	réservé	réservé	réservé	

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
REVOKE	non réservé	réservé	réservé	réservé	réservé
RIGHT	réservé (peut être une fonction ou un type)	réservé	réservé	réservé	réservé
ROLE	non réservé	non réservé	non réservé	réservé	
ROLLBACK	non réservé	réservé	réservé	réservé	réservé
ROLLUP		réservé	réservé	réservé	
ROUTINE		non réservé	non réservé	réservé	
ROUTINE_CATALOG		non réservé	non réservé	non réservé	
ROUTINE_NAME		non réservé	non réservé	non réservé	
ROUTINE_SCHEMA		non réservé	non réservé	non réservé	
ROW	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	
ROWS	non réservé	réservé	réservé	réservé	réservé
ROW_COUNT		non réservé	non réservé	non réservé	non réservé
ROW_NUMBER		réservé	réservé		
RULE	non réservé				
SAVEPOINT	non réservé	réservé	réservé	réservé	
SCALE		non réservé	non réservé	non réservé	non réservé
SCHEMA	non réservé	non réservé	non réservé	réservé	réservé
SCHEMA_NAME		non réservé	non réservé	non réservé	non réservé
SCOPE		réservé	réservé	réservé	
SCOPE_CATALOG		non réservé	non réservé		
SCOPE_NAME		non réservé	non réservé		
SCOPE_SCHEMA		non réservé	non réservé		
SCROLL	non réservé	réservé	réservé	réservé	réservé
SEARCH	non réservé	réservé	réservé	réservé	
SECOND	non réservé	réservé	réservé	réservé	réservé
SECTION		non réservé	non réservé	réservé	réservé
SECURITY	non réservé	non réservé	non réservé	non réservé	
SELECT	réservé	réservé	réservé	réservé	réservé
SELECTIVE		non réservé	non réservé		
SELF		non réservé	non réservé	non réservé	
SENSITIVE		réservé	réservé	non réservé	
SEQUENCE	non réservé	non réservé	non réservé	réservé	
SEQUENCES	non réservé				
SERIALIZABLE	non réservé	non réservé	non réservé	non réservé	non réservé
SERVER	non réservé	non réservé	non réservé		
SERVER_NAME		non réservé	non réservé	non réservé	non réservé
SESSION	non réservé	non réservé	non réservé	réservé	réservé
SESSION_USER	réservé	réservé	réservé	réservé	réservé
SET	non réservé	réservé	réservé	réservé	réservé
SETOF	non réservé (ne peut pas être une fonction ou un type)				
SETS		non réservé	non réservé	réservé	

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
SHARE	non réservé				
SHOW	non réservé				
SIMILAR	réservé (peut être une fonction ou un type)	réservé	réservé	non réservé	
SIMPLE	non réservé	non réservé	non réservé	non réservé	
SIZE		non réservé	non réservé	réservé	réservé
SMALLINT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
SOME	réservé	réservé	réservé	réservé	réservé
SOURCE		non réservé	non réservé	non réservé	
SPACE		non réservé	non réservé	réservé	réservé
SPECIFIC		réservé	réservé	réservé	
SPECIFICTYPE		réservé	réservé	réservé	
SPECIFIC_NAME		non réservé	non réservé	non réservé	
SQL		réservé	réservé	réservé	réservé
SQLCODE					réservé
SQLERROR					réservé
SQLEXCEPTION		réservé	réservé	réservé	
SQLSTATE		réservé	réservé	réservé	réservé
SQLWARNING		réservé	réservé	réservé	
SQRT		réservé	réservé		
STABLE	non réservé				
STANDALONE	non réservé	non réservé	non réservé		
START	non réservé	réservé	réservé	réservé	
STATE		non réservé	non réservé	réservé	
STATEMENT	non réservé	non réservé	non réservé	réservé	
STATIC		réservé	réservé	réservé	
STATISTICS	non réservé				
STDDEV_POP		réservé	réservé		
STDDEV_SAMP		réservé	réservé		
STDIN	non réservé				
STDOUT	non réservé				
STORAGE	non réservé				
STRICT	non réservé				
STRIP	non réservé	non réservé	non réservé		
STRUCTURE		non réservé	non réservé	réservé	
STYLE		non réservé	non réservé	non réservé	
SUBCLASS_ORIGIN		non réservé	non réservé	non réservé	non réservé
SUBLIST				non réservé	
SUBMULTISET		réservé	réservé		
SUBSTRING	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
SUBSTRING_REGEX		réservé			
SUM		réservé	réservé	non réservé	réservé

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
SUPERUSER	non réservé				
SYMMETRIC	réservé	réservé	réservé	non réservé	
SYSID	non réservé				
SYSTEM	non réservé	réservé	réservé	non réservé	
SYSTEM_USER		réservé	réservé	réservé	réservé
T		non réservé			
TABLE	réservé	réservé	réservé	réservé	réservé
TABLES	non réservé				
TABLESAMPLE		réservé	réservé		
TABLESPACE	non réservé				
TABLE_NAME		non réservé	non réservé	non réservé	non réservé
TEMP	non réservé				
TEMPLATE	non réservé				
TEMPORARY	non réservé	non réservé	non réservé	réservé	réservé
TERMINATE				réservé	
TEXT	non réservé				
THAN				réservé	
THEN	réservé	réservé	réservé	réservé	réservé
TIES		non réservé	non réservé		
TIME	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
TIMESTAMP	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
TIMEZONE_HOUR		réservé	réservé	réservé	réservé
TIMEZONE_MINUTE		réservé	réservé	réservé	réservé
TO	réservé	réservé	réservé	réservé	réservé
TOKEN		non réservé	non réservé		
TOP_LEVEL_COUNT		non réservé	non réservé		
TRAILING	réservé	réservé	réservé	réservé	réservé
TRANSACTION	non réservé	non réservé	non réservé	réservé	réservé
TRANSACTIONS_COMMITTED		non réservé	non réservé	non réservé	
TRANSACTIONS_ROLLED_BACK		non réservé	non réservé	non réservé	
TRANSACTION_ACTIVE		non réservé	non réservé	non réservé	
TRANSFORM		non réservé	non réservé	non réservé	
TRANSFORMS		non réservé	non réservé	non réservé	
TRANSLATE		réservé	réservé	non réservé	réservé
TRANSLATE_REGEX		réservé			
TRANSLATION		réservé	réservé	réservé	réservé
TREAT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	
TRIGGER	non réservé	réservé	réservé	réservé	
TRIGGER_CATALOG		non réservé	non réservé	non réservé	
TRIGGER_NAME		non réservé	non réservé	non réservé	

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
TRIGGER_SCHEMA		non réservé	non réservé	non réservé	
TRIM	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	non réservé	réservé
TRIM_ARRAY		réservé			
TRUE	réservé	réservé	réservé	réservé	réservé
TRUNCATE	non réservé	réservé			
TRUSTED	non réservé				
TYPE	non réservé	non réservé	non réservé	non réservé	non réservé
UESCAPE		réservé	réservé		
UNBOUNDED	non réservé	non réservé	non réservé		
UNCOMMITTED	non réservé	non réservé	non réservé	non réservé	non réservé
UNDER		non réservé	non réservé	réservé	
UNENCRYPTED	non réservé				
UNION	réservé	réservé	réservé	réservé	réservé
UNIQUE	réservé	réservé	réservé	réservé	réservé
UNKNOWN	non réservé	réservé	réservé	réservé	réservé
UNLINK		non réservé	non réservé		
UNLISTEN	non réservé				
UNNAMED		non réservé	non réservé	non réservé	non réservé
UNNEST		réservé	réservé	réservé	
UNTIL	non réservé				
UNTYPED		non réservé			
UPDATE	non réservé	réservé	réservé	réservé	réservé
UPPER		réservé	réservé	non réservé	réservé
URI		non réservé			
USAGE		non réservé	non réservé	réservé	réservé
USER	réservé	réservé	réservé	réservé	réservé
USER_DEFINED_TYPE_CATALOG		non réservé	non réservé	non réservé	
USER_DEFINED_TYPE_CODE		non réservé	non réservé		
USER_DEFINED_TYPE_NAME		non réservé	non réservé	non réservé	
USER_DEFINED_TYPE_SCHEMA		non réservé	non réservé	non réservé	
USING	réservé	réservé	réservé	réservé	réservé
VACUUM	non réservé				
VALID	non réservé	non réservé			
VALIDATOR	non réservé				
VALUE	non réservé	réservé	réservé	réservé	réservé
VALUES	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
VARBINARY		réservé			
VARCHAR	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé	réservé
VARIABLE				réservé	
VARIADIC	réservé				

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
VARYING	non réservé	réservé	réservé	réservé	réservé
VAR_POP		réservé	réservé		
VAR_SAMP		réservé	réservé		
VERBOSE	réservé (peut être une fonction ou un type)				
VERSION	non réservé	non réservé	non réservé		
VIEW	non réservé	non réservé	non réservé	réservé	réservé
VOLATILE	non réservé				
WHEN	réservé	réservé	réservé	réservé	réservé
WHENEVER		réservé	réservé	réservé	réservé
WHERE	réservé	réservé	réservé	réservé	réservé
WHITESPACE	non réservé	non réservé	non réservé		
WIDTH_BUCKET		réservé	réservé		
WINDOW	réservé	réservé	réservé		
WITH	réservé	réservé	réservé	réservé	réservé
WITHIN		réservé	réservé		
WITHOUT	non réservé	réservé	réservé	réservé	
WORK	non réservé	non réservé	non réservé	réservé	réservé
WRAPPER	non réservé	non réservé	non réservé		
WRITE	non réservé	non réservé	non réservé	réservé	réservé
XML	non réservé	réservé	réservé		
XMLAGG		réservé	réservé		
XMLATTRIBUTES	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
XMLBINARY		réservé	réservé		
XMLCAST		réservé			
XMLCOMMENT		réservé	réservé		
XMLCONCAT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
XMLDECLARATION		non réservé			
XMLDOCUMENT		réservé			
XMLELEMENT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
XMLEXISTS		réservé			
XMLFOREST	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
XMLITERATE		réservé			
XMLNAMESPACES		réservé	réservé		
XMLPARSE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
XMLPI	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		

Mot-clé	PostgreSQL™	SQL:2008	SQL:2003	SQL:1999	SQL-92
XMLQUERY		réservé			
XMLROOT	non réservé (ne peut pas être une fonction ou un type)		réservé		
XMLSCHEMA		non réservé			
XMLSERIALIZE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé		
XMLTABLE		réservé			
XMLTEXT		réservé			
XMLVALIDATE		réservé			
YEAR	non réservé	réservé	réservé	réservé	réservé
YES	non réservé	non réservé			
ZONE	non réservé	non réservé	non réservé	réservé	réservé

Annexe D. Conformité SQL

Cette section explique dans quelle mesure PostgreSQL™ se conforme à la norme SQL en vigueur. Les informations qui suivent ne représentent pas une liste exhaustive de conformance, mais présentent les thèmes principaux utilement et raisonnablement détaillés.

Le nom complet du standard SQL est ISO/IEC 9075 « Database Language SQL ». Le standard est modifié de temps en temps. La mise à jour la plus récente apparaît en 2008. La version 2008 est référencée ISO/IEC 9075:2008, ou plus simplement SQL:2008. Les versions antérieures étaient SQL:2003, SQL:1999 et SQL-92. Chaque version remplace la précédente. Il n'y a donc aucun mérite à revendiquer une compatibilité avec une version antérieure du standard.

Le développement de PostgreSQL™ respecte le standard en vigueur, tant que celui-ci ne s'oppose pas aux fonctionnalités traditionnelles ou au bon sens. Le projet PostgreSQL n'est pas représenté au sein du groupe de travail ISO/IEC 9075 lors de la préparation des versions du standard SQL. Malgré cela, un grand nombre des fonctionnalités requises par le standard SQL sont déjà supportées. Parfois avec une syntaxe ou un fonctionnement légèrement différents. Une meilleure compatibilité est attendue pour les prochaines versions.

SQL-92 définit trois niveaux de conformité : basique (*Entry*), intermédiaire (*Intermediate*) et complète (*Full*). La majorité des systèmes de gestion de bases de données se prétendaient compatibles au standard SQL dès lors qu'ils se conformaient au niveau Entry ; l'ensemble des fonctionnalités des niveaux Intermediate et Full étaient, soit trop volumineux, soit en conflit avec les fonctionnalités implantées.

À partir de SQL99, le standard SQL définit un vaste ensemble de fonctionnalités individuelles à la place des trois niveaux de fonctionnalités définis dans SQL-92. Une grande partie représente les fonctionnalités « centrales » que chaque implantation conforme de SQL doit fournir. Les fonctionnalités restantes sont purement optionnelles. Certaines sont regroupées au sein de « paquetages » auxquels une implantation peut se déclarer conforme. On parle alors de conformité à un groupe de fonctionnalités.

Les standards SQL:2008 et SQL:2003 sont également divisé en parties. Chacune est connue par un pseudonyme. Leur numérotation n'est pas consécutive :

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

PostgreSQL™ couvre les parties 1, 2, 9, 11 et 14. La partie 3 est couverte par l'interface ODBC, et la partie 13 est couverte par le plugin PL/Java, mais une conformance exacte n'est pas actuellement vérifiée par ses composants. Il n'y a pas actuellement d'implantations des parties 4 et 10 pour PostgreSQL™.

PostgreSQL supporte la plupart des fonctionnalités majeures de SQL:2008. Sur les 179 fonctionnalités requises pour une conformité « centrale » complète (*full Core conformance*), PostgreSQL se conforme à plus de 160. De plus, il existe une longue liste de fonctionnalités optionelles supportées. À la date de rédaction de ce document, aucune version de quelque système de gestion de bases de données que ce soit n'affiche une totale conformité au cœur de SQL:2008.

Les deux sections qui suivent présentent la liste des fonctionnalités supportées par PostgreSQL™ et celle des fonctionnalités définies dans SQL:2008 qui ne sont pas encore prises en compte. Ces deux listes sont approximatives : certains détails d'une fonctionnalité présentée comme supportée peuvent ne pas être conformes, alors que de grandes parties d'une fonctionnalité non supportée peuvent être implantées. La documentation principale fournit les informations précises sur ce qui est, ou non, supporté.

Note

Les codes de fonctionnalité contenant un tiret sont des sous-fonctionnalités. Si une sous-fonctionnalité n'est pas supportée, la fonctionnalité elle-même sera déclarée non supportée, alors même que d'autres de ses sous-fonctionnalités le sont.

D.1. Fonctionnalités supportées

Identifiant	Paquetage	Description	Commentaire
B012		Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	

Identifiant	Paquetage	Description	Commentaire
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-05	Core	UPDATE privilege at the column level	
E081-06	Core	REFERENCES privilege at the table level	
E081-07	Core	REFERENCES privilege at the column level	
E081-08	Core	WITH GRANT OPTION	
E081-10	Core	EXECUTE privilege	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121	Core	Basic cursor support	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	

Identifiant	Paquetage	Description	Commentaire
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	

Identifiant	Paquetage	Description	Commentaire
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced date-time facilities	Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	

Identifiant	Paquetage	Description	Commentaire
F200		TRUNCATE TABLE statement	
F201	Core	CAST function	
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F271		Compound character literals	
F281		LIKE enhancements	
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F382		Alter column data type	
F391		Long identifiers	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		Named column joins for LOBs, arrays, and multisets	
F411	Enhanced date-time facilities	Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	

Identifiant	Paquetage	Description	Commentaire
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F442		Mixed column references in set functions	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced documentation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_INFO view	
F502-03		SQL_PACKAGES view	
F531		Temporary tables	
F555	Enhanced date-time facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591		Derived tables	
F611		Indicator data types	
F651		Catalog name qualifiers	
F661		Simple tables	
F672		Retrospective check constraints	
F701	Enhanced integrity management	Referential update actions	
F711		ALTER domain	
F731		INSERT column privileges	
F761		Session management	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Connection management	
F781		Self-referencing operations	
F791		Insensitive cursors	
F801		Full set function	
F850		Top-level <order by clause> in <query expression>	
F851		<order by clause> in subqueries	
F852		Top-level <order by clause> in views	
F855		Nested <order by clause> in <query expression>	

Identifiant	Paquetage	Description	Commentaire
F856		Nested <fetch first clause> in <query expression>	
F857		Top-level <fetch first clause> in <query expression>	
F858		<fetch first clause> in subqueries	
F859		Top-level <fetch first clause> in views	
F860		<fetch first row count> in <fetch first clause>	
F861		Top-level <result offset clause> in <query expression>	
F862		<result offset clause> in subqueries	
F863		Nested <result offset clause> in <query expression>	
F864		Top-level <result offset clause> in views	
F865		<offset row count> in <result offset clause>	
S071	Enhanced object support	SQL paths in function and type name resolution	
S092		Arrays of user-defined types	
S095		Array constructors by query	
S096		Optional array bounds	
S098		ARRAY_AGG	
S111	Enhanced object support	ONLY in query expressions	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S211	Enhanced object support	User-defined cast functions	
T031		BOOLEAN data type	
T071		BIGINT data type	
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T152		DISTINCT predicate with negation	
T171		LIKE clause in table definition	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constraints	
T211-01	Active database, Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02	Active database, Enhanced integrity management	BEFORE triggers	
T211-03	Active database, Enhanced integri-	AFTER triggers	

Identifiant	Paquetage	Description	Commentaire
	ty management		
T211-04	Active database, Enhanced integrity management	FOR EACH ROW triggers	
T211-05	Active database, Enhanced integrity management	Ability to specify a search condition that must be true before the trigger is invoked	
T211-07	Active database, Enhanced integrity management	TRIGGER privilege	
T212	Enhanced integrity management	Enhanced trigger capability	
T231		Sensitive cursors	
T241		START TRANSACTION statement	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-03	Core	Function invocation	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T322	PSM	Overloading of SQL-invoked functions and procedures	
T323		Explicit security for external routines	
T351		Bracketed SQL comments (/*...*/ comments)	
T441		ABS and MOD functions	
T461		Symmetric BETWEEN predicate	
T501		Enhanced EXISTS predicate	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	
T614		NTILE function	
T615		LEAD and LAG functions	
T617		FIRST_VALUE and LAST_VALUE function	
T621		Enhanced numeric functions	
T631	Core	IN predicate with one list element	
T651		SQL-schema statements in SQL routines	
T655		Cyclically dependent routines	
X010		XML type	
X011		Arrays of XML type	
X016		Persistent XML values	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: ORDER BY option	
X036		XMLComment	

Identifiant	Paquetage	Description	Commentaire
X037		XMLPI	
X040		Basic table mapping	
X041		Basic table mapping: nulls absent	
X042		Basic table mapping: null as nil	
X043		Basic table mapping: table as forest	
X044		Basic table mapping: table as element	
X045		Basic table mapping: with target namespace	
X046		Basic table mapping: data mapping	
X047		Basic table mapping: metadata mapping	
X048		Basic table mapping: base64 encoding of binary strings	
X049		Basic table mapping: hex encoding of binary strings	
X050		Advanced table mapping	
X051		Advanced table mapping: nulls absent	
X052		Advanced table mapping: null as nil	
X053		Advanced table mapping: table as forest	
X054		Advanced table mapping: table as element	
X055		Advanced table mapping: target namespace	
X056		Advanced table mapping: data mapping	
X057		Advanced table mapping: metadata mapping	
X058		Advanced table mapping: base64 encoding of binary strings	
X059		Advanced table mapping: hex encoding of binary strings	
X060		XMLParse: Character string input and CONTENT option	
X061		XMLParse: Character string input and DOCUMENT option	
X070		XMLSerialize: Character string serialization and CONTENT option	
X071		XMLSerialize: Character string serialization and DOCUMENT option	
X072		XMLSerialize: Character string serialization	
X090		XML document predicate	
X120		XML parameters in SQL routines	
X121		XML parameters in external routines	

D.2. Fonctionnalités non supportées

Les fonctionnalités suivantes définies dans SQL:2008 ne sont pas implantées dans cette version de PostgreSQL™. Dans certains cas, des fonctionnalités similaires sont disponibles.

Identifiant	Paquetage	Description	Commentaire
B011		Embedded Ada	
B013		Embedded COBOL	
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	
B017		Embedded PL/I	

Identifiant	Paquetage	Description	Commentaire
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B035		Non-extended descriptor names	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	
B116		Module language Pascal	
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B128		Routine language SQL	
E081	Core	Basic Privileges	
E081-09	Core	USAGE privilege	
E153	Core	Updatable queries with subqueries	
E182	Core	Module language	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F122		Enhanced diagnostics management	
F123		All diagnostics	
F181	Core	Multiple module support	
F202		TRUNCATE TABLE: identity column restart option	
F262		Extended CASE expression	
F263		Comma-separated predicates in simple CASE expression	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F311	Core	Schema definition statement	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F312		MERGE statement	
F313		Enhanced MERGE statement	
F341		Usage tables	

Identifiant	Paquetage	Description	Commentaire
F394		Optional normal form specification	
F403		Partitioned joined tables	
F451		Character set definition	
F461		Named character sets	
F521	Enhanced integrity management	Assertions	
F641		Row and table constructors	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F690		Collation support	
F692		Enhanced collation support	
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign and unique keys only
F741		Referential MATCH types	no partial match yet
F751		View CHECK enhancements	
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
F841		LIKE_REGEX predicate	
F842		OCCURENCES_REGEX function	
F843		POSITION_REGEX function	
F844		SUBSTRING_REGEX function	
F845		TRANSLATE_REGEX function	
F846		Octet support in regular expression operators	
F847		Nonconstant regular expressions	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support	Basic structured types	
S024	Enhanced object support	Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	

Identifiant	Paquetage	Description	Commentaire
S081	Enhanced object support	Subtables	
S091		Basic array support	partially supported
S091-01		Arrays of built-in data types	
S091-02		Arrays of distinct types	
S091-03		Array expressions	
S094		Arrays of reference types	
S097		Array element assignment	
S151	Basic object support	Type predicate	
S161	Enhanced object support	Subtype treatment	
S162		Subtype treatment for references	
S202		SQL-invoked routines on multisets	
S231	Enhanced object support	Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
S301		Enhanced UNNEST	
S401		Distinct types based on array types	
S402		Distinct types based on distinct types	
S403		MAX_CARDINALITY	
S404		TRIM_ARRAY	
T011		Timestamp in Information Schema	
T021		BINARY and VARBINARY data types	
T022		Advanced support for BINARY and VARBINARY data types	
T023		Compound binary literal	
T024		Spaces in binary literals	
T041	Basic object support	Basic LOB data type support	
T041-01	Basic object support	BLOB data type	
T041-02	Basic object support	CLOB data type	
T041-03	Basic object sup-	POSITION, LENGTH, LOWER, TRIM, UPPER, and	

Identifiant	Paquetage	Description	Commentaire
	port	SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T043		Multiplier T	
T044		Multiplier P	
T051		Row types	
T052		MAX and MIN for row types	
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T101		Enhanced nullability determiniation	
T111		Updatable joins, unions, and columns	
T174		Identity columns	
T175		Generated columns	
T176		Sequence generator support	
T177		Sequence generator support: simple restart option	
T178		Identity columns: simple restart option	
T211	Active database, Enhanced integrity management	Basic trigger capability	
T211-06	Active database, Enhanced integrity management	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Active database, Enhanced integrity management	Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T213		INSTEAD OF triggers	
T251		SET TRANSACTION statement: LOCAL option	
T261		Chained transactions	
T272		Enhanced savepoint management	
T285		Enhanced derived column names	
T301		Functional dependencies	
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	
T324		Explicit security for SQL routines	
T325		Qualified SQL parameter references	
T326		Table functions	
T331		Basic roles	
T332		Extended roles	
T431	OLAP	Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	
T433		Multiargument GROUPING function	

Identifiant	Paquetage	Description	Commentaire
T434		GROUP BY DISTINCT	
T471		Result sets return value	
T491		LATERAL derived table	
T511		Transaction counts	
T541		Updatable table references	
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T611	OLAP	Elementary OLAP operations	most forms supported
T612		Advanced OLAP operations	some forms supported
T613		Sampling	
T616		Null treatment option for LEAD and LAG functions	
T618		NTH_VALUE function	function exists, but some options missing
T641		Multiple column assignment	only some syntax variants supported
T652		SQL-dynamic statements in SQL routines	
T653		SQL-schema statements in external routines	
T654		SQL-dynamic statements in external routines	
M001		Datalinks	
M002		Datalinks via SQL/CLI	
M003		Datalinks via Embedded SQL	
M004		Foreign data support	
M005		Foreign schema support	
M006		GetSQLString routine	
M007		TransmitRequest	
M009		GetOpts and GetStatistics routines	
M010		Foreign data wrapper support	
M011		Datalinks via Ada	
M012		Datalinks via C	
M013		Datalinks via COBOL	
M014		Datalinks via Fortran	
M015		Datalinks via M	
M016		Datalinks via Pascal	
M017		Datalinks via PL/I	
M018		Foreign data wrapper interface routines in Ada	
M019		Foreign data wrapper interface routines in C	
M020		Foreign data wrapper interface routines in COBOL	
M021		Foreign data wrapper interface routines in Fortran	
M022		Foreign data wrapper interface routines in MUMPS	
M023		Foreign data wrapper interface routines in Pascal	
M024		Foreign data wrapper interface routines in PL/I	
M030		SQL-server foreign data support	

Identifiant	Paquetage	Description	Commentaire
M031		Foreign data wrapper general routines	
X012		Multisets of XML type	
X013		Distinct types of XML type	
X014		Attributes of XML type	
X015		Fields of XML type	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse: BLOB input and CONTENT option	
X066		XMLParse: BLOB input and DOCUMENT option	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOB serialization and CONTENT option	
X074		XMLSerialize: BLOB serialization and DOCUMENT option	
X075		XMLSerialize: BLOB serialization	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: explicit ENCODING option	
X078		XMLSerialize: explicit XML declaration	
X080		Namespaces in XML publishing	
X081		Query-level XML namespace declarations	
X082		XML namespace declarations in DML	
X083		XML namespace declarations in DDL	
X084		XML namespace declarations in compound statements	
X085		Predefined namespace prefixes	
X086		XML namespace declarations in XMLTable	
X091		XML content predicate	
X096		XMLExists	
X100		Host language support for XML: CONTENT option	
X101		Host language support for XML: DOCUMENT option	
X110		Host language support for XML: VARCHAR mapping	
X111		Host language support for XML: CLOB mapping	
X112		Host language support for XML: BLOB mapping	
X113		Host language support for XML: STRIP WHITESPACE option	
X114		Host language support for XML: PRESERVE WHITESPACE option	
X131		Query-level XMLBINARY clause	
X132		XMLBINARY clause in DML	
X133		XMLBINARY clause in DDL	
X134		XMLBINARY clause in compound statements	
X135		XMLBINARY clause in subqueries	
X141		IS VALID predicate: data-driven case	
X142		IS VALID predicate: ACCORDING TO clause	

Identifiant	Paquetage	Description	Commentaire
X143		IS VALID predicate: ELEMENT clause	
X144		IS VALID predicate: schema location	
X145		IS VALID predicate outside check constraints	
X151		IS VALID predicate with DOCUMENT option	
X152		IS VALID predicate with CONTENT option	
X153		IS VALID predicate with SEQUENCE option	
X155		IS VALID predicate: NAMESPACE without ELEMENT clause	
X157		IS VALID predicate: NO NAMESPACE with ELEMENT clause	
X160		Basic Information Schema for registered XML Schemas	
X161		Advanced Information Schema for registered XML Schemas	
X170		XML null handling options	
X171		NIL ON NO CONTENT option	
X181		XML(DOCUMENT(UNTYPED)) type	
X182		XML(DOCUMENT(ANY)) type	
X190		XML(SEQUENCE) type	
X191		XML(DOCUMENT(XMLSCHEMA)) type	
X192		XML(CONTENT(XMLSCHEMA)) type	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: passing a context item	
X204		XMLQuery: initializing an XQuery variable	
X205		XMLQuery: EMPTY ON EMPTY option	
X206		XMLQuery: NULL ON EMPTY option	
X211		XML 1.1 support	
X221		XML passing mechanism BY VALUE	
X222		XML passing mechanism BY REF	
X231		XML(CONTENT(UNTYPED)) type	
X232		XML(CONTENT(ANY)) type	
X241		RETURNING CONTENT in XML publishing	
X242		RETURNING SEQUENCE in XML publishing	
X251		Persistent XML values of XML(DOCUMENT(UNTYPED)) type	
X252		Persistent XML values of XML(DOCUMENT(ANY)) type	
X253		Persistent XML values of XML(CONTENT(UNTYPED)) type	
X254		Persistent XML values of XML(CONTENT(ANY)) type	
X255		Persistent XML values of XML(SEQUENCE) type	
X256		Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	
X257		Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	

Identifiant	Paquetage	Description	Commentaire
X260		XML type: ELEMENT clause	
X261		XML type: NAMESPACE without ELEMENT clause	
X263		XML type: NO NAMESPACE with ELEMENT clause	
X264		XML type: schema location	
X271		XMLValidate: data-driven case	
X272		XMLValidate: ACCORDING TO clause	
X273		XMLValidate: ELEMENT clause	
X274		XMLValidate: schema location	
X281		XMLValidate: with DOCUMENT option	
X282		XMLValidate with CONTENT option	
X283		XMLValidate with SEQUENCE option	
X284		XMLValidate NAMESPACE without ELEMENT clause	
X286		XMLValidate: NO NAMESPACE with ELEMENT clause	
X300		XMLTable	
X301		XMLTable: derived column list option	
X302		XMLTable: ordinality column option	
X303		XMLTable: column default option	
X304		XMLTable: passing a context item	
X305		XMLTable: initializing an XQuery variable	
X400		Name and identifier mapping	

Annexe E. Notes de version

Les notes de version contiennent les modifications significatives apparaissant dans chaque version de PostgreSQL™. Elles contiennent aussi les fonctionnalités majeures et les problèmes de migration éventuels. Les notes de version ne contiennent pas les modifications qui n'affectent que peu d'utilisateurs ainsi que les modifications internes, non visibles pour les utilisateurs. Par exemple, l'optimiseur est amélioré dans pratiquement chaque version, mais les améliorations ne sont visibles par les utilisateurs que par la plus grande rapidité des requêtes.

Une liste complète de modifications est récupérable pour chaque version en lisant les validations Git. La *liste de diffusion* `pgsql-committers` enregistre en plus toutes les modifications du code source. Il existe aussi une *interface web* montrant les modifications sur chaque fichier.

Le nom apparaissant auprès de chaque élément précise le développeur principal de cet élément. Bien sûr, toutes les modifications impliquent des discussions de la communauté et une relecture des correctifs, donc chaque élément est vraiment un travail de la communauté.

E.1. Release 9.0.18

Release Date

2014-07-24

This release contains a variety of fixes from 9.0.17. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.1.1. Migration to Version 9.0.18

A dump/restore is not required for those running 9.0.X.

However, this release corrects an index corruption problem in some GiST indexes. See the first changelog entry below to find out whether your installation has been affected and what steps you should take if so.

Also, if you are upgrading from a version earlier than 9.0.15, see Section E.4, « Release 9.0.15 ».

E.1.2. Changes

- Correctly initialize padding bytes in `contrib/btree_gist` indexes on bit columns (Heikki Linnakangas)

  This error could result in incorrect query results due to values that should compare equal not being seen as equal. Users with GiST indexes on bit or bit varying columns should **REINDEX** those indexes after installing this update.

- Protect against torn pages when deleting GIN list pages (Heikki Linnakangas)

  This fix prevents possible index corruption if a system crash occurs while the page update is being written to disk.

- Don't clear the right-link of a GiST index page while replaying updates from WAL (Heikki Linnakangas)

  This error could lead to transiently wrong answers from GiST index scans performed in Hot Standby.

- Fix possibly-incorrect cache invalidation during nested calls to `ReceiveSharedInvalidMessages` (Andres Freund)

- Don't assume a subquery's output is unique if there's a set-returning function in its targetlist (David Rowley)

  This oversight could lead to misoptimization of constructs like `WHERE x IN (SELECT y, generate_series(1,10) FROM t GROUP BY y)`.

- Fix failure to detoast fields in composite elements of structured types (Tom Lane)

  This corrects cases where TOAST pointers could be copied into other tables without being dereferenced. If the original data is later deleted, it would lead to errors like « missing chunk number 0 for toast value ... » when the now-dangling pointer is used.

- Fix « record type has not been registered » failures with whole-row references to the output of Append plan nodes (Tom Lane)

- Fix possible crash when invoking a user-defined function while rewinding a cursor (Tom Lane)

- Fix query-lifespan memory leak while evaluating the arguments for a function in `FROM` (Tom Lane)

- Fix session-lifespan memory leaks in regular-expression processing (Tom Lane, Arthur O'Dwyer, Greg Stark)

- Fix data encoding error in `hungarian.stop` (Tom Lane)
- Fix liveness checks for rows that were inserted in the current transaction and then deleted by a now-rolled-back subtransaction (Andres Freund)

  This could cause problems (at least spurious warnings, and at worst an infinite loop) if **CREATE INDEX** or **CLUSTER** were done later in the same transaction.
- Clear pg_stat_activity.*`xact_start`* during **PREPARE TRANSACTION** (Andres Freund)

  After the **PREPARE**, the originating session is no longer in a transaction, so it should not continue to display a transaction start time.
- Fix **REASSIGN OWNED** to not fail for text search objects (Álvaro Herrera)
- Block signals during postmaster startup (Tom Lane)

  This ensures that the postmaster will properly clean up after itself if, for example, it receives `SIGINT` while still starting up.
- Secure Unix-domain sockets of temporary postmasters started during `make check` (Noah Misch)

  Any local user able to access the socket file could connect as the server's bootstrap superuser, then proceed to execute arbitrary code as the operating-system user running the test, as we previously noted in CVE-2014-0067. This change defends against that risk by placing the server's socket in a temporary, mode 0700 subdirectory of `/tmp`. The hazard remains however on platforms where Unix sockets are not supported, notably Windows, because then the temporary postmaster must accept local TCP connections.

  A useful side effect of this change is to simplify `make check` testing in builds that override `DEFAULT_PGSOCKET_DIR`. Popular non-default values like `/var/run/postgresql` are often not writable by the build user, requiring workarounds that will no longer be necessary.
- Fix tablespace creation WAL replay to work on Windows (MauMau)
- Fix detection of socket creation failures on Windows (Bruce Momjian)
- On Windows, allow new sessions to absorb values of PGC_BACKEND parameters (such as log_connections) from the configuration file (Amit Kapila)

  Previously, if such a parameter were changed in the file post-startup, the change would have no effect.
- Properly quote executable path names on Windows (Nikhil Deshpande)

  This oversight could cause initdb and pg_upgrade to fail on Windows, if the installation path contained both spaces and `@` signs.
- Fix linking of libpython on OS X (Tom Lane)

  The method we previously used can fail with the Python library supplied by Xcode 5.0 and later.
- Avoid buffer bloat in libpq when the server consistently sends data faster than the client can absorb it (Shin-ichi Morita, Tom Lane)

  libpq could be coerced into enlarging its input buffer until it runs out of memory (which would be reported misleadingly as « lost synchronization with server »). Under ordinary circumstances it's quite far-fetched that data could be continuously transmitted more quickly than the `recv()` loop can absorb it, but this has been observed when the client is artificially slowed by scheduler constraints.
- Ensure that LDAP lookup attempts in libpq time out as intended (Laurenz Albe)
- Fix ecpg to do the right thing when an array of char * is the target for a FETCH statement returning more than one row, as well as some other array-handling fixes (Ashutosh Bapat)
- Fix pg_restore's processing of old-style large object comments (Tom Lane)

  A direct-to-database restore from an archive file generated by a pre-9.0 version of pg_dump would usually fail if the archive contained more than a few comments for large objects.
- In `contrib/pgcrypto` functions, ensure sensitive information is cleared from stack variables before returning (Marko Kreen)
- In `contrib/uuid-ossp`, cache the state of the OSSP UUID library across calls (Tom Lane)

  This improves the efficiency of UUID generation and reduces the amount of entropy drawn from `/dev/urandom`, on platforms that have that.

- Update time zone data files to tzdata release 2014e for DST law changes in Crimea, Egypt, and Morocco.

E.2. Release 9.0.17

Release Date

2014-03-20

This release contains a variety of fixes from 9.0.16. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.2.1. Migration to Version 9.0.17

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.15, see Section E.4, « Release 9.0.15 ».

E.2.2. Changes

- Restore GIN metapages unconditionally to avoid torn-page risk (Heikki Linnakangas)

  Although this oversight could theoretically result in a corrupted index, it is unlikely to have caused any problems in practice, since the active part of a GIN metapage is smaller than a standard 512-byte disk sector.

- Avoid race condition in checking transaction commit status during receipt of a **NOTIFY** message (Marko Tiikkaja)

  This prevents a scenario wherein a sufficiently fast client might respond to a notification before database updates made by the notifier have become visible to the recipient.

- Allow regular-expression operators to be terminated early by query cancel requests (Tom Lane)

  This prevents scenarios wherein a pathological regular expression could lock up a server process uninterruptably for a long time.

- Remove incorrect code that tried to allow `OVERLAPS` with single-element row arguments (Joshua Yanovski)

  This code never worked correctly, and since the case is neither specified by the SQL standard nor documented, it seemed better to remove it than fix it.

- Avoid getting more than `AccessShareLock` when de-parsing a rule or view (Dean Rasheed)

  This oversight resulted in pg_dump unexpectedly acquiring `RowExclusiveLock` locks on tables mentioned as the targets of `INSERT`/`UPDATE`/`DELETE` commands in rules. While usually harmless, that could interfere with concurrent transactions that tried to acquire, for example, `ShareLock` on those tables.

- Improve performance of index endpoint probes during planning (Tom Lane)

  This change fixes a significant performance problem that occurred when there were many not-yet-committed rows at the end of the index, which is a common situation for indexes on sequentially-assigned values such as timestamps or sequence-generated identifiers.

- Fix test to see if hot standby connections can be allowed immediately after a crash (Heikki Linnakangas)

- Prevent interrupts while reporting non-`ERROR` messages (Tom Lane)

  This guards against rare server-process freezeups due to recursive entry to `syslog()`, and perhaps other related problems.

- Prevent intermittent « could not reserve shared memory region » failures on recent Windows versions (MauMau)

- Update time zone data files to tzdata release 2014a for DST law changes in Fiji and Turkey, plus historical changes in Israel and Ukraine.

E.3. Release 9.0.16

Release Date

2014-02-20

This release contains a variety of fixes from 9.0.15. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.3.1. Migration to Version 9.0.16

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.15, see Section E.4, « Release 9.0.15 ».

E.3.2. Changes

- Shore up `GRANT ... WITH ADMIN OPTION` restrictions (Noah Misch)

  Granting a role without `ADMIN OPTION` is supposed to prevent the grantee from adding or removing members from the granted role, but this restriction was easily bypassed by doing `SET ROLE` first. The security impact is mostly that a role member can revoke the access of others, contrary to the wishes of his grantor. Unapproved role member additions are a lesser concern, since an uncooperative role member could provide most of his rights to others anyway by creating views or `SECURITY DEFINER` functions. (CVE-2014-0060)

- Prevent privilege escalation via manual calls to PL validator functions (Andres Freund)

  The primary role of PL validator functions is to be called implicitly during **CREATE FUNCTION**, but they are also normal SQL functions that a user can call explicitly. Calling a validator on a function actually written in some other language was not checked for and could be exploited for privilege-escalation purposes. The fix involves adding a call to a privilege-checking function in each validator function. Non-core procedural languages will also need to make this change to their own validator functions, if any. (CVE-2014-0061)

- Avoid multiple name lookups during table and index DDL (Robert Haas, Andres Freund)

  If the name lookups come to different conclusions due to concurrent activity, we might perform some parts of the DDL on a different table than other parts. At least in the case of **CREATE INDEX**, this can be used to cause the permissions checks to be performed against a different table than the index creation, allowing for a privilege escalation attack. (CVE-2014-0062)

- Prevent buffer overrun with long datetime strings (Noah Misch)

  The `MAXDATELEN` constant was too small for the longest possible value of type interval, allowing a buffer overrun in `interval_out()`. Although the datetime input functions were more careful about avoiding buffer overrun, the limit was short enough to cause them to reject some valid inputs, such as input containing a very long timezone name. The ecpg library contained these vulnerabilities along with some of its own. (CVE-2014-0063)

- Prevent buffer overrun due to integer overflow in size calculations (Noah Misch, Heikki Linnakangas)

  Several functions, mostly type input functions, calculated an allocation size without checking for overflow. If overflow did occur, a too-small buffer would be allocated and then written past. (CVE-2014-0064)

- Prevent overruns of fixed-size buffers (Peter Eisentraut, Jozef Mlich)

  Use `strlcpy()` and related functions to provide a clear guarantee that fixed-size buffers are not overrun. Unlike the preceding items, it is unclear whether these cases really represent live issues, since in most cases there appear to be previous constraints on the size of the input string. Nonetheless it seems prudent to silence all Coverity warnings of this type. (CVE-2014-0065)

- Avoid crashing if `crypt()` returns NULL (Honza Horak, Bruce Momjian)

  There are relatively few scenarios in which `crypt()` could return NULL, but `contrib/chkpass` would crash if it did. One practical case in which this could be an issue is if libc is configured to refuse to execute unapproved hashing algorithms (e.g., « FIPS mode »). (CVE-2014-0066)

- Document risks of `make check` in the regression testing instructions (Noah Misch, Tom Lane)

  Since the temporary server started by `make check` uses « trust » authentication, another user on the same machine could connect to it as database superuser, and then potentially exploit the privileges of the operating-system user who started the tests. A future release will probably incorporate changes in the testing procedure to prevent this risk, but some public discussion is needed first. So for the moment, just warn people against using `make check` when there are untrusted users on the same machine. (CVE-2014-0067)

- Fix possible mis-replay of WAL records when some segments of a relation aren't full size (Greg Stark, Tom Lane)

The WAL update could be applied to the wrong page, potentially many pages past where it should have been. Aside from corrupting data, this error has been observed to result in significant « bloat » of standby servers compared to their masters, due to updates being applied far beyond where the end-of-file should have been. This failure mode does not appear to be a significant risk during crash recovery, only when initially synchronizing a standby created from a base backup taken from a quickly-changing master.

- Fix bug in determining when recovery has reached consistency (Tomonari Katsumata, Heikki Linnakangas)

  In some cases WAL replay would mistakenly conclude that the database was already consistent at the start of replay, thus possibly allowing hot-standby queries before the database was really consistent. Other symptoms such as « PANIC: WAL contains references to invalid pages » were also possible.

- Fix improper locking of btree index pages while replaying a `VACUUM` operation in hot-standby mode (Andres Freund, Heikki Linnakangas, Tom Lane)

  This error could result in « PANIC: WAL contains references to invalid pages » failures.

- Ensure that insertions into non-leaf GIN index pages write a full-page WAL record when appropriate (Heikki Linnakangas)

  The previous coding risked index corruption in the event of a partial-page write during a system crash.

- Fix race conditions during server process exit (Robert Haas)

  Ensure that signal handlers don't attempt to use the process's `MyProc` pointer after it's no longer valid.

- Fix unsafe references to `errno` within error reporting logic (Christian Kruse)

  This would typically lead to odd behaviors such as missing or inappropriate `HINT` fields.

- Fix possible crashes from using `ereport()` too early during server startup (Tom Lane)

  The principal case we've seen in the field is a crash if the server is started in a directory it doesn't have permission to read.

- Clear retry flags properly in OpenSSL socket write function (Alexander Kukushkin)

  This omission could result in a server lockup after unexpected loss of an SSL-encrypted connection.

- Fix length checking for Unicode identifiers (`U&"..."` syntax) containing escapes (Tom Lane)

  A spurious truncation warning would be printed for such identifiers if the escaped form of the identifier was too long, but the identifier actually didn't need truncation after de-escaping.

- Allow keywords that are type names to be used in lists of roles (Stephen Frost)

  A previous patch allowed such keywords to be used without quoting in places such as role identifiers; but it missed cases where a list of role identifiers was permitted, such as `DROP ROLE`.

- Fix possible crash due to invalid plan for nested sub-selects, such as `WHERE (... x IN (SELECT ...) ...) IN (SELECT ...)` (Tom Lane)

- Ensure that **ANALYZE** creates statistics for a table column even when all the values in it are « too wide » (Tom Lane)

  ANALYZE intentionally omits very wide values from its histogram and most-common-values calculations, but it neglected to do something sane in the case that all the sampled entries are too wide.

- In `ALTER TABLE ... SET TABLESPACE`, allow the database's default tablespace to be used without a permissions check (Stephen Frost)

  `CREATE TABLE` has always allowed such usage, but `ALTER TABLE` didn't get the memo.

- Fix « cannot accept a set » error when some arms of a `CASE` return a set and others don't (Tom Lane)

- Fix checks for all-zero client addresses in pgstat functions (Kevin Grittner)

- Fix possible misclassification of multibyte characters by the text search parser (Tom Lane)

  Non-ASCII characters could be misclassified when using C locale with a multibyte encoding. On Cygwin, non-C locales could fail as well.

- Fix possible misbehavior in `plainto_tsquery()` (Heikki Linnakangas)

  Use `memmove()` not `memcpy()` for copying overlapping memory regions. There have been no field reports of this actually causing trouble, but it's certainly risky.

- Accept `SHIFT_JIS` as an encoding name for locale checking purposes (Tatsuo Ishii)

- Fix misbehavior of `PQhost()` on Windows (Fujii Masao)

  It should return `localhost` if no host has been specified.
- Improve error handling in libpq and psql for failures during `COPY TO STDOUT/FROM STDIN` (Tom Lane)

  In particular this fixes an infinite loop that could occur in 9.2 and up if the server connection was lost during `COPY FROM STDIN`. Variants of that scenario might be possible in older versions, or with other client applications.
- Fix misaligned descriptors in ecpg (MauMau)
- In ecpg, handle lack of a hostname in the connection parameters properly (Michael Meskes)
- Fix performance regression in `contrib/dblink` connection startup (Joe Conway)

  Avoid an unnecessary round trip when client and server encodings match.
- In `contrib/isn`, fix incorrect calculation of the check digit for ISMN values (Fabien Coelho)
- Ensure client-code-only installation procedure works as documented (Peter Eisentraut)
- In Mingw and Cygwin builds, install the libpq DLL in the `bin` directory (Andrew Dunstan)

  This duplicates what the MSVC build has long done. It should fix problems with programs like psql failing to start because they can't find the DLL.
- Avoid using the deprecated `dllwrap` tool in Cygwin builds (Marco Atzeri)
- Don't generate plain-text `HISTORY` and `src/test/regress/README` files anymore (Tom Lane)

  These text files duplicated the main HTML and PDF documentation formats. The trouble involved in maintaining them greatly outweighs the likely audience for plain-text format. Distribution tarballs will still contain files by these names, but they'll just be stubs directing the reader to consult the main documentation. The plain-text `INSTALL` file will still be maintained, as there is arguably a use-case for that.
- Update time zone data files to tzdata release 2013i for DST law changes in Jordan and historical changes in Cuba.

  In addition, the zones `Asia/Riyadh87`, `Asia/Riyadh88`, and `Asia/Riyadh89` have been removed, as they are no longer maintained by IANA, and never represented actual civil timekeeping practice.

E.4. Release 9.0.15

Release Date

2013-12-05

This release contains a variety of fixes from 9.0.14. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.4.1. Migration to Version 9.0.15

A dump/restore is not required for those running 9.0.X.

However, this release corrects a number of potential data corruption issues. See the first two changelog entries below to find out whether your installation has been affected and what steps you can take if so.

Also, if you are upgrading from a version earlier than 9.0.13, see Section E.6, « Release 9.0.13 ».

E.4.2. Changes

- Fix **VACUUM**'s tests to see whether it can update *`relfrozenxid`* (Andres Freund)

  In some cases **VACUUM** (either manual or autovacuum) could incorrectly advance a table's *`relfrozenxid`* value, allowing tuples to escape freezing, causing those rows to become invisible once 2^31 transactions have elapsed. The probability of data loss is fairly low since multiple incorrect advancements would need to happen before actual loss occurs, but it's not zero. Users upgrading from releases 9.0.4 or 8.4.8 or earlier are not affected, but all later versions contain the bug.

  The issue can be ameliorated by, after upgrading, vacuuming all tables in all databases while having `vacuum_freeze_table_age` set to zero. This will fix any latent corruption but will not be able to fix all pre-existing data errors. However, an installation can be presumed safe after performing this vacuuming if it has executed fewer than 2^31 up-

date transactions in its lifetime (check this with `SELECT txid_current() < 2^31`).

- Fix initialization of `pg_clog` and `pg_subtrans` during hot standby startup (Andres Freund, Heikki Linnakangas)

  This bug can cause data loss on standby servers at the moment they start to accept hot-standby queries, by marking committed transactions as uncommitted. The likelihood of such corruption is small unless, at the time of standby startup, the primary server has executed many updating transactions since its last checkpoint. Symptoms include missing rows, rows that should have been deleted being still visible, and obsolete versions of updated rows being still visible alongside their newer versions.

  This bug was introduced in versions 9.3.0, 9.2.5, 9.1.10, and 9.0.14. Standby servers that have only been running earlier releases are not at risk. It's recommended that standby servers that have ever run any of the buggy releases be re-cloned from the primary (e.g., with a new base backup) after upgrading.

- Truncate `pg_multixact` contents during WAL replay (Andres Freund)

  This avoids ever-increasing disk space consumption in standby servers.

- Fix race condition in GIN index posting tree page deletion (Heikki Linnakangas)

  This could lead to transient wrong answers or query failures.

- Avoid flattening a subquery whose `SELECT` list contains a volatile function wrapped inside a sub-`SELECT` (Tom Lane)

  This avoids unexpected results due to extra evaluations of the volatile function.

- Fix planner's processing of non-simple-variable subquery outputs nested within outer joins (Tom Lane)

  This error could lead to incorrect plans for queries involving multiple levels of subqueries within `JOIN` syntax.

- Fix premature deletion of temporary files (Andres Freund)
- Fix possible read past end of memory in rule printing (Peter Eisentraut)
- Fix array slicing of int2vector and oidvector values (Tom Lane)

  Expressions of this kind are now implicitly promoted to regular int2 or oid arrays.

- Fix incorrect behaviors when using a SQL-standard, simple GMT offset timezone (Tom Lane)

  In some cases, the system would use the simple GMT offset value when it should have used the regular timezone setting that had prevailed before the simple offset was selected. This change also causes the `timeofday` function to honor the simple GMT offset zone.

- Prevent possible misbehavior when logging translations of Windows error codes (Tom Lane)
- Properly quote generated command lines in pg_ctl (Naoya Anzai and Tom Lane)

  This fix applies only to Windows.

- Fix pg_dumpall to work when a source database sets `default_transaction_read_only` via **ALTER DATABASE SET** (Kevin Grittner)

  Previously, the generated script would fail during restore.

- Fix ecpg's processing of lists of variables declared varchar (Zoltán Böszörményi)
- Make `contrib/lo` defend against incorrect trigger definitions (Marc Cousin)
- Update time zone data files to tzdata release 2013h for DST law changes in Argentina, Brazil, Jordan, Libya, Liechtenstein, Morocco, and Palestine. Also, new timezone abbreviations WIB, WIT, WITA for Indonesia.

E.5. Release 9.0.14

Release Date

2013-10-10

This release contains a variety of fixes from 9.0.13. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.5.1. Migration to Version 9.0.14

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.13, see Section E.6, « Release 9.0.13 ».

E.5.2. Changes

- Prevent corruption of multi-byte characters when attempting to case-fold identifiers (Andrew Dunstan)

  PostgreSQL™ case-folds non-ASCII characters only when using a single-byte server encoding.
- Fix checkpoint memory leak in background writer when `wal_level = hot_standby` (Naoya Anzai)
- Fix memory leak caused by `lo_open()` failure (Heikki Linnakangas)
- Fix memory overcommit bug when `work_mem` is using more than 24GB of memory (Stephen Frost)
- Fix deadlock bug in libpq when using SSL (Stephen Frost)
- Fix possible SSL state corruption in threaded libpq applications (Nick Phillips, Stephen Frost)
- Properly compute row estimates for boolean columns containing many NULL values (Andrew Gierth)

  Previously tests like `col IS NOT TRUE` and `col IS NOT FALSE` did not properly factor in NULL values when estimating plan costs.
- Prevent pushing down `WHERE` clauses into unsafe `UNION/INTERSECT` subqueries (Tom Lane)

  Subqueries of a `UNION` or `INTERSECT` that contain set-returning functions or volatile functions in their `SELECT` lists could be improperly optimized, leading to run-time errors or incorrect query results.
- Fix rare case of « failed to locate grouping columns » planner failure (Tom Lane)
- Improve view dumping code's handling of dropped columns in referenced tables (Tom Lane)
- Properly record index comments created using `UNIQUE` and `PRIMARY KEY` syntax (Andres Freund)

  This fixes a parallel pg_restore failure.
- Fix **REINDEX TABLE** and **REINDEX DATABASE** to properly revalidate constraints and mark invalidated indexes as valid (Noah Misch)

  REINDEX INDEX has always worked properly.
- Fix possible deadlock during concurrent **CREATE INDEX CONCURRENTLY** operations (Tom Lane)
- Fix `regexp_matches()` handling of zero-length matches (Jeevan Chalke)

  Previously, zero-length matches like '^' could return too many matches.
- Fix crash for overly-complex regular expressions (Heikki Linnakangas)
- Fix regular expression match failures for back references combined with non-greedy quantifiers (Jeevan Chalke)
- Prevent **CREATE FUNCTION** from checking **SET** variables unless function body checking is enabled (Tom Lane)
- Allow **ALTER DEFAULT PRIVILEGES** to operate on schemas without requiring CREATE permission (Tom Lane)
- Loosen restriction on keywords used in queries (Tom Lane)

  Specifically, lessen keyword restrictions for role names, language names, **EXPLAIN** and **COPY** options, and **SET** values. This allows `COPY ... (FORMAT BINARY)` to work as expected; previously `BINARY` needed to be quoted.
- Fix `pgp_pub_decrypt()` so it works for secret keys with passwords (Marko Kreen)
- Remove rare inaccurate warning during vacuum of index-less tables (Heikki Linnakangas)
- Ensure that **VACUUM ANALYZE** still runs the ANALYZE phase if its attempt to truncate the file is cancelled due to lock conflicts (Kevin Grittner)
- Avoid possible failure when performing transaction control commands (e.g ROLLBACK) in prepared queries (Tom Lane)
- Ensure that floating-point data input accepts standard spellings of « infinity » on all platforms (Tom Lane)

  The C99 standard says that allowable spellings are `inf`, `+inf`, `-inf`, `infinity`, `+infinity`, and `-infinity`. Make sure we recognize these even if the platform's `strtod` function doesn't.
- Expand ability to compare rows to records and arrays (Rafal Rzepecki, Tom Lane)

- Update time zone data files to tzdata release 2013d for DST law changes in Israel, Morocco, Palestine, and Paraguay. Also, historical zone data corrections for Macquarie Island.

E.6. Release 9.0.13

Release Date

2013-04-04

This release contains a variety of fixes from 9.0.12. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.6.1. Migration to Version 9.0.13

A dump/restore is not required for those running 9.0.X.

However, this release corrects several errors in management of GiST indexes. After installing this update, it is advisable to **REINDEX** any GiST indexes that meet one or more of the conditions described below.

Also, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.6.2. Changes

- Fix insecure parsing of server command-line switches (Mitsumasa Kondo, Kyotaro Horiguchi)

  A connection request containing a database name that begins with « `-` » could be crafted to damage or destroy files within the server's data directory, even if the request is eventually rejected. (CVE-2013-1899)

- Reset OpenSSL randomness state in each postmaster child process (Marko Kreen)

  This avoids a scenario wherein random numbers generated by `contrib/pgcrypto` functions might be relatively easy for another database user to guess. The risk is only significant when the postmaster is configured with `ssl` = `on` but most connections don't use SSL encryption. (CVE-2013-1900)

- Fix GiST indexes to not use « fuzzy » geometric comparisons when it's not appropriate to do so (Alexander Korotkov)

  The core geometric types perform comparisons using « fuzzy » equality, but `gist_box_same` must do exact comparisons, else GiST indexes using it might become inconsistent. After installing this update, users should **REINDEX** any GiST indexes on box, polygon, circle, or point columns, since all of these use `gist_box_same`.

- Fix erroneous range-union and penalty logic in GiST indexes that use `contrib/btree_gist` for variable-width data types, that is text, bytea, bit, and numeric columns (Tom Lane)

  These errors could result in inconsistent indexes in which some keys that are present would not be found by searches, and also in useless index bloat. Users are advised to **REINDEX** such indexes after installing this update.

- Fix bugs in GiST page splitting code for multi-column indexes (Tom Lane)

  These errors could result in inconsistent indexes in which some keys that are present would not be found by searches, and also in indexes that are unnecessarily inefficient to search. Users are advised to **REINDEX** multi-column GiST indexes after installing this update.

- Fix `gist_point_consistent` to handle fuzziness consistently (Alexander Korotkov)

  Index scans on GiST indexes on point columns would sometimes yield results different from a sequential scan, because `gist_point_consistent` disagreed with the underlying operator code about whether to do comparisons exactly or fuzzily.

- Fix buffer leak in WAL replay (Heikki Linnakangas)

  This bug could result in « incorrect local pin count » errors during replay, making recovery impossible.

- Fix race condition in **DELETE RETURNING** (Tom Lane)

  Under the right circumstances, **DELETE RETURNING** could attempt to fetch data from a shared buffer that the current process no longer has any pin on. If some other process changed the buffer meanwhile, this would lead to garbage `RETURNING` output, or even a crash.

- Fix infinite-loop risk in regular expression compilation (Tom Lane, Don Porter)

- Fix potential null-pointer dereference in regular expression compilation (Tom Lane)
- Fix `to_char()` to use ASCII-only case-folding rules where appropriate (Tom Lane)

  This fixes misbehavior of some template patterns that should be locale-independent, but mishandled « `I` » and « `i` » in Turkish locales.
- Fix unwanted rejection of timestamp `1999-12-31 24:00:00` (Tom Lane)
- Fix logic error when a single transaction does **UNLISTEN** then **LISTEN** (Tom Lane)

  The session wound up not listening for notify events at all, though it surely should listen in this case.
- Remove useless « picksplit doesn't support secondary split » log messages (Josh Hansen, Tom Lane)

  This message seems to have been added in expectation of code that was never written, and probably never will be, since GiST's default handling of secondary splits is actually pretty good. So stop nagging end users about it.
- Fix possible failure to send a session's last few transaction commit/abort counts to the statistics collector (Tom Lane)
- Eliminate memory leaks in PL/Perl's `spi_prepare()` function (Alex Hunsaker, Tom Lane)
- Fix pg_dumpall to handle database names containing « `=` » correctly (Heikki Linnakangas)
- Avoid crash in pg_dump when an incorrect connection string is given (Heikki Linnakangas)
- Ignore invalid indexes in pg_dump and pg_upgrade (Michael Paquier, Bruce Momjian)

  Dumping invalid indexes can cause problems at restore time, for example if the reason the index creation failed was because it tried to enforce a uniqueness condition not satisfied by the table's data. Also, if the index creation is in fact still in progress, it seems reasonable to consider it to be an uncommitted DDL change, which pg_dump wouldn't be expected to dump anyway. pg_upgrade now also skips invalid indexes rather than failing.
- Fix `contrib/pg_trgm`'s `similarity()` function to return zero for trigram-less strings (Tom Lane)

  Previously it returned `NaN` due to internal division by zero.
- Update time zone data files to tzdata release 2013b for DST law changes in Chile, Haiti, Morocco, Paraguay, and some Russian areas. Also, historical zone data corrections for numerous places.

  Also, update the time zone abbreviation files for recent changes in Russia and elsewhere: `CHOT`, `GET`, `IRKT`, `KGT`, `KRAT`, `MAGT`, `MAWT`, `MSK`, `NOVT`, `OMST`, `TKT`, `VLAT`, `WST`, `YAKT`, `YEKT` now follow their current meanings, and `VOLT` (Europe/Volgograd) and `MIST` (Antarctica/Macquarie) are added to the default abbreviations list.

E.7. Release 9.0.12

Release Date

2013-02-07

This release contains a variety of fixes from 9.0.11. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.7.1. Migration to Version 9.0.12

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.7.2. Changes

- Prevent execution of `enum_recv` from SQL (Tom Lane)

  The function was misdeclared, allowing a simple SQL command to crash the server. In principle an attacker might be able to use it to examine the contents of server memory. Our thanks to Sumit Soni (via Secunia SVCRP) for reporting this issue. (CVE-2013-0255)
- Fix multiple problems in detection of when a consistent database state has been reached during WAL replay (Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)

- Update minimum recovery point when truncating a relation file (Heikki Linnakangas)

  Once data has been discarded, it's no longer safe to stop recovery at an earlier point in the timeline.
- Fix missing cancellations in hot standby mode (Noah Misch, Simon Riggs)

  The need to cancel conflicting hot-standby queries would sometimes be missed, allowing those queries to see inconsistent data.
- Fix SQL grammar to allow subscripting or field selection from a sub-SELECT result (Tom Lane)
- Fix performance problems with autovacuum truncation in busy workloads (Jan Wieck)

  Truncation of empty pages at the end of a table requires exclusive lock, but autovacuum was coded to fail (and release the table lock) when there are conflicting lock requests. Under load, it is easily possible that truncation would never occur, resulting in table bloat. Fix by performing a partial truncation, releasing the lock, then attempting to re-acquire the lock and continue. This fix also greatly reduces the average time before autovacuum releases the lock after a conflicting request arrives.
- Protect against race conditions when scanning pg_tablespace (Stephen Frost, Tom Lane)

  CREATE DATABASE and **DROP DATABASE** could misbehave if there were concurrent updates of pg_tablespace entries.
- Prevent **DROP OWNED** from trying to drop whole databases or tablespaces (Álvaro Herrera)

  For safety, ownership of these objects must be reassigned, not dropped.
- Fix error in `vacuum_freeze_table_age` implementation (Andres Freund)

  In installations that have existed for more than `vacuum_freeze_min_age` transactions, this mistake prevented autovacuum from using partial-table scans, so that a full-table scan would always happen instead.
- Prevent misbehavior when a RowExpr or XmlExpr is parse-analyzed twice (Andres Freund, Tom Lane)

  This mistake could be user-visible in contexts such as `CREATE TABLE LIKE INCLUDING INDEXES`.
- Improve defenses against integer overflow in hashtable sizing calculations (Jeff Davis)
- Reject out-of-range dates in `to_date()` (Hitoshi Harada)
- Ensure that non-ASCII prompt strings are translated to the correct code page on Windows (Alexander Law, Noah Misch)

  This bug affected psql and some other client programs.
- Fix possible crash in psql's **\?** command when not connected to a database (Meng Qingzhong)
- Fix pg_upgrade to deal with invalid indexes safely (Bruce Momjian)
- Fix one-byte buffer overrun in libpq's `PQprintTuples` (Xi Wang)

  This ancient function is not used anywhere by PostgreSQL™ itself, but it might still be used by some client code.
- Make ecpglib use translated messages properly (Chen Huajun)
- Properly install ecpg_compat and pgtypes libraries on MSVC (Jiang Guiqing)
- Include our version of `isinf()` in libecpg if it's not provided by the system (Jiang Guiqing)
- Rearrange configure's tests for supplied functions so it is not fooled by bogus exports from libedit/libreadline (Christoph Berg)
- Ensure Windows build number increases over time (Magnus Hagander)
- Make pgxs build executables with the right `.exe` suffix when cross-compiling for Windows (Zoltan Boszormenyi)
- Add new timezone abbreviation `FET` (Tom Lane)

  This is now used in some eastern-European time zones.

E.8. Release 9.0.11

Release Date

2012-12-06

This release contains a variety of fixes from 9.0.10. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.8.1. Migration to Version 9.0.11

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.8.2. Changes

- Fix multiple bugs associated with **CREATE INDEX CONCURRENTLY** (Andres Freund, Tom Lane)

  Fix **CREATE INDEX CONCURRENTLY** to use in-place updates when changing the state of an index's pg_index row. This prevents race conditions that could cause concurrent sessions to miss updating the target index, thus resulting in corrupt concurrently-created indexes.

  Also, fix various other operations to ensure that they ignore invalid indexes resulting from a failed **CREATE INDEX CONCURRENTLY** command. The most important of these is **VACUUM**, because an auto-vacuum could easily be launched on the table before corrective action can be taken to fix or remove the invalid index.

- Fix buffer locking during WAL replay (Tom Lane)

  The WAL replay code was insufficiently careful about locking buffers when replaying WAL records that affect more than one page. This could result in hot standby queries transiently seeing inconsistent states, resulting in wrong answers or unexpected failures.

- Fix an error in WAL generation logic for GIN indexes (Tom Lane)

  This could result in index corruption, if a torn-page failure occurred.

- Properly remove startup process's virtual XID lock when promoting a hot standby server to normal running (Simon Riggs)

  This oversight could prevent subsequent execution of certain operations such as **CREATE INDEX CONCURRENTLY**.

- Avoid bogus « out-of-sequence timeline ID » errors in standby mode (Heikki Linnakangas)

- Prevent the postmaster from launching new child processes after it's received a shutdown signal (Tom Lane)

  This mistake could result in shutdown taking longer than it should, or even never completing at all without additional user action.

- Avoid corruption of internal hash tables when out of memory (Hitoshi Harada)

- Fix planning of non-strict equivalence clauses above outer joins (Tom Lane)

  The planner could derive incorrect constraints from a clause equating a non-strict construct to something else, for example `WHERE COALESCE(foo, 0) = 0` when `foo` is coming from the nullable side of an outer join.

- Improve planner's ability to prove exclusion constraints from equivalence classes (Tom Lane)

- Fix partial-row matching in hashed subplans to handle cross-type cases correctly (Tom Lane)

  This affects multicolumn `NOT IN` subplans, such as `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` when for instance `b` and `y` are int4 and int8 respectively. This mistake led to wrong answers or crashes depending on the specific datatypes involved.

- Acquire buffer lock when re-fetching the old tuple for an `AFTER ROW UPDATE/DELETE` trigger (Andres Freund)

  In very unusual circumstances, this oversight could result in passing incorrect data to the precheck logic for a foreign-key enforcement trigger. That could result in a crash, or in an incorrect decision about whether to fire the trigger.

- Fix **ALTER COLUMN TYPE** to handle alled check constraints properly (Pavan Deolasee)

  This worked correctly in pre-8.4 releases, and now works correctly in 8.4 and later.

- Fix **REASSIGN OWNED** to handle grants on tablespaces (Álvaro Herrera)

- Ignore incorrect pg_attribute entries for system columns for views (Tom Lane)

  Views do not have any system columns. However, we forgot to remove such entries when converting a table to a view. That's fixed properly for 9.3 and later, but in previous branches we need to defend against existing mis-converted views.

- Fix rule printing to dump `INSERT INTO` *`table`* `DEFAULT VALUES` correctly (Tom Lane)

- Guard against stack overflow when there are too many `UNION`/`INTERSECT`/`EXCEPT` clauses in a query (Tom Lane)

- Prevent platform-dependent failures when dividing the minimum possible integer value by -1 (Xi Wang, Tom Lane)

- Fix possible access past end of string in date parsing (Hitoshi Harada)
- Fix failure to advance XID epoch if XID wraparound happens during a checkpoint and `wal_level` is `hot_standby` (Tom Lane, Andres Freund)

  While this mistake had no particular impact on PostgreSQL™ itself, it was bad for applications that rely on `txid_current()` and related functions: the TXID value would appear to go backwards.
- Produce an understandable error message if the length of the path name for a Unix-domain socket exceeds the platform-specific limit (Tom Lane, Andrew Dunstan)

  Formerly, this would result in something quite unhelpful, such as « Non-recoverable failure in name resolution ».
- Fix memory leaks when sending composite column values to the client (Tom Lane)
- Make pg_ctl more robust about reading the `postmaster.pid` file (Heikki Linnakangas)

  Fix race conditions and possible file descriptor leakage.
- Fix possible crash in psql if incorrectly-encoded data is presented and the `client_encoding` setting is a client-only encoding, such as SJIS (Jiang Guiqing)
- Fix bugs in the `restore.sql` script emitted by pg_dump in `tar` output format (Tom Lane)

  The script would fail outright on tables whose names include upper-case characters. Also, make the script capable of restoring data in `--inserts` mode as well as the regular COPY mode.
- Fix pg_restore to accept POSIX-conformant `tar` files (Brian Weaver, Tom Lane)

  The original coding of pg_dump's `tar` output mode produced files that are not fully conformant with the POSIX standard. This has been corrected for version 9.3. This patch updates previous branches so that they will accept both the incorrect and the corrected formats, in hopes of avoiding compatibility problems when 9.3 comes out.
- Fix pg_resetxlog to locate `postmaster.pid` correctly when given a relative path to the data directory (Tom Lane)

  This mistake could lead to pg_resetxlog not noticing that there is an active postmaster using the data directory.
- Fix libpq's `lo_import()` and `lo_export()` functions to report file I/O errors properly (Tom Lane)
- Fix ecpg's processing of nested structure pointer variables (Muhammad Usama)
- Fix ecpg's `ecpg_get_data` function to handle arrays properly (Michael Meskes)
- Make `contrib/pageinspect`'s btree page inspection functions take buffer locks while examining pages (Tom Lane)
- Fix pgxs support for building loadable modules on AIX (Tom Lane)

  Building modules outside the original source tree didn't work on AIX.
- Update time zone data files to tzdata release 2012j for DST law changes in Cuba, Israel, Jordan, Libya, Palestine, Western Samoa, and portions of Brazil.

E.9. Release 9.0.10

Release Date

2012-09-24

This release contains a variety of fixes from 9.0.9. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.9.1. Migration to Version 9.0.10

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.9.2. Changes

- Fix planner's assignment of executor parameters, and fix executor's rescan logic for CTE plan nodes (Tom Lane)

These errors could result in wrong answers from queries that scan the same `WITH` subquery multiple times.

- Improve page-splitting decisions in GiST indexes (Alexander Korotkov, Robert Haas, Tom Lane)

  Multi-column GiST indexes might suffer unexpected bloat due to this error.

- Fix cascading privilege revoke to stop if privileges are still held (Tom Lane)

  If we revoke a grant option from some role *X*, but *X* still holds that option via a grant from someone else, we should not recursively revoke the corresponding privilege from role(s) *Y* that *X* had granted it to.

- Improve error messages for Hot Standby misconfiguration errors (Gurjeet Singh)
- Fix handling of `SIGFPE` when PL/Perl is in use (Andres Freund)

  Perl resets the process's `SIGFPE` handler to `SIG_IGN`, which could result in crashes later on. Restore the normal Postgres signal handler after initializing PL/Perl.

- Prevent PL/Perl from crashing if a recursive PL/Perl function is redefined while being executed (Tom Lane)
- Work around possible misoptimization in PL/Perl (Tom Lane)

  Some Linux distributions contain an incorrect version of `pthread.h` that results in incorrect compiled code in PL/Perl, leading to crashes if a PL/Perl function calls another one that throws an error.

- Fix pg_upgrade's handling of line endings on Windows (Andrew Dunstan)

  Previously, pg_upgrade might add or remove carriage returns in places such as function bodies.

- On Windows, make pg_upgrade use backslash path separators in the scripts it emits (Andrew Dunstan)
- Update time zone data files to tzdata release 2012f for DST law changes in Fiji

E.10. Release 9.0.9

Release Date

2012-08-17

This release contains a variety of fixes from 9.0.8. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.10.1. Migration to Version 9.0.9

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.10.2. Changes

- Prevent access to external files/URLs via XML entity references (Noah Misch, Tom Lane)

  `xml_parse()` would attempt to fetch external files or URLs as needed to resolve DTD and entity references in an XML value, thus allowing unprivileged database users to attempt to fetch data with the privileges of the database server. While the external data wouldn't get returned directly to the user, portions of it could be exposed in error messages if the data didn't parse as valid XML; and in any case the mere ability to check existence of a file might be useful to an attacker. (CVE-2012-3489)

- Prevent access to external files/URLs via `contrib/xml2`'s `xslt_process()` (Peter Eisentraut)

  libxslt offers the ability to read and write both files and URLs through stylesheet commands, thus allowing unprivileged database users to both read and write data with the privileges of the database server. Disable that through proper use of libxslt's security options. (CVE-2012-3488)

  Also, remove `xslt_process()`'s ability to fetch documents and stylesheets from external files/URLs. While this was a documented « feature », it was long regarded as a bad idea. The fix for CVE-2012-3489 broke that capability, and rather than expend effort on trying to fix it, we're just going to summarily remove it.

- Prevent too-early recycling of btree index pages (Noah Misch)

  When we allowed read-only transactions to skip assigning XIDs, we introduced the possibility that a deleted btree page could

be recycled while a read-only transaction was still in flight to it. This would result in incorrect index search results. The probability of such an error occurring in the field seems very low because of the timing requirements, but nonetheless it should be fixed.

- Fix crash-safety bug with newly-created-or-reset sequences (Tom Lane)

  If **ALTER SEQUENCE** was executed on a freshly created or reset sequence, and then precisely one `nextval()` call was made on it, and then the server crashed, WAL replay would restore the sequence to a state in which it appeared that no `nextval()` had been done, thus allowing the first sequence value to be returned again by the next `nextval()` call. In particular this could manifest for serial columns, since creation of a serial column's sequence includes an **ALTER SEQUENCE OWNED BY** step.

- Fix `txid_current()` to report the correct epoch when not in hot standby (Heikki Linnakangas)

  This fixes a regression introduced in the previous minor release.

- Fix bug in startup of Hot Standby when a master transaction has many subtransactions (Andres Freund)

  This mistake led to failures reported as « out-of-order XID insertion in KnownAssignedXids ».

- Ensure the `backup_label` file is fsync'd after `pg_start_backup()` (Dave Kerr)

- Fix timeout handling in walsender processes (Tom Lane)

  WAL sender background processes neglected to establish a `SIGALRM` handler, meaning they would wait forever in some corner cases where a timeout ought to happen.

- Back-patch 9.1 improvement to compress the fsync request queue (Robert Haas)

  This improves performance during checkpoints. The 9.1 change has now seen enough field testing to seem safe to back-patch.

- Fix `LISTEN`/`NOTIFY` to cope better with I/O problems, such as out of disk space (Tom Lane)

  After a write failure, all subsequent attempts to send more `NOTIFY` messages would fail with messages like « Could not read from file "pg_notify/*nnnn*" at offset *nnnnn*: Success ».

- Only allow autovacuum to be auto-canceled by a directly blocked process (Tom Lane)

  The original coding could allow inconsistent behavior in some cases; in particular, an autovacuum could get canceled after less than `deadlock_timeout` grace period.

- Improve logging of autovacuum cancels (Robert Haas)

- Fix log collector so that `log_truncate_on_rotation` works during the very first log rotation after server start (Tom Lane)

- Fix `WITH` attached to a nested set operation (`UNION`/`INTERSECT`/`EXCEPT`) (Tom Lane)

- Ensure that a whole-row reference to a subquery doesn't include any extra `GROUP BY` or `ORDER BY` columns (Tom Lane)

- Disallow copying whole-row references in `CHECK` constraints and index definitions during **CREATE TABLE** (Tom Lane)

  This situation can arise in **CREATE TABLE** with `LIKE` or `INHERITS`. The copied whole-row variable was incorrectly labeled with the row type of the original table not the new one. Rejecting the case seems reasonable for `LIKE`, since the row types might well diverge later. For `INHERITS` we should ideally allow it, with an implicit coercion to the parent table's row type; but that will require more work than seems safe to back-patch.

- Fix memory leak in `ARRAY(SELECT ...)` subqueries (Heikki Linnakangas, Tom Lane)

- Fix extraction of common prefixes from regular expressions (Tom Lane)

  The code could get confused by quantified parenthesized subexpressions, such as `^(foo)?bar`. This would lead to incorrect index optimization of searches for such patterns.

- Fix bugs with parsing signed *hh*:*mm* and *hh*:*mm*:*ss* fields in interval constants (Amit Kapila, Tom Lane)

- Use Postgres' encoding conversion functions, not Python's, when converting a Python Unicode string to the server encoding in PL/Python (Jan Urbanski)

  This avoids some corner-case problems, notably that Python doesn't support all the encodings Postgres does. A notable functional change is that if the server encoding is SQL_ASCII, you will get the UTF-8 representation of the string; formerly, any non-ASCII characters in the string would result in an error.

- Fix mapping of PostgreSQL encodings to Python encodings in PL/Python (Jan Urbanski)

- Report errors properly in `contrib/xml2`'s `xslt_process()` (Tom Lane)
- Update time zone data files to tzdata release 2012e for DST law changes in Morocco and Tokelau

E.11. Release 9.0.8

Release Date

2012-06-04

This release contains a variety of fixes from 9.0.7. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.11.1. Migration to Version 9.0.8

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.11.2. Changes

- Fix incorrect password transformation in `contrib/pgcrypto`'s DES `crypt()` function (Solar Designer)

  If a password string contained the byte value `0x80`, the remainder of the password was ignored, causing the password to be much weaker than it appeared. With this fix, the rest of the string is properly included in the DES hash. Any stored password values that are affected by this bug will thus no longer match, so the stored values may need to be updated. (CVE-2012-2143)

- Ignore `SECURITY DEFINER` and `SET` attributes for a procedural language's call handler (Tom Lane)

  Applying such attributes to a call handler could crash the server. (CVE-2012-2655)

- Allow numeric timezone offsets in timestamp input to be up to 16 hours away from UTC (Tom Lane)

  Some historical time zones have offsets larger than 15 hours, the previous limit. This could result in dumped data values being rejected during reload.

- Fix timestamp conversion to cope when the given time is exactly the last DST transition time for the current timezone (Tom Lane)

  This oversight has been there a long time, but was not noticed previously because most DST-using zones are presumed to have an indefinite sequence of future DST transitions.

- Fix text to name and char to name casts to perform string truncation correctly in multibyte encodings (Karl Schnaitter)
- Fix memory copying bug in `to_tsquery()` (Heikki Linnakangas)
- Ensure `txid_current()` reports the correct epoch when executed in hot standby (Simon Riggs)
- Fix planner's handling of outer PlaceHolderVars within subqueries (Tom Lane)

  This bug concerns sub-SELECTs that reference variables coming from the nullable side of an outer join of the surrounding query. In 9.1, queries affected by this bug would fail with « ERROR: Upper-level PlaceHolderVar found where not expected ». But in 9.0 and 8.4, you'd silently get possibly-wrong answers, since the value transmitted into the subquery wouldn't go to null when it should.

- Fix slow session startup when pg_attribute is very large (Tom Lane)

  If pg_attribute exceeds one-fourth of `shared_buffers`, cache rebuilding code that is sometimes needed during session start would trigger the synchronized-scan logic, causing it to take many times longer than normal. The problem was particularly acute if many new sessions were starting at once.

- Ensure sequential scans check for query cancel reasonably often (Merlin Moncure)

  A scan encountering many consecutive pages that contain no live tuples would not respond to interrupts meanwhile.

- Ensure the Windows implementation of `PGSemaphoreLock()` clears `ImmediateInterruptOK` before returning (Tom Lane)

  This oversight meant that a query-cancel interrupt received later in the same query could be accepted at an unsafe time, with unpredictable but not good consequences.

- Show whole-row variables safely when printing views or rules (Abbas Butt, Tom Lane)

  Corner cases involving ambiguous names (that is, the name could be either a table or column name of the query) were printed in an ambiguous way, risking that the view or rule would be interpreted differently after dump and reload. Avoid the ambiguous case by attaching a no-op cast.
- Fix **COPY FROM** to properly handle null marker strings that correspond to invalid encoding (Tom Lane)

  A null marker string such as `E'\\0'` should work, and did work in the past, but the case got broken in 8.4.
- Ensure autovacuum worker processes perform stack depth checking properly (Heikki Linnakangas)

  Previously, infinite recursion in a function invoked by auto-**ANALYZE** could crash worker processes.
- Fix logging collector to not lose log coherency under high load (Andrew Dunstan)

  The collector previously could fail to reassemble large messages if it got too busy.
- Fix logging collector to ensure it will restart file rotation after receiving `SIGHUP` (Tom Lane)
- Fix WAL replay logic for GIN indexes to not fail if the index was subsequently dropped (Tom Lane)
- Fix memory leak in PL/pgSQL's **RETURN NEXT** command (Joe Conway)
- Fix PL/pgSQL's **GET DIAGNOSTICS** command when the target is the function's first variable (Tom Lane)
- Fix potential access off the end of memory in psql's expanded display (**x**) mode (Peter Eisentraut)
- Fix several performance problems in pg_dump when the database contains many objects (Jeff Janes, Tom Lane)

  pg_dump could get very slow if the database contained many schemas, or if many objects are in dependency loops, or if there are many owned sequences.
- Fix pg_upgrade for the case that a database stored in a non-default tablespace contains a table in the cluster's default tablespace (Bruce Momjian)
- In ecpg, fix rare memory leaks and possible overwrite of one byte after the sqlca_t structure (Peter Eisentraut)
- Fix `contrib/dblink`'s `dblink_exec()` to not leak temporary database connections upon error (Tom Lane)
- Fix `contrib/dblink` to report the correct connection name in error messages (Kyotaro Horiguchi)
- Fix `contrib/vacuumlo` to use multiple transactions when dropping many large objects (Tim Lewis, Robert Haas, Tom Lane)

  This change avoids exceeding `max_locks_per_transaction` when many objects need to be dropped. The behavior can be adjusted with the new `-l` (limit) option.
- Update time zone data files to tzdata release 2012c for DST law changes in Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria, and Tokelau Islands; also historical corrections for Canada.

E.12. Release 9.0.7

Release Date

2012-02-27

This release contains a variety of fixes from 9.0.6. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.12.1. Migration to Version 9.0.7

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.6, see Section E.13, « Release 9.0.6 ».

E.12.2. Changes

- Require execute permission on the trigger function for **CREATE TRIGGER** (Robert Haas)

  This missing check could allow another user to execute a trigger function with forged input data, by installing it on a table he

owns. This is only of significance for trigger functions marked `SECURITY DEFINER`, since otherwise trigger functions run as the table owner anyway. (CVE-2012-0866)

- Remove arbitrary limitation on length of common name in SSL certificates (Heikki Linnakangas)

  Both libpq and the server truncated the common name extracted from an SSL certificate at 32 bytes. Normally this would cause nothing worse than an unexpected verification failure, but there are some rather-implausible scenarios in which it might allow one certificate holder to impersonate another. The victim would have to have a common name exactly 32 bytes long, and the attacker would have to persuade a trusted CA to issue a certificate in which the common name has that string as a prefix. Impersonating a server would also require some additional exploit to redirect client connections. (CVE-2012-0867)

- Convert newlines to spaces in names written in pg_dump comments (Robert Haas)

  pg_dump was incautious about sanitizing object names that are emitted within SQL comments in its output script. A name containing a newline would at least render the script syntactically incorrect. Maliciously crafted object names could present a SQL injection risk when the script is reloaded. (CVE-2012-0868)

- Fix btree index corruption from insertions concurrent with vacuuming (Tom Lane)

  An index page split caused by an insertion could sometimes cause a concurrently-running **VACUUM** to miss removing index entries that it should remove. After the corresponding table rows are removed, the dangling index entries would cause errors (such as « could not read block N in file ... ») or worse, silently wrong query results after unrelated rows are re-inserted at the now-free table locations. This bug has been present since release 8.2, but occurs so infrequently that it was not diagnosed until now. If you have reason to suspect that it has happened in your database, reindexing the affected index will fix things.

- Fix transient zeroing of shared buffers during WAL replay (Tom Lane)

  The replay logic would sometimes zero and refill a shared buffer, so that the contents were transiently invalid. In hot standby mode this can result in a query that's executing in parallel seeing garbage data. Various symptoms could result from that, but the most common one seems to be « invalid memory alloc request size ».

- Fix postmaster to attempt restart after a hot-standby crash (Tom Lane)

  A logic error caused the postmaster to terminate, rather than attempt to restart the cluster, if any backend process crashed while operating in hot standby mode.

- Fix **CLUSTER/VACUUM FULL** handling of toast values owned by recently-updated rows (Tom Lane)

  This oversight could lead to « duplicate key value violates unique constraint » errors being reported against the toast table's index during one of these commands.

- Update per-column permissions, not only per-table permissions, when changing table owner (Tom Lane)

  Failure to do this meant that any previously granted column permissions were still shown as having been granted by the old owner. This meant that neither the new owner nor a superuser could revoke the now-untraceable-to-table-owner permissions.

- Support foreign data wrappers and foreign servers in **REASSIGN OWNED** (Alvaro Herrera)

  This command failed with « unexpected classid » errors if it needed to change the ownership of any such objects.

- Allow non-existent values for some settings in **ALTER USER/DATABASE SET** (Heikki Linnakangas)

  Allow `default_text_search_config`, `default_tablespace`, and `temp_tablespaces` to be set to names that are not known. This is because they might be known in another database where the setting is intended to be used, or for the tablespace cases because the tablespace might not be created yet. The same issue was previously recognized for `search_path`, and these settings now act like that one.

- Avoid crashing when we have problems deleting table files post-commit (Tom Lane)

  Dropping a table should lead to deleting the underlying disk files only after the transaction commits. In event of failure then (for instance, because of wrong file permissions) the code is supposed to just emit a warning message and go on, since it's too late to abort the transaction. This logic got broken as of release 8.4, causing such situations to result in a PANIC and an unrestartable database.

- Recover from errors occurring during WAL replay of **DROP TABLESPACE** (Tom Lane)

  Replay will attempt to remove the tablespace's directories, but there are various reasons why this might fail (for example, incorrect ownership or permissions on those directories). Formerly the replay code would panic, rendering the database unrestartable without manual intervention. It seems better to log the problem and continue, since the only consequence of failure to remove the directories is some wasted disk space.

- Fix race condition in logging AccessExclusiveLocks for hot standby (Simon Riggs)

Sometimes a lock would be logged as being held by « transaction zero ». This is at least known to produce assertion failures on slave servers, and might be the cause of more serious problems.

- Track the OID counter correctly during WAL replay, even when it wraps around (Tom Lane)

  Previously the OID counter would remain stuck at a high value until the system exited replay mode. The practical consequences of that are usually nil, but there are scenarios wherein a standby server that's been promoted to master might take a long time to advance the OID counter to a reasonable value once values are needed.

- Prevent emitting misleading « consistent recovery state reached » log message at the beginning of crash recovery (Heikki Linnakangas)

- Fix initial value of pg_stat_replication.*replay_location* (Fujii Masao)

  Previously, the value shown would be wrong until at least one WAL record had been replayed.

- Fix regular expression back-references with `*` attached (Tom Lane)

  Rather than enforcing an exact string match, the code would effectively accept any string that satisfies the pattern subexpression referenced by the back-reference symbol.

  A similar problem still afflicts back-references that are embedded in a larger quantified expression, rather than being the immediate subject of the quantifier. This will be addressed in a future PostgreSQL™ release.

- Fix recently-introduced memory leak in processing of inet/cidr values (Heikki Linnakangas)

  A patch in the December 2011 releases of PostgreSQL™ caused memory leakage in these operations, which could be significant in scenarios such as building a btree index on such a column.

- Fix dangling pointer after **CREATE TABLE AS**/**SELECT INTO** in a SQL-language function (Tom Lane)

  In most cases this only led to an assertion failure in assert-enabled builds, but worse consequences seem possible.

- Avoid double close of file handle in syslogger on Windows (MauMau)

  Ordinarily this error was invisible, but it would cause an exception when running on a debug version of Windows.

- Fix I/O-conversion-related memory leaks in plpgsql (Andres Freund, Jan Urbanski, Tom Lane)

  Certain operations would leak memory until the end of the current function.

- Improve pg_dump's handling of alled table columns (Tom Lane)

  pg_dump mishandled situations where a child column has a different default expression than its parent column. If the default is textually identical to the parent's default, but not actually the same (for instance, because of schema search path differences) it would not be recognized as different, so that after dump and restore the child would be allowed to all the parent's default. Child columns that are `NOT NULL` where their parent is not could also be restored subtly incorrectly.

- Fix pg_restore's direct-to-database mode for INSERT-style table data (Tom Lane)

  Direct-to-database restores from archive files made with `--inserts` or `--column-inserts` options fail when using pg_restore from a release dated September or December 2011, as a result of an oversight in a fix for another problem. The archive file itself is not at fault, and text-mode output is okay.

- Allow pg_upgrade to process tables containing regclass columns (Bruce Momjian)

  Since pg_upgrade now takes care to preserve pg_class OIDs, there was no longer any reason for this restriction.

- Make libpq ignore `ENOTDIR` errors when looking for an SSL client certificate file (Magnus Hagander)

  This allows SSL connections to be established, though without a certificate, even when the user's home directory is set to something like `/dev/null`.

- Fix some more field alignment issues in ecpg's SQLDA area (Zoltan Boszormenyi)

- Allow `AT` option in ecpg `DEALLOCATE` statements (Michael Meskes)

  The infrastructure to support this has been there for awhile, but through an oversight there was still an error check rejecting the case.

- Do not use the variable name when defining a varchar structure in ecpg (Michael Meskes)

- Fix `contrib/auto_explain`'s JSON output mode to produce valid JSON (Andrew Dunstan)

  The output used brackets at the top level, when it should have used braces.

- Fix error in `contrib/intarray`'s `int[] & int[]` operator (Guillaume Lelarge)

  If the smallest integer the two input arrays have in common is 1, and there are smaller values in either array, then 1 would be incorrectly omitted from the result.

- Fix error detection in `contrib/pgcrypto`'s `encrypt_iv()` and `decrypt_iv()` (Marko Kreen)

  These functions failed to report certain types of invalid-input errors, and would instead return random garbage values for incorrect input.

- Fix one-byte buffer overrun in `contrib/test_parser` (Paul Guyot)

  The code would try to read one more byte than it should, which would crash in corner cases. Since `contrib/test_parser` is only example code, this is not a security issue in itself, but bad example code is still bad.

- Use `__sync_lock_test_and_set()` for spinlocks on ARM, if available (Martin Pitt)

  This function replaces our previous use of the `SWPB` instruction, which is deprecated and not available on ARMv6 and later. Reports suggest that the old code doesn't fail in an obvious way on recent ARM boards, but simply doesn't interlock concurrent accesses, leading to bizarre failures in multiprocess operation.

- Use `-fexcess-precision=standard` option when building with gcc versions that accept it (Andrew Dunstan)

  This prevents assorted scenarios wherein recent versions of gcc will produce creative results.

- Allow use of threaded Python on FreeBSD (Chris Rees)

  Our configure script previously believed that this combination wouldn't work; but FreeBSD fixed the problem, so remove that error check.

E.13. Release 9.0.6

Release Date

2011-12-05

This release contains a variety of fixes from 9.0.5. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.13.1. Migration to Version 9.0.6

A dump/restore is not required for those running 9.0.X.

However, a longstanding error was discovered in the definition of the `information_schema.referential_constraints` view. If you rely on correct results from that view, you should replace its definition as explained in the first changelog item below.

Also, if you are upgrading from a version earlier than 9.0.4, see Section E.15, « Release 9.0.4 ».

E.13.2. Changes

- Fix bugs in `information_schema.referential_constraints` view (Tom Lane)

  This view was being insufficiently careful about matching the foreign-key constraint to the depended-on primary or unique key constraint. That could result in failure to show a foreign key constraint at all, or showing it multiple times, or claiming that it depends on a different constraint than the one it really does.

  Since the view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can (as a superuser) drop the `information_schema` schema then re-create it by sourcing *`SHAREDIR`*`/information_schema.sql`. (Run `pg_config --sharedir` if you're uncertain where *`SHAREDIR`* is.) This must be repeated in each database to be fixed.

- Fix possible crash during **UPDATE** or **DELETE** that joins to the output of a scalar-returning function (Tom Lane)

  A crash could only occur if the target row had been concurrently updated, so this problem surfaced only intermittently.

- Fix incorrect replay of WAL records for GIN index updates (Tom Lane)

  This could result in transiently failing to find index entries after a crash, or on a hot-standby server. The problem would be re-

paired by the next **VACUUM** of the index, however.

- Fix TOAST-related data corruption during `CREATE TABLE dest AS SELECT * FROM src` or `INSERT INTO dest SELECT * FROM src` (Tom Lane)

  If a table has been modified by **ALTER TABLE ADD COLUMN**, attempts to copy its data verbatim to another table could produce corrupt results in certain corner cases. The problem can only manifest in this precise form in 8.4 and later, but we patched earlier versions as well in case there are other code paths that could trigger the same bug.

- Fix possible failures during hot standby startup (Simon Riggs)
- Start hot standby faster when initial snapshot is incomplete (Simon Riggs)
- Fix race condition during toast table access from stale syscache entries (Tom Lane)

  The typical symptom was transient errors like « missing chunk number 0 for toast value NNNNN in pg_toast_2619 », where the cited toast table would always belong to a system catalog.

- Track dependencies of functions on items used in parameter default expressions (Tom Lane)

  Previously, a referenced object could be dropped without having dropped or modified the function, leading to misbehavior when the function was used. Note that merely installing this update will not fix the missing dependency entries; to do that, you'd need to **CREATE OR REPLACE** each such function afterwards. If you have functions whose defaults depend on non-built-in objects, doing so is recommended.

- Allow inlining of set-returning SQL functions with multiple OUT parameters (Tom Lane)
- Don't trust deferred-unique indexes for join removal (Tom Lane and Marti Raudsepp)

  A deferred uniqueness constraint might not hold intra-transaction, so assuming that it does could give incorrect query results.

- Make `DatumGetInetP()` unpack inet datums that have a 1-byte header, and add a new macro, `DatumGetInetPP()`, that does not (Heikki Linnakangas)

  This change affects no core code, but might prevent crashes in add-on code that expects `DatumGetInetP()` to produce an unpacked datum as per usual convention.

- Improve locale support in money type's input and output (Tom Lane)

  Aside from not supporting all standard `lc_monetary` formatting options, the input and output functions were inconsistent, meaning there were locales in which dumped money values could not be re-read.

- Don't let `transform_null_equals` affect `CASE foo WHEN NULL ...` constructs (Heikki Linnakangas)

  `transform_null_equals` is only supposed to affect `foo = NULL` expressions written directly by the user, not equality checks generated internally by this form of `CASE`.

- Change foreign-key trigger creation order to better support self-referential foreign keys (Tom Lane)

  For a cascading foreign key that references its own table, a row update will fire both the `ON UPDATE` trigger and the `CHECK` trigger as one event. The `ON UPDATE` trigger must execute first, else the `CHECK` will check a non-final state of the row and possibly throw an inappropriate error. However, the firing order of these triggers is determined by their names, which generally sort in creation order since the triggers have auto-generated names following the convention « RI_ConstraintTrigger_NNNN ». A proper fix would require modifying that convention, which we will do in 9.2, but it seems risky to change it in existing releases. So this patch just changes the creation order of the triggers. Users encountering this type of error should drop and re-create the foreign key constraint to get its triggers into the right order.

- Avoid floating-point underflow while tracking buffer allocation rate (Greg Matthews)

  While harmless in itself, on certain platforms this would result in annoying kernel log messages.

- Preserve configuration file name and line number values when starting child processes under Windows (Tom Lane)

  Formerly, these would not be displayed correctly in the pg_settings view.

- Fix incorrect field alignment in ecpg's SQLDA area (Zoltan Boszormenyi)
- Preserve blank lines within commands in psql's command history (Robert Haas)

  The former behavior could cause problems if an empty line was removed from within a string literal, for example.

- Fix pg_dump to dump user-defined casts between auto-generated types, such as table rowtypes (Tom Lane)
- Assorted fixes for pg_upgrade (Bruce Momjian)

Handle exclusion constraints correctly, avoid failures on Windows, don't complain about mismatched toast table names in 8.4 databases.

- Use the preferred version of xsubpp to build PL/Perl, not necessarily the operating system's main copy (David Wheeler and Alex Hunsaker)
- Fix incorrect coding in `contrib/dict_int` and `contrib/dict_xsyn` (Tom Lane)

  Some functions incorrectly assumed that memory returned by `palloc()` is guaranteed zeroed.
- Fix assorted errors in `contrib/unaccent`'s configuration file parsing (Tom Lane)
- Honor query cancel interrupts promptly in `pgstatindex()` (Robert Haas)
- Fix incorrect quoting of log file name in Mac OS X start script (Sidar Lopez)
- Ensure VPATH builds properly install all server header files (Peter Eisentraut)
- Shorten file names reported in verbose error messages (Peter Eisentraut)

  Regular builds have always reported just the name of the C file containing the error message call, but VPATH builds formerly reported an absolute path name.
- Fix interpretation of Windows timezone names for Central America (Tom Lane)

  Map « Central America Standard Time » to `CST6`, not `CST6CDT`, because DST is generally not observed anywhere in Central America.
- Update time zone data files to tzdata release 2011n for DST law changes in Brazil, Cuba, Fiji, Palestine, Russia, and Samoa; also historical corrections for Alaska and British East Africa.

E.14. Release 9.0.5

Release Date

2011-09-26

This release contains a variety of fixes from 9.0.4. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.14.1. Migration to Version 9.0.5

A dump/restore is not required for those running 9.0.X.

However, if you are upgrading from a version earlier than 9.0.4, see Section E.15, « Release 9.0.4 ».

E.14.2. Changes

- Fix catalog cache invalidation after a **VACUUM FULL** or **CLUSTER** on a system catalog (Tom Lane)

  In some cases the relocation of a system catalog row to another place would not be recognized by concurrent server processes, allowing catalog corruption to occur if they then tried to update that row. The worst-case outcome could be as bad as complete loss of a table.
- Fix incorrect order of operations during sinval reset processing, and ensure that TOAST OIDs are preserved in system catalogs (Tom Lane)

  These mistakes could lead to transient failures after a **VACUUM FULL** or **CLUSTER** on a system catalog.
- Fix bugs in indexing of in-doubt HOT-updated tuples (Tom Lane)

  These bugs could result in index corruption after reindexing a system catalog. They are not believed to affect user indexes.
- Fix multiple bugs in GiST index page split processing (Heikki Linnakangas)

  The probability of occurrence was low, but these could lead to index corruption.
- Fix possible buffer overrun in `tsvector_concat()` (Tom Lane)

  The function could underestimate the amount of memory needed for its result, leading to server crashes.

- Fix crash in `xml_recv` when processing a « standalone » parameter (Tom Lane)
- Make `pg_options_to_table` return NULL for an option with no value (Tom Lane)

  Previously such cases would result in a server crash.
- Avoid possibly accessing off the end of memory in **ANALYZE** and in SJIS-2004 encoding conversion (Noah Misch)

  This fixes some very-low-probability server crash scenarios.
- Protect `pg_stat_reset_shared()` against NULL input (Magnus Hagander)
- Fix possible failure when a recovery conflict deadlock is detected within a sub-transaction (Tom Lane)
- Avoid spurious conflicts while recycling btree index pages during hot standby (Noah Misch, Simon Riggs)
- Shut down WAL receiver if it's still running at end of recovery (Heikki Linnakangas)

  The postmaster formerly panicked in this situation, but it's actually a legitimate case.
- Fix race condition in relcache init file invalidation (Tom Lane)

  There was a window wherein a new backend process could read a stale init file but miss the inval messages that would tell it the data is stale. The result would be bizarre failures in catalog accesses, typically « could not read block 0 in file ... » later during startup.
- Fix memory leak at end of a GiST index scan (Tom Lane)

  Commands that perform many separate GiST index scans, such as verification of a new GiST-based exclusion constraint on a table already containing many rows, could transiently require large amounts of memory due to this leak.
- Fix memory leak when encoding conversion has to be done on incoming command strings and **LISTEN** is active (Tom Lane)
- Fix incorrect memory accounting (leading to possible memory bloat) in tuplestores supporting holdable cursors and plpgsql's `RETURN NEXT` command (Tom Lane)
- Fix trigger `WHEN` conditions when both `BEFORE` and `AFTER` triggers exist (Tom Lane)

  Evaluation of `WHEN` conditions for `AFTER ROW UPDATE` triggers could crash if there had been a `BEFORE ROW` trigger fired for the same update.
- Fix performance problem when constructing a large, lossy bitmap (Tom Lane)
- Fix join selectivity estimation for unique columns (Tom Lane)

  This fixes an erroneous planner heuristic that could lead to poor estimates of the result size of a join.
- Fix nested PlaceHolderVar expressions that appear only in sub-select target lists (Tom Lane)

  This mistake could result in outputs of an outer join incorrectly appearing as NULL.
- Allow the planner to assume that empty parent tables really are empty (Tom Lane)

  Normally an empty table is assumed to have a certain minimum size for planning purposes; but this heuristic seems to do more harm than good for the parent table of an allance hierarchy, which often is permanently empty.
- Allow nested `EXISTS` queries to be optimized properly (Tom Lane)
- Fix array- and path-creating functions to ensure padding bytes are zeroes (Tom Lane)

  This avoids some situations where the planner will think that semantically-equal constants are not equal, resulting in poor optimization.
- Fix **EXPLAIN** to handle gating Result nodes within inner-indexscan subplans (Tom Lane)

  The usual symptom of this oversight was « bogus varno » errors.
- Fix btree preprocessing of *`indexedcol`* `IS NULL` conditions (Dean Rasheed)

  Such a condition is unsatisfiable if combined with any other type of btree-indexable condition on the same index column. The case was handled incorrectly in 9.0.0 and later, leading to query output where there should be none.
- Work around gcc 4.6.0 bug that breaks WAL replay (Tom Lane)

  This could lead to loss of committed transactions after a server crash.
- Fix dump bug for `VALUES` in a view (Tom Lane)

- Disallow `SELECT FOR UPDATE/SHARE` on sequences (Tom Lane)

  This operation doesn't work as expected and can lead to failures.
- Fix **VACUUM** so that it always updates `pg_class.reltuples/relpages` (Tom Lane)

  This fixes some scenarios where autovacuum could make increasingly poor decisions about when to vacuum tables.
- Defend against integer overflow when computing size of a hash table (Tom Lane)
- Fix cases where **CLUSTER** might attempt to access already-removed TOAST data (Tom Lane)
- Fix premature timeout failures during initial authentication transaction (Tom Lane)
- Fix portability bugs in use of credentials control messages for « peer » authentication (Tom Lane)
- Fix SSPI login when multiple roundtrips are required (Ahmed Shinwari, Magnus Hagander)

  The typical symptom of this problem was « The function requested is not supported » errors during SSPI login.
- Fix failure when adding a new variable of a custom variable class to `postgresql.conf` (Tom Lane)
- Throw an error if `pg_hba.conf` contains `hostssl` but SSL is disabled (Tom Lane)

  This was concluded to be more user-friendly than the previous behavior of silently ignoring such lines.
- Fix failure when **DROP OWNED BY** attempts to remove default privileges on sequences (Shigeru Hanada)
- Fix typo in `pg_srand48` seed initialization (Andres Freund)

  This led to failure to use all bits of the provided seed. This function is not used on most platforms (only those without `srandom`), and the potential security exposure from a less-random-than-expected seed seems minimal in any case.
- Avoid integer overflow when the sum of `LIMIT` and `OFFSET` values exceeds 2^63 (Heikki Linnakangas)
- Add overflow checks to int4 and int8 versions of `generate_series()` (Robert Haas)
- Fix trailing-zero removal in `to_char()` (Marti Raudsepp)

  In a format with `FM` and no digit positions after the decimal point, zeroes to the left of the decimal point could be removed incorrectly.
- Fix `pg_size_pretty()` to avoid overflow for inputs close to 2^63 (Tom Lane)
- Weaken plpgsql's check for typmod matching in record values (Tom Lane)

  An overly enthusiastic check could lead to discarding length modifiers that should have been kept.
- Correctly handle quotes in locale names during initdb (Heikki Linnakangas)

  The case can arise with some Windows locales, such as « People's Republic of China ».
- In pg_upgrade, avoid dumping orphaned temporary tables (Bruce Momjian)

  This prevents situations wherein table OID assignments could get out of sync between old and new installations.
- Fix pg_upgrade to preserve toast tables' relfrozenxids during an upgrade from 8.3 (Bruce Momjian)

  Failure to do this could lead to `pg_clog` files being removed too soon after the upgrade.
- In pg_upgrade, fix the `-l` (log) option to work on Windows (Bruce Momjian)
- In pg_ctl, support silent mode for service registrations on Windows (MauMau)
- Fix psql's counting of script file line numbers during `COPY` from a different file (Tom Lane)
- Fix pg_restore's direct-to-database mode for `standard_conforming_strings` (Tom Lane)

  pg_restore could emit incorrect commands when restoring directly to a database server from an archive file that had been made with `standard_conforming_strings` set to `on`.
- Be more user-friendly about unsupported cases for parallel pg_restore (Tom Lane)

  This change ensures that such cases are detected and reported before any restore actions have been taken.
- Fix write-past-buffer-end and memory leak in libpq's LDAP service lookup code (Albe Laurenz)
- In libpq, avoid failures when using nonblocking I/O and an SSL connection (Martin Pihlak, Tom Lane)
- Improve libpq's handling of failures during connection startup (Tom Lane)

In particular, the response to a server report of `fork()` failure during SSL connection startup is now saner.

- Improve libpq's error reporting for SSL failures (Tom Lane)
- Fix `PQsetvalue()` to avoid possible crash when adding a new tuple to a PGresult originally obtained from a server query (Andrew Chernow)
- Make ecpglib write double values with 15 digits precision (Akira Kurosawa)
- In ecpglib, be sure `LC_NUMERIC` setting is restored after an error (Michael Meskes)
- Apply upstream fix for blowfish signed-character bug (CVE-2011-2483) (Tom Lane)

  `contrib/pg_crypto`'s blowfish encryption code could give wrong results on platforms where char is signed (which is most), leading to encrypted passwords being weaker than they should be.
- Fix memory leak in `contrib/seg` (Heikki Linnakangas)
- Fix `pgstatindex()` to give consistent results for empty indexes (Tom Lane)
- Allow building with perl 5.14 (Alex Hunsaker)
- Fix assorted issues with build and install file paths containing spaces (Tom Lane)
- Update time zone data files to tzdata release 2011i for DST law changes in Canada, Egypt, Russia, Samoa, and South Sudan.

E.15. Release 9.0.4

Release Date

2011-04-18

This release contains a variety of fixes from 9.0.3. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.15.1. Migration to Version 9.0.4

A dump/restore is not required for those running 9.0.X.

However, if your installation was upgraded from a previous major release by running pg_upgrade, you should take action to prevent possible data loss due to a now-fixed bug in pg_upgrade. The recommended solution is to run **VACUUM FREEZE** on all TOAST tables. More information is available at *http://wiki.postgresql.org/wiki/20110408pg_upgrade_fix*.

E.15.2. Changes

- Fix pg_upgrade's handling of TOAST tables (Bruce Momjian)

  The pg_class.*`relfrozenxid`* value for TOAST tables was not correctly copied into the new installation during pg_upgrade. This could later result in `pg_clog` files being discarded while they were still needed to validate tuples in the TOAST tables, leading to « could not access status of transaction » failures.

  This error poses a significant risk of data loss for installations that have been upgraded with pg_upgrade. This patch corrects the problem for future uses of pg_upgrade, but does not in itself cure the issue in installations that have been processed with a buggy version of pg_upgrade.
- Suppress incorrect « PD_ALL_VISIBLE flag was incorrectly set » warning (Heikki Linnakangas)

  VACUUM would sometimes issue this warning in cases that are actually valid.
- Use better SQLSTATE error codes for hot standby conflict cases (Tatsuo Ishii and Simon Riggs)

  All retryable conflict errors now have an error code that indicates that a retry is possible. Also, session closure due to the database being dropped on the master is now reported as `ERRCODE_DATABASE_DROPPED`, rather than `ERRCODE_ADMIN_SHUTDOWN`, so that connection poolers can handle the situation correctly.
- Prevent intermittent hang in interactions of startup process with bgwriter process (Simon Riggs)

  This affected recovery in non-hot-standby cases.
- Disallow including a composite type in itself (Tom Lane)

This prevents scenarios wherein the server could recurse infinitely while processing the composite type. While there are some possible uses for such a structure, they don't seem compelling enough to justify the effort required to make sure it always works safely.

- Avoid potential deadlock during catalog cache initialization (Nikhil Sontakke)

  In some cases the cache loading code would acquire share lock on a system index before locking the index's catalog. This could deadlock against processes trying to acquire exclusive locks in the other, more standard order.

- Fix dangling-pointer problem in `BEFORE ROW UPDATE` trigger handling when there was a concurrent update to the target tuple (Tom Lane)

  This bug has been observed to result in intermittent « cannot extract system attribute from virtual tuple » failures while trying to do `UPDATE RETURNING ctid`. There is a very small probability of more serious errors, such as generating incorrect index entries for the updated tuple.

- Disallow **DROP TABLE** when there are pending deferred trigger events for the table (Tom Lane)

  Formerly the **DROP** would go through, leading to « could not open relation with OID nnn » errors when the triggers were eventually fired.

- Allow « replication » as a user name in `pg_hba.conf` (Andrew Dunstan)

  « replication » is special in the database name column, but it was mistakenly also treated as special in the user name column.

- Prevent crash triggered by constant-false WHERE conditions during GEQO optimization (Tom Lane)

- Improve planner's handling of semi-join and anti-join cases (Tom Lane)
- Fix handling of `SELECT FOR UPDATE` in a sub-SELECT (Tom Lane)

  This bug typically led to « cannot extract system attribute from virtual tuple » errors.

- Fix selectivity estimation for text search to account for NULLs (Jesper Krogh)
- Fix get_actual_variable_range() to support hypothetical indexes injected by an index adviser plugin (Gurjeet Singh)
- Fix PL/Python memory leak involving array slices (Daniel Popowich)
- Allow libpq's SSL initialization to succeed when user's home directory is unavailable (Tom Lane)

  If the SSL mode is such that a root certificate file is not required, there is no need to fail. This change restores the behavior to what it was in pre-9.0 releases.

- Fix libpq to return a useful error message for errors detected in `conninfo_array_parse` (Joseph Adams)

  A typo caused the library to return NULL, rather than the PGconn structure containing the error message, to the application.

- Fix ecpg preprocessor's handling of float constants (Heikki Linnakangas)
- Fix parallel pg_restore to handle comments on POST_DATA items correctly (Arnd Hannemann)
- Fix pg_restore to cope with long lines (over 1KB) in TOC files (Tom Lane)
- Put in more safeguards against crashing due to division-by-zero with overly enthusiastic compiler optimization (Aurelien Jarno)
- Support use of dlopen() in FreeBSD and OpenBSD on MIPS (Tom Lane)

  There was a hard-wired assumption that this system function was not available on MIPS hardware on these systems. Use a compile-time test instead, since more recent versions have it.

- Fix compilation failures on HP-UX (Heikki Linnakangas)
- Avoid crash when trying to write to the Windows console very early in process startup (Rushabh Lathia)
- Support building with MinGW 64 bit compiler for Windows (Andrew Dunstan)
- Fix version-incompatibility problem with libintl on Windows (Hiroshi Inoue)
- Fix usage of xcopy in Windows build scripts to work correctly under Windows 7 (Andrew Dunstan)

  This affects the build scripts only, not installation or usage.

- Fix path separator used by pg_regress on Cygwin (Andrew Dunstan)

- Update time zone data files to tzdata release 2011f for DST law changes in Chile, Cuba, Falkland Islands, Morocco, Samoa, and Turkey; also historical corrections for South Australia, Alaska, and Hawaii.

E.16. Release 9.0.3

Release Date

2011-01-31

This release contains a variety of fixes from 9.0.2. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.16.1. Migration to Version 9.0.3

A dump/restore is not required for those running 9.0.X.

E.16.2. Changes

- Before exiting walreceiver, ensure all the received WAL is fsync'd to disk (Heikki Linnakangas)

  Otherwise the standby server could replay some un-synced WAL, conceivably leading to data corruption if the system crashes just at that point.

- Avoid excess fsync activity in walreceiver (Heikki Linnakangas)
- Make **ALTER TABLE** revalidate uniqueness and exclusion constraints when needed (Noah Misch)

  This was broken in 9.0 by a change that was intended to suppress revalidation during **VACUUM FULL** and **CLUSTER**, but unintentionally affected **ALTER TABLE** as well.

- Fix EvalPlanQual for **UPDATE** of an allance tree in which the tables are not all alike (Tom Lane)

  Any variation in the table row types (including dropped columns present in only some child tables) would confuse the EvalPlanQual code, leading to misbehavior or even crashes. Since EvalPlanQual is only executed during concurrent updates to the same row, the problem was only seen intermittently.

- Avoid failures when **EXPLAIN** tries to display a simple-form `CASE` expression (Tom Lane)

  If the `CASE`'s test expression was a constant, the planner could simplify the `CASE` into a form that confused the expression-display code, resulting in « unexpected CASE WHEN clause » errors.

- Fix assignment to an array slice that is before the existing range of subscripts (Tom Lane)

  If there was a gap between the newly added subscripts and the first pre-existing subscript, the code miscalculated how many entries needed to be copied from the old array's null bitmap, potentially leading to data corruption or crash.

- Avoid unexpected conversion overflow in planner for very distant date values (Tom Lane)

  The date type supports a wider range of dates than can be represented by the timestamp types, but the planner assumed it could always convert a date to timestamp with impunity.

- Fix PL/Python crash when an array contains null entries (Alex Hunsaker)
- Remove ecpg's fixed length limit for constants defining an array dimension (Michael Meskes)
- Fix erroneous parsing of tsquery values containing `... & !(subexpression) | ...` (Tom Lane)

  Queries containing this combination of operators were not executed correctly. The same error existed in `contrib/intarray`'s query_int type and `contrib/ltree`'s ltxtquery type.

- Fix buffer overrun in `contrib/intarray`'s input function for the query_int type (Apple)

  This bug is a security risk since the function's return address could be overwritten. Thanks to Apple Inc's security team for reporting this issue and supplying the fix. (CVE-2010-4015)

- Fix bug in `contrib/seg`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a seg column. If you have such an index, consider **REINDEX**ing it after installing this update. (This is identical to the bug that was fixed in `contrib/cube` in the previous update.)

E.17. Release 9.0.2

Release Date

2010-12-16

This release contains a variety of fixes from 9.0.1. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.17.1. Migration to Version 9.0.2

A dump/restore is not required for those running 9.0.X.

E.17.2. Changes

- Force the default `wal_sync_method` to be `fdatasync` on Linux (Tom Lane, Marti Raudsepp)

  The default on Linux has actually been `fdatasync` for many years, but recent kernel changes caused PostgreSQL™ to choose `open_datasync` instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably `ext4` with the `data=journal` mount option.

- Fix « too many KnownAssignedXids » error during Hot Standby replay (Heikki Linnakangas)
- Fix race condition in lock acquisition during Hot Standby (Simon Riggs)
- Avoid unnecessary conflicts during Hot Standby (Simon Riggs)

  This fixes some cases where replay was considered to conflict with standby queries (causing delay of replay or possibly cancellation of the queries), but there was no real conflict.

- Fix assorted bugs in WAL replay logic for GIN indexes (Tom Lane)

  This could result in « bad buffer id: 0 » failures or corruption of index contents during replication.

- Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)
- Fix corner-case bug when streaming replication is enabled immediately after creating the master database cluster (Heikki Linnakangas)
- Fix persistent slowdown of autovacuum workers when multiple workers remain active for a long time (Tom Lane)

  The effective `vacuum_cost_limit` for an autovacuum worker could drop to nearly zero if it processed enough tables, causing it to run extremely slowly.

- Fix long-term memory leak in autovacuum launcher (Alvaro Herrera)
- Avoid failure when trying to report an impending transaction wraparound condition from outside a transaction (Tom Lane)

  This oversight prevented recovery after transaction wraparound got too close, because database startup processing would fail.

- Add support for detecting register-stack overrun on `IA64` (Tom Lane)

  The `IA64` architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.

- Add a check for stack overflow in `copyObject()` (Tom Lane)

  Certain code paths could crash due to stack overflow given a sufficiently complex query.

- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)

  It is possible to have a « concurrent » page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.

- Fix error checking during early connection processing (Tom Lane)

  The check for too many child processes was skipped in some cases, possibly leading to postmaster crash when attempting to add the new child process to fixed-size arrays.

- Improve efficiency of window functions (Tom Lane)

Certain cases where a large number of tuples needed to be read in advance, but `work_mem` was large enough to allow them all to be held in memory, were unexpectedly slow. `percent_rank()`, `cume_dist()` and `ntile()` in particular were subject to this problem.

- Avoid memory leakage while **ANALYZE**'ing complex index expressions (Tom Lane)
- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)

  An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.
- Add missing support in **DROP OWNED BY** for removing foreign data wrapper/server privileges belonging to a user (Heikki Linnakangas)
- Do not « inline » a SQL function with multiple `OUT` parameters (Tom Lane)

  This avoids a possible crash due to loss of information about the expected result rowtype.
- Fix crash when inline-ing a set-returning function whose argument list contains a reference to an inline-able user function (Tom Lane)
- Behave correctly if `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `WITH` is attached to the `VALUES` part of `INSERT ... VALUES` (Tom Lane)
- Make the `OFF` keyword unreserved (Heikki Linnakangas)

  This prevents problems with using `off` as a variable name in PL/pgSQL. That worked before 9.0, but was now broken because PL/pgSQL now treats all core reserved words as reserved.
- Fix constant-folding of `COALESCE()` expressions (Tom Lane)

  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.
- Fix « could not find pathkey item to sort » planner failure with comparison of whole-row Vars (Tom Lane)
- Fix postmaster crash when connection acceptance (`accept()` or one of the calls made immediately after it) fails, and the postmaster was compiled with GSSAPI support (Alexander Chernikov)
- Retry after receiving an invalid response packet from a RADIUS authentication server (Magnus Hagander)

  This fixes a low-risk potential denial of service condition.
- Fix missed unlink of temporary files when `log_temp_files` is active (Tom Lane)

  If an error occurred while attempting to emit the log message, the unlink was not done, resulting in accumulation of temp files.
- Add print functionality for InhRelation nodes (Tom Lane)

  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.
- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)

  This bug affected several different geometric distance-measurement operators.
- Fix incorrect calculation of transaction status in ecpg (Itagaki Takahiro)
- Fix errors in psql's Unicode-escape support (Tom Lane)
- Speed up parallel pg_restore when the archive contains many large objects (blobs) (Tom Lane)
- Fix PL/pgSQL's handling of « simple » expressions to not fail in recursion or error-recovery cases (Tom Lane)
- Fix PL/pgSQL's error reporting for no-such-column cases (Tom Lane)

  As of 9.0, it would sometimes report « missing FROM-clause entry for table foo » when « record foo has no field bar » would be more appropriate.
- Fix PL/Python to honor typmod (i.e., length or precision restrictions) when assigning to tuple fields (Tom Lane)

  This fixes a regression from 8.4.
- Fix PL/Python's handling of set-returning functions (Jan Urbanski)

  Attempts to call SPI functions within the iterator generating a set result would fail.
- Fix bug in `contrib/cube`'s GiST picksplit algorithm (Alexander Korotkov)

This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a cube column. If you have such an index, consider **REINDEX**ing it after installing this update.

- Don't emit « identifier will be truncated » notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)
- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)
- Fix buffer overrun in `contrib/pg_upgrade` (Hernan Gonzalez)
- Fix memory leak in `contrib/xml2`'s XPath query functions (Tom Lane)
- Update time zone data files to tzdata release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.

E.18. Release 9.0.1

Release Date

2010-10-04

This release contains a variety of fixes from 9.0.0. For information about new features in the 9.0 major release, see Section E.19, « Release 9.0 ».

E.18.1. Migration to Version 9.0.1

A dump/restore is not required for those running 9.0.X.

E.18.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).
- Improve `pg_get_expr()` security fix so that the function can still be used on the output of a sub-select (Tom Lane)
- Fix incorrect placement of placeholder evaluation (Tom Lane)

  This bug could result in query outputs being non-null when they should be null, in cases where the inner side of an outer join is a sub-select with non-strict expressions in its output list.
- Fix join removal's handling of placeholder expressions (Tom Lane)
- Fix possible duplicate scans of `UNION ALL` member relations (Tom Lane)
- Prevent infinite loop in ProcessIncomingNotify() after unlistening (Jeff Davis)
- Prevent show_session_authorization() from crashing within autovacuum processes (Tom Lane)
- Re-allow input of Julian dates prior to 0001-01-01 AD (Tom Lane)

  Input such as `'J100000'::date` worked before 8.4, but was unintentionally broken by added error-checking.
- Make psql recognize **DISCARD ALL** as a command that should not be encased in a transaction block in autocommit-off mode (Itagaki Takahiro)

- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)

E.19. Release 9.0

Release Date

2010-09-20

E.19.1. Overview

This release of PostgreSQL™ adds features that have been requested for years, such as easy-to-use replication, a mass permission-changing facility, and anonymous code blocks. While past major releases have been conservative in their scope, this release shows a bold new desire to provide facilities that new and existing users of PostgreSQL™ will embrace. This has all been done with few incompatibilities. Major enhancements include:

- Built-in replication based on log shipping. This advance consists of two features: Streaming Replication, allowing continuous archive (WAL) files to be streamed over a network connection to a standby server, and Hot Standby, allowing continuous archive standby servers to execute read-only queries. The net effect is to support a single master with multiple read-only slave servers.
- Easier database object permissions management. **GRANT/REVOKE IN SCHEMA** supports mass permissions changes on existing objects, while **ALTER DEFAULT PRIVILEGES** allows control of privileges for objects created in the future. Large objects (BLOBs) now support permissions management as well.
- Broadly enhanced stored procedure support. The **DO** statement supports ad-hoc or « anonymous » code blocks. Functions can now be called using named parameters. PL/pgSQL is now installed by default, and PL/Perl and PL/Python have been enhanced in several ways, including support for Python3.
- Full support for 64-bit Windows™.
- More advanced reporting queries, including additional windowing options (`PRECEDING` and `FOLLOWING`) and the ability to control the order in which values are fed to aggregate functions.
- New trigger features, including SQL-standard-compliant per-column triggers and conditional trigger execution.
- Deferrable unique constraints. Mass updates to unique keys are now possible without trickery.
- Exclusion constraints. These provide a generalized version of unique constraints, allowing enforcement of complex conditions.
- New and enhanced security features, including RADIUS authentication, LDAP authentication improvements, and a new contrib module `passwordcheck` for testing password strength.
- New high-performance implementation of the **LISTEN/NOTIFY** feature. Pending events are now stored in a memory-based queue rather than a table. Also, a « payload » string can be sent with each event, rather than transmitting just an event name as before.
- New implementation of **VACUUM FULL**. This command now rewrites the entire table and indexes, rather than moving individual rows to compact space. It is substantially faster in most cases, and no longer results in index bloat.
- New contrib module `pg_upgrade` to support in-place upgrades from 8.3 or 8.4 to 9.0.
- Multiple performance enhancements for specific types of queries, including elimination of unnecessary joins. This helps optimize some automatically-generated queries, such as those produced by object-relational mappers (ORMs).
- **EXPLAIN** enhancements. The output is now available in JSON, XML, or YAML format, and includes buffer utilization and other data not previously available.
- `hstore` improvements, including new functions and greater data capacity.

The above items are explained in more detail in the sections below.

E.19.2. Migration to Version 9.0

A dump/restore using pg_dump, or use of pg_upgrade, is required for those wishing to migrate data from any previous release.

Version 9.0 contains a number of changes that selectively break backwards compatibility in order to support new features and

code quality improvements. In particular, users who make extensive use of PL/pgSQL, Point-In-Time Recovery (PITR), or Warm Standby should test their applications because of slight user-visible changes in those areas. Observe the following incompatibilities:

E.19.2.1. Server Settings

- Remove server parameter `add_missing_from`, which was defaulted to off for many years (Tom Lane)
- Remove server parameter `regex_flavor`, which was defaulted to `advanced` for many years (Tom Lane)
- `archive_mode` now only affects `archive_command`; a new setting, `wal_level`, affects the contents of the write-ahead log (Heikki Linnakangas)
- `log_temp_files` now uses default file size units of kilobytes (Robert Haas)

E.19.2.2. Queries

- When querying a parent table, do not do any separate permission checks on child tables scanned as part of the query (Peter Eisentraut)

  The SQL standard specifies this behavior, and it is also much more convenient in practice than the former behavior of checking permissions on each child as well as the parent.

E.19.2.3. Data Types

- bytea output now appears in hex format by default (Peter Eisentraut)

  The server parameter `bytea_output` can be used to select the traditional output format if needed for compatibility.
- Array input now considers only plain ASCII whitespace characters to be potentially ignorable; it will never ignore non-ASCII characters, even if they are whitespace according to some locales (Tom Lane)

  This avoids some corner cases where array values could be interpreted differently depending on the server's locale settings.
- Improve standards compliance of `SIMILAR TO` patterns and SQL-style `substring()` patterns (Tom Lane)

  This includes treating `?` and `{...}` as pattern metacharacters, while they were simple literal characters before; that corresponds to new features added in SQL:2008. Also, `^` and `$` are now treated as simple literal characters; formerly they were treated as metacharacters, as if the pattern were following POSIX rather than SQL rules. Also, in SQL-standard `substring()`, use of parentheses for nesting no longer interferes with capturing of a substring. Also, processing of bracket expressions (character classes) is now more standards-compliant.
- Reject negative length values in 3-parameter `substring()` for bit strings, per the SQL standard (Tom Lane)
- Make `date_trunc` truncate rather than round when reducing precision of fractional seconds (Tom Lane)

  The code always acted this way for integer-based dates/times. Now float-based dates/times behave similarly.

E.19.2.4. Object Renaming

- Tighten enforcement of column name consistency during **RENAME** when a child table alls the same column from multiple unrelated parents (KaiGai Kohei)
- No longer automatically rename indexes and index columns when the underlying table columns are renamed (Tom Lane)

  Administrators can still rename such indexes and columns manually. This change will require an update of the JDBC driver, and possibly other drivers, so that unique indexes are correctly recognized after a rename.
- **CREATE OR REPLACE FUNCTION** can no longer change the declared names of function parameters (Pavel Stehule)

  In order to avoid creating ambiguity in named-parameter calls, it is no longer allowed to change the aliases for input parameters in the declaration of an existing function (although names can still be assigned to previously unnamed parameters). You now have to **DROP** and recreate the function to do that.

E.19.2.5. PL/pgSQL

- PL/pgSQL now throws an error if a variable name conflicts with a column name used in a query (Tom Lane)

The former behavior was to bind ambiguous names to PL/pgSQL variables in preference to query columns, which often resulted in surprising misbehavior. Throwing an error allows easy detection of ambiguous situations. Although it's recommended that functions encountering this type of error be modified to remove the conflict, the old behavior can be restored if necessary via the configuration parameter `plpgsql.variable_conflict`, or via the per-function option `#variable_conflict`.

- PL/pgSQL no longer allows variable names that match certain SQL reserved words (Tom Lane)

  This is a consequence of aligning the PL/pgSQL parser to match the core SQL parser more closely. If necessary, variable names can be double-quoted to avoid this restriction.

- PL/pgSQL now requires columns of composite results to match the expected type modifier as well as base type (Pavel Stehule, Tom Lane)

  For example, if a column of the result type is declared as `NUMERIC(30,2)`, it is no longer acceptable to return a `NUMERIC` of some other precision in that column. Previous versions neglected to check the type modifier and would thus allow result rows that didn't actually conform to the declared restrictions.

- PL/pgSQL now treats selection into composite fields more consistently (Tom Lane)

  Formerly, a statement like `SELECT ... INTO` *`rec.fld`* `FROM ...` was treated as a scalar assignment even if the record field *`fld`* was of composite type. Now it is treated as a record assignment, the same as when the `INTO` target is a regular variable of composite type. So the values to be assigned to the field's subfields should be written as separate columns of the **SELECT** list, not as a `ROW(...)` construct as in previous versions.

  If you need to do this in a way that will work in both 9.0 and previous releases, you can write something like *`rec.fld`* `:= ROW(...) FROM ....`

- Remove PL/pgSQL's `RENAME` declaration (Tom Lane)

  Instead of `RENAME`, use `ALIAS`, which can now create an alias for any variable, not only dollar sign parameter names (such as `$1`) as before.

E.19.2.6. Other Incompatibilities

- Deprecate use of `=>` as an operator name (Robert Haas)

  Future versions of PostgreSQL™ will probably reject this operator name entirely, in order to support the SQL-standard notation for named function parameters. For the moment, it is still allowed, but a warning is emitted when such an operator is defined.

- Remove support for platforms that don't have a working 64-bit integer data type (Tom Lane)

  It is believed all still-supported platforms have working 64-bit integer data types.

E.19.3. Changes

Version 9.0 has an unprecedented number of new major features, and over 200 enhancements, improvements, new commands, new functions, and other changes.

E.19.3.1. Server

E.19.3.1.1. Continuous Archiving and Streaming Replication

PostgreSQL's existing standby-server capability has been expanded both to support read-only queries on standby servers and to greatly reduce the lag between master and standby servers. For many users, this will be a useful and low-administration form of replication, either for high availability or for horizontal scalability.

- Allow a standby server to accept read-only queries (Simon Riggs, Heikki Linnakangas)

  This feature is called Hot Standby. There are new `postgresql.conf` and `recovery.conf` settings to control this feature, as well as extensive documentation.

- Allow write-ahead log (WAL) data to be streamed to a standby server (Fujii Masao, Heikki Linnakangas)

  This feature is called Streaming Replication. Previously WAL data could be sent to standby servers only in units of entire WAL files (normally 16 megabytes each). Streaming Replication eliminates this inefficiency and allows updates on the master to be propagated to standby servers with very little delay. There are new `postgresql.conf` and `recovery.conf` set-

tings to control this feature, as well as extensive documentation.

- Add `pg_last_xlog_receive_location()` and `pg_last_xlog_replay_location()`, which can be used to monitor standby server WAL activity (Simon Riggs, Fujii Masao, Heikki Linnakangas)

E.19.3.1.2. Performance

- Allow per-tablespace values to be set for sequential and random page cost estimates (`seq_page_cost`/`random_page_cost`) via **ALTER TABLESPACE ... SET/RESET** (Robert Haas)
- Improve performance and reliability of EvalPlanQual rechecks in join queries (Tom Lane)

  UPDATE, **DELETE**, and **SELECT FOR UPDATE/SHARE** queries that involve joins will now behave much better when encountering freshly-updated rows.
- Improve performance of **TRUNCATE** when the table was created or truncated earlier in the same transaction (Tom Lane)
- Improve performance of finding allance child tables (Tom Lane)

E.19.3.1.3. Optimizer

- Remove unnecessary outer joins (Robert Haas)

  Outer joins where the inner side is unique and not referenced above the join are unnecessary and are therefore now removed. This will accelerate many automatically generated queries, such as those created by object-relational mappers (ORMs).
- Allow `IS NOT NULL` restrictions to use indexes (Tom Lane)

  This is particularly useful for finding `MAX()`/`MIN()` values in indexes that contain many null values.
- Improve the optimizer's choices about when to use materialize nodes, and when to use sorting versus hashing for `DISTINCT` (Tom Lane)
- Improve the optimizer's equivalence detection for expressions involving boolean `<>` operators (Tom Lane)

E.19.3.1.4. GEQO

- Use the same random seed every time GEQO plans a query (Andres Freund)

  While the Genetic Query Optimizer (GEQO) still selects random plans, it now always selects the same random plans for identical queries, thus giving more consistent performance. You can modify `geqo_seed` to experiment with alternative plans.
- Improve GEQO plan selection (Tom Lane)

  This avoids the rare error « failed to make a valid plan », and should also improve planning speed.

E.19.3.1.5. Optimizer Statistics

- Improve **ANALYZE** to support allance-tree statistics (Tom Lane)

  This is particularly useful for partitioned tables. However, autovacuum does not yet automatically re-analyze parent tables when child tables change.
- Improve autovacuum's detection of when re-analyze is necessary (Tom Lane)
- Improve optimizer's estimation for greater/less-than comparisons (Tom Lane)

  When looking up statistics for greater/less-than comparisons, if the comparison value is in the first or last histogram bucket, use an index (if available) to fetch the current actual column minimum or maximum. This greatly improves the accuracy of estimates for comparison values near the ends of the data range, particularly if the range is constantly changing due to addition of new data.
- Allow setting of number-of-distinct-values statistics using **ALTER TABLE** (Robert Haas)

  This allows users to override the estimated number or percentage of distinct values for a column. This statistic is normally computed by **ANALYZE**, but the estimate can be poor, especially on tables with very large numbers of rows.

E.19.3.1.6. Authentication

- Add support for RADIUS (Remote Authentication Dial In User Service) authentication (Magnus Hagander)
- Allow LDAP (Lightweight Directory Access Protocol) authentication to operate in « search/bind » mode (Robert Fleming, Magnus Hagander)

  This allows the user to be looked up first, then the system uses the DN (Distinguished Name) returned for that user.
- Add `samehost` and `samenet` designations to `pg_hba.conf` (Stef Walter)

  These match the server's IP address and subnet address respectively.
- Pass trusted SSL root certificate names to the client so the client can return an appropriate client certificate (Craig Ringer)

E.19.3.1.7. Monitoring

- Add the ability for clients to set an application name, which is displayed in pg_stat_activity (Dave Page)

  This allows administrators to characterize database traffic and troubleshoot problems by source application.
- Add a SQLSTATE option (`%e`) to `log_line_prefix` (Guillaume Smet)

  This allows users to compile statistics on errors and messages by error code number.
- Write to the Windows event log in UTF16 encoding (Itagaki Takahiro)

  Now there is true multilingual support for PostgreSQL log messages on Windows.

E.19.3.1.8. Statistics Counters

- Add `pg_stat_reset_shared('bgwriter')` to reset the cluster-wide shared statistics for the background writer (Greg Smith)
- Add `pg_stat_reset_single_table_counters()` and `pg_stat_reset_single_function_counters()` to allow resetting the statistics counters for individual tables and functions (Magnus Hagander)

E.19.3.1.9. Server Settings

- Allow setting of configuration parameters based on database/role combinations (Alvaro Herrera)

  Previously only per-database and per-role settings were possible, not combinations. All role and database settings are now stored in the new pg_db_role_setting system catalog. A new psql command `\drds` shows these settings. The legacy system views pg_roles, pg_shadow, and pg_user do not show combination settings, and therefore no longer completely represent the configuration for a user or database.
- Add server parameter `bonjour`, which controls whether a Bonjour-enabled server advertises itself via Bonjour™ (Tom Lane)

  The default is off, meaning it does not advertise. This allows packagers to distribute Bonjour-enabled builds without worrying that individual users might not want the feature.
- Add server parameter `enable_material`, which controls the use of materialize nodes in the optimizer (Robert Haas)

  The default is on. When off, the optimizer will not add materialize nodes purely for performance reasons, though they will still be used when necessary for correctness.
- Change server parameter `log_temp_files` to use default file size units of kilobytes (Robert Haas)

  Previously this setting was interpreted in bytes if no units were specified.
- Log changes of parameter values when `postgresql.conf` is reloaded (Peter Eisentraut)

  This lets administrators and security staff audit changes of database settings, and is also very convenient for checking the effects of `postgresql.conf` edits.
- Properly enforce superuser permissions for custom server parameters (Tom Lane)

  Non-superusers can no longer issue **ALTER ROLE/DATABASE SET** for parameters that are not currently known to the server. This allows the server to correctly check that superuser-only parameters are only set by superusers. Previously, the `SET` would be allowed and then ignored at session start, making superuser-only custom parameters much less useful than they should be.

E.19.3.2. Queries

- Perform **SELECT FOR UPDATE**/`SHARE` processing after applying `LIMIT`, so the number of rows returned is always predictable (Tom Lane)

  Previously, changes made by concurrent transactions could cause a **SELECT FOR UPDATE** to unexpectedly return fewer rows than specified by its `LIMIT`. `FOR UPDATE` in combination with `ORDER BY` can still produce surprising results, but that can be corrected by placing `FOR UPDATE` in a subquery.
- Allow mixing of traditional and SQL-standard `LIMIT`/`OFFSET` syntax (Tom Lane)
- Extend the supported frame options in window functions (Hitoshi Harada)

  Frames can now start with `CURRENT ROW`, and the `ROWS` *`n`* `PRECEDING`/`FOLLOWING` options are now supported.
- Make **SELECT INTO** and **CREATE TABLE AS** return row counts to the client in their command tags (Boszormenyi Zoltan)

  This can save an entire round-trip to the client, allowing result counts and pagination to be calculated without an additional **COUNT** query.

E.19.3.2.1. Unicode Strings

- Support Unicode surrogate pairs (dual 16-bit representation) in `U&` strings and identifiers (Peter Eisentraut)
- Support Unicode escapes in `E'...'` strings (Marko Kreen)

E.19.3.3. Object Manipulation

- Speed up **CREATE DATABASE** by deferring flushes to disk (Andres Freund, Greg Stark)
- Allow comments on columns of tables, views, and composite types only, not other relation types such as indexes and TOAST tables (Tom Lane)
- Allow the creation of enumerated types containing no values (Bruce Momjian)
- Let values of columns having storage type `MAIN` remain on the main heap page unless the row cannot fit on a page (Kevin Grittner)

  Previously `MAIN` values were forced out to TOAST tables until the row size was less than one-quarter of the page size.

E.19.3.3.1. ALTER TABLE

- Implement `IF EXISTS` for `ALTER TABLE DROP COLUMN` and `ALTER TABLE DROP CONSTRAINT` (Andres Freund)
- Allow **ALTER TABLE** commands that rewrite tables to skip WAL logging (Itagaki Takahiro)

  Such operations either produce a new copy of the table or are rolled back, so WAL archiving can be skipped, unless running in continuous archiving mode. This reduces I/O overhead and improves performance.
- Fix failure of `ALTER TABLE` *`table`* `ADD COLUMN` *`col`* `serial` when done by non-owner of table (Tom Lane)

E.19.3.3.2. CREATE TABLE

- Add support for copying `COMMENTS` and `STORAGE` settings in **CREATE TABLE ... LIKE** commands (Itagaki Takahiro)
- Add a shortcut for copying all properties in **CREATE TABLE ... LIKE** commands (Itagaki Takahiro)
- Add the SQL-standard `CREATE TABLE ... OF` *`type`* command (Peter Eisentraut)

  This allows creation of a table that matches an existing composite type. Additional constraints and defaults can be specified in the command.

E.19.3.3.3. Constraints

- Add deferrable unique constraints (Dean Rasheed)

This allows mass updates, such as `UPDATE tab SET col = col + 1`, to work reliably on columns that have unique indexes or are marked as primary keys. If the constraint is specified as `DEFERRABLE` it will be checked at the end of the statement, rather than after each row is updated. The constraint check can also be deferred until the end of the current transaction, allowing such updates to be spread over multiple SQL commands.

- Add exclusion constraints (Jeff Davis)

  Exclusion constraints generalize uniqueness constraints by allowing arbitrary comparison operators, not just equality. They are created with the **CREATE TABLE CONSTRAINT ... EXCLUDE** clause. The most common use of exclusion constraints is to specify that column entries must not overlap, rather than simply not be equal. This is useful for time periods and other ranges, as well as arrays. This feature enhances checking of data integrity for many calendaring, time-management, and scientific applications.

- Improve uniqueness-constraint violation error messages to report the values causing the failure (Itagaki Takahiro)

  For example, a uniqueness constraint violation might now report `Key (x)=(2) already exists`.

E.19.3.3.4. Object Permissions

- Add the ability to make mass permission changes across a whole schema using the new **GRANT**/**REVOKE IN SCHEMA** clause (Petr Jelinek)

  This simplifies management of object permissions and makes it easier to utilize database roles for application data security.

- Add **ALTER DEFAULT PRIVILEGES** command to control privileges of objects created later (Petr Jelinek)

  This greatly simplifies the assignment of object privileges in a complex database application. Default privileges can be set for tables, views, sequences, and functions. Defaults may be assigned on a per-schema basis, or database-wide.

- Add the ability to control large object (BLOB) permissions with **GRANT**/**REVOKE** (KaiGai Kohei)

  Formerly, any database user could read or modify any large object. Read and write permissions can now be granted and revoked per large object, and the ownership of large objects is tracked.

E.19.3.4. Utility Operations

- Make **LISTEN**/**NOTIFY** store pending events in a memory queue, rather than in a system table (Joachim Wieland)

  This substantially improves performance, while retaining the existing features of transactional support and guaranteed delivery.

- Allow **NOTIFY** to pass an optional « payload » string to listeners (Joachim Wieland)

  This greatly improves the usefulness of **LISTEN**/**NOTIFY** as a general-purpose event queue system.

- Allow **CLUSTER** on all per-database system catalogs (Tom Lane)

  Shared catalogs still cannot be clustered.

E.19.3.4.1. COPY

- Accept `COPY ... CSV FORCE QUOTE *` (Itagaki Takahiro)

  Now `*` can be used as shorthand for « all columns » in the `FORCE QUOTE` clause.

- Add new **COPY** syntax that allows options to be specified inside parentheses (Robert Haas, Emmanuel Cecchet)

  This allows greater flexibility for future **COPY** options. The old syntax is still supported, but only for pre-existing options.

E.19.3.4.2. EXPLAIN

- Allow **EXPLAIN** to output in XML, JSON, or YAML format (Robert Haas, Greg Sabino Mullane)

  The new output formats are easily machine-readable, supporting the development of new tools for analysis of **EXPLAIN** output.

- Add new `BUFFERS` option to report query buffer usage during **EXPLAIN ANALYZE** (Itagaki Takahiro)

  This allows better query profiling for individual queries. Buffer usage is no longer reported in the output for

log_statement_stats and related settings.

- Add hash usage information to **EXPLAIN** output (Robert Haas)
- Add new **EXPLAIN** syntax that allows options to be specified inside parentheses (Robert Haas)

  This allows greater flexibility for future **EXPLAIN** options. The old syntax is still supported, but only for pre-existing options.

E.19.3.4.3. VACUUM

- Change **VACUUM FULL** to rewrite the entire table and rebuild its indexes, rather than moving individual rows around to compact space (Itagaki Takahiro, Tom Lane)

  The previous method was usually slower and caused index bloat. Note that the new method will use more disk space transiently during **VACUUM FULL**; potentially as much as twice the space normally occupied by the table and its indexes.
- Add new **VACUUM** syntax that allows options to be specified inside parentheses (Itagaki Takahiro)

  This allows greater flexibility for future **VACUUM** options. The old syntax is still supported, but only for pre-existing options.

E.19.3.4.4. Indexes

- Allow an index to be named automatically by omitting the index name in **CREATE INDEX** (Tom Lane)
- By default, multicolumn indexes are now named after all their columns; and index expression columns are now named based on their expressions (Tom Lane)
- Reindexing shared system catalogs is now fully transactional and crash-safe (Tom Lane)

  Formerly, reindexing a shared index was only allowed in standalone mode, and a crash during the operation could leave the index in worse condition than it was before.
- Add `point_ops` operator class for GiST (Teodor Sigaev)

  This feature permits GiST indexing of point columns. The index can be used for several types of queries such as *`point <@ polygon`* (point is in polygon). This should make many PostGIS™ queries faster.
- Use red-black binary trees for GIN index creation (Teodor Sigaev)

  Red-black trees are self-balancing. This avoids slowdowns in cases where the input is in nonrandom order.

E.19.3.5. Data Types

- Allow bytea values to be written in hex notation (Peter Eisentraut)

  The server parameter `bytea_output` controls whether hex or traditional format is used for bytea output. Libpq's `PQescapeByteaConn()` function automatically uses the hex format when connected to PostgreSQL™ 9.0 or newer servers. However, pre-9.0 libpq versions will not correctly process hex format from newer servers.

  The new hex format will be directly compatible with more applications that use binary data, allowing them to store and retrieve it without extra conversion. It is also significantly faster to read and write than the traditional format.
- Allow server parameter extra_float_digits to be increased to `3` (Tom Lane)

  The previous maximum `extra_float_digits` setting was `2`. There are cases where 3 digits are needed to dump and restore float4 values exactly. pg_dump will now use the setting of 3 when dumping from a server that allows it.
- Tighten input checking for int2vector values (Caleb Welton)

E.19.3.5.1. Full Text Search

- Add prefix support in `synonym` dictionaries (Teodor Sigaev)
- Add *filtering* dictionaries (Teodor Sigaev)

  Filtering dictionaries allow tokens to be modified then passed to subsequent dictionaries.
- Allow underscores in email-address tokens (Teodor Sigaev)

- Use more standards-compliant rules for parsing URL tokens (Tom Lane)

E.19.3.6. Functions

- Allow function calls to supply parameter names and match them to named parameters in the function definition (Pavel Stehule)

  For example, if a function is defined to take parameters `a` and `b`, it can be called with `func(a := 7, b := 12)` or `func(b := 12, a := 7)`.
- Support locale-specific regular expression processing with UTF-8 server encoding (Tom Lane)

  Locale-specific regular expression functionality includes case-insensitive matching and locale-specific character classes. Previously, these features worked correctly for non-ASCII characters only if the database used a single-byte server encoding (such as LATIN1). They will still misbehave in multi-byte encodings other than UTF-8.
- Add support for scientific notation in `to_char()` (`EEEE` specification) (Pavel Stehule, Brendan Jurd)
- Make `to_char()` honor `FM` (fill mode) in `Y`, `YY`, and `YYY` specifications (Bruce Momjian, Tom Lane)

  It was already honored by `YYYY`.
- Fix `to_char()` to output localized numeric and monetary strings in the correct encoding on Windows™ (Hiroshi Inoue, Itagaki Takahiro, Bruce Momjian)
- Correct calculations of « overlaps » and « contains » operations for polygons (Teodor Sigaev)

  The polygon `&&` (overlaps) operator formerly just checked to see if the two polygons' bounding boxes overlapped. It now does a more correct check. The polygon `@>` and `<@` (contains/contained by) operators formerly checked to see if one polygon's vertexes were all contained in the other; this can wrongly report « true » for some non-convex polygons. Now they check that all line segments of one polygon are contained in the other.

E.19.3.6.1. Aggregates

- Allow aggregate functions to use `ORDER BY` (Andrew Gierth)

  For example, this is now supported: `array_agg(a ORDER BY b)`. This is useful with aggregates for which the order of input values is significant, and eliminates the need to use a nonstandard subquery to determine the ordering.
- Multi-argument aggregate functions can now use `DISTINCT` (Andrew Gierth)
- Add the `string_agg()` aggregate function to combine values into a single string (Pavel Stehule)
- Aggregate functions that are called with `DISTINCT` are now passed NULL values if the aggregate transition function is not marked as `STRICT` (Andrew Gierth)

  For example, `agg(DISTINCT x)` might pass a NULL `x` value to `agg()`. This is more consistent with the behavior in non-`DISTINCT` cases.

E.19.3.6.2. Bit Strings

- Add `get_bit()` and `set_bit()` functions for bit strings, mirroring those for bytea (Leonardo F)
- Implement `OVERLAY()` (replace) for bit strings and bytea (Leonardo F)

E.19.3.6.3. Object Information Functions

- Add `pg_table_size()` and `pg_indexes_size()` to provide a more user-friendly interface to the `pg_relation_size()` function (Bernd Helmle)
- Add `has_sequence_privilege()` for sequence permission checking (Abhijit Menon-Sen)
- Update the information_schema views to conform to SQL:2008 (Peter Eisentraut)
- Make the `information_schema` views correctly display maximum octet lengths for char and varchar columns (Peter Eisentraut)
- Speed up `information_schema` privilege views (Joachim Wieland)

E.19.3.6.4. Function and Trigger Creation

- Support execution of anonymous code blocks using the **DO** statement (Petr Jelinek, Joshua Tolley, Hannu Valtonen)

  This allows execution of server-side code without the need to create and delete a temporary function definition. Code can be executed in any language for which the user has permissions to define a function.
- Implement SQL-standard-compliant per-column triggers (Itagaki Takahiro)

  Such triggers are fired only when the specified column(s) are affected by the query, e.g. appear in an **UPDATE**'s `SET` list.
- Add the `WHEN` clause to **CREATE TRIGGER** to allow control over whether a trigger is fired (Itagaki Takahiro)

  While the same type of check can always be performed inside the trigger, doing it in an external `WHEN` clause can have performance benefits.

E.19.3.7. Server-Side Languages

- Add the `OR REPLACE` clause to **CREATE LANGUAGE** (Tom Lane)

  This is helpful to optionally install a language if it does not already exist, and is particularly helpful now that PL/pgSQL is installed by default.

E.19.3.7.1. PL/pgSQL Server-Side Language

- Install PL/pgSQL by default (Bruce Momjian)

  The language can still be removed from a particular database if the administrator has security or performance concerns about making it available.
- Improve handling of cases where PL/pgSQL variable names conflict with identifiers used in queries within a function (Tom Lane)

  The default behavior is now to throw an error when there is a conflict, so as to avoid surprising behaviors. This can be modified, via the configuration parameter `plpgsql.variable_conflict` or the per-function option `#variable_conflict`, to allow either the variable or the query-supplied column to be used. In any case PL/pgSQL will no longer attempt to substitute variables in places where they would not be syntactically valid.
- Make PL/pgSQL use the main lexer, rather than its own version (Tom Lane)

  This ensures accurate tracking of the main system's behavior for details such as string escaping. Some user-visible details, such as the set of keywords considered reserved in PL/pgSQL, have changed in consequence.
- Avoid throwing an unnecessary error for an invalid record reference (Tom Lane)

  An error is now thrown only if the reference is actually fetched, rather than whenever the enclosing expression is reached. For example, many people have tried to do this in triggers:

  ```
  if TG_OP = 'INSERT' and NEW.col1 = ... then
  ```

  This will now actually work as expected.
- Improve PL/pgSQL's ability to handle row types with dropped columns (Pavel Stehule)
- Allow input parameters to be assigned values within PL/pgSQL functions (Steve Prentice)

  Formerly, input parameters were treated as being declared `CONST`, so the function's code could not change their values. This restriction has been removed to simplify porting of functions from other DBMSes that do not impose the equivalent restriction. An input parameter now acts like a local variable initialized to the passed-in value.
- Improve error location reporting in PL/pgSQL (Tom Lane)
- Add *`count`* and `ALL` options to **MOVE FORWARD**/`BACKWARD` in PL/pgSQL (Pavel Stehule)
- Allow PL/pgSQL's `WHERE CURRENT OF` to use a cursor variable (Tom Lane)
- Allow PL/pgSQL's **OPEN *`cursor`* FOR EXECUTE** to use parameters (Pavel Stehule, Itagaki Takahiro)

  This is accomplished with a new `USING` clause.

E.19.3.7.2. PL/Perl Server-Side Language

- Add new PL/Perl functions: `quote_literal()`, `quote_nullable()`, `quote_ident()`, `encode_bytea()`, `decode_bytea()`, `looks_like_number()`, `encode_array_literal()`, `encode_array_constructor()` (Tim Bunce)
- Add server parameter `plperl.on_init` to specify a PL/Perl initialization function (Tim Bunce)

  `plperl.on_plperl_init` and `plperl.on_plperlu_init` are also available for initialization that is specific to the trusted or untrusted language respectively.
- Support **END** blocks in PL/Perl (Tim Bunce)

  END blocks do not currently allow database access.
- Allow **use strict** in PL/Perl (Tim Bunce)

  Perl `strict` checks can also be globally enabled with the new server parameter `plperl.use_strict`.
- Allow **require** in PL/Perl (Tim Bunce)

  This basically tests to see if the module is loaded, and if not, generates an error. It will not allow loading of modules that the administrator has not preloaded via the initialization parameters.
- Allow **use feature** in PL/Perl if Perl version 5.10 or later is used (Tim Bunce)
- Verify that PL/Perl return values are valid in the server encoding (Andrew Dunstan)

E.19.3.7.3. PL/Python Server-Side Language

- Add Unicode support in PL/Python (Peter Eisentraut)

  Strings are automatically converted from/to the server encoding as necessary.
- Improve bytea support in PL/Python (Caleb Welton)

  Bytea values passed into PL/Python are now represented as binary, rather than the PostgreSQL bytea text format. Bytea values containing null bytes are now also output properly from PL/Python. Passing of boolean, integer, and float values was also improved.
- Support arrays as parameters and return values in PL/Python (Peter Eisentraut)
- Improve mapping of SQL domains to Python types (Peter Eisentraut)
- Add Python 3 support to PL/Python (Peter Eisentraut)

  The new server-side language is called `plpython3u`. This cannot be used in the same session with the Python 2 server-side language.
- Improve error location and exception reporting in PL/Python (Peter Eisentraut)

E.19.3.8. Client Applications

- Add an `--analyze-only` option to **vacuumdb**, to analyze without vacuuming (Bruce Momjian)

E.19.3.8.1. psql

- Add support for quoting/escaping the values of psql variables as SQL strings or identifiers (Pavel Stehule, Robert Haas)

  For example, `:'var'` will produce the value of `var` quoted and properly escaped as a literal string, while `:"var"` will produce its value quoted and escaped as an identifier.
- Ignore a leading UTF-8-encoded Unicode byte-order marker in script files read by psql (Itagaki Takahiro)

  This is enabled when the client encoding is UTF-8. It improves compatibility with certain editors, mostly on Windows, that insist on inserting such markers.
- Fix **psql --file -** to properly honor `--single-transaction` (Bruce Momjian)
- Avoid overwriting of psql's command-line history when two psql sessions are run concurrently (Tom Lane)

- Improve psql's tab completion support (Itagaki Takahiro)
- Show `\timing` output when it is enabled, regardless of « quiet » mode (Peter Eisentraut)

E.19.3.8.1.1. psql Display

- Improve display of wrapped columns in psql (Roger Leigh)

  This behavior is now the default. The previous formatting is available by using **\pset linestyle old-ascii**.
- Allow psql to use fancy Unicode line-drawing characters via **\pset linestyle unicode** (Roger Leigh)

E.19.3.8.1.2. psql \d Commands

- Make **\d** show child tables that all from the specified parent (Damien Clochard)

  \d shows only the number of child tables, while **\d+** shows the names of all child tables.
- Show definitions of index columns in **\d index_name** (Khee Chin)

  The definition is useful for expression indexes.
- Show a view's defining query only in **\d+**, not in **\d** (Peter Eisentraut)

  Always including the query was deemed overly verbose.

E.19.3.8.2. pg_dump

- Make pg_dump/pg_restore `--clean` also remove large objects (Itagaki Takahiro)
- Fix pg_dump to properly dump large objects when `standard_conforming_strings` is enabled (Tom Lane)

  The previous coding could fail when dumping to an archive file and then generating script output from pg_restore.
- pg_restore now emits large-object data in hex format when generating script output (Tom Lane)

  This could cause compatibility problems if the script is then loaded into a pre-9.0 server. To work around that, restore directly to the server, instead.
- Allow pg_dump to dump comments attached to columns of composite types (Taro Minowa (Higepon))
- Make pg_dump `--verbose` output the pg_dump and server versions in text output mode (Jim Cox, Tom Lane)

  These were already provided in custom output mode.
- pg_restore now complains if any command-line arguments remain after the switches and optional file name (Tom Lane)

  Previously, it silently ignored any such arguments.

E.19.3.8.3. pg_ctl

- Allow pg_ctl to be used safely to start the postmaster during a system reboot (Tom Lane)

  Previously, pg_ctl's parent process could have been mistakenly identified as a running postmaster based on a stale postmaster lock file, resulting in a transient failure to start the database.
- Give pg_ctl the ability to initialize the database (by invoking initdb) (Zdenek Kotala)

E.19.3.9. Development Tools

E.19.3.9.1. libpq

- Add new libpq functions `PQconnectdbParams()` and `PQconnectStartParams()` (Guillaume Lelarge)

  These functions are similar to `PQconnectdb()` and `PQconnectStart()` except that they accept a null-terminated array of connection options, rather than requiring all options to be provided in a single string.
- Add libpq functions `PQescapeLiteral()` and `PQescapeIdentifier()` (Robert Haas)

  These functions return appropriately quoted and escaped SQL string literals and identifiers. The caller is not required to pre-

allocate the string result, as is required by `PQescapeStringConn()`.

- Add support for a per-user service file (`.pg_service.conf`), which is checked before the site-wide service file (Peter Eisentraut)
- Properly report an error if the specified libpq service cannot be found (Peter Eisentraut)
- Add TCP keepalive settings in libpq (Tollef Fog Heen, Fujii Masao, Robert Haas)

  Keepalive settings were already supported on the server end of TCP connections.
- Avoid extra system calls to block and unblock `SIGPIPE` in libpq, on platforms that offer alternative methods (Jeremy Kerr)
- When a `.pgpass`-supplied password fails, mention where the password came from in the error message (Bruce Momjian)
- Load all SSL certificates given in the client certificate file (Tom Lane)

  This improves support for indirectly-signed SSL certificates.

E.19.3.9.2. ecpg

- Add SQLDA (SQL Descriptor Area) support to ecpg (Boszormenyi Zoltan)
- Add the **DESCRIBE** [`OUTPUT`] statement to ecpg (Boszormenyi Zoltan)
- Add an ECPGtransactionStatus function to return the current transaction status (Bernd Helmle)
- Add the `string` data type in ecpg Informix-compatibility mode (Boszormenyi Zoltan)
- Allow ecpg to use `new` and `old` variable names without restriction (Michael Meskes)
- Allow ecpg to use variable names in `free()` (Michael Meskes)
- Make `ecpg_dynamic_type()` return zero for non-SQL3 data types (Michael Meskes)

  Previously it returned the negative of the data type OID. This could be confused with valid type OIDs, however.
- Support long long types on platforms that already have 64-bit long (Michael Meskes)

E.19.3.9.2.1. ecpg Cursors

- Add out-of-scope cursor support in ecpg's native mode (Boszormenyi Zoltan)

  This allows **DECLARE** to use variables that are not in scope when **OPEN** is called. This facility already existed in ecpg's Informix-compatibility mode.
- Allow dynamic cursor names in ecpg (Boszormenyi Zoltan)
- Allow ecpg to use noise words `FROM` and `IN` in **FETCH** and **MOVE** (Boszormenyi Zoltan)

E.19.3.10. Build Options

- Enable client thread safety by default (Bruce Momjian)

  The thread-safety option can be disabled with `configure --disable-thread-safety`.
- Add support for controlling the Linux out-of-memory killer (Alex Hunsaker, Tom Lane)

  Now that `/proc/self/oom_adj` allows disabling of the Linux™ out-of-memory (OOM) killer, it's recommendable to disable OOM kills for the postmaster. It may then be desirable to re-enable OOM kills for the postmaster's child processes. The new compile-time option `LINUX_OOM_ADJ` allows the killer to be reactivated for child processes.

E.19.3.10.1. Makefiles

- New `Makefile` targets `world`, `install-world`, and `installcheck-world` (Andrew Dunstan)

  These are similar to the existing `all`, `install`, and `installcheck` targets, but they also build the HTML documentation, build and test `contrib`, and test server-side languages and ecpg.
- Add data and documentation installation location control to PGXS Makefiles (Mark Cave-Ayland)

- Add Makefile rules to build the PostgreSQL™ documentation as a single HTML file or as a single plain-text file (Peter Eisentraut, Bruce Momjian)

E.19.3.10.2. Windows

- Support compiling on 64-bit Windows™ and running in 64-bit mode (Tsutomu Yamada, Magnus Hagander)

  This allows for large shared memory sizes on Windows™.
- Support server builds using Visual Studio 2008™ (Magnus Hagander)

E.19.3.11. Source Code

- Distribute prebuilt documentation in a subdirectory tree, rather than as tar archive files inside the distribution tarball (Peter Eisentraut)

  For example, the prebuilt HTML documentation is now in `doc/src/sgml/html/`; the manual pages are packaged similarly.
- Make the server's lexer reentrant (Tom Lane)

  This was needed for use of the lexer by PL/pgSQL.
- Improve speed of memory allocation (Tom Lane, Greg Stark)
- User-defined constraint triggers now have entries in pg_constraint as well as pg_trigger (Tom Lane)

  Because of this change, pg_constraint.*`pgconstrname`* is now redundant and has been removed.
- Add system catalog columns pg_constraint.*`conindid`* and pg_trigger.*`tgconstrindid`* to better document the use of indexes for constraint enforcement (Tom Lane)
- Allow multiple conditions to be communicated to backends using a single operating system signal (Fujii Masao)

  This allows new features to be added without a platform-specific constraint on the number of signal conditions.
- Improve source code test coverage, including `contrib`, PL/Python, and PL/Perl (Peter Eisentraut, Andrew Dunstan)
- Remove the use of flat files for system table bootstrapping (Tom Lane, Alvaro Herrera)

  This improves performance when using many roles or databases, and eliminates some possible failure conditions.
- Automatically generate the initial contents of pg_attribute for « bootstrapped » catalogs (John Naylor)

  This greatly simplifies changes to these catalogs.
- Split the processing of **INSERT**/**UPDATE**/**DELETE** operations out of `execMain.c` (Marko Tiikkaja)

  Updates are now executed in a separate ModifyTable node. This change is necessary infrastructure for future improvements.
- Simplify translation of psql's SQL help text (Peter Eisentraut)
- Reduce the lengths of some file names so that all file paths in the distribution tarball are less than 100 characters (Tom Lane)

  Some decompression programs have problems with longer file paths.
- Add a new `ERRCODE_INVALID_PASSWORD` `SQLSTATE` error code (Bruce Momjian)
- With authors' permissions, remove the few remaining personal source code copyright notices (Bruce Momjian)

  The personal copyright notices were insignificant but the community occasionally had to answer questions about them.
- Add new documentation section about running PostgreSQL™ in non-durable mode to improve performance (Bruce Momjian)
- Restructure the HTML documentation `Makefile` rules to make their dependency checks work correctly, avoiding unnecessary rebuilds (Peter Eisentraut)
- Use DocBook™ XSL stylesheets for man page building, rather than Docbook2X™ (Peter Eisentraut)

  This changes the set of tools needed to build the man pages.
- Improve PL/Perl code structure (Tim Bunce)
- Improve error context reports in PL/Perl (Alexey Klyukin)

E.19.3.11.1. New Build Requirements

Note that these requirements do not apply when building from a distribution tarball, since tarballs include the files that these programs are used to build.

- Require Autoconf 2.63 to build configure (Peter Eisentraut)
- Require Flex 2.5.31 or later to build from a CVS checkout (Tom Lane)
- Require Perl version 5.8 or later to build from a CVS checkout (John Naylor, Andrew Dunstan)

E.19.3.11.2. Portability

- Use a more modern API for Bonjour (Tom Lane)

  Bonjour support now requires OS X™ 10.3 or later. The older API has been deprecated by Apple.
- Add spinlock support for the SuperH™ architecture (Nobuhiro Iwamatsu)
- Allow non-GCC compilers to use inline functions if they support them (Kurt Harriman)
- Remove support for platforms that don't have a working 64-bit integer data type (Tom Lane)
- Restructure use of `LDFLAGS` to be more consistent across platforms (Tom Lane)

  `LDFLAGS` is now used for linking both executables and shared libraries, and we add on `LDFLAGS_EX` when linking executables, or `LDFLAGS_SL` when linking shared libraries.

E.19.3.11.3. Server Programming

- Make backend header files safe to include in C++™ (Kurt Harriman, Peter Eisentraut)

  These changes remove keyword conflicts that previously made C++™ usage difficult in backend code. However, there are still other complexities when using C++™ for backend functions. `extern "C" { }` is still necessary in appropriate places, and memory management and error handling are still problematic.
- Add `AggCheckCallContext()` for use in detecting if a C™ function is being called as an aggregate (Hitoshi Harada)
- Change calling convention for `SearchSysCache()` and related functions to avoid hard-wiring the maximum number of cache keys (Robert Haas)

  Existing calls will still work for the moment, but can be expected to break in 9.1 or later if not converted to the new style.
- Require calls of `fastgetattr()` and `heap_getattr()` backend macros to provide a non-NULL fourth argument (Robert Haas)
- Custom typanalyze functions should no longer rely on VacAttrStats.*`attr`* to determine the type of data they will be passed (Tom Lane)

  This was changed to allow collection of statistics on index columns for which the storage type is different from the underlying column data type. There are new fields that tell the actual datatype being analyzed.

E.19.3.11.4. Server Hooks

- Add parser hooks for processing ColumnRef and ParamRef nodes (Tom Lane)
- Add a ProcessUtility hook so loadable modules can control utility commands (Itagaki Takahiro)

E.19.3.11.5. Binary Upgrade Support

- Add `contrib/pg_upgrade` to support in-place upgrades (Bruce Momjian)

  This avoids the requirement of dumping/reloading the database when upgrading to a new major release of PostgreSQL, thus reducing downtime by orders of magnitude. It supports upgrades to 9.0 from PostgreSQL 8.3 and 8.4.
- Add support for preserving relation relfilenode values during binary upgrades (Bruce Momjian)
- Add support for preserving pg_type and pg_enum OIDs during binary upgrades (Bruce Momjian)

- Move data files within tablespaces into PostgreSQL™-version-specific subdirectories (Bruce Momjian)

  This simplifies binary upgrades.

E.19.3.12. Contrib

- Add multithreading option (`-j`) to `contrib/pgbench` (Itagaki Takahiro)

  This allows multiple CPUs to be used by pgbench, reducing the risk of pgbench itself becoming the test bottleneck.
- Add **\shell** and **\setshell** meta commands to `contrib/pgbench` (Michael Paquier)
- New features for `contrib/dict_xsyn` (Sergey Karpov)

  The new options are `matchorig`, `matchsynonyms`, and `keepsynonyms`.
- Add full text dictionary `contrib/unaccent` (Teodor Sigaev)

  This filtering dictionary removes accents from letters, which makes full-text searches over multiple languages much easier.
- Add `dblink_get_notify()` to `contrib/dblink` (Marcus Kempe)

  This allows asynchronous notifications in dblink™.
- Improve `contrib/dblink`'s handling of dropped columns (Tom Lane)

  This affects `dblink_build_sql_insert()` and related functions. These functions now number columns according to logical not physical column numbers.
- Greatly increase `contrib/hstore`'s data length limit, and add B-tree and hash support so `GROUP BY` and `DISTINCT` operations are possible on hstore columns (Andrew Gierth)

  New functions and operators were also added. These improvements make hstore a full-function key-value store embedded in PostgreSQL™.
- Add `contrib/passwordcheck` to support site-specific password strength policies (Laurenz Albe)

  The source code of this module should be modified to implement site-specific password policies.
- Add `contrib/pg_archivecleanup` tool (Simon Riggs)

  This is designed to be used in the `archive_cleanup_command` server parameter, to remove no-longer-needed archive files.
- Add query text to `contrib/auto_explain` output (Andrew Dunstan)
- Add buffer access counters to `contrib/pg_stat_statements` (Itagaki Takahiro)
- Update `contrib/start-scripts/linux` to use `/proc/self/oom_adj` to disable the Linux™ out-of-memory (OOM) killer (Alex Hunsaker, Tom Lane)

E.20. Release 8.4.22

Release Date

2014-07-24

This release contains a variety of fixes from 8.4.21. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

This is expected to be the last PostgreSQL™ release in the 8.4.X series. Users are encouraged to update to a newer release branch soon.

E.20.1. Migration to Version 8.4.22

A dump/restore is not required for those running 8.4.X.

However, this release corrects an index corruption problem in some GiST indexes. See the first changelog entry below to find out whether your installation has been affected and what steps you should take if so.

Also, if you are upgrading from a version earlier than 8.4.19, see Section E.23, « Release 8.4.19 ».

E.20.2. Changes

- Correctly initialize padding bytes in `contrib/btree_gist` indexes on bit columns (Heikki Linnakangas)

  This error could result in incorrect query results due to values that should compare equal not being seen as equal. Users with GiST indexes on bit or bit varying columns should **REINDEX** those indexes after installing this update.

- Protect against torn pages when deleting GIN list pages (Heikki Linnakangas)

  This fix prevents possible index corruption if a system crash occurs while the page update is being written to disk.

- Fix possibly-incorrect cache invalidation during nested calls to `ReceiveSharedInvalidMessages` (Andres Freund)

- Don't assume a subquery's output is unique if there's a set-returning function in its targetlist (David Rowley)

  This oversight could lead to misoptimization of constructs like `WHERE x IN (SELECT y, generate_series(1,10) FROM t GROUP BY y)`.

- Fix failure to detoast fields in composite elements of structured types (Tom Lane)

  This corrects cases where TOAST pointers could be copied into other tables without being dereferenced. If the original data is later deleted, it would lead to errors like « missing chunk number 0 for toast value ... » when the now-dangling pointer is used.

- Fix « record type has not been registered » failures with whole-row references to the output of Append plan nodes (Tom Lane)

- Fix possible crash when invoking a user-defined function while rewinding a cursor (Tom Lane)

- Fix query-lifespan memory leak while evaluating the arguments for a function in `FROM` (Tom Lane)

- Fix session-lifespan memory leaks in regular-expression processing (Tom Lane, Arthur O'Dwyer, Greg Stark)

- Fix data encoding error in `hungarian.stop` (Tom Lane)

- Fix liveness checks for rows that were inserted in the current transaction and then deleted by a now-rolled-back subtransaction (Andres Freund)

  This could cause problems (at least spurious warnings, and at worst an infinite loop) if **CREATE INDEX** or **CLUSTER** were done later in the same transaction.

- Clear pg_stat_activity.*`xact_start`* during **PREPARE TRANSACTION** (Andres Freund)

  After the **PREPARE**, the originating session is no longer in a transaction, so it should not continue to display a transaction start time.

- Fix **REASSIGN OWNED** to not fail for text search objects (Álvaro Herrera)

- Block signals during postmaster startup (Tom Lane)

  This ensures that the postmaster will properly clean up after itself if, for example, it receives `SIGINT` while still starting up.

- Secure Unix-domain sockets of temporary postmasters started during `make check` (Noah Misch)

  Any local user able to access the socket file could connect as the server's bootstrap superuser, then proceed to execute arbitrary code as the operating-system user running the test, as we previously noted in CVE-2014-0067. This change defends against that risk by placing the server's socket in a temporary, mode 0700 subdirectory of `/tmp`. The hazard remains however on platforms where Unix sockets are not supported, notably Windows, because then the temporary postmaster must accept local TCP connections.

  A useful side effect of this change is to simplify `make check` testing in builds that override `DEFAULT_PGSOCKET_DIR`. Popular non-default values like `/var/run/postgresql` are often not writable by the build user, requiring workarounds that will no longer be necessary.

- On Windows, allow new sessions to absorb values of PGC_BACKEND parameters (such as log_connections) from the configuration file (Amit Kapila)

  Previously, if such a parameter were changed in the file post-startup, the change would have no effect.

- Properly quote executable path names on Windows (Nikhil Deshpande)

  This oversight could cause initdb and pg_upgrade to fail on Windows, if the installation path contained both spaces and `@` signs.

- Fix linking of libpython on OS X (Tom Lane)

  The method we previously used can fail with the Python library supplied by Xcode 5.0 and later.

- Avoid buffer bloat in libpq when the server consistently sends data faster than the client can absorb it (Shin-ichi Morita, Tom Lane)

  libpq could be coerced into enlarging its input buffer until it runs out of memory (which would be reported misleadingly as « lost synchronization with server »). Under ordinary circumstances it's quite far-fetched that data could be continuously transmitted more quickly than the `recv()` loop can absorb it, but this has been observed when the client is artificially slowed by scheduler constraints.
- Ensure that LDAP lookup attempts in libpq time out as intended (Laurenz Albe)
- Fix pg_restore's processing of old-style large object comments (Tom Lane)

  A direct-to-database restore from an archive file generated by a pre-9.0 version of pg_dump would usually fail if the archive contained more than a few comments for large objects.
- In `contrib/pgcrypto` functions, ensure sensitive information is cleared from stack variables before returning (Marko Kreen)
- In `contrib/uuid-ossp`, cache the state of the OSSP UUID library across calls (Tom Lane)

  This improves the efficiency of UUID generation and reduces the amount of entropy drawn from `/dev/urandom`, on platforms that have that.
- Update time zone data files to tzdata release 2014e for DST law changes in Crimea, Egypt, and Morocco.

E.21. Release 8.4.21

Release Date

2014-03-20

This release contains a variety of fixes from 8.4.20. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

The PostgreSQL™ community will stop releasing updates for the 8.4.X release series in July 2014. Users are encouraged to update to a newer release branch soon.

E.21.1. Migration to Version 8.4.21

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.19, see Section E.23, « Release 8.4.19 ».

E.21.2. Changes

- Restore GIN metapages unconditionally to avoid torn-page risk (Heikki Linnakangas)

  Although this oversight could theoretically result in a corrupted index, it is unlikely to have caused any problems in practice, since the active part of a GIN metapage is smaller than a standard 512-byte disk sector.
- Allow regular-expression operators to be terminated early by query cancel requests (Tom Lane)

  This prevents scenarios wherein a pathological regular expression could lock up a server process uninterruptably for a long time.
- Remove incorrect code that tried to allow `OVERLAPS` with single-element row arguments (Joshua Yanovski)

  This code never worked correctly, and since the case is neither specified by the SQL standard nor documented, it seemed better to remove it than fix it.
- Avoid getting more than `AccessShareLock` when de-parsing a rule or view (Dean Rasheed)

  This oversight resulted in pg_dump unexpectedly acquiring `RowExclusiveLock` locks on tables mentioned as the targets of `INSERT`/`UPDATE`/`DELETE` commands in rules. While usually harmless, that could interfere with concurrent transactions that tried to acquire, for example, `ShareLock` on those tables.
- Prevent interrupts while reporting non-`ERROR` messages (Tom Lane)

  This guards against rare server-process freezeups due to recursive entry to `syslog()`, and perhaps other related problems.

- Update time zone data files to tzdata release 2014a for DST law changes in Fiji and Turkey, plus historical changes in Israel and Ukraine.

E.22. Release 8.4.20

Release Date

2014-02-20

This release contains a variety of fixes from 8.4.19. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

The PostgreSQL™ community will stop releasing updates for the 8.4.X release series in July 2014. Users are encouraged to update to a newer release branch soon.

E.22.1. Migration to Version 8.4.20

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.19, see Section E.23, « Release 8.4.19 ».

E.22.2. Changes

- Shore up `GRANT ... WITH ADMIN OPTION` restrictions (Noah Misch)

  Granting a role without `ADMIN OPTION` is supposed to prevent the grantee from adding or removing members from the granted role, but this restriction was easily bypassed by doing `SET ROLE` first. The security impact is mostly that a role member can revoke the access of others, contrary to the wishes of his grantor. Unapproved role member additions are a lesser concern, since an uncooperative role member could provide most of his rights to others anyway by creating views or `SECURITY DEFINER` functions. (CVE-2014-0060)

- Prevent privilege escalation via manual calls to PL validator functions (Andres Freund)

  The primary role of PL validator functions is to be called implicitly during **CREATE FUNCTION**, but they are also normal SQL functions that a user can call explicitly. Calling a validator on a function actually written in some other language was not checked for and could be exploited for privilege-escalation purposes. The fix involves adding a call to a privilege-checking function in each validator function. Non-core procedural languages will also need to make this change to their own validator functions, if any. (CVE-2014-0061)

- Avoid multiple name lookups during table and index DDL (Robert Haas, Andres Freund)

  If the name lookups come to different conclusions due to concurrent activity, we might perform some parts of the DDL on a different table than other parts. At least in the case of **CREATE INDEX**, this can be used to cause the permissions checks to be performed against a different table than the index creation, allowing for a privilege escalation attack. (CVE-2014-0062)

- Prevent buffer overrun with long datetime strings (Noah Misch)

  The `MAXDATELEN` constant was too small for the longest possible value of type interval, allowing a buffer overrun in `interval_out()`. Although the datetime input functions were more careful about avoiding buffer overrun, the limit was short enough to cause them to reject some valid inputs, such as input containing a very long timezone name. The ecpg library contained these vulnerabilities along with some of its own. (CVE-2014-0063)

- Prevent buffer overrun due to integer overflow in size calculations (Noah Misch, Heikki Linnakangas)

  Several functions, mostly type input functions, calculated an allocation size without checking for overflow. If overflow did occur, a too-small buffer would be allocated and then written past. (CVE-2014-0064)

- Prevent overruns of fixed-size buffers (Peter Eisentraut, Jozef Mlich)

  Use `strlcpy()` and related functions to provide a clear guarantee that fixed-size buffers are not overrun. Unlike the preceding items, it is unclear whether these cases really represent live issues, since in most cases there appear to be previous constraints on the size of the input string. Nonetheless it seems prudent to silence all Coverity warnings of this type. (CVE-2014-0065)

- Avoid crashing if `crypt()` returns NULL (Honza Horak, Bruce Momjian)

  There are relatively few scenarios in which `crypt()` could return NULL, but `contrib/chkpass` would crash if it did.

One practical case in which this could be an issue is if libc is configured to refuse to execute unapproved hashing algorithms (e.g., « FIPS mode »). (CVE-2014-0066)

- Document risks of `make check` in the regression testing instructions (Noah Misch, Tom Lane)

  Since the temporary server started by `make check` uses « trust » authentication, another user on the same machine could connect to it as database superuser, and then potentially exploit the privileges of the operating-system user who started the tests. A future release will probably incorporate changes in the testing procedure to prevent this risk, but some public discussion is needed first. So for the moment, just warn people against using `make check` when there are untrusted users on the same machine. (CVE-2014-0067)

- Fix possible mis-replay of WAL records when some segments of a relation aren't full size (Greg Stark, Tom Lane)

  The WAL update could be applied to the wrong page, potentially many pages past where it should have been. Aside from corrupting data, this error has been observed to result in significant « bloat » of standby servers compared to their masters, due to updates being applied far beyond where the end-of-file should have been. This failure mode does not appear to be a significant risk during crash recovery, only when initially synchronizing a standby created from a base backup taken from a quickly-changing master.

- Ensure that insertions into non-leaf GIN index pages write a full-page WAL record when appropriate (Heikki Linnakangas)

  The previous coding risked index corruption in the event of a partial-page write during a system crash.

- Fix race conditions during server process exit (Robert Haas)

  Ensure that signal handlers don't attempt to use the process's `MyProc` pointer after it's no longer valid.

- Fix unsafe references to `errno` within error reporting logic (Christian Kruse)

  This would typically lead to odd behaviors such as missing or inappropriate `HINT` fields.

- Fix possible crashes from using `ereport()` too early during server startup (Tom Lane)

  The principal case we've seen in the field is a crash if the server is started in a directory it doesn't have permission to read.

- Clear retry flags properly in OpenSSL socket write function (Alexander Kukushkin)

  This omission could result in a server lockup after unexpected loss of an SSL-encrypted connection.

- Fix length checking for Unicode identifiers (`U&"..."` syntax) containing escapes (Tom Lane)

  A spurious truncation warning would be printed for such identifiers if the escaped form of the identifier was too long, but the identifier actually didn't need truncation after de-escaping.

- Fix possible crash due to invalid plan for nested sub-selects, such as `WHERE (... x IN (SELECT ...) ...) IN (SELECT ...)` (Tom Lane)

- Ensure that **ANALYZE** creates statistics for a table column even when all the values in it are « too wide » (Tom Lane)

  ANALYZE intentionally omits very wide values from its histogram and most-common-values calculations, but it neglected to do something sane in the case that all the sampled entries are too wide.

- In `ALTER TABLE ... SET TABLESPACE`, allow the database's default tablespace to be used without a permissions check (Stephen Frost)

  `CREATE TABLE` has always allowed such usage, but `ALTER TABLE` didn't get the memo.

- Fix « cannot accept a set » error when some arms of a `CASE` return a set and others don't (Tom Lane)

- Fix checks for all-zero client addresses in pgstat functions (Kevin Grittner)

- Fix possible misclassification of multibyte characters by the text search parser (Tom Lane)

  Non-ASCII characters could be misclassified when using C locale with a multibyte encoding. On Cygwin, non-C locales could fail as well.

- Fix possible misbehavior in `plainto_tsquery()` (Heikki Linnakangas)

  Use `memmove()` not `memcpy()` for copying overlapping memory regions. There have been no field reports of this actually causing trouble, but it's certainly risky.

- Accept `SHIFT_JIS` as an encoding name for locale checking purposes (Tatsuo Ishii)

- Fix misbehavior of `PQhost()` on Windows (Fujii Masao)

  It should return `localhost` if no host has been specified.

- Improve error handling in libpq and psql for failures during `COPY TO STDOUT/FROM STDIN` (Tom Lane)

  In particular this fixes an infinite loop that could occur in 9.2 and up if the server connection was lost during `COPY FROM STDIN`. Variants of that scenario might be possible in older versions, or with other client applications.
- Fix misaligned descriptors in ecpg (MauMau)
- In ecpg, handle lack of a hostname in the connection parameters properly (Michael Meskes)
- Fix performance regression in `contrib/dblink` connection startup (Joe Conway)

  Avoid an unnecessary round trip when client and server encodings match.
- In `contrib/isn`, fix incorrect calculation of the check digit for ISMN values (Fabien Coelho)
- Ensure client-code-only installation procedure works as documented (Peter Eisentraut)
- In Mingw and Cygwin builds, install the libpq DLL in the `bin` directory (Andrew Dunstan)

  This duplicates what the MSVC build has long done. It should fix problems with programs like psql failing to start because they can't find the DLL.
- Don't generate plain-text `HISTORY` and `src/test/regress/README` files anymore (Tom Lane)

  These text files duplicated the main HTML and PDF documentation formats. The trouble involved in maintaining them greatly outweighs the likely audience for plain-text format. Distribution tarballs will still contain files by these names, but they'll just be stubs directing the reader to consult the main documentation. The plain-text `INSTALL` file will still be maintained, as there is arguably a use-case for that.
- Update time zone data files to tzdata release 2013i for DST law changes in Jordan and historical changes in Cuba.

  In addition, the zones `Asia/Riyadh87`, `Asia/Riyadh88`, and `Asia/Riyadh89` have been removed, as they are no longer maintained by IANA, and never represented actual civil timekeeping practice.

E.23. Release 8.4.19

Release Date

2013-12-05

This release contains a variety of fixes from 8.4.18. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.23.1. Migration to Version 8.4.19

A dump/restore is not required for those running 8.4.X.

However, this release corrects a potential data corruption issue. See the first changelog entry below to find out whether your installation has been affected and what steps you can take if so.

Also, if you are upgrading from a version earlier than 8.4.17, see Section E.25, « Release 8.4.17 ».

E.23.2. Changes

- Fix **VACUUM**'s tests to see whether it can update *`relfrozenxid`* (Andres Freund)

  In some cases **VACUUM** (either manual or autovacuum) could incorrectly advance a table's *`relfrozenxid`* value, allowing tuples to escape freezing, causing those rows to become invisible once 2^31 transactions have elapsed. The probability of data loss is fairly low since multiple incorrect advancements would need to happen before actual loss occurs, but it's not zero. Users upgrading from release 8.4.8 or earlier are not affected, but all later versions contain the bug.

  The issue can be ameliorated by, after upgrading, vacuuming all tables in all databases while having `vacuum_freeze_table_age` set to zero. This will fix any latent corruption but will not be able to fix all pre-existing data errors. However, an installation can be presumed safe after performing this vacuuming if it has executed fewer than 2^31 update transactions in its lifetime (check this with `SELECT txid_current() < 2^31`).
- Fix race condition in GIN index posting tree page deletion (Heikki Linnakangas)

  This could lead to transient wrong answers or query failures.

- Avoid flattening a subquery whose `SELECT` list contains a volatile function wrapped inside a sub-`SELECT` (Tom Lane)

  This avoids unexpected results due to extra evaluations of the volatile function.
- Fix planner's processing of non-simple-variable subquery outputs nested within outer joins (Tom Lane)

  This error could lead to incorrect plans for queries involving multiple levels of subqueries within `JOIN` syntax.
- Fix premature deletion of temporary files (Andres Freund)
- Fix possible read past end of memory in rule printing (Peter Eisentraut)
- Fix array slicing of int2vector and oidvector values (Tom Lane)

  Expressions of this kind are now implicitly promoted to regular int2 or oid arrays.
- Fix incorrect behaviors when using a SQL-standard, simple GMT offset timezone (Tom Lane)

  In some cases, the system would use the simple GMT offset value when it should have used the regular timezone setting that had prevailed before the simple offset was selected. This change also causes the `timeofday` function to honor the simple GMT offset zone.
- Prevent possible misbehavior when logging translations of Windows error codes (Tom Lane)
- Properly quote generated command lines in pg_ctl (Naoya Anzai and Tom Lane)

  This fix applies only to Windows.
- Fix pg_dumpall to work when a source database sets `default_transaction_read_only` via **ALTER DATABASE SET** (Kevin Grittner)

  Previously, the generated script would fail during restore.
- Fix ecpg's processing of lists of variables declared varchar (Zoltán Böszörményi)
- Make `contrib/lo` defend against incorrect trigger definitions (Marc Cousin)
- Update time zone data files to tzdata release 2013h for DST law changes in Argentina, Brazil, Jordan, Libya, Liechtenstein, Morocco, and Palestine. Also, new timezone abbreviations WIB, WIT, WITA for Indonesia.

E.24. Release 8.4.18

Release Date

2013-10-10

This release contains a variety of fixes from 8.4.17. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.24.1. Migration to Version 8.4.18

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.17, see Section E.25, « Release 8.4.17 ».

E.24.2. Changes

- Prevent corruption of multi-byte characters when attempting to case-fold identifiers (Andrew Dunstan)

  PostgreSQL™ case-folds non-ASCII characters only when using a single-byte server encoding.
- Fix memory leak caused by `lo_open()` failure (Heikki Linnakangas)
- Fix memory overcommit bug when `work_mem` is using more than 24GB of memory (Stephen Frost)
- Fix deadlock bug in libpq when using SSL (Stephen Frost)
- Properly compute row estimates for boolean columns containing many NULL values (Andrew Gierth)

  Previously tests like `col IS NOT TRUE` and `col IS NOT FALSE` did not properly factor in NULL values when estimating plan costs.

- Prevent pushing down `WHERE` clauses into unsafe `UNION/INTERSECT` subqueries (Tom Lane)

  Subqueries of a `UNION` or `INTERSECT` that contain set-returning functions or volatile functions in their `SELECT` lists could be improperly optimized, leading to run-time errors or incorrect query results.
- Fix rare case of « failed to locate grouping columns » planner failure (Tom Lane)
- Improve view dumping code's handling of dropped columns in referenced tables (Tom Lane)
- Fix possible deadlock during concurrent **CREATE INDEX CONCURRENTLY** operations (Tom Lane)
- Fix `regexp_matches()` handling of zero-length matches (Jeevan Chalke)

  Previously, zero-length matches like '^' could return too many matches.
- Fix crash for overly-complex regular expressions (Heikki Linnakangas)
- Fix regular expression match failures for back references combined with non-greedy quantifiers (Jeevan Chalke)
- Prevent **CREATE FUNCTION** from checking **SET** variables unless function body checking is enabled (Tom Lane)
- Fix `pgp_pub_decrypt()` so it works for secret keys with passwords (Marko Kreen)
- Remove rare inaccurate warning during vacuum of index-less tables (Heikki Linnakangas)
- Avoid possible failure when performing transaction control commands (e.g **ROLLBACK**) in prepared queries (Tom Lane)
- Ensure that floating-point data input accepts standard spellings of « infinity » on all platforms (Tom Lane)

  The C99 standard says that allowable spellings are `inf`, `+inf`, `-inf`, `infinity`, `+infinity`, and `-infinity`. Make sure we recognize these even if the platform's `strtod` function doesn't.
- Expand ability to compare rows to records and arrays (Rafal Rzepecki, Tom Lane)
- Update time zone data files to tzdata release 2013d for DST law changes in Israel, Morocco, Palestine, and Paraguay. Also, historical zone data corrections for Macquarie Island.

E.25. Release 8.4.17

Release Date

2013-04-04

This release contains a variety of fixes from 8.4.16. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.25.1. Migration to Version 8.4.17

A dump/restore is not required for those running 8.4.X.

However, this release corrects several errors in management of GiST indexes. After installing this update, it is advisable to **REINDEX** any GiST indexes that meet one or more of the conditions described below.

Also, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.25.2. Changes

- Reset OpenSSL randomness state in each postmaster child process (Marko Kreen)

  This avoids a scenario wherein random numbers generated by `contrib/pgcrypto` functions might be relatively easy for another database user to guess. The risk is only significant when the postmaster is configured with `ssl` = `on` but most connections don't use SSL encryption. (CVE-2013-1900)
- Fix GiST indexes to not use « fuzzy » geometric comparisons when it's not appropriate to do so (Alexander Korotkov)

  The core geometric types perform comparisons using « fuzzy » equality, but `gist_box_same` must do exact comparisons, else GiST indexes using it might become inconsistent. After installing this update, users should **REINDEX** any GiST indexes on box, polygon, circle, or point columns, since all of these use `gist_box_same`.
- Fix erroneous range-union and penalty logic in GiST indexes that use `contrib/btree_gist` for variable-width data types, that is text, bytea, bit, and numeric columns (Tom Lane)

These errors could result in inconsistent indexes in which some keys that are present would not be found by searches, and also in useless index bloat. Users are advised to **REINDEX** such indexes after installing this update.

- Fix bugs in GiST page splitting code for multi-column indexes (Tom Lane)

  These errors could result in inconsistent indexes in which some keys that are present would not be found by searches, and also in indexes that are unnecessarily inefficient to search. Users are advised to **REINDEX** multi-column GiST indexes after installing this update.

- Fix infinite-loop risk in regular expression compilation (Tom Lane, Don Porter)
- Fix potential null-pointer dereference in regular expression compilation (Tom Lane)
- Fix `to_char()` to use ASCII-only case-folding rules where appropriate (Tom Lane)

  This fixes misbehavior of some template patterns that should be locale-independent, but mishandled « `I` » and « `i` » in Turkish locales.

- Fix unwanted rejection of timestamp `1999-12-31 24:00:00` (Tom Lane)
- Remove useless « picksplit doesn't support secondary split » log messages (Josh Hansen, Tom Lane)

  This message seems to have been added in expectation of code that was never written, and probably never will be, since GiST's default handling of secondary splits is actually pretty good. So stop nagging end users about it.

- Fix possible failure to send a session's last few transaction commit/abort counts to the statistics collector (Tom Lane)
- Eliminate memory leaks in PL/Perl's `spi_prepare()` function (Alex Hunsaker, Tom Lane)
- Fix pg_dumpall to handle database names containing « `=` » correctly (Heikki Linnakangas)
- Avoid crash in pg_dump when an incorrect connection string is given (Heikki Linnakangas)
- Ignore invalid indexes in pg_dump (Michael Paquier)

  Dumping invalid indexes can cause problems at restore time, for example if the reason the index creation failed was because it tried to enforce a uniqueness condition not satisfied by the table's data. Also, if the index creation is in fact still in progress, it seems reasonable to consider it to be an uncommitted DDL change, which pg_dump wouldn't be expected to dump anyway.

- Fix `contrib/pg_trgm`'s `similarity()` function to return zero for trigram-less strings (Tom Lane)

  Previously it returned `NaN` due to internal division by zero.

- Update time zone data files to tzdata release 2013b for DST law changes in Chile, Haiti, Morocco, Paraguay, and some Russian areas. Also, historical zone data corrections for numerous places.

  Also, update the time zone abbreviation files for recent changes in Russia and elsewhere: `CHOT`, `GET`, `IRKT`, `KGT`, `KRAT`, `MAGT`, `MAWT`, `MSK`, `NOVT`, `OMST`, `TKT`, `VLAT`, `WST`, `YAKT`, `YEKT` now follow their current meanings, and `VOLT` (Europe/Volgograd) and `MIST` (Antarctica/Macquarie) are added to the default abbreviations list.

E.26. Release 8.4.16

Release Date

2013-02-07

This release contains a variety of fixes from 8.4.15. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.26.1. Migration to Version 8.4.16

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.26.2. Changes

- Prevent execution of `enum_recv` from SQL (Tom Lane)

  The function was misdeclared, allowing a simple SQL command to crash the server. In principle an attacker might be able to

use it to examine the contents of server memory. Our thanks to Sumit Soni (via Secunia SVCRP) for reporting this issue. (CVE-2013-0255)

- Update minimum recovery point when truncating a relation file (Heikki Linnakangas)

  Once data has been discarded, it's no longer safe to stop recovery at an earlier point in the timeline.
- Fix SQL grammar to allow subscripting or field selection from a sub-SELECT result (Tom Lane)
- Protect against race conditions when scanning pg_tablespace (Stephen Frost, Tom Lane)

  CREATE DATABASE and **DROP DATABASE** could misbehave if there were concurrent updates of pg_tablespace entries.
- Prevent **DROP OWNED** from trying to drop whole databases or tablespaces (Álvaro Herrera)

  For safety, ownership of these objects must be reassigned, not dropped.
- Fix error in `vacuum_freeze_table_age` implementation (Andres Freund)

  In installations that have existed for more than `vacuum_freeze_min_age` transactions, this mistake prevented autovacuum from using partial-table scans, so that a full-table scan would always happen instead.
- Prevent misbehavior when a RowExpr or XmlExpr is parse-analyzed twice (Andres Freund, Tom Lane)

  This mistake could be user-visible in contexts such as `CREATE TABLE LIKE INCLUDING INDEXES`.
- Improve defenses against integer overflow in hashtable sizing calculations (Jeff Davis)
- Reject out-of-range dates in `to_date()` (Hitoshi Harada)
- Ensure that non-ASCII prompt strings are translated to the correct code page on Windows (Alexander Law, Noah Misch)

  This bug affected psql and some other client programs.
- Fix possible crash in psql's **\?** command when not connected to a database (Meng Qingzhong)
- Fix one-byte buffer overrun in libpq's `PQprintTuples` (Xi Wang)

  This ancient function is not used anywhere by PostgreSQL™ itself, but it might still be used by some client code.
- Make ecpglib use translated messages properly (Chen Huajun)
- Properly install ecpg_compat and pgtypes libraries on MSVC (Jiang Guiqing)
- Rearrange configure's tests for supplied functions so it is not fooled by bogus exports from libedit/libreadline (Christoph Berg)
- Ensure Windows build number increases over time (Magnus Hagander)
- Make pgxs build executables with the right `.exe` suffix when cross-compiling for Windows (Zoltan Boszormenyi)
- Add new timezone abbreviation `FET` (Tom Lane)

  This is now used in some eastern-European time zones.

E.27. Release 8.4.15

Release Date

2012-12-06

This release contains a variety of fixes from 8.4.14. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.27.1. Migration to Version 8.4.15

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.27.2. Changes

- Fix multiple bugs associated with **CREATE INDEX CONCURRENTLY** (Andres Freund, Tom Lane)

Fix **CREATE INDEX CONCURRENTLY** to use in-place updates when changing the state of an index's pg_index row. This prevents race conditions that could cause concurrent sessions to miss updating the target index, thus resulting in corrupt concurrently-created indexes.

Also, fix various other operations to ensure that they ignore invalid indexes resulting from a failed **CREATE INDEX CONCURRENTLY** command. The most important of these is **VACUUM**, because an auto-vacuum could easily be launched on the table before corrective action can be taken to fix or remove the invalid index.

- Avoid corruption of internal hash tables when out of memory (Hitoshi Harada)
- Fix planning of non-strict equivalence clauses above outer joins (Tom Lane)

  The planner could derive incorrect constraints from a clause equating a non-strict construct to something else, for example `WHERE COALESCE(foo, 0) = 0` when `foo` is coming from the nullable side of an outer join.
- Improve planner's ability to prove exclusion constraints from equivalence classes (Tom Lane)
- Fix partial-row matching in hashed subplans to handle cross-type cases correctly (Tom Lane)

  This affects multicolumn `NOT IN` subplans, such as `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` when for instance `b` and `y` are int4 and int8 respectively. This mistake led to wrong answers or crashes depending on the specific data-types involved.
- Acquire buffer lock when re-fetching the old tuple for an `AFTER ROW UPDATE/DELETE` trigger (Andres Freund)

  In very unusual circumstances, this oversight could result in passing incorrect data to the precheck logic for a foreign-key enforcement trigger. That could result in a crash, or in an incorrect decision about whether to fire the trigger.
- Fix **ALTER COLUMN TYPE** to handle alled check constraints properly (Pavan Deolasee)

  This worked correctly in pre-8.4 releases, and now works correctly in 8.4 and later.
- Fix **REASSIGN OWNED** to handle grants on tablespaces (Álvaro Herrera)
- Ignore incorrect pg_attribute entries for system columns for views (Tom Lane)

  Views do not have any system columns. However, we forgot to remove such entries when converting a table to a view. That's fixed properly for 9.3 and later, but in previous branches we need to defend against existing mis-converted views.
- Fix rule printing to dump `INSERT INTO` *`table`* `DEFAULT VALUES` correctly (Tom Lane)
- Guard against stack overflow when there are too many `UNION`/`INTERSECT`/`EXCEPT` clauses in a query (Tom Lane)
- Prevent platform-dependent failures when dividing the minimum possible integer value by -1 (Xi Wang, Tom Lane)
- Fix possible access past end of string in date parsing (Hitoshi Harada)
- Produce an understandable error message if the length of the path name for a Unix-domain socket exceeds the platform-specific limit (Tom Lane, Andrew Dunstan)

  Formerly, this would result in something quite unhelpful, such as « Non-recoverable failure in name resolution ».
- Fix memory leaks when sending composite column values to the client (Tom Lane)
- Make pg_ctl more robust about reading the `postmaster.pid` file (Heikki Linnakangas)

  Fix race conditions and possible file descriptor leakage.
- Fix possible crash in psql if incorrectly-encoded data is presented and the `client_encoding` setting is a client-only encoding, such as SJIS (Jiang Guiqing)
- Fix bugs in the `restore.sql` script emitted by pg_dump in `tar` output format (Tom Lane)

  The script would fail outright on tables whose names include upper-case characters. Also, make the script capable of restoring data in `--inserts` mode as well as the regular COPY mode.
- Fix pg_restore to accept POSIX-conformant `tar` files (Brian Weaver, Tom Lane)

  The original coding of pg_dump's `tar` output mode produced files that are not fully conformant with the POSIX standard. This has been corrected for version 9.3. This patch updates previous branches so that they will accept both the incorrect and the corrected formats, in hopes of avoiding compatibility problems when 9.3 comes out.
- Fix pg_resetxlog to locate `postmaster.pid` correctly when given a relative path to the data directory (Tom Lane)

  This mistake could lead to pg_resetxlog not noticing that there is an active postmaster using the data directory.

- Fix libpq's `lo_import()` and `lo_export()` functions to report file I/O errors properly (Tom Lane)
- Fix ecpg's processing of nested structure pointer variables (Muhammad Usama)
- Make `contrib/pageinspect`'s btree page inspection functions take buffer locks while examining pages (Tom Lane)
- Fix pgxs support for building loadable modules on AIX (Tom Lane)

  Building modules outside the original source tree didn't work on AIX.
- Update time zone data files to tzdata release 2012j for DST law changes in Cuba, Israel, Jordan, Libya, Palestine, Western Samoa, and portions of Brazil.

E.28. Release 8.4.14

Release Date

2012-09-24

This release contains a variety of fixes from 8.4.13. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.28.1. Migration to Version 8.4.14

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.28.2. Changes

- Fix planner's assignment of executor parameters, and fix executor's rescan logic for CTE plan nodes (Tom Lane)

  These errors could result in wrong answers from queries that scan the same `WITH` subquery multiple times.
- Improve page-splitting decisions in GiST indexes (Alexander Korotkov, Robert Haas, Tom Lane)

  Multi-column GiST indexes might suffer unexpected bloat due to this error.
- Fix cascading privilege revoke to stop if privileges are still held (Tom Lane)

  If we revoke a grant option from some role *X*, but *X* still holds that option via a grant from someone else, we should not recursively revoke the corresponding privilege from role(s) *Y* that *X* had granted it to.
- Fix handling of `SIGFPE` when PL/Perl is in use (Andres Freund)

  Perl resets the process's `SIGFPE` handler to `SIG_IGN`, which could result in crashes later on. Restore the normal Postgres signal handler after initializing PL/Perl.
- Prevent PL/Perl from crashing if a recursive PL/Perl function is redefined while being executed (Tom Lane)
- Work around possible misoptimization in PL/Perl (Tom Lane)

  Some Linux distributions contain an incorrect version of `pthread.h` that results in incorrect compiled code in PL/Perl, leading to crashes if a PL/Perl function calls another one that throws an error.
- Update time zone data files to tzdata release 2012f for DST law changes in Fiji

E.29. Release 8.4.13

Release Date

2012-08-17

This release contains a variety of fixes from 8.4.12. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.29.1. Migration to Version 8.4.13

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.29.2. Changes

- Prevent access to external files/URLs via XML entity references (Noah Misch, Tom Lane)

  `xml_parse()` would attempt to fetch external files or URLs as needed to resolve DTD and entity references in an XML value, thus allowing unprivileged database users to attempt to fetch data with the privileges of the database server. While the external data wouldn't get returned directly to the user, portions of it could be exposed in error messages if the data didn't parse as valid XML; and in any case the mere ability to check existence of a file might be useful to an attacker. (CVE-2012-3489)

- Prevent access to external files/URLs via `contrib/xml2`'s `xslt_process()` (Peter Eisentraut)

  libxslt offers the ability to read and write both files and URLs through stylesheet commands, thus allowing unprivileged database users to both read and write data with the privileges of the database server. Disable that through proper use of libxslt's security options. (CVE-2012-3488)

  Also, remove `xslt_process()`'s ability to fetch documents and stylesheets from external files/URLs. While this was a documented « feature », it was long regarded as a bad idea. The fix for CVE-2012-3489 broke that capability, and rather than expend effort on trying to fix it, we're just going to summarily remove it.

- Prevent too-early recycling of btree index pages (Noah Misch)

  When we allowed read-only transactions to skip assigning XIDs, we introduced the possibility that a deleted btree page could be recycled while a read-only transaction was still in flight to it. This would result in incorrect index search results. The probability of such an error occurring in the field seems very low because of the timing requirements, but nonetheless it should be fixed.

- Fix crash-safety bug with newly-created-or-reset sequences (Tom Lane)

  If **ALTER SEQUENCE** was executed on a freshly created or reset sequence, and then precisely one `nextval()` call was made on it, and then the server crashed, WAL replay would restore the sequence to a state in which it appeared that no `nextval()` had been done, thus allowing the first sequence value to be returned again by the next `nextval()` call. In particular this could manifest for serial columns, since creation of a serial column's sequence includes an **ALTER SEQUENCE OWNED BY** step.

- Ensure the `backup_label` file is fsync'd after `pg_start_backup()` (Dave Kerr)

- Back-patch 9.1 improvement to compress the fsync request queue (Robert Haas)

  This improves performance during checkpoints. The 9.1 change has now seen enough field testing to seem safe to back-patch.

- Only allow autovacuum to be auto-canceled by a directly blocked process (Tom Lane)

  The original coding could allow inconsistent behavior in some cases; in particular, an autovacuum could get canceled after less than `deadlock_timeout` grace period.

- Improve logging of autovacuum cancels (Robert Haas)

- Fix log collector so that `log_truncate_on_rotation` works during the very first log rotation after server start (Tom Lane)

- Fix `WITH` attached to a nested set operation (`UNION`/`INTERSECT`/`EXCEPT`) (Tom Lane)

- Ensure that a whole-row reference to a subquery doesn't include any extra `GROUP BY` or `ORDER BY` columns (Tom Lane)

- Disallow copying whole-row references in `CHECK` constraints and index definitions during **CREATE TABLE** (Tom Lane)

  This situation can arise in **CREATE TABLE** with `LIKE` or `INHERITS`. The copied whole-row variable was incorrectly labeled with the row type of the original table not the new one. Rejecting the case seems reasonable for `LIKE`, since the row types might well diverge later. For `INHERITS` we should ideally allow it, with an implicit coercion to the parent table's row type; but that will require more work than seems safe to back-patch.

- Fix memory leak in `ARRAY(SELECT ...)` subqueries (Heikki Linnakangas, Tom Lane)

- Fix extraction of common prefixes from regular expressions (Tom Lane)

  The code could get confused by quantified parenthesized subexpressions, such as `^(foo)?bar`. This would lead to incorrect index optimization of searches for such patterns.

- Fix bugs with parsing signed *hh*`:`*mm* and *hh*`:`*mm*`:`*ss* fields in interval constants (Amit Kapila, Tom Lane)

- Report errors properly in `contrib/xml2`'s `xslt_process()` (Tom Lane)
- Update time zone data files to tzdata release 2012e for DST law changes in Morocco and Tokelau

E.30. Release 8.4.12

Release Date

2012-06-04

This release contains a variety of fixes from 8.4.11. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.30.1. Migration to Version 8.4.12

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.30.2. Changes

- Fix incorrect password transformation in `contrib/pgcrypto`'s DES `crypt()` function (Solar Designer)

  If a password string contained the byte value `0x80`, the remainder of the password was ignored, causing the password to be much weaker than it appeared. With this fix, the rest of the string is properly included in the DES hash. Any stored password values that are affected by this bug will thus no longer match, so the stored values may need to be updated. (CVE-2012-2143)

- Ignore `SECURITY DEFINER` and `SET` attributes for a procedural language's call handler (Tom Lane)

  Applying such attributes to a call handler could crash the server. (CVE-2012-2655)

- Allow numeric timezone offsets in timestamp input to be up to 16 hours away from UTC (Tom Lane)

  Some historical time zones have offsets larger than 15 hours, the previous limit. This could result in dumped data values being rejected during reload.

- Fix timestamp conversion to cope when the given time is exactly the last DST transition time for the current timezone (Tom Lane)

  This oversight has been there a long time, but was not noticed previously because most DST-using zones are presumed to have an indefinite sequence of future DST transitions.

- Fix text to name and char to name casts to perform string truncation correctly in multibyte encodings (Karl Schnaitter)
- Fix memory copying bug in `to_tsquery()` (Heikki Linnakangas)
- Fix planner's handling of outer PlaceHolderVars within subqueries (Tom Lane)

  This bug concerns sub-SELECTs that reference variables coming from the nullable side of an outer join of the surrounding query. In 9.1, queries affected by this bug would fail with « ERROR: Upper-level PlaceHolderVar found where not expected ». But in 9.0 and 8.4, you'd silently get possibly-wrong answers, since the value transmitted into the subquery wouldn't go to null when it should.

- Fix slow session startup when pg_attribute is very large (Tom Lane)

  If pg_attribute exceeds one-fourth of `shared_buffers`, cache rebuilding code that is sometimes needed during session start would trigger the synchronized-scan logic, causing it to take many times longer than normal. The problem was particularly acute if many new sessions were starting at once.

- Ensure sequential scans check for query cancel reasonably often (Merlin Moncure)

  A scan encountering many consecutive pages that contain no live tuples would not respond to interrupts meanwhile.

- Ensure the Windows implementation of `PGSemaphoreLock()` clears `ImmediateInterruptOK` before returning (Tom Lane)

  This oversight meant that a query-cancel interrupt received later in the same query could be accepted at an unsafe time, with unpredictable but not good consequences.

- Show whole-row variables safely when printing views or rules (Abbas Butt, Tom Lane)

Corner cases involving ambiguous names (that is, the name could be either a table or column name of the query) were printed in an ambiguous way, risking that the view or rule would be interpreted differently after dump and reload. Avoid the ambiguous case by attaching a no-op cast.

- Fix **COPY FROM** to properly handle null marker strings that correspond to invalid encoding (Tom Lane)

  A null marker string such as `E'\\0'` should work, and did work in the past, but the case got broken in 8.4.

- Ensure autovacuum worker processes perform stack depth checking properly (Heikki Linnakangas)

  Previously, infinite recursion in a function invoked by auto-**ANALYZE** could crash worker processes.

- Fix logging collector to not lose log coherency under high load (Andrew Dunstan)

  The collector previously could fail to reassemble large messages if it got too busy.

- Fix logging collector to ensure it will restart file rotation after receiving `SIGHUP` (Tom Lane)
- Fix WAL replay logic for GIN indexes to not fail if the index was subsequently dropped (Tom Lane)
- Fix memory leak in PL/pgSQL's **RETURN NEXT** command (Joe Conway)
- Fix PL/pgSQL's **GET DIAGNOSTICS** command when the target is the function's first variable (Tom Lane)
- Fix potential access off the end of memory in psql's expanded display (**x**) mode (Peter Eisentraut)
- Fix several performance problems in pg_dump when the database contains many objects (Jeff Janes, Tom Lane)

  pg_dump could get very slow if the database contained many schemas, or if many objects are in dependency loops, or if there are many owned sequences.

- Fix `contrib/dblink`'s `dblink_exec()` to not leak temporary database connections upon error (Tom Lane)
- Fix `contrib/dblink` to report the correct connection name in error messages (Kyotaro Horiguchi)
- Update time zone data files to tzdata release 2012c for DST law changes in Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria, and Tokelau Islands; also historical corrections for Canada.

E.31. Release 8.4.11

Release Date

2012-02-27

This release contains a variety of fixes from 8.4.10. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.31.1. Migration to Version 8.4.11

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.10, see Section E.32, « Release 8.4.10 ».

E.31.2. Changes

- Require execute permission on the trigger function for **CREATE TRIGGER** (Robert Haas)

  This missing check could allow another user to execute a trigger function with forged input data, by installing it on a table he owns. This is only of significance for trigger functions marked `SECURITY DEFINER`, since otherwise trigger functions run as the table owner anyway. (CVE-2012-0866)

- Remove arbitrary limitation on length of common name in SSL certificates (Heikki Linnakangas)

  Both libpq and the server truncated the common name extracted from an SSL certificate at 32 bytes. Normally this would cause nothing worse than an unexpected verification failure, but there are some rather-implausible scenarios in which it might allow one certificate holder to impersonate another. The victim would have to have a common name exactly 32 bytes long, and the attacker would have to persuade a trusted CA to issue a certificate in which the common name has that string as a prefix. Impersonating a server would also require some additional exploit to redirect client connections. (CVE-2012-0867)

- Convert newlines to spaces in names written in pg_dump comments (Robert Haas)

pg_dump was incautious about sanitizing object names that are emitted within SQL comments in its output script. A name containing a newline would at least render the script syntactically incorrect. Maliciously crafted object names could present a SQL injection risk when the script is reloaded. (CVE-2012-0868)

- Fix btree index corruption from insertions concurrent with vacuuming (Tom Lane)

  An index page split caused by an insertion could sometimes cause a concurrently-running **VACUUM** to miss removing index entries that it should remove. After the corresponding table rows are removed, the dangling index entries would cause errors (such as « could not read block N in file ... ») or worse, silently wrong query results after unrelated rows are re-inserted at the now-free table locations. This bug has been present since release 8.2, but occurs so infrequently that it was not diagnosed until now. If you have reason to suspect that it has happened in your database, reindexing the affected index will fix things.

- Update per-column permissions, not only per-table permissions, when changing table owner (Tom Lane)

  Failure to do this meant that any previously granted column permissions were still shown as having been granted by the old owner. This meant that neither the new owner nor a superuser could revoke the now-untraceable-to-table-owner permissions.

- Allow non-existent values for some settings in **ALTER USER/DATABASE SET** (Heikki Linnakangas)

  Allow `default_text_search_config`, `default_tablespace`, and `temp_tablespaces` to be set to names that are not known. This is because they might be known in another database where the setting is intended to be used, or for the tablespace cases because the tablespace might not be created yet. The same issue was previously recognized for `search_path`, and these settings now act like that one.

- Avoid crashing when we have problems deleting table files post-commit (Tom Lane)

  Dropping a table should lead to deleting the underlying disk files only after the transaction commits. In event of failure then (for instance, because of wrong file permissions) the code is supposed to just emit a warning message and go on, since it's too late to abort the transaction. This logic got broken as of release 8.4, causing such situations to result in a PANIC and an unrestartable database.

- Track the OID counter correctly during WAL replay, even when it wraps around (Tom Lane)

  Previously the OID counter would remain stuck at a high value until the system exited replay mode. The practical consequences of that are usually nil, but there are scenarios wherein a standby server that's been promoted to master might take a long time to advance the OID counter to a reasonable value once values are needed.

- Fix regular expression back-references with `*` attached (Tom Lane)

  Rather than enforcing an exact string match, the code would effectively accept any string that satisfies the pattern subexpression referenced by the back-reference symbol.

  A similar problem still afflicts back-references that are embedded in a larger quantified expression, rather than being the immediate subject of the quantifier. This will be addressed in a future PostgreSQL™ release.

- Fix recently-introduced memory leak in processing of inet/cidr values (Heikki Linnakangas)

  A patch in the December 2011 releases of PostgreSQL™ caused memory leakage in these operations, which could be significant in scenarios such as building a btree index on such a column.

- Fix dangling pointer after **CREATE TABLE AS/SELECT INTO** in a SQL-language function (Tom Lane)

  In most cases this only led to an assertion failure in assert-enabled builds, but worse consequences seem possible.

- Avoid double close of file handle in syslogger on Windows (MauMau)

  Ordinarily this error was invisible, but it would cause an exception when running on a debug version of Windows.

- Fix I/O-conversion-related memory leaks in plpgsql (Andres Freund, Jan Urbanski, Tom Lane)

  Certain operations would leak memory until the end of the current function.

- Improve pg_dump's handling of alled table columns (Tom Lane)

  pg_dump mishandled situations where a child column has a different default expression than its parent column. If the default is textually identical to the parent's default, but not actually the same (for instance, because of schema search path differences) it would not be recognized as different, so that after dump and restore the child would be allowed to all the parent's default. Child columns that are `NOT NULL` where their parent is not could also be restored subtly incorrectly.

- Fix pg_restore's direct-to-database mode for INSERT-style table data (Tom Lane)

  Direct-to-database restores from archive files made with `--inserts` or `--column-inserts` options fail when using pg_restore from a release dated September or December 2011, as a result of an oversight in a fix for another problem. The ar-

chive file itself is not at fault, and text-mode output is okay.

- Allow `AT` option in ecpg `DEALLOCATE` statements (Michael Meskes)

  The infrastructure to support this has been there for awhile, but through an oversight there was still an error check rejecting the case.

- Fix error in `contrib/intarray`'s `int[] & int[]` operator (Guillaume Lelarge)

  If the smallest integer the two input arrays have in common is 1, and there are smaller values in either array, then 1 would be incorrectly omitted from the result.

- Fix error detection in `contrib/pgcrypto`'s `encrypt_iv()` and `decrypt_iv()` (Marko Kreen)

  These functions failed to report certain types of invalid-input errors, and would instead return random garbage values for incorrect input.

- Fix one-byte buffer overrun in `contrib/test_parser` (Paul Guyot)

  The code would try to read one more byte than it should, which would crash in corner cases. Since `contrib/test_parser` is only example code, this is not a security issue in itself, but bad example code is still bad.

- Use `__sync_lock_test_and_set()` for spinlocks on ARM, if available (Martin Pitt)

  This function replaces our previous use of the `SWPB` instruction, which is deprecated and not available on ARMv6 and later. Reports suggest that the old code doesn't fail in an obvious way on recent ARM boards, but simply doesn't interlock concurrent accesses, leading to bizarre failures in multiprocess operation.

- Use `-fexcess-precision=standard` option when building with gcc versions that accept it (Andrew Dunstan)

  This prevents assorted scenarios wherein recent versions of gcc will produce creative results.

- Allow use of threaded Python on FreeBSD (Chris Rees)

  Our configure script previously believed that this combination wouldn't work; but FreeBSD fixed the problem, so remove that error check.

E.32. Release 8.4.10

Release Date

2011-12-05

This release contains a variety of fixes from 8.4.9. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.32.1. Migration to Version 8.4.10

A dump/restore is not required for those running 8.4.X.

However, a longstanding error was discovered in the definition of the `information_schema.referential_constraints` view. If you rely on correct results from that view, you should replace its definition as explained in the first changelog item below.

Also, if you are upgrading from a version earlier than 8.4.8, see Section E.34, « Release 8.4.8 ».

E.32.2. Changes

- Fix bugs in `information_schema.referential_constraints` view (Tom Lane)

  This view was being insufficiently careful about matching the foreign-key constraint to the depended-on primary or unique key constraint. That could result in failure to show a foreign key constraint at all, or showing it multiple times, or claiming that it depends on a different constraint than the one it really does.

  Since the view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can (as a superuser) drop the `information_schema` schema then re-create it by sourcing *`SHAREDIR`*`/information_schema.sql`. (Run `pg_config --sharedir` if you're uncertain where *`SHAREDIR`* is.) This must be repeated in each database to be fixed.

- Fix incorrect replay of WAL records for GIN index updates (Tom Lane)

  This could result in transiently failing to find index entries after a crash, or on a hot-standby server. The problem would be repaired by the next **VACUUM** of the index, however.

- Fix TOAST-related data corruption during `CREATE TABLE dest AS SELECT * FROM src` or `INSERT INTO dest SELECT * FROM src` (Tom Lane)

  If a table has been modified by **ALTER TABLE ADD COLUMN**, attempts to copy its data verbatim to another table could produce corrupt results in certain corner cases. The problem can only manifest in this precise form in 8.4 and later, but we patched earlier versions as well in case there are other code paths that could trigger the same bug.

- Fix race condition during toast table access from stale syscache entries (Tom Lane)

  The typical symptom was transient errors like « missing chunk number 0 for toast value NNNNN in pg_toast_2619 », where the cited toast table would always belong to a system catalog.

- Track dependencies of functions on items used in parameter default expressions (Tom Lane)

  Previously, a referenced object could be dropped without having dropped or modified the function, leading to misbehavior when the function was used. Note that merely installing this update will not fix the missing dependency entries; to do that, you'd need to **CREATE OR REPLACE** each such function afterwards. If you have functions whose defaults depend on non-built-in objects, doing so is recommended.

- Allow inlining of set-returning SQL functions with multiple OUT parameters (Tom Lane)

- Make `DatumGetInetP()` unpack inet datums that have a 1-byte header, and add a new macro, `DatumGetInetPP()`, that does not (Heikki Linnakangas)

  This change affects no core code, but might prevent crashes in add-on code that expects `DatumGetInetP()` to produce an unpacked datum as per usual convention.

- Improve locale support in money type's input and output (Tom Lane)

  Aside from not supporting all standard `lc_monetary` formatting options, the input and output functions were inconsistent, meaning there were locales in which dumped money values could not be re-read.

- Don't let `transform_null_equals` affect `CASE foo WHEN NULL ...` constructs (Heikki Linnakangas)

  `transform_null_equals` is only supposed to affect `foo = NULL` expressions written directly by the user, not equality checks generated internally by this form of `CASE`.

- Change foreign-key trigger creation order to better support self-referential foreign keys (Tom Lane)

  For a cascading foreign key that references its own table, a row update will fire both the `ON UPDATE` trigger and the `CHECK` trigger as one event. The `ON UPDATE` trigger must execute first, else the `CHECK` will check a non-final state of the row and possibly throw an inappropriate error. However, the firing order of these triggers is determined by their names, which generally sort in creation order since the triggers have auto-generated names following the convention « RI_ConstraintTrigger_NNNN ». A proper fix would require modifying that convention, which we will do in 9.2, but it seems risky to change it in existing releases. So this patch just changes the creation order of the triggers. Users encountering this type of error should drop and re-create the foreign key constraint to get its triggers into the right order.

- Avoid floating-point underflow while tracking buffer allocation rate (Greg Matthews)

  While harmless in itself, on certain platforms this would result in annoying kernel log messages.

- Preserve configuration file name and line number values when starting child processes under Windows (Tom Lane)

  Formerly, these would not be displayed correctly in the pg_settings view.

- Preserve blank lines within commands in psql's command history (Robert Haas)

  The former behavior could cause problems if an empty line was removed from within a string literal, for example.

- Fix pg_dump to dump user-defined casts between auto-generated types, such as table rowtypes (Tom Lane)

- Use the preferred version of xsubpp to build PL/Perl, not necessarily the operating system's main copy (David Wheeler and Alex Hunsaker)

- Fix incorrect coding in `contrib/dict_int` and `contrib/dict_xsyn` (Tom Lane)

  Some functions incorrectly assumed that memory returned by `palloc()` is guaranteed zeroed.

- Honor query cancel interrupts promptly in `pgstatindex()` (Robert Haas)

- Ensure VPATH builds properly install all server header files (Peter Eisentraut)
- Shorten file names reported in verbose error messages (Peter Eisentraut)

  Regular builds have always reported just the name of the C file containing the error message call, but VPATH builds formerly reported an absolute path name.
- Fix interpretation of Windows timezone names for Central America (Tom Lane)

  Map « Central America Standard Time » to `CST6`, not `CST6CDT`, because DST is generally not observed anywhere in Central America.
- Update time zone data files to tzdata release 2011n for DST law changes in Brazil, Cuba, Fiji, Palestine, Russia, and Samoa; also historical corrections for Alaska and British East Africa.

E.33. Release 8.4.9

Release Date

2011-09-26

This release contains a variety of fixes from 8.4.8. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.33.1. Migration to Version 8.4.9

A dump/restore is not required for those running 8.4.X.

However, if you are upgrading from a version earlier than 8.4.8, see Section E.34, « Release 8.4.8 ».

E.33.2. Changes

- Fix bugs in indexing of in-doubt HOT-updated tuples (Tom Lane)

  These bugs could result in index corruption after reindexing a system catalog. They are not believed to affect user indexes.
- Fix multiple bugs in GiST index page split processing (Heikki Linnakangas)

  The probability of occurrence was low, but these could lead to index corruption.
- Fix possible buffer overrun in `tsvector_concat()` (Tom Lane)

  The function could underestimate the amount of memory needed for its result, leading to server crashes.
- Fix crash in `xml_recv` when processing a « standalone » parameter (Tom Lane)
- Make `pg_options_to_table` return NULL for an option with no value (Tom Lane)

  Previously such cases would result in a server crash.
- Avoid possibly accessing off the end of memory in **ANALYZE** and in SJIS-2004 encoding conversion (Noah Misch)

  This fixes some very-low-probability server crash scenarios.
- Prevent intermittent hang in interactions of startup process with bgwriter process (Simon Riggs)

  This affected recovery in non-hot-standby cases.
- Fix race condition in relcache init file invalidation (Tom Lane)

  There was a window wherein a new backend process could read a stale init file but miss the inval messages that would tell it the data is stale. The result would be bizarre failures in catalog accesses, typically « could not read block 0 in file ... » later during startup.
- Fix memory leak at end of a GiST index scan (Tom Lane)

  Commands that perform many separate GiST index scans, such as verification of a new GiST-based exclusion constraint on a table already containing many rows, could transiently require large amounts of memory due to this leak.
- Fix incorrect memory accounting (leading to possible memory bloat) in tuplestores supporting holdable cursors and plpgsql's `RETURN NEXT` command (Tom Lane)

- Fix performance problem when constructing a large, lossy bitmap (Tom Lane)
- Fix join selectivity estimation for unique columns (Tom Lane)

  This fixes an erroneous planner heuristic that could lead to poor estimates of the result size of a join.
- Fix nested PlaceHolderVar expressions that appear only in sub-select target lists (Tom Lane)

  This mistake could result in outputs of an outer join incorrectly appearing as NULL.
- Allow nested `EXISTS` queries to be optimized properly (Tom Lane)
- Fix array- and path-creating functions to ensure padding bytes are zeroes (Tom Lane)

  This avoids some situations where the planner will think that semantically-equal constants are not equal, resulting in poor optimization.
- Fix **EXPLAIN** to handle gating Result nodes within inner-indexscan subplans (Tom Lane)

  The usual symptom of this oversight was « bogus varno » errors.
- Work around gcc 4.6.0 bug that breaks WAL replay (Tom Lane)

  This could lead to loss of committed transactions after a server crash.
- Fix dump bug for `VALUES` in a view (Tom Lane)
- Disallow `SELECT FOR UPDATE/SHARE` on sequences (Tom Lane)

  This operation doesn't work as expected and can lead to failures.
- Fix **VACUUM** so that it always updates `pg_class.reltuples/relpages` (Tom Lane)

  This fixes some scenarios where autovacuum could make increasingly poor decisions about when to vacuum tables.
- Defend against integer overflow when computing size of a hash table (Tom Lane)
- Fix cases where **CLUSTER** might attempt to access already-removed TOAST data (Tom Lane)
- Fix portability bugs in use of credentials control messages for « peer » authentication (Tom Lane)
- Fix SSPI login when multiple roundtrips are required (Ahmed Shinwari, Magnus Hagander)

  The typical symptom of this problem was « The function requested is not supported » errors during SSPI login.
- Throw an error if `pg_hba.conf` contains `hostssl` but SSL is disabled (Tom Lane)

  This was concluded to be more user-friendly than the previous behavior of silently ignoring such lines.
- Fix typo in `pg_srand48` seed initialization (Andres Freund)

  This led to failure to use all bits of the provided seed. This function is not used on most platforms (only those without `srandom`), and the potential security exposure from a less-random-than-expected seed seems minimal in any case.
- Avoid integer overflow when the sum of `LIMIT` and `OFFSET` values exceeds 2^63 (Heikki Linnakangas)
- Add overflow checks to int4 and int8 versions of `generate_series()` (Robert Haas)
- Fix trailing-zero removal in `to_char()` (Marti Raudsepp)

  In a format with `FM` and no digit positions after the decimal point, zeroes to the left of the decimal point could be removed incorrectly.
- Fix `pg_size_pretty()` to avoid overflow for inputs close to 2^63 (Tom Lane)
- Weaken plpgsql's check for typmod matching in record values (Tom Lane)

  An overly enthusiastic check could lead to discarding length modifiers that should have been kept.
- Correctly handle quotes in locale names during initdb (Heikki Linnakangas)

  The case can arise with some Windows locales, such as « People's Republic of China ».
- Fix pg_upgrade to preserve toast tables' relfrozenxids during an upgrade from 8.3 (Bruce Momjian)

  Failure to do this could lead to `pg_clog` files being removed too soon after the upgrade.
- In pg_ctl, support silent mode for service registrations on Windows (MauMau)
- Fix psql's counting of script file line numbers during `COPY` from a different file (Tom Lane)

- Fix pg_restore's direct-to-database mode for `standard_conforming_strings` (Tom Lane)

  pg_restore could emit incorrect commands when restoring directly to a database server from an archive file that had been made with `standard_conforming_strings` set to `on`.
- Be more user-friendly about unsupported cases for parallel pg_restore (Tom Lane)

  This change ensures that such cases are detected and reported before any restore actions have been taken.
- Fix write-past-buffer-end and memory leak in libpq's LDAP service lookup code (Albe Laurenz)
- In libpq, avoid failures when using nonblocking I/O and an SSL connection (Martin Pihlak, Tom Lane)
- Improve libpq's handling of failures during connection startup (Tom Lane)

  In particular, the response to a server report of `fork()` failure during SSL connection startup is now saner.
- Improve libpq's error reporting for SSL failures (Tom Lane)
- Fix `PQsetvalue()` to avoid possible crash when adding a new tuple to a PGresult originally obtained from a server query (Andrew Chernow)
- Make ecpglib write double values with 15 digits precision (Akira Kurosawa)
- In ecpglib, be sure `LC_NUMERIC` setting is restored after an error (Michael Meskes)
- Apply upstream fix for blowfish signed-character bug (CVE-2011-2483) (Tom Lane)

  `contrib/pg_crypto`'s blowfish encryption code could give wrong results on platforms where char is signed (which is most), leading to encrypted passwords being weaker than they should be.
- Fix memory leak in `contrib/seg` (Heikki Linnakangas)
- Fix `pgstatindex()` to give consistent results for empty indexes (Tom Lane)
- Allow building with perl 5.14 (Alex Hunsaker)
- Update configure script's method for probing existence of system functions (Tom Lane)

  The version of autoconf we used in 8.3 and 8.2 could be fooled by compilers that perform link-time optimization.
- Fix assorted issues with build and install file paths containing spaces (Tom Lane)
- Update time zone data files to tzdata release 2011i for DST law changes in Canada, Egypt, Russia, Samoa, and South Sudan.

E.34. Release 8.4.8

Release Date

2011-04-18

This release contains a variety of fixes from 8.4.7. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.34.1. Migration to Version 8.4.8

A dump/restore is not required for those running 8.4.X.

However, if your installation was upgraded from a previous major release by running pg_upgrade, you should take action to prevent possible data loss due to a now-fixed bug in pg_upgrade. The recommended solution is to run **VACUUM FREEZE** on all TOAST tables. More information is available at *http://wiki.postgresql.org/wiki/20110408pg_upgrade_fix*.

Also, if you are upgrading from a version earlier than 8.4.2, see Section E.40, « Release 8.4.2 ».

E.34.2. Changes

- Fix pg_upgrade's handling of TOAST tables (Bruce Momjian)

  The pg_class.*`relfrozenxid`* value for TOAST tables was not correctly copied into the new installation during pg_upgrade. This could later result in `pg_clog` files being discarded while they were still needed to validate tuples in the TOAST tables, leading to « could not access status of transaction » failures.

This error poses a significant risk of data loss for installations that have been upgraded with pg_upgrade. This patch corrects the problem for future uses of pg_upgrade, but does not in itself cure the issue in installations that have been processed with a buggy version of pg_upgrade.

- Suppress incorrect « PD_ALL_VISIBLE flag was incorrectly set » warning (Heikki Linnakangas)

  VACUUM would sometimes issue this warning in cases that are actually valid.

- Disallow including a composite type in itself (Tom Lane)

  This prevents scenarios wherein the server could recurse infinitely while processing the composite type. While there are some possible uses for such a structure, they don't seem compelling enough to justify the effort required to make sure it always works safely.

- Avoid potential deadlock during catalog cache initialization (Nikhil Sontakke)

  In some cases the cache loading code would acquire share lock on a system index before locking the index's catalog. This could deadlock against processes trying to acquire exclusive locks in the other, more standard order.

- Fix dangling-pointer problem in `BEFORE ROW UPDATE` trigger handling when there was a concurrent update to the target tuple (Tom Lane)

  This bug has been observed to result in intermittent « cannot extract system attribute from virtual tuple » failures while trying to do `UPDATE RETURNING ctid`. There is a very small probability of more serious errors, such as generating incorrect index entries for the updated tuple.

- Disallow **DROP TABLE** when there are pending deferred trigger events for the table (Tom Lane)

  Formerly the **DROP** would go through, leading to « could not open relation with OID nnn » errors when the triggers were eventually fired.

- Prevent crash triggered by constant-false WHERE conditions during GEQO optimization (Tom Lane)

- Improve planner's handling of semi-join and anti-join cases (Tom Lane)
- Fix selectivity estimation for text search to account for NULLs (Jesper Krogh)
- Improve PL/pgSQL's ability to handle row types with dropped columns (Pavel Stehule)

  This is a back-patch of fixes previously made in 9.0.

- Fix PL/Python memory leak involving array slices (Daniel Popowich)
- Fix pg_restore to cope with long lines (over 1KB) in TOC files (Tom Lane)
- Put in more safeguards against crashing due to division-by-zero with overly enthusiastic compiler optimization (Aurelien Jarno)
- Support use of dlopen() in FreeBSD and OpenBSD on MIPS (Tom Lane)

  There was a hard-wired assumption that this system function was not available on MIPS hardware on these systems. Use a compile-time test instead, since more recent versions have it.

- Fix compilation failures on HP-UX (Heikki Linnakangas)
- Fix version-incompatibility problem with libintl on Windows (Hiroshi Inoue)
- Fix usage of xcopy in Windows build scripts to work correctly under Windows 7 (Andrew Dunstan)

  This affects the build scripts only, not installation or usage.

- Fix path separator used by pg_regress on Cygwin (Andrew Dunstan)
- Update time zone data files to tzdata release 2011f for DST law changes in Chile, Cuba, Falkland Islands, Morocco, Samoa, and Turkey; also historical corrections for South Australia, Alaska, and Hawaii.

E.35. Release 8.4.7

Release Date

2011-01-31

This release contains a variety of fixes from 8.4.6. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.35.1. Migration to Version 8.4.7

A dump/restore is not required for those running 8.4.X. However, if you are upgrading from a version earlier than 8.4.2, see Section E.40, « Release 8.4.2 ».

E.35.2. Changes

- Avoid failures when **EXPLAIN** tries to display a simple-form `CASE` expression (Tom Lane)

  If the `CASE`'s test expression was a constant, the planner could simplify the `CASE` into a form that confused the expression-display code, resulting in « unexpected CASE WHEN clause » errors.

- Fix assignment to an array slice that is before the existing range of subscripts (Tom Lane)

  If there was a gap between the newly added subscripts and the first pre-existing subscript, the code miscalculated how many entries needed to be copied from the old array's null bitmap, potentially leading to data corruption or crash.

- Avoid unexpected conversion overflow in planner for very distant date values (Tom Lane)

  The date type supports a wider range of dates than can be represented by the timestamp types, but the planner assumed it could always convert a date to timestamp with impunity.

- Fix pg_restore's text output for large objects (BLOBs) when `standard_conforming_strings` is on (Tom Lane)

  Although restoring directly to a database worked correctly, string escaping was incorrect if pg_restore was asked for SQL text output and `standard_conforming_strings` had been enabled in the source database.

- Fix erroneous parsing of tsquery values containing `... & !(subexpression) | ...` (Tom Lane)

  Queries containing this combination of operators were not executed correctly. The same error existed in `contrib/intarray`'s query_int type and `contrib/ltree`'s ltxtquery type.

- Fix buffer overrun in `contrib/intarray`'s input function for the query_int type (Apple)

  This bug is a security risk since the function's return address could be overwritten. Thanks to Apple Inc's security team for reporting this issue and supplying the fix. (CVE-2010-4015)

- Fix bug in `contrib/seg`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a seg column. If you have such an index, consider **REINDEX**ing it after installing this update. (This is identical to the bug that was fixed in `contrib/cube` in the previous update.)

E.36. Release 8.4.6

Release Date

2010-12-16

This release contains a variety of fixes from 8.4.5. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.36.1. Migration to Version 8.4.6

A dump/restore is not required for those running 8.4.X. However, if you are upgrading from a version earlier than 8.4.2, see Section E.40, « Release 8.4.2 ».

E.36.2. Changes

- Force the default `wal_sync_method` to be `fdatasync` on Linux (Tom Lane, Marti Raudsepp)

  The default on Linux has actually been `fdatasync` for many years, but recent kernel changes caused PostgreSQL™ to choose `open_datasync` instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably `ext4` with the `data=journal` mount option.

- Fix assorted bugs in WAL replay logic for GIN indexes (Tom Lane)

  This could result in « bad buffer id: 0 » failures or corruption of index contents during replication.
- Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)
- Fix persistent slowdown of autovacuum workers when multiple workers remain active for a long time (Tom Lane)

  The effective `vacuum_cost_limit` for an autovacuum worker could drop to nearly zero if it processed enough tables, causing it to run extremely slowly.
- Add support for detecting register-stack overrun on `IA64` (Tom Lane)

  The `IA64` architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.
- Add a check for stack overflow in `copyObject()` (Tom Lane)

  Certain code paths could crash due to stack overflow given a sufficiently complex query.
- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)

  It is possible to have a « concurrent » page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.
- Fix error checking during early connection processing (Tom Lane)

  The check for too many child processes was skipped in some cases, possibly leading to postmaster crash when attempting to add the new child process to fixed-size arrays.
- Improve efficiency of window functions (Tom Lane)

  Certain cases where a large number of tuples needed to be read in advance, but `work_mem` was large enough to allow them all to be held in memory, were unexpectedly slow. `percent_rank()`, `cume_dist()` and `ntile()` in particular were subject to this problem.
- Avoid memory leakage while **ANALYZE**'ing complex index expressions (Tom Lane)
- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)

  An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.
- Do not « inline » a SQL function with multiple `OUT` parameters (Tom Lane)

  This avoids a possible crash due to loss of information about the expected result rowtype.
- Behave correctly if `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `WITH` is attached to the `VALUES` part of `INSERT ... VALUES` (Tom Lane)
- Fix constant-folding of `COALESCE()` expressions (Tom Lane)

  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.
- Fix postmaster crash when connection acceptance (`accept()` or one of the calls made immediately after it) fails, and the postmaster was compiled with GSSAPI support (Alexander Chernikov)
- Fix missed unlink of temporary files when `log_temp_files` is active (Tom Lane)

  If an error occurred while attempting to emit the log message, the unlink was not done, resulting in accumulation of temp files.
- Add print functionality for InhRelation nodes (Tom Lane)

  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.
- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)

  This bug affected several different geometric distance-measurement operators.
- Fix incorrect calculation of transaction status in ecpg (Itagaki Takahiro)
- Fix PL/pgSQL's handling of « simple » expressions to not fail in recursion or error-recovery cases (Tom Lane)
- Fix PL/Python's handling of set-returning functions (Jan Urbanski)

Attempts to call SPI functions within the iterator generating a set result would fail.

- Fix bug in `contrib/cube`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a cube column. If you have such an index, consider **REINDEX**ing it after installing this update.

- Don't emit « identifier will be truncated » notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)
- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)
- Fix memory leak in `contrib/xml2`'s XPath query functions (Tom Lane)
- Update time zone data files to tzdata release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.

E.37. Release 8.4.5

Release Date

2010-10-04

This release contains a variety of fixes from 8.4.4. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.37.1. Migration to Version 8.4.5

A dump/restore is not required for those running 8.4.X. However, if you are upgrading from a version earlier than 8.4.2, see Section E.40, « Release 8.4.2 ».

E.37.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

- Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it's intended to be used with (Heikki Linnakangas, Tom Lane)
- Treat exit code 128 (`ERROR_WAIT_NO_CHILDREN`) as non-fatal on Windows (Magnus Hagander)

  Under high load, Windows processes will sometimes fail at startup with this error code. Formerly the postmaster treated this as a panic condition and restarted the whole database, but that seems to be an overreaction.

- Fix incorrect placement of placeholder evaluation (Tom Lane)

  This bug could result in query outputs being non-null when they should be null, in cases where the inner side of an outer join is a sub-select with non-strict expressions in its output list.

- Fix possible duplicate scans of `UNION ALL` member relations (Tom Lane)
- Fix « cannot handle unplanned sub-select » error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.

- Fix mishandling of whole-row Vars that reference a view or sub-select and appear within a nested sub-select (Tom Lane)
- Fix mishandling of cross-type `IN` comparisons (Tom Lane)

  This could result in failures if the planner tried to implement an `IN` join with a sort-then-unique-then-plain-join plan.
- Fix computation of **ANALYZE** statistics for tsvector columns (Jan Urbanski)

  The original coding could produce incorrect statistics, leading to poor plan choices later.
- Improve planner's estimate of memory used by `array_agg()`, `string_agg()`, and similar aggregate functions (Hitoshi Harada)

  The previous drastic underestimate could lead to out-of-memory failures due to inappropriate choice of a hash-aggregation plan.
- Fix failure to mark cached plans as transient (Tom Lane)

  If a plan is prepared while **CREATE INDEX CONCURRENTLY** is in progress for one of the referenced tables, it is supposed to be re-planned once the index is ready for use. This was not happening reliably.
- Reduce PANIC to ERROR in some occasionally-reported btree failure cases, and provide additional detail in the resulting error messages (Tom Lane)

  This should improve the system's robustness with corrupted indexes.
- Fix incorrect search logic for partial-match queries with GIN indexes (Tom Lane)

  Cases involving AND/OR combination of several GIN index conditions didn't always give the right answer, and were sometimes much slower than necessary.
- Prevent show_session_authorization() from crashing within autovacuum processes (Tom Lane)
- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)
- Fix possible corruption of pending trigger event lists during subtransaction rollback (Tom Lane)

  This could lead to a crash or incorrect firing of triggers.
- Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)
- Improve merge join's handling of NULLs in the join columns (Tom Lane)

  A merge join can now stop entirely upon reaching the first NULL, if the sort order is such that NULLs sort high.
- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.
- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.
- Avoid holding open old WAL segments in the walwriter process (Magnus Hagander, Heikki Linnakangas)

  The previous coding would prevent removal of no-longer-needed segments.
- Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)
- Prevent misinterpretation of partially-specified relation options for TOAST tables (Itagaki Takahiro)

  In particular, `fillfactor` would be read as zero if any other reloption had been set for the table, leading to serious bloat.
- Fix allance count tracking in **ALTER TABLE ... ADD CONSTRAINT** (Robert Haas)
- Fix possible data corruption in **ALTER TABLE ... SET TABLESPACE** when archiving is enabled (Jeff Davis)
- Allow **CREATE DATABASE** and **ALTER DATABASE ... SET TABLESPACE** to be interrupted by query-cancel (Guillaume Lelarge)
- Improve **CREATE INDEX**'s checking of whether proposed index expressions are immutable (Tom Lane)
- Fix **REASSIGN OWNED** to handle operator classes and families (Asko Tiidumaa)
- Fix possible core dump when comparing two empty tsquery values (Tom Lane)
- Fix `LIKE`'s handling of patterns containing `%` followed by _ (Tom Lane)

We've fixed this before, but there were still some incorrectly-handled cases.

- Re-allow input of Julian dates prior to 0001-01-01 AD (Tom Lane)

  Input such as `'J100000'::date` worked before 8.4, but was unintentionally broken by added error-checking.

- Fix PL/pgSQL to throw an error, not crash, if a cursor is closed within a `FOR` loop that is iterating over that cursor (Heikki Linnakangas)
- In PL/Python, defend against null pointer results from `PyCObject_AsVoidPtr` and `PyCObject_FromVoidPtr` (Peter Eisentraut)
- In libpq, fix full SSL certificate verification for the case where both `host` and `hostaddr` are specified (Tom Lane)
- Make psql recognize **DISCARD ALL** as a command that should not be encased in a transaction block in autocommit-off mode (Itagaki Takahiro)
- Fix some issues in pg_dump's handling of SQL/MED objects (Tom Lane)

  Notably, pg_dump would always fail if run by a non-superuser, which was not intended.

- Improve pg_dump and pg_restore's handling of non-seekable archive files (Tom Lane, Robert Haas)

  This is important for proper functioning of parallel restore.

- Improve parallel pg_restore's ability to cope with selective restore (`-L` option) (Tom Lane)

  The original code tended to fail if the `-L` file commanded a non-default restore ordering.

- Fix ecpg to process data from `RETURNING` clauses correctly (Michael Meskes)
- Fix some memory leaks in ecpg (Zoltan Boszormenyi)
- Improve `contrib/dblink`'s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after « duplicate connection name » errors in `contrib/dblink` (Itagaki Takahiro)
- Fix `contrib/dblink` to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
- Add `hstore(text, text)` function to `contrib/hstore` (Robert Haas)

  This function is the recommended substitute for the now-deprecated `=>` operator. It was back-patched so that future-proofed code can be used with older server versions. Note that the patch will be effective only after `contrib/hstore` is installed or reinstalled in a particular database. Users might prefer to execute the **CREATE FUNCTION** command by hand, instead.

- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)
- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.

- Make Windows' « N. Central Asia Standard Time » timezone map to Asia/Novosibirsk, not Asia/Almaty (Magnus Hagander)

  Microsoft changed the DST behavior of this zone in the timezone update from KB976098. Asia/Novosibirsk is a better match to its new behavior.

E.38. Release 8.4.4

Release Date

2010-05-17

This release contains a variety of fixes from 8.4.3. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.38.1. Migration to Version 8.4.4

A dump/restore is not required for those running 8.4.X. However, if you are upgrading from a version earlier than 8.4.2, see Sec-

tion E.40, « Release 8.4.2 ».

E.38.2. Changes

- Enforce restrictions in `plperl` using an opmask applied to the whole interpreter, instead of using `Safe.pm` (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that `Safe.pm` is too insecure to rely on for making `plperl` trustable. This change removes use of `Safe.pm` altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's `strict` pragma in a natural way in `plperl`, and that Perl's `$a` and `$b` variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted « normal » Tcl interpreter unless we are really going to execute a `pltclu` function. (CVE-2010-1170)

- Fix data corruption during WAL replay of `ALTER ... SET TABLESPACE` (Tom)

  When `archive_mode` is on, `ALTER ... SET TABLESPACE` generates a WAL record whose replay logic was incorrect. It could write the data to the wrong place, leading to possibly-unrecoverable data corruption. Data corruption would be observed on standby slaves, and could occur on the master as well if a database crash and recovery occurred after committing the `ALTER` and before the next checkpoint.

- Fix possible crash if a cache reset message is received during rebuild of a relcache entry (Heikki)

  This error was introduced in 8.4.3 while fixing a related failure.

- Apply per-function GUC settings while running the language validator for the function (Itagaki Takahiro)

  This avoids failures if the function's code is invalid without the setting; an example is that SQL functions may not parse if the `search_path` is not correct.

- Do constraint exclusion for alled **UPDATE** and **DELETE** target tables when `constraint_exclusion` = `partition` (Tom)

  Due to an oversight, this setting previously only caused constraint exclusion to be checked in **SELECT** commands.

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran `ALTER USER ... RESET ALL` for himself, or `ALTER DATABASE ... RESET ALL` for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the **ALTER** will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a `CONTEXT` addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Fix erroneous handling of `%r` parameter in `recovery_end_command` (Heikki)

  The value always came out zero.

- Ensure the archiver process responds to changes in `archive_command` as soon as possible (Tom)
- Fix pl/pgsql's `CASE` statement to not fail when the case expression is a query that returns no rows (Tom)
- Update pl/perl's `ppport.h` for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Handle empty-string connect parameters properly in ecpg (Michael)
- Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)
- Fix psql's `\copy` to not add spaces around a dot within `\copy (select ...)` (Tom)

  Addition of spaces around the decimal point in a numeric literal would result in a syntax error.

- Avoid formatting failure in psql when running in a locale context that doesn't match the `client_encoding` (Tom)
- Fix unnecessary « GIN indexes do not support whole-index scans » errors for unsatisfiable queries using `contrib/intarray` operators (Tom)
- Ensure that `contrib/pgstattuple` functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that `shmget()` returns `EINVAL` for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.
- Avoid possible crashes in syslogger process on Windows (Heikki)
- Deal more robustly with incomplete time zone information in the Windows registry (Magnus)
- Update the set of known Windows time zone names (Magnus)
- Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

  Also, add `PKST` (Pakistan Summer Time) to the default set of timezone abbreviations.

E.39. Release 8.4.3

Release Date

2010-03-15

This release contains a variety of fixes from 8.4.2. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.39.1. Migration to Version 8.4.3

A dump/restore is not required for those running 8.4.X. However, if you are upgrading from a version earlier than 8.4.2, see Section E.40, « Release 8.4.2 ».

E.39.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for an SSL connection (Magnus)

  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.
- Fix possible deadlock during backend startup (Tom)
- Fix possible crashes due to not handling errors during relcache reload cleanly (Tom)
- Fix possible crash due to use of dangling pointer to a cached plan (Tatsuo)
- Fix possible crash due to overenthusiastic invalidation of cached plan for **ROLLBACK** (Tom)
- Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)
- Fix server memory leak associated with use of savepoints and a client encoding different from server's encoding (Tom)
- Fix incorrect WAL data emitted during end-of-recovery cleanup of a GIST index page split (Yoichi Hirai)

  This would result in index corruption, or even more likely an error during WAL replay, if we were unlucky enough to crash during end-of-recovery cleanup after having completed an incomplete GIST insertion.
- Fix bug in WAL redo cleanup method for GIN indexes (Heikki)
- Fix incorrect comparison of scan key in GIN index search (Teodor)
- Make `substring()` for bit types treat any negative length as meaning « all the rest of the string » (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).

- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix bug occurring when trying to inline a SQL function that returns a set of a composite type that contains dropped columns (Tom)
- Fix bug with trying to update a field of an element of a composite-type array column (Tom)
- Avoid failure when **EXPLAIN** has to print a FieldStore or assignment ArrayRef expression (Tom)

  These cases can arise now that **EXPLAIN VERBOSE** tries to print plan node target lists.
- Avoid an unnecessary coercion failure in some cases where an undecorated literal string appears in a subquery within **UNION/INTERSECT/EXCEPT** (Tom)

  This fixes a regression for some cases that worked before 8.4.
- Avoid undesirable rowtype compatibility check failures in some cases where a whole-row Var has a rowtype that contains dropped columns (Tom)
- Fix the `STOP WAL LOCATION` entry in backup history files to report the next WAL segment's name when the end location is exactly at a segment boundary (Itagaki Takahiro)
- Always pass the catalog ID to an option validator function specified in **CREATE FOREIGN DATA WRAPPER** (Martin Pihlak)
- Fix some more cases of temporary-file leakage (Heikki)

  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function's exception handler.
- Add support for doing `FULL JOIN ON FALSE` (Tom)

  This prevents a regression from pre-8.4 releases for some queries that can now be simplified to a constant-false join condition.
- Improve constraint exclusion processing of boolean-variable cases, in particular make it possible to exclude a partition that has a « bool_column = false » constraint (Tom)
- Prevent treating an `INOUT` cast as representing binary compatibility (Heikki)
- Include column name in the message when warning about inability to grant or revoke column-level privileges (Stephen Frost)

  This is more useful than before and helps to prevent confusion when a **REVOKE** generates multiple messages, which formerly appeared to be duplicates.
- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.
- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)
- Fix possible infinite loop if `SSL_read` or `SSL_write` fails without setting `errno` (Tom)

  This is reportedly possible with some Windows versions of openssl.
- Disallow GSSAPI authentication on local connections, since it requires a hostname to function correctly (Magnus)
- Protect ecpg against applications freeing strings unexpectedly (Michael)
- Make ecpg report the proper SQLSTATE if the connection disappears (Michael)
- Fix translation of cell contents in psql `\d` output (Heikki)
- Fix psql's `numericlocale` option to not format strings it shouldn't in latex and troff output formats (Heikki)
- Fix a small per-query memory leak in psql (Tom)
- Make psql return the correct exit status (3) when `ON_ERROR_STOP` and `--single-transaction` are both specified and an error occurs during the implied **COMMIT** (Bruce)

- Fix pg_dump's output of permissions for foreign servers (Heikki)
- Fix possible crash in parallel pg_restore due to out-of-range dependency IDs (Tom)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Fix possible failure when calling PL/Perl functions from PL/PerlU or vice versa (Tim Bunce)
- Add `volatile` markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.
- Prevent `ExecutorEnd` from being run on portals created within a failed transaction or subtransaction (Tom)

  This is known to cause issues when using `contrib/auto_explain`.
- Prevent crash in `contrib/dblink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)
- Allow zero-dimensional arrays in `contrib/ltree` operations (Tom)

  This case was formerly rejected as an error, but it's more convenient to treat it the same as a zero-element array. In particular this avoids unnecessary failures when an ltree operation is applied to the result of `ARRAY(SELECT ...)` and the sub-select returns no rows.
- Fix assorted crashes in `contrib/xml2` caused by sloppy memory management (Tom)
- Make building of `contrib/xml2` more robust on Windows (Andrew)
- Fix race condition in Windows signal handling (Radu Ilie)

  One known symptom of this bug is that rows in pg_listener could be dropped under heavy load.
- Make the configure script report failure if the C compiler does not provide a working 64-bit integer datatype (Tom)

  This case has been broken for some time, and no longer seems worth supporting, so just reject it at configure time instead.
- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.40. Release 8.4.2

Release Date

2009-12-14

This release contains a variety of fixes from 8.4.1. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.40.1. Migration to Version 8.4.2

A dump/restore is not required for those running 8.4.X. However, if you have any hash indexes, you should **REINDEX** them after updating to 8.4.2, to repair possible damage.

E.40.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

  This change prevents allegedly-immutable index functions from possibly subverting a superuser's session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).
- Fix hash index corruption (Tom)

  The 8.4 change that made hash indexes keep entries sorted by hash value failed to update the bucket splitting and compaction routines to preserve the ordering. So application of either of those operations could lead to permanent corruption of an index, in the sense that searches might fail to find entries that are present. To deal with this, it is recommended to `REINDEX` any hash indexes you may have after installing this update.

- Fix possible crash during backend-startup-time cache initialization (Tom)
- Avoid crash on empty thesaurus dictionary (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)

  This fix prevents a PANIC if a `VACUUM FULL` is canceled after it's already committed its tuple movements, as well as transient errors if a plain `VACUUM` is interrupted after having truncated the table.
- Fix possible crash due to integer overflow in hash table size calculation (Tom)

  This could occur with extremely large planner estimates for the size of a hashjoin's result.
- Fix crash if a `DROP` is attempted on an internally-dependent object (Tom)
- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix memory leak in syslogger process when rotating to a new CSV logfile (Tom)
- Fix memory leak in postmaster when re-parsing `pg_hba.conf` (Tom)
- Fix Windows permission-downgrade logic (Jesse Morris)

  This fixes some cases where the database failed to start on Windows, often with misleading error messages such as « could not locate matching postgres executable ».
- Make `FOR UPDATE/SHARE` in the primary query not propagate into `WITH` queries (Tom)

  For example, in

  ```
  WITH w AS (SELECT * FROM foo) SELECT * FROM w, bar ... FOR UPDATE
  ```

  the `FOR UPDATE` will now affect `bar` but not `foo`. This is more useful and consistent than the original 8.4 behavior, which tried to propagate `FOR UPDATE` into the `WITH` query but always failed due to assorted implementation restrictions. It also follows the design rule that `WITH` queries are executed as if independent of the main query.
- Fix bug with a `WITH RECURSIVE` query immediately inside another one (Tom)
- Fix concurrency bug in hash indexes (Tom)

  Concurrent insertions could cause index scans to transiently report wrong results.
- Fix incorrect logic for GiST index page splits, when the split depends on a non-first column of the index (Paul Ramsey)
- Fix wrong search results for a multi-column GIN index with `fastupdate` enabled (Teodor)
- Fix bugs in WAL entry creation for GIN indexes (Tom)

  These bugs were masked when `full_page_writes` was on, but with it off a WAL replay failure was certain if a crash occurred before the next checkpoint.
- Don't error out if recycling or removing an old WAL file fails at the end of checkpoint (Heikki)

  It's better to treat the problem as non-fatal and allow the checkpoint to complete. Future checkpoints will retry the removal. Such problems are not expected in normal operation, but have been seen to be caused by misdesigned Windows anti-virus and backup software.
- Ensure WAL files aren't repeatedly archived on Windows (Heikki)

  This is another symptom that could happen if some other process interfered with deletion of a no-longer-needed file.
- Fix PAM password processing to be more robust (Tom)

  The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Raise the maximum authentication token (Kerberos ticket) size in GSSAPI and SSPI authentication methods (Ian Turner)

  While the old 2000-byte limit was more than enough for Unix Kerberos implementations, tickets issued by Windows Domain Controllers can be much larger.
- Ensure that domain constraints are enforced in constructs like `ARRAY[...]::domain`, where the domain is over an array

type (Heikki)

- Fix foreign-key logic for some cases involving composite-type columns as foreign keys (Tom)
- Ensure that a cursor's snapshot is not modified after it is created (Alvaro)

  This could lead to a cursor delivering wrong results if later operations in the same transaction modify the data the cursor is supposed to return.
- Fix `CREATE TABLE` to properly merge default expressions coming from different allance parent tables (Tom)

  This used to work but was broken in 8.4.
- Re-enable collection of access statistics for sequences (Akira Kurosawa)

  This used to work but was broken in 8.3.
- Fix processing of ownership dependencies during `CREATE OR REPLACE FUNCTION` (Tom)
- Fix incorrect handling of `WHERE` *x*=*x* conditions (Tom)

  In some cases these could get ignored as redundant, but they aren't -- they're equivalent to *x* `IS NOT NULL`.
- Fix incorrect plan construction when using hash aggregation to implement `DISTINCT` for textually identical volatile expressions (Tom)
- Fix Assert failure for a volatile `SELECT DISTINCT ON` expression (Tom)
- Fix `ts_stat()` to not fail on an empty tsvector value (Tom)
- Make text search parser accept underscores in XML attributes (Peter)
- Fix encoding handling in xml binary input (Heikki)

  If the XML header doesn't specify an encoding, we now assume UTF-8 by default; the previous handling was inconsistent.
- Fix bug with calling `plperl` from `plperlu` or vice versa (Tom)

  An error exit from the inner function could result in crashes due to failure to re-select the correct Perl interpreter for the outer function.
- Fix session-lifespan memory leak when a PL/Perl function is redefined (Tom)
- Ensure that Perl arrays are properly converted to PostgreSQL™ arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)

  This worked correctly already for non-set-returning functions.
- Fix rare crash in exception processing in PL/Python (Peter)
- Fix ecpg problem with comments in `DECLARE CURSOR` statements (Michael)
- Fix ecpg to not treat recently-added keywords as reserved words (Tom)

  This affected the keywords `CALLED`, `CATALOG`, `DEFINER`, `ENUM`, `FOLLOWING`, `INVOKER`, `OPTIONS`, `PARTITION`, `PRECEDING`, `RANGE`, `SECURITY`, `SERVER`, `UNBOUNDED`, and `WRAPPER`.
- Re-allow regular expression special characters in psql's `\df` function name parameter (Tom)
- In `contrib/fuzzystrmatch`, correct the calculation of `levenshtein` distances with non-default costs (Marcin Mank)
- In `contrib/pg_standby`, disable triggering failover with a signal on Windows (Fujii Masao)

  This never did anything useful, because Windows doesn't have Unix-style signals, but recent changes made it actually crash.
- Put `FREEZE` and `VERBOSE` options in the right order in the `VACUUM` command that `contrib/vacuumdb` produces (Heikki)
- Fix possible leak of connections when `contrib/dblink` encounters an error (Tatsuhito Kasahara)
- Ensure psql's flex module is compiled with the correct system header definitions (Tom)

  This fixes build failures on platforms where `--enable-largefile` causes incompatible changes in the generated code.
- Make the postmaster ignore any `application_name` parameter in connection request packets, to improve compatibility with future libpq versions (Tom)
- Update the timezone abbreviation files to match current reality (Joachim Wieland)

This includes adding `IDT` to the default timezone abbreviation set.

- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.

E.41. Release 8.4.1

Release Date

2009-09-09

This release contains a variety of fixes from 8.4. For information about new features in the 8.4 major release, see Section E.42, « Release 8.4 ».

E.41.1. Migration to Version 8.4.1

A dump/restore is not required for those running 8.4.X.

E.41.2. Changes

- Fix WAL page header initialization at the end of archive recovery (Heikki)

  This could lead to failure to process the WAL in a subsequent archive recovery.

- Fix « cannot make new WAL entries during recovery » error (Tom)
- Fix problem that could make expired rows visible after a crash (Tom)

  This bug involved a page status bit potentially not being set correctly after a server crash.

- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)

- Make **LOAD** of an already-loaded loadable module into a no-op (Tom)

  Formerly, **LOAD** would attempt to unload and re-load the module, but this is unsafe and not all that useful.

- Make window function `PARTITION BY` and `ORDER BY` items always be interpreted as simple expressions (Tom)

  In 8.4.0 these lists were parsed following the rules used for top-level `GROUP BY` and `ORDER BY` lists. But this was not correct per the SQL standard, and it led to possible circularity.

- Fix several errors in planning of semi-joins (Tom)

  These led to wrong query results in some cases where `IN` or `EXISTS` was used together with another join.

- Fix handling of whole-row references to subqueries that are within an outer join (Tom)

  An example is `SELECT COUNT(ss.*) FROM ... LEFT JOIN (SELECT ...) ss ON ...`. Here, `ss.*` would be treated as `ROW(NULL,NULL,...)` for null-extended join rows, which is not the same as a simple NULL. Now it is treated as a simple NULL.

- Fix Windows shared-memory allocation code (Tsutomu Yamada, Magnus)

  This bug led to the often-reported « could not reattach to shared memory » error message.

- Fix locale handling with plperl (Heikki)

  This bug could cause the server's locale setting to change when a plperl function is called, leading to data corruption.

- Fix handling of reloptions to ensure setting one option doesn't force default values for others (Itagaki Takahiro)
- Ensure that a « fast shutdown » request will forcibly terminate open sessions, even if a « smart shutdown » was already in progress (Fujii Masao)
- Avoid memory leak for `array_agg()` in `GROUP BY` queries (Tom)
- Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with `'HH'/'HH12'` (Heikki)

It was previously handled as `'th'` (lowercase).

- Include the fractional part in the result of `EXTRACT(second)` and `EXTRACT(milliseconds)` for time and time with time zone inputs (Tom)

  This has always worked for floating-point datetime configurations, but was broken in the integer datetime code.
- Fix overflow for `INTERVAL 'x ms'` when *x* is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Improve performance when processing toasted values in index scans (Tom)

  This is particularly useful for *PostGIS*.
- Fix a typo that disabled `commit_delay` (Jeff Janes)
- Output early-startup messages to `postmaster.log` if the server is started in silent mode (Tom)

  Previously such error messages were discarded, leading to difficulty in debugging.
- Remove translated FAQs (Peter)

  They are now on the *wiki*. The main FAQ was moved to the wiki some time ago.
- Fix pg_ctl to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)
- Fix several errors in pg_dump's `--binary-upgrade` mode (Bruce, Tom)

  `pg_dump --binary-upgrade` is used by pg_migrator.
- Fix `contrib/xml2`'s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)
- Improve robustness of libpq's code to recover from errors during **COPY FROM STDIN** (Tom)
- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)
- Work around gcc bug that causes « floating-point exception » instead of « division by zero » on some platforms (Tom)
- Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Mauritius.

E.42. Release 8.4

Release Date

2009-07-01

E.42.1. Overview

After many years of development, PostgreSQL™ has become feature-complete in many areas. This release shows a targeted approach to adding features (e.g., authentication, monitoring, space reuse), and adds capabilities defined in the later SQL standards. The major areas of enhancement are:

- Windowing Functions
- Common Table Expressions and Recursive Queries
- Default and variadic parameters for functions
- Parallel Restore
- Column Permissions
- Per-database locale settings
- Improved hash indexes
- Improved join performance for `EXISTS` and `NOT EXISTS` queries
- Easier-to-use Warm Standby
- Automatic sizing of the Free Space Map
- Visibility Map (greatly reduces vacuum overhead for slowly-changing tables)

- Version-aware psql (backslash commands work against older servers)
- Support SSL certificates for user authentication
- Per-function runtime statistics
- Easy editing of functions in psql
- New contrib modules: pg_stat_statements, auto_explain, citext, btree_gin

The above items are explained in more detail in the sections below.

E.42.2. Migration to Version 8.4

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

E.42.2.1. General

- Use 64-bit integer datetimes by default (Neil Conway)

  Previously this was selected by configure's `--enable-integer-datetimes` option. To retain the old behavior, build with `--disable-integer-datetimes`.
- Remove ipcclean utility command (Bruce)

  The utility only worked on a few platforms. Users should use their operating system tools instead.

E.42.2.2. Server Settings

- Change default setting for `log_min_messages` to `warning` (previously it was `notice`) to reduce log file volume (Tom)
- Change default setting for `max_prepared_transactions` to zero (previously it was 5) (Tom)
- Make `debug_print_parse`, `debug_print_rewritten`, and `debug_print_plan` output appear at `LOG` message level, not `DEBUG1` as formerly (Tom)
- Make `debug_pretty_print` default to `on` (Tom)
- Remove `explain_pretty_print` parameter (no longer needed) (Tom)
- Make `log_temp_files` settable by superusers only, like other logging options (Simon Riggs)
- Remove automatic appending of the epoch timestamp when no `%` escapes are present in `log_filename` (Robert Haas)

  This change was made because some users wanted a fixed log filename, for use with an external log rotation tool.
- Remove `log_restartpoints` from `recovery.conf`; instead use `log_checkpoints` (Simon)
- Remove `krb_realm` and `krb_server_hostname`; these are now set in `pg_hba.conf` instead (Magnus)
- There are also significant changes in `pg_hba.conf`, as described below.

E.42.2.3. Queries

- Change **TRUNCATE** and **LOCK** to apply to child tables of the specified table(s) (Peter)

  These commands now accept an `ONLY` option that prevents processing child tables; this option must be used if the old behavior is needed.
- **SELECT DISTINCT** and `UNION`/`INTERSECT`/`EXCEPT` no longer always produce sorted output (Tom)

  Previously, these types of queries always removed duplicate rows by means of Sort/Unique processing (i.e., sort then remove adjacent duplicates). Now they can be implemented by hashing, which will not produce sorted output. If an application relied on the output being in sorted order, the recommended fix is to add an `ORDER BY` clause. As a short-term workaround, the previous behavior can be restored by disabling `enable_hashagg`, but that is a very performance-expensive fix. `SELECT DISTINCT ON` never uses hashing, however, so its behavior is unchanged.
- Force child tables to all `CHECK` constraints from parents (Alex Hunsaker, Nikhil Sontakke, Tom)

Formerly it was possible to drop such a constraint from a child table, allowing rows that violate the constraint to be visible when scanning the parent table. This was deemed inconsistent, as well as contrary to SQL standard.

- Disallow negative `LIMIT` or `OFFSET` values, rather than treating them as zero (Simon)
- Disallow **LOCK TABLE** outside a transaction block (Tom)

  Such an operation is useless because the lock would be released immediately.
- Sequences now contain an additional *`start_value`* column (Zoltan Boszormenyi)

  This supports **ALTER SEQUENCE ... RESTART**.

E.42.2.4. Functions and Operators

- Make numeric zero raised to a fractional power return `0`, rather than throwing an error, and make numeric zero raised to the zero power return `1`, rather than error (Bruce)

  This matches the longstanding float8 behavior.
- Allow unary minus of floating-point values to produce minus zero (Tom)

  The changed behavior is more IEEE-standard compliant.
- Throw an error if an escape character is the last character in a `LIKE` pattern (i.e., it has nothing to escape) (Tom)

  Previously, such an escape character was silently ignored, thus possibly masking application logic errors.
- Remove `~=~` and `~<>~` operators formerly used for `LIKE` index comparisons (Tom)

  Pattern indexes now use the regular equality operator.
- `xpath()` now passes its arguments to libxml without any changes (Andrew)

  This means that the XML argument must be a well-formed XML document. The previous coding attempted to allow XML fragments, but it did not work well.
- Make `xmlelement()` format attribute values just like content values (Peter)

  Previously, attribute values were formatted according to the normal SQL output behavior, which is sometimes at odds with XML rules.
- Rewrite memory management for libxml-using functions (Tom)

  This change should avoid some compatibility problems with use of libxml in PL/Perl and other add-on code.
- Adopt a faster algorithm for hash functions (Kenneth Marshall, based on work of Bob Jenkins)

  Many of the built-in hash functions now deliver different results on little-endian and big-endian platforms.

E.42.2.4.1. Temporal Functions and Operators

- `DateStyle` no longer controls interval output formatting; instead there is a new variable `IntervalStyle` (Ron Mayer)
- Improve consistency of handling of fractional seconds in timestamp and interval output (Ron Mayer)

  This may result in displaying a different number of fractional digits than before, or rounding instead of truncating.
- Make `to_char()`'s localized month/day names depend on `LC_TIME`, not `LC_MESSAGES` (Euler Taveira de Oliveira)
- Cause `to_date()` and `to_timestamp()` to more consistently report errors for invalid input (Brendan Jurd)

  Previous versions would often ignore or silently misread input that did not match the format string. Such cases will now result in an error.
- Fix `to_timestamp()` to not require upper/lower case matching for meridian (`AM`/`PM`) and era (`BC`/`AD`) format designations (Brendan Jurd)

  For example, input value `ad` now matches the format string `AD`.

E.42.3. Changes

Below you will find a detailed account of the changes between PostgreSQL™ 8.4 and the previous major release.

E.42.3.1. Performance

- Improve optimizer statistics calculations (Jan Urbanski, Tom)

  In particular, estimates for full-text-search operators are greatly improved.
- Allow **SELECT DISTINCT** and `UNION`/`INTERSECT`/`EXCEPT` to use hashing (Tom)

  This means that these types of queries no longer automatically produce sorted output.
- Create explicit concepts of semi-joins and anti-joins (Tom)

  This work formalizes our previous ad-hoc treatment of `IN (SELECT ...)` clauses, and extends it to `EXISTS` and `NOT EXISTS` clauses. It should result in significantly better planning of `EXISTS` and `NOT EXISTS` queries. In general, logically equivalent `IN` and `EXISTS` clauses should now have similar performance, whereas previously `IN` often won.
- Improve optimization of sub-selects beneath outer joins (Tom)

  Formerly, a sub-select or view could not be optimized very well if it appeared within the nullable side of an outer join and contained non-strict expressions (for instance, constants) in its result list.
- Improve the performance of `text_position()` and related functions by using Boyer-Moore-Horspool searching (David Rowley)

  This is particularly helpful for long search patterns.
- Reduce I/O load of writing the statistics collection file by writing the file only when requested (Martin Pihlak)
- Improve performance for bulk inserts (Robert Haas, Simon)
- Increase the default value of `default_statistics_target` from `10` to `100` (Greg Sabino Mullane, Tom)

  The maximum value was also increased from `1000` to `10000`.
- Perform `constraint_exclusion` checking by default in queries involving allance or `UNION ALL` (Tom)

  A new `constraint_exclusion` setting, `partition`, was added to specify this behavior.
- Allow I/O read-ahead for bitmap index scans (Greg Stark)

  The amount of read-ahead is controlled by `effective_io_concurrency`. This feature is available only if the kernel has `posix_fadvise()` support.
- Inline simple set-returning SQL functions in `FROM` clauses (Richard Rowell)
- Improve performance of multi-batch hash joins by providing a special case for join key values that are especially common in the outer relation (Bryce Cutt, Ramon Lawrence)
- Reduce volume of temporary data in multi-batch hash joins by suppressing « physical tlist » optimization (Michael Henderson, Ramon Lawrence)
- Avoid waiting for idle-in-transaction sessions during **CREATE INDEX CONCURRENTLY** (Simon)
- Improve performance of shared cache invalidation (Tom)

E.42.3.2. Server

E.42.3.2.1. Settings

- Convert many `postgresql.conf` settings to enumerated values so that `pg_settings` can display the valid values (Magnus)
- Add `cursor_tuple_fraction` parameter to control the fraction of a cursor's rows that the planner assumes will be fetched (Robert Hell)
- Allow underscores in the names of custom variable classes in `postgresql.conf` (Tom)

E.42.3.2.2. Authentication and security

- Remove support for the (insecure) `crypt` authentication method (Magnus)

This effectively obsoletes pre-PostgreSQL™ 7.2 client libraries, as there is no longer any non-plaintext password method that they can use.

- Support regular expressions in `pg_ident.conf` (Magnus)
- Allow Kerberos™/GSSAPI parameters to be changed without restarting the postmaster (Magnus)
- Support SSL certificate chains in server certificate file (Andrew Gierth)

  Including the full certificate chain makes the client able to verify the certificate without having all intermediate CA certificates present in the local store, which is often the case for commercial CAs.
- Report appropriate error message for combination of `MD5` authentication and `db_user_namespace` enabled (Bruce)

E.42.3.2.3. `pg_hba.conf`

- Change all authentication options to use `name=value` syntax (Magnus)

  This makes incompatible changes to the `ldap`, `pam` and `ident` authentication methods. All `pg_hba.conf` entries with these methods need to be rewritten using the new format.
- Remove the `ident sameuser` option, instead making that behavior the default if no usermap is specified (Magnus)
- Allow a usermap parameter for all external authentication methods (Magnus)

  Previously a usermap was only supported for `ident` authentication.
- Add `clientcert` option to control requesting of a client certificate (Magnus)

  Previously this was controlled by the presence of a root certificate file in the server's data directory.
- Add `cert` authentication method to allow *user* authentication via SSL certificates (Magnus)

  Previously SSL certificates could only verify that the client had access to a certificate, not authenticate a user.
- Allow `krb5`, `gssapi` and `sspi` realm and `krb5` host settings to be specified in `pg_hba.conf` (Magnus)

  These override the settings in `postgresql.conf`.
- Add `include_realm` parameter for `krb5`, `gssapi`, and `sspi` methods (Magnus)

  This allows identical usernames from different realms to be authenticated as different database users using usermaps.
- Parse `pg_hba.conf` fully when it is loaded, so that errors are reported immediately (Magnus)

  Previously, most errors in the file wouldn't be detected until clients tried to connect, so an erroneous file could render the system unusable. With the new behavior, if an error is detected during reload then the bad file is rejected and the postmaster continues to use its old copy.
- Show all parsing errors in `pg_hba.conf` instead of aborting after the first one (Selena Deckelmann)
- Support `ident` authentication over Unix-domain sockets on Solaris™ (Garick Hamlin)

E.42.3.2.4. Continuous Archiving

- Provide an option to `pg_start_backup()` to force its implied checkpoint to finish as quickly as possible (Tom)

  The default behavior avoids excess I/O consumption, but that is pointless if no concurrent query activity is going on.
- Make `pg_stop_backup()` wait for modified WAL files to be archived (Simon)

  This guarantees that the backup is valid at the time `pg_stop_backup()` completes.
- When archiving is enabled, rotate the last WAL segment at shutdown so that all transactions can be archived immediately (Guillaume Smet, Heikki)
- Delay « smart » shutdown while a continuous archiving base backup is in progress (Laurenz Albe)
- Cancel a continuous archiving base backup if « fast » shutdown is requested (Laurenz Albe)
- Allow `recovery.conf` boolean variables to take the same range of string values as `postgresql.conf` boolean variables (Bruce)

E.42.3.2.5. Monitoring

- Add `pg_conf_load_time()` to report when the PostgreSQL™ configuration files were last loaded (George Gensure)
- Add `pg_terminate_backend()` to safely terminate a backend (the `SIGTERM` signal works also) (Tom, Bruce)

  While it's always been possible to `SIGTERM` a single backend, this was previously considered unsupported; and testing of the case found some bugs that are now fixed.
- Add ability to track user-defined functions' call counts and runtimes (Martin Pihlak)

  Function statistics appear in a new system view, `pg_stat_user_functions`. Tracking is controlled by the new parameter `track_functions`.
- Allow specification of the maximum query string size in `pg_stat_activity` via new `track_activity_query_size` parameter (Thomas Lee)
- Increase the maximum line length sent to syslog, in hopes of improving performance (Tom)
- Add read-only configuration variables `segment_size`, `wal_block_size`, and `wal_segment_size` (Bernd Helmle)
- When reporting a deadlock, report the text of all queries involved in the deadlock to the server log (Itagaki Takahiro)
- Add `pg_stat_get_activity(pid)` function to return information about a specific process id (Magnus)
- Allow the location of the server's statistics file to be specified via `stats_temp_directory` (Magnus)

  This allows the statistics file to be placed in a RAM-resident directory to reduce I/O requirements. On startup/shutdown, the file is copied to its traditional location (`$PGDATA/global/`) so it is preserved across restarts.

E.42.3.3. Queries

- Add support for `WINDOW` functions (Hitoshi Harada)
- Add support for `WITH` clauses (CTEs), including `WITH RECURSIVE` (Yoshiyuki Asaba, Tatsuo Ishii, Tom)
- Add **TABLE** command (Peter)

  `TABLE tablename` is a SQL standard short-hand for `SELECT * FROM tablename`.
- Allow `AS` to be optional when specifying a **SELECT** (or `RETURNING`) column output label (Hiroshi Saito)

  This works so long as the column label is not any PostgreSQL™ keyword; otherwise `AS` is still needed.
- Support set-returning functions in **SELECT** result lists even for functions that return their result via a tuplestore (Tom)

  In particular, this means that functions written in PL/pgSQL and other PL languages can now be called this way.
- Support set-returning functions in the output of aggregation and grouping queries (Tom)
- Allow **SELECT FOR UPDATE**/`SHARE` to work on allance trees (Tom)
- Add infrastructure for SQL/MED (Martin Pihlak, Peter)

  There are no remote or external SQL/MED capabilities yet, but this change provides a standardized and future-proof system for managing connection information for modules like `dblink` and `plproxy`.
- Invalidate cached plans when referenced schemas, functions, operators, or operator classes are modified (Martin Pihlak, Tom)

  This improves the system's ability to respond to on-the-fly DDL changes.
- Allow comparison of composite types and allow arrays of anonymous composite types (Tom)

  This allows constructs such as `row(1, 1.1) = any (array[row(7, 7.7), row(1, 1.0)])`. This is particularly useful in recursive queries.
- Add support for Unicode string literal and identifier specifications using code points, e.g. `U&'d\0061t\+000061'` (Peter)
- Reject `\000` in string literals and **COPY** data (Tom)

  Previously, this was accepted but had the effect of terminating the string contents.
- Improve the parser's ability to report error locations (Tom)

  An error location is now reported for many semantic errors, such as mismatched datatypes, that previously could not be locali-

zed.

E.42.3.3.1. TRUNCATE

- Support statement-level `ON TRUNCATE` triggers (Simon)
- Add `RESTART`/`CONTINUE IDENTITY` options for **TRUNCATE TABLE** (Zoltan Boszormenyi)

  The start value of a sequence can be changed by **ALTER SEQUENCE START WITH**.
- Allow **TRUNCATE tab1, tab1** to succeed (Bruce)
- Add a separate **TRUNCATE** permission (Robert Haas)

E.42.3.3.2. EXPLAIN

- Make **EXPLAIN VERBOSE** show the output columns of each plan node (Tom)

  Previously **EXPLAIN VERBOSE** output an internal representation of the query plan. (That behavior is now available via `debug_print_plan`.)
- Make **EXPLAIN** identify subplans and initplans with individual labels (Tom)
- Make **EXPLAIN** honor `debug_print_plan` (Tom)
- Allow **EXPLAIN** on **CREATE TABLE AS** (Peter)

E.42.3.3.3. `LIMIT`/`OFFSET`

- Allow sub-selects in `LIMIT` and `OFFSET` (Tom)
- Add SQL-standard syntax for `LIMIT`/`OFFSET` capabilities (Peter)

  To wit, `OFFSET num {ROW|ROWS} FETCH {FIRST|NEXT} [num] {ROW|ROWS} ONLY`.

E.42.3.4. Object Manipulation

- Add support for column-level privileges (Stephen Frost, KaiGai Kohei)
- Refactor multi-object **DROP** operations to reduce the need for `CASCADE` (Alex Hunsaker)

  For example, if table `B` has a dependency on table `A`, the command `DROP TABLE A, B` no longer requires the `CASCADE` option.
- Fix various problems with concurrent **DROP** commands by ensuring that locks are taken before we begin to drop dependencies of an object (Tom)
- Improve reporting of dependencies during **DROP** commands (Tom)
- Add `WITH [NO] DATA` clause to **CREATE TABLE AS**, per the SQL standard (Peter, Tom)
- Add support for user-defined I/O conversion casts (Heikki)
- Allow **CREATE AGGREGATE** to use an internal transition datatype (Tom)
- Add `LIKE` clause to **CREATE TYPE** (Tom)

  This simplifies creation of data types that use the same internal representation as an existing type.
- Allow specification of the type category and « preferred » status for user-defined base types (Tom)

  This allows more control over the coercion behavior of user-defined types.
- Allow **CREATE OR REPLACE VIEW** to add columns to the end of a view (Robert Haas)

E.42.3.4.1. ALTER

- Add **ALTER TYPE RENAME** (Petr Jelinek)

- Add **ALTER SEQUENCE ... RESTART** (with no parameter) to reset a sequence to its initial value (Zoltan Boszormenyi)
- Modify the **ALTER TABLE** syntax to allow all reasonable combinations for tables, indexes, sequences, and views (Tom)

  This change allows the following new syntaxes:

  - **ALTER SEQUENCE OWNER TO**
  - **ALTER VIEW ALTER COLUMN SET/DROP DEFAULT**
  - **ALTER VIEW OWNER TO**
  - **ALTER VIEW SET SCHEMA**

  There is no actual new functionality here, but formerly you had to say **ALTER TABLE** to do these things, which was confusing.
- Add support for the syntax **ALTER TABLE ... ALTER COLUMN ... SET DATA TYPE** (Peter)

  This is SQL-standard syntax for functionality that was already supported.
- Make **ALTER TABLE SET WITHOUT OIDS** rewrite the table to physically remove OID values (Tom)

  Also, add **ALTER TABLE SET WITH OIDS** to rewrite the table to add OIDs.

E.42.3.4.2. Database Manipulation

- Improve reporting of **CREATE/DROP/RENAME DATABASE** failure when uncommitted prepared transactions are the cause (Tom)
- Make `LC_COLLATE` and `LC_CTYPE` into per-database settings (Radek Strnad, Heikki)

  This makes collation similar to encoding, which was always configurable per database.
- Improve checks that the database encoding, collation (`LC_COLLATE`), and character classes (`LC_CTYPE`) match (Heikki, Tom)

  Note in particular that a new database's encoding and locale settings can be changed only when copying from `template0`. This prevents possibly copying data that doesn't match the settings.
- Add **ALTER DATABASE SET TABLESPACE** to move a database to a new tablespace (Guillaume Lelarge, Bernd Helmle)

E.42.3.5. Utility Operations

- Add a `VERBOSE` option to the **CLUSTER** command and clusterdb (Jim Cox)
- Decrease memory requirements for recording pending trigger events (Tom)

E.42.3.5.1. Indexes

- Dramatically improve the speed of building and accessing hash indexes (Tom Raney, Shreya Bhargava)

  This allows hash indexes to be sometimes faster than btree indexes. However, hash indexes are still not crash-safe.
- Make hash indexes store only the hash code, not the full value of the indexed column (Xiao Meng)

  This greatly reduces the size of hash indexes for long indexed values, improving performance.
- Implement fast update option for GIN indexes (Teodor, Oleg)

  This option greatly improves update speed at a small penalty in search speed.
- `xxx_pattern_ops` indexes can now be used for simple equality comparisons, not only for `LIKE` (Tom)

E.42.3.5.2. Full Text Indexes

- Remove the requirement to use `@@@` when doing GIN weighted lookups on full text indexes (Tom, Teodor)

  The normal `@@` text search operator can be used instead.
- Add an optimizer selectivity function for `@@` text search operations (Jan Urbanski)

- Allow prefix matching in full text searches (Teodor Sigaev, Oleg Bartunov)
- Support multi-column GIN indexes (Teodor Sigaev)
- Improve support for Nepali language and Devanagari alphabet (Teodor)

E.42.3.5.3. VACUUM

- Track free space in separate per-relation « fork » files (Heikki)

  Free space discovered by **VACUUM** is now recorded in `*_fsm` files, rather than in a fixed-sized shared memory area. The `max_fsm_pages` and `max_fsm_relations` settings have been removed, greatly simplifying administration of free space management.
- Add a visibility map to track pages that do not require vacuuming (Heikki)

  This allows **VACUUM** to avoid scanning all of a table when only a portion of the table needs vacuuming. The visibility map is stored in per-relation « fork » files.
- Add `vacuum_freeze_table_age` parameter to control when **VACUUM** should ignore the visibility map and do a full table scan to freeze tuples (Heikki)
- Track transaction snapshots more carefully (Alvaro)

  This improves **VACUUM**'s ability to reclaim space in the presence of long-running transactions.
- Add ability to specify per-relation autovacuum and TOAST parameters in **CREATE TABLE** (Alvaro, Euler Taveira de Oliveira)

  Autovacuum options used to be stored in a system table.
- Add `--freeze` option to vacuumdb (Bruce)

E.42.3.6. Data Types

- Add a `CaseSensitive` option for text search synonym dictionaries (Simon)
- Improve the precision of NUMERIC division (Tom)
- Add basic arithmetic operators for int2 with int8 (Tom)

  This eliminates the need for explicit casting in some situations.
- Allow UUID input to accept an optional hyphen after every fourth digit (Robert Haas)
- Allow `on`/`off` as input for the boolean data type (Itagaki Takahiro)
- Allow spaces around `NaN` in the input string for type numeric (Sam Mason)

E.42.3.6.1. Temporal Data Types

- Reject year `0 BC` and years `000` and `0000` (Tom)

  Previously these were interpreted as `1 BC`. (Note: years `0` and `00` are still assumed to be the year 2000.)
- Include `SGT` (Singapore time) in the default list of known time zone abbreviations (Tom)
- Support `infinity` and `-infinity` as values of type date (Tom)
- Make parsing of interval literals more standard-compliant (Tom, Ron Mayer)

  For example, `INTERVAL '1' YEAR` now does what it's supposed to.
- Allow interval fractional-seconds precision to be specified after the `second` keyword, for SQL standard compliance (Tom)

  Formerly the precision had to be specified after the keyword interval. (For backwards compatibility, this syntax is still supported, though deprecated.) Data type definitions will now be output using the standard format.
- Support the IS0 8601 interval syntax (Ron Mayer, Kevin Grittner)

  For example, `INTERVAL 'P1Y2M3DT4H5M6.7S'` is now supported.
- Add `IntervalStyle` parameter which controls how interval values are output (Ron Mayer)

Valid values are: `postgres`, `postgres_verbose`, `sql_standard`, `iso_8601`. This setting also controls the handling of negative interval input when only some fields have positive/negative designations.

- Improve consistency of handling of fractional seconds in timestamp and interval output (Ron Mayer)

E.42.3.6.2. Arrays

- Improve the handling of casts applied to `ARRAY[]` constructs, such as `ARRAY[...]::integer[]` (Brendan Jurd)

  Formerly PostgreSQL™ attempted to determine a data type for the `ARRAY[]` construct without reference to the ensuing cast. This could fail unnecessarily in many cases, in particular when the `ARRAY[]` construct was empty or contained only ambiguous entries such as `NULL`. Now the cast is consulted to determine the type that the array elements must be.
- Make SQL-syntax ARRAY dimensions optional to match the SQL standard (Peter)
- Add `array_ndims()` to return the number of dimensions of an array (Robert Haas)
- Add `array_length()` to return the length of an array for a specified dimension (Jim Nasby, Robert Haas, Peter Eisentraut)
- Add aggregate function `array_agg()`, which returns all aggregated values as a single array (Robert Haas, Jeff Davis, Peter)
- Add `unnest()`, which converts an array to individual row values (Tom)

  This is the opposite of `array_agg()`.
- Add `array_fill()` to create arrays initialized with a value (Pavel Stehule)
- Add `generate_subscripts()` to simplify generating the range of an array's subscripts (Pavel Stehule)

E.42.3.6.3. Wide-Value Storage (TOAST)

- Consider TOAST compression on values as short as 32 bytes (previously 256 bytes) (Greg Stark)
- Require 25% minimum space savings before using TOAST compression (previously 20% for small values and any-savings-at-all for large values) (Greg)
- Improve TOAST heuristics for rows that have a mix of large and small toastable fields, so that we prefer to push large values out of line and don't compress small values unnecessarily (Greg, Tom)

E.42.3.7. Functions

- Document that `setseed()` allows values from `-1` to `1` (not just `0` to `1`), and enforce the valid range (Kris Jurka)
- Add server-side function `lo_import(filename, oid)` (Tatsuo)
- Add `quote_nullable()`, which behaves like `quote_literal()` but returns the string `NULL` for a null argument (Brendan Jurd)
- Improve full text search `headline()` function to allow extracting several fragments of text (Sushant Sinha)
- Add `suppress_redundant_updates_trigger()` trigger function to avoid overhead for non-data-changing updates (Andrew)
- Add `div(numeric, numeric)` to perform numeric division without rounding (Tom)
- Add timestamp and timestamptz versions of `generate_series()` (Hitoshi Harada)

E.42.3.7.1. Object Information Functions

- Implement `current_query()` for use by functions that need to know the currently running query (Tomas Doran)
- Add `pg_get_keywords()` to return a list of the parser keywords (Dave Page)
- Add `pg_get_functiondef()` to see a function's definition (Abhijit Menon-Sen)
- Allow the second argument of `pg_get_expr()` to be zero when deparsing an expression that does not contain variables (Tom)
- Modify `pg_relation_size()` to use `regclass` (Heikki)

pg_relation_size(data_type_name) no longer works.

- Add boot_val and reset_val columns to pg_settings output (Greg Smith)
- Add source file name and line number columns to pg_settings output for variables set in a configuration file (Magnus, Alvaro)

  For security reasons, these columns are only visible to superusers.
- Add support for CURRENT_CATALOG, CURRENT_SCHEMA, SET CATALOG, SET SCHEMA (Peter)

  These provide SQL-standard syntax for existing features.
- Add pg_typeof() which returns the data type of any value (Brendan Jurd)
- Make version() return information about whether the server is a 32- or 64-bit binary (Bruce)
- Fix the behavior of information schema columns *is_insertable_into* and *is_updatable* to be consistent (Peter)
- Improve the behavior of information schema *datetime_precision* columns (Peter)

  These columns now show zero for date columns, and 6 (the default precision) for time, timestamp, and interval without a declared precision, rather than showing null as formerly.
- Convert remaining builtin set-returning functions to use OUT parameters (Jaime Casanova)

  This makes it possible to call these functions without specifying a column list: pg_show_all_settings(), pg_lock_status(), pg_prepared_xact(), pg_prepared_statement(), pg_cursor()
- Make pg_*_is_visible() and has_*_privilege() functions return NULL for invalid OIDs, rather than reporting an error (Tom)
- Extend has_*_privilege() functions to allow inquiring about the OR of multiple privileges in one call (Stephen Frost, Tom)
- Add has_column_privilege() and has_any_column_privilege() functions (Stephen Frost, Tom)

E.42.3.7.2. Function Creation

- Support variadic functions (functions with a variable number of arguments) (Pavel Stehule)

  Only trailing arguments can be optional, and they all must be of the same data type.
- Support default values for function arguments (Pavel Stehule)
- Add **CREATE FUNCTION ... RETURNS TABLE** clause (Pavel Stehule)
- Allow SQL-language functions to return the output of an **INSERT**/**UPDATE**/**DELETE** RETURNING clause (Tom)

E.42.3.7.3. PL/pgSQL Server-Side Language

- Support EXECUTE USING for easier insertion of data values into a dynamic query string (Pavel Stehule)
- Allow looping over the results of a cursor using a FOR loop (Pavel Stehule)
- Support RETURN QUERY EXECUTE (Pavel Stehule)
- Improve the RAISE command (Pavel Stehule)
  - Support DETAIL and HINT fields
  - Support specification of the SQLSTATE error code
  - Support an exception name parameter
  - Allow RAISE without parameters in an exception block to re-throw the current error
- Allow specification of SQLSTATE codes in EXCEPTION lists (Pavel Stehule)

  This is useful for handling custom SQLSTATE codes.
- Support the CASE statement (Pavel Stehule)

- Make **RETURN QUERY** set the special `FOUND` and **GET DIAGNOSTICS** `ROW_COUNT` variables (Pavel Stehule)
- Make **FETCH** and **MOVE** set the **GET DIAGNOSTICS** `ROW_COUNT` variable (Andrew Gierth)
- Make **EXIT** without a label always exit the innermost loop (Tom)

  Formerly, if there were a `BEGIN` block more closely nested than any loop, it would exit that block instead. The new behavior matches Oracle(TM) and is also what was previously stated by our own documentation.
- Make processing of string literals and nested block comments match the main SQL parser's processing (Tom)

  In particular, the format string in **RAISE** now works the same as any other string literal, including being subject to `standard_conforming_strings`. This change also fixes other cases in which valid commands would fail when `standard_conforming_strings` is on.
- Avoid memory leakage when the same function is called at varying exception-block nesting depths (Tom)

E.42.3.8. Client Applications

- Fix `pg_ctl restart` to preserve command-line arguments (Bruce)
- Add `-w`/`--no-password` option that prevents password prompting in all utilities that have a `-W`/`--password` option (Peter)
- Remove `-q` (quiet) option of createdb, createuser, dropdb, dropuser (Peter)

  These options have had no effect since PostgreSQL™ 8.3.

E.42.3.8.1. psql

- Remove verbose startup banner; now just suggest `help` (Joshua Drake)
- Make `help` show common backslash commands (Greg Sabino Mullane)
- Add `\pset format wrapped` mode to wrap output to the screen width, or file/pipe output too if `\pset columns` is set (Bryce Nesbitt)
- Allow all supported spellings of boolean values in **\pset**, rather than just `on` and `off` (Bruce)

  Formerly, any string other than « off » was silently taken to mean `true`. psql will now complain about unrecognized spellings (but still take them as `true`).
- Use the pager for wide output (Bruce)
- Require a space between a one-letter backslash command and its first argument (Bernd Helmle)

  This removes a historical source of ambiguity.
- Improve tab completion support for schema-qualified and quoted identifiers (Greg Sabino Mullane)
- Add optional `on`/`off` argument for **\timing** (David Fetter)
- Display access control rights on multiple lines (Brendan Jurd, Andreas Scherbaum)
- Make **\l** show database access privileges (Andrew Gilligan)
- Make **\l+** show database sizes, if permissions allow (Andrew Gilligan)
- Add the **\ef** command to edit function definitions (Abhijit Menon-Sen)

E.42.3.8.2. psql \d* commands

- Make **\d*** commands that do not have a pattern argument show system objects only if the `S` modifier is specified (Greg Sabino Mullane, Bruce)

  The former behavior was inconsistent across different variants of **\d**, and in most cases it provided no easy way to see just user objects.
- Improve **\d*** commands to work with older PostgreSQL™ server versions (back to 7.4), not only the current server version (Guillaume Lelarge)
- Make **\d** show foreign-key constraints that reference the selected table (Kenneth D'Souza)

- Make **\d** on a sequence show its column values (Euler Taveira de Oliveira)
- Add column storage type and other relation options to the **\d+** display (Gregory Stark, Euler Taveira de Oliveira)
- Show relation size in **\dt+** output (Dickson S. Guedes)
- Show the possible values of `enum` types in **\dT+** (David Fetter)
- Allow **\dC** to accept a wildcard pattern, which matches either datatype involved in the cast (Tom)
- Add a function type column to **\df**'s output, and add options to list only selected types of functions (David Fetter)
- Make **\df** not hide functions that take or return type cstring (Tom)

  Previously, such functions were hidden because most of them are datatype I/O functions, which were deemed uninteresting. The new policy about hiding system functions by default makes this wart unnecessary.

E.42.3.8.3. pg_dump

- Add a `--no-tablespaces` option to pg_dump/pg_dumpall/pg_restore so that dumps can be restored to clusters that have non-matching tablespace layouts (Gavin Roy)
- Remove `-d` and `-D` options from pg_dump and pg_dumpall (Tom)

  These options were too frequently confused with the option to select a database name in other PostgreSQL™ client applications. The functionality is still available, but you must now spell out the long option name `--inserts` or `--column-inserts`.
- Remove `-i`/`--ignore-version` option from pg_dump and pg_dumpall (Tom)

  Use of this option does not throw an error, but it has no effect. This option was removed because the version checks are necessary for safety.
- Disable `statement_timeout` during dump and restore (Joshua Drake)
- Add pg_dump/pg_dumpall option `--lock-wait-timeout` (David Gould)

  This allows dumps to fail if unable to acquire a shared lock within the specified amount of time.
- Reorder pg_dump `--data-only` output to dump tables referenced by foreign keys before the referencing tables (Tom)

  This allows data loads when foreign keys are already present. If circular references make a safe ordering impossible, a `NOTICE` is issued.
- Allow pg_dump, pg_dumpall, and pg_restore to use a specified role (Benedek László)
- Allow pg_restore to use multiple concurrent connections to do the restore (Andrew)

  The number of concurrent connections is controlled by the option `--jobs`. This is supported only for custom-format archives.

E.42.3.9. Programming Tools

E.42.3.9.1. libpq

- Allow the OID to be specified when importing a large object, via new function `lo_import_with_oid()` (Tatsuo)
- Add « events » support (Andrew Chernow, Merlin Moncure)

  This adds the ability to register callbacks to manage private data associated with PGconn and PGresult objects.
- Improve error handling to allow the return of multiple error messages as multi-line error reports (Magnus)
- Make `PQexecParams()` and related functions return `PGRES_EMPTY_QUERY` for an empty query (Tom)

  They previously returned `PGRES_COMMAND_OK`.
- Document how to avoid the overhead of `WSACleanup()` on Windows (Andrew Chernow)
- Do not rely on Kerberos tickets to determine the default database username (Magnus)

  Previously, a Kerberos-capable build of libpq would use the principal name from any available Kerberos ticket as default database username, even if the connection wasn't using Kerberos authentication. This was deemed inconsistent and confusing. The

default username is now determined the same way with or without Kerberos. Note however that the database username must still match the ticket when Kerberos authentication is used.

E.42.3.9.2. libpq SSL (Secure Sockets Layer) support

- Fix certificate validation for SSL connections (Magnus)

  libpq now supports verifying both the certificate and the name of the server when making SSL connections. If a root certificate is not available to use for verification, SSL connections will fail. The `sslmode` parameter is used to enable certificate verification and set the level of checking. The default is still not to do any verification, allowing connections to SSL-enabled servers without requiring a root certificate on the client.

- Support wildcard server certificates (Magnus)

  If a certificate CN starts with `*`, it will be treated as a wildcard when matching the hostname, allowing the use of the same certificate for multiple servers.

- Allow the file locations for client certificates to be specified (Mark Woodward, Alvaro, Magnus)
- Add a `PQinitOpenSSL` function to allow greater control over OpenSSL/libcrypto initialization (Andrew Chernow)
- Make libpq unregister its OpenSSL callbacks when no database connections remain open (Bruce, Magnus, Russell Smith)

  This is required for applications that unload the libpq library, otherwise invalid OpenSSL callbacks will remain.

E.42.3.9.3. ecpg

- Add localization support for messages (Euler Taveira de Oliveira)
- ecpg parser is now automatically generated from the server parser (Michael)

  Previously the ecpg parser was hand-maintained.

E.42.3.9.4. Server Programming Interface (SPI)

- Add support for single-use plans with out-of-line parameters (Tom)
- Add new `SPI_OK_REWRITTEN` return code for `SPI_execute()` (Heikki)

  This is used when a command is rewritten to another type of command.

- Remove unnecessary inclusions from `executor/spi.h` (Tom)

  SPI-using modules might need to add some `#include` lines if they were depending on `spi.h` to include things for them.

E.42.3.10. Build Options

- Update build system to use Autoconf™ 2.61 (Peter)
- Require GNU bison™ for source code builds (Peter)

  This has effectively been required for several years, but now there is no infrastructure claiming to support other parser tools.

- Add pg_config `--htmldir` option (Peter)
- Pass float4 by value inside the server (Zoltan Boszormenyi)

  Add configure option `--disable-float4-byval` to use the old behavior. External C functions that use old-style (version 0) call convention and pass or return float4 values will be broken by this change, so you may need the configure option if you have such functions and don't want to update them.

- Pass float8, int8, and related datatypes by value inside the server on 64-bit platforms (Zoltan Boszormenyi)

  Add configure option `--disable-float8-byval` to use the old behavior. As above, this change might break old-style external C functions.

- Add configure options `--with-segsize`, `--with-blocksize`, `--with-wal-blocksize`, `--with-wal-segsize` (Zdenek Kotala, Tom)

This simplifies build-time control over several constants that previously could only be changed by editing `pg_config_manual.h`.

- Allow threaded builds on Solaris™ 2.5 (Bruce)
- Use the system's `getopt_long()` on Solaris™ (Zdenek Kotala, Tom)

  This makes option processing more consistent with what Solaris users expect.
- Add support for the Sun Studio™ compiler on Linux™ (Julius Stroffek)
- Append the major version number to the backend gettext domain, and the `soname` major version number to libraries' gettext domain (Peter)

  This simplifies parallel installations of multiple versions.
- Add support for code coverage testing with gcov (Michelle Caisse)
- Allow out-of-tree builds on Mingw™ and Cygwin™ (Richard Evans)
- Fix the use of Mingw™ as a cross-compiling source platform (Peter)

E.42.3.11. Source Code

- Support 64-bit time zone data files (Heikki)

  This adds support for daylight saving time (DST) calculations beyond the year 2038.
- Deprecate use of platform's time_t data type (Tom)

  Some platforms have migrated to 64-bit time_t, some have not, and Windows can't make up its mind what it's doing. Define pg_time_t to have the same meaning as time_t, but always be 64 bits (unless the platform has no 64-bit integer type), and use that type in all module APIs and on-disk data formats.
- Fix bug in handling of the time zone database when cross-compiling (Richard Evans)
- Link backend object files in one step, rather than in stages (Peter)
- Improve gettext support to allow better translation of plurals (Peter)
- Add message translation support to the PL languages (Alvaro, Peter)
- Add more DTrace probes (Robert Lor)
- Enable DTrace support on Mac OS X Leopard and other non-Solaris platforms (Robert Lor)
- Simplify and standardize conversions between C strings and text datums, by providing common functions for the purpose (Brendan Jurd, Tom)
- Clean up the `include/catalog/` header files so that frontend programs can include them without including `postgres.h` (Zdenek Kotala)
- Make name char-aligned, and suppress zero-padding of name entries in indexes (Tom)
- Recover better if dynamically-loaded code executes `exit()` (Tom)
- Add a hook to let plug-ins monitor the executor (Itagaki Takahiro)
- Add a hook to allow the planner's statistics lookup behavior to be overridden (Simon Riggs)
- Add `shmem_startup_hook()` for custom shared memory requirements (Tom)
- Replace the index access method `amgetmulti` entry point with `amgetbitmap`, and extend the API for `amgettuple` to support run-time determination of operator lossiness (Heikki, Tom, Teodor)

  The API for GIN and GiST opclass `consistent` functions has been extended as well.
- Add support for partial-match searches in GIN indexes (Teodor Sigaev, Oleg Bartunov)
- Replace pg_class column *`reltriggers`* with boolean *`relhastriggers`* (Simon)

  Also remove unused pg_class columns *`relukeys`*, *`relfkeys`*, and *`relrefs`*.
- Add a *`relistemp`* column to pg_class to ease identification of temporary tables (Tom)
- Move platform FAQs into the main documentation (Peter)

- Prevent parser input files from being built with any conflicts (Peter)
- Add support for the `KOI8U` (Ukrainian) encoding (Peter)
- Add Japanese message translations (Japan PostgreSQL Users Group)

  This used to be maintained as a separate project.
- Fix problem when setting `LC_MESSAGES` on MSVC-built systems (Hiroshi Inoue, Hiroshi Saito, Magnus)

E.42.3.12. Contrib

- Add `contrib/auto_explain` to automatically run **EXPLAIN** on queries exceeding a specified duration (Itagaki Takahiro, Tom)
- Add `contrib/btree_gin` to allow GIN indexes to handle more datatypes (Oleg, Teodor)
- Add `contrib/citext` to provide a case-insensitive, multibyte-aware text data type (David Wheeler)
- Add `contrib/pg_stat_statements` for server-wide tracking of statement execution statistics (Itagaki Takahiro)
- Add duration and query mode options to `contrib/pgbench` (Itagaki Takahiro)
- Make `contrib/pgbench` use table names pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers, rather than just accounts, branches, history, and tellers (Tom)

  This is to reduce the risk of accidentally destroying real data by running pgbench.
- Fix `contrib/pgstattuple` to handle tables and indexes with over 2 billion pages (Tatsuhito Kasahara)
- In `contrib/fuzzystrmatch`, add a version of the Levenshtein string-distance function that allows the user to specify the costs of insertion, deletion, and substitution (Volkan Yazici)
- Make `contrib/ltree` support multibyte encodings (laser)
- Enable `contrib/dblink` to use connection information stored in the SQL/MED catalogs (Joe Conway)
- Improve `contrib/dblink`'s reporting of errors from the remote server (Joe Conway)
- Make `contrib/dblink` set `client_encoding` to match the local database's encoding (Joe Conway)

  This prevents encoding problems when communicating with a remote database that uses a different encoding.
- Make sure `contrib/dblink` uses a password supplied by the user, and not accidentally taken from the server's `.pgpass` file (Joe Conway)

  This is a minor security enhancement.
- Add `fsm_page_contents()` to `contrib/pageinspect` (Heikki)
- Modify `get_raw_page()` to support free space map (`*_fsm`) files. Also update `contrib/pg_freespacemap`.
- Add support for multibyte encodings to `contrib/pg_trgm` (Teodor)
- Rewrite `contrib/intagg` to use new functions `array_agg()` and `unnest()` (Tom)
- Make `contrib/pg_standby` recover all available WAL before failover (Fujii Masao, Simon, Heikki)

  To make this work safely, you now need to set the new `recovery_end_command` option in `recovery.conf` to clean up the trigger file after failover. pg_standby will no longer remove the trigger file itself.
- `contrib/pg_standby`'s `-l` option is now a no-op, because it is unsafe to use a symlink (Simon)

E.43. Release 8.3.23

Release Date

2013-02-07

This release contains a variety of fixes from 8.3.22. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

This is expected to be the last PostgreSQL™ release in the 8.3.X series. Users are encouraged to update to a newer release branch soon.

E.43.1. Migration to Version 8.3.23

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see Section E.49, « Release 8.3.17 ».

E.43.2. Changes

- Prevent execution of `enum_recv` from SQL (Tom Lane)

  The function was misdeclared, allowing a simple SQL command to crash the server. In principle an attacker might be able to use it to examine the contents of server memory. Our thanks to Sumit Soni (via Secunia SVCRP) for reporting this issue. (CVE-2013-0255)
- Fix SQL grammar to allow subscripting or field selection from a sub-SELECT result (Tom Lane)
- Protect against race conditions when scanning pg_tablespace (Stephen Frost, Tom Lane)

  CREATE DATABASE and **DROP DATABASE** could misbehave if there were concurrent updates of pg_tablespace entries.
- Prevent **DROP OWNED** from trying to drop whole databases or tablespaces (Álvaro Herrera)

  For safety, ownership of these objects must be reassigned, not dropped.
- Prevent misbehavior when a RowExpr or XmlExpr is parse-analyzed twice (Andres Freund, Tom Lane)

  This mistake could be user-visible in contexts such as `CREATE TABLE LIKE INCLUDING INDEXES`.
- Improve defenses against integer overflow in hashtable sizing calculations (Jeff Davis)
- Ensure that non-ASCII prompt strings are translated to the correct code page on Windows (Alexander Law, Noah Misch)

  This bug affected psql and some other client programs.
- Fix possible crash in psql's **\?** command when not connected to a database (Meng Qingzhong)
- Fix one-byte buffer overrun in libpq's `PQprintTuples` (Xi Wang)

  This ancient function is not used anywhere by PostgreSQL™ itself, but it might still be used by some client code.
- Rearrange configure's tests for supplied functions so it is not fooled by bogus exports from libedit/libreadline (Christoph Berg)
- Ensure Windows build number increases over time (Magnus Hagander)
- Make pgxs build executables with the right `.exe` suffix when cross-compiling for Windows (Zoltan Boszormenyi)
- Add new timezone abbreviation `FET` (Tom Lane)

  This is now used in some eastern-European time zones.

E.44. Release 8.3.22

Release Date

2012-12-06

This release contains a variety of fixes from 8.3.21. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

The PostgreSQL™ community will stop releasing updates for the 8.3.X release series in February 2013. Users are encouraged to update to a newer release branch soon.

E.44.1. Migration to Version 8.3.22

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see Section E.49, « Release 8.3.17 ».

E.44.2. Changes

- Fix multiple bugs associated with **CREATE INDEX CONCURRENTLY** (Andres Freund, Tom Lane)

  Fix **CREATE INDEX CONCURRENTLY** to use in-place updates when changing the state of an index's pg_index row. This prevents race conditions that could cause concurrent sessions to miss updating the target index, thus resulting in corrupt concurrently-created indexes.

  Also, fix various other operations to ensure that they ignore invalid indexes resulting from a failed **CREATE INDEX CONCURRENTLY** command. The most important of these is **VACUUM**, because an auto-vacuum could easily be launched on the table before corrective action can be taken to fix or remove the invalid index.
- Avoid corruption of internal hash tables when out of memory (Hitoshi Harada)
- Fix planning of non-strict equivalence clauses above outer joins (Tom Lane)

  The planner could derive incorrect constraints from a clause equating a non-strict construct to something else, for example `WHERE COALESCE(foo, 0) = 0` when `foo` is coming from the nullable side of an outer join.
- Improve planner's ability to prove exclusion constraints from equivalence classes (Tom Lane)
- Fix partial-row matching in hashed subplans to handle cross-type cases correctly (Tom Lane)

  This affects multicolumn `NOT IN` subplans, such as `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` when for instance `b` and `y` are int4 and int8 respectively. This mistake led to wrong answers or crashes depending on the specific datatypes involved.
- Acquire buffer lock when re-fetching the old tuple for an `AFTER ROW UPDATE/DELETE` trigger (Andres Freund)

  In very unusual circumstances, this oversight could result in passing incorrect data to the precheck logic for a foreign-key enforcement trigger. That could result in a crash, or in an incorrect decision about whether to fire the trigger.
- Fix **REASSIGN OWNED** to handle grants on tablespaces (Álvaro Herrera)
- Ignore incorrect pg_attribute entries for system columns for views (Tom Lane)

  Views do not have any system columns. However, we forgot to remove such entries when converting a table to a view. That's fixed properly for 9.3 and later, but in previous branches we need to defend against existing mis-converted views.
- Fix rule printing to dump `INSERT INTO` *`table`* `DEFAULT VALUES` correctly (Tom Lane)
- Guard against stack overflow when there are too many `UNION`/`INTERSECT`/`EXCEPT` clauses in a query (Tom Lane)
- Prevent platform-dependent failures when dividing the minimum possible integer value by -1 (Xi Wang, Tom Lane)
- Fix possible access past end of string in date parsing (Hitoshi Harada)
- Produce an understandable error message if the length of the path name for a Unix-domain socket exceeds the platform-specific limit (Tom Lane, Andrew Dunstan)

  Formerly, this would result in something quite unhelpful, such as « Non-recoverable failure in name resolution ».
- Fix memory leaks when sending composite column values to the client (Tom Lane)
- Make pg_ctl more robust about reading the `postmaster.pid` file (Heikki Linnakangas)

  Fix race conditions and possible file descriptor leakage.
- Fix possible crash in psql if incorrectly-encoded data is presented and the `client_encoding` setting is a client-only encoding, such as SJIS (Jiang Guiqing)
- Fix bugs in the `restore.sql` script emitted by pg_dump in `tar` output format (Tom Lane)

  The script would fail outright on tables whose names include upper-case characters. Also, make the script capable of restoring data in `--inserts` mode as well as the regular COPY mode.
- Fix pg_restore to accept POSIX-conformant `tar` files (Brian Weaver, Tom Lane)

  The original coding of pg_dump's `tar` output mode produced files that are not fully conformant with the POSIX standard. This has been corrected for version 9.3. This patch updates previous branches so that they will accept both the incorrect and the corrected formats, in hopes of avoiding compatibility problems when 9.3 comes out.
- Fix pg_resetxlog to locate `postmaster.pid` correctly when given a relative path to the data directory (Tom Lane)

  This mistake could lead to pg_resetxlog not noticing that there is an active postmaster using the data directory.

- Fix libpq's `lo_import()` and `lo_export()` functions to report file I/O errors properly (Tom Lane)
- Fix ecpg's processing of nested structure pointer variables (Muhammad Usama)
- Make `contrib/pageinspect`'s btree page inspection functions take buffer locks while examining pages (Tom Lane)
- Fix pgxs support for building loadable modules on AIX (Tom Lane)

  Building modules outside the original source tree didn't work on AIX.
- Update time zone data files to tzdata release 2012j for DST law changes in Cuba, Israel, Jordan, Libya, Palestine, Western Samoa, and portions of Brazil.

E.45. Release 8.3.21

Release Date

2012-09-24

This release contains a variety of fixes from 8.3.20. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

The PostgreSQL™ community will stop releasing updates for the 8.3.X release series in February 2013. Users are encouraged to update to a newer release branch soon.

E.45.1. Migration to Version 8.3.21

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see Section E.49, « Release 8.3.17 ».

E.45.2. Changes

- Improve page-splitting decisions in GiST indexes (Alexander Korotkov, Robert Haas, Tom Lane)

  Multi-column GiST indexes might suffer unexpected bloat due to this error.
- Fix cascading privilege revoke to stop if privileges are still held (Tom Lane)

  If we revoke a grant option from some role *X*, but *X* still holds that option via a grant from someone else, we should not recursively revoke the corresponding privilege from role(s) *Y* that *X* had granted it to.
- Fix handling of `SIGFPE` when PL/Perl is in use (Andres Freund)

  Perl resets the process's `SIGFPE` handler to `SIG_IGN`, which could result in crashes later on. Restore the normal Postgres signal handler after initializing PL/Perl.
- Prevent PL/Perl from crashing if a recursive PL/Perl function is redefined while being executed (Tom Lane)
- Work around possible misoptimization in PL/Perl (Tom Lane)

  Some Linux distributions contain an incorrect version of `pthread.h` that results in incorrect compiled code in PL/Perl, leading to crashes if a PL/Perl function calls another one that throws an error.
- Update time zone data files to tzdata release 2012f for DST law changes in Fiji

E.46. Release 8.3.20

Release Date

2012-08-17

This release contains a variety of fixes from 8.3.19. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

The PostgreSQL™ community will stop releasing updates for the 8.3.X release series in February 2013. Users are encouraged to update to a newer release branch soon.

E.46.1. Migration to Version 8.3.20

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see Section E.49, « Release 8.3.17 ».

E.46.2. Changes

- Prevent access to external files/URLs via XML entity references (Noah Misch, Tom Lane)

  `xml_parse()` would attempt to fetch external files or URLs as needed to resolve DTD and entity references in an XML value, thus allowing unprivileged database users to attempt to fetch data with the privileges of the database server. While the external data wouldn't get returned directly to the user, portions of it could be exposed in error messages if the data didn't parse as valid XML; and in any case the mere ability to check existence of a file might be useful to an attacker. (CVE-2012-3489)

- Prevent access to external files/URLs via `contrib/xml2`'s `xslt_process()` (Peter Eisentraut)

  libxslt offers the ability to read and write both files and URLs through stylesheet commands, thus allowing unprivileged database users to both read and write data with the privileges of the database server. Disable that through proper use of libxslt's security options. (CVE-2012-3488)

  Also, remove `xslt_process()`'s ability to fetch documents and stylesheets from external files/URLs. While this was a documented « feature », it was long regarded as a bad idea. The fix for CVE-2012-3489 broke that capability, and rather than expend effort on trying to fix it, we're just going to summarily remove it.

- Prevent too-early recycling of btree index pages (Noah Misch)

  When we allowed read-only transactions to skip assigning XIDs, we introduced the possibility that a deleted btree page could be recycled while a read-only transaction was still in flight to it. This would result in incorrect index search results. The probability of such an error occurring in the field seems very low because of the timing requirements, but nonetheless it should be fixed.

- Fix crash-safety bug with newly-created-or-reset sequences (Tom Lane)

  If **ALTER SEQUENCE** was executed on a freshly created or reset sequence, and then precisely one `nextval()` call was made on it, and then the server crashed, WAL replay would restore the sequence to a state in which it appeared that no `nextval()` had been done, thus allowing the first sequence value to be returned again by the next `nextval()` call. In particular this could manifest for serial columns, since creation of a serial column's sequence includes an **ALTER SEQUENCE OWNED BY** step.

- Ensure the `backup_label` file is fsync'd after `pg_start_backup()` (Dave Kerr)

- Back-patch 9.1 improvement to compress the fsync request queue (Robert Haas)

  This improves performance during checkpoints. The 9.1 change has now seen enough field testing to seem safe to back-patch.

- Only allow autovacuum to be auto-canceled by a directly blocked process (Tom Lane)

  The original coding could allow inconsistent behavior in some cases; in particular, an autovacuum could get canceled after less than `deadlock_timeout` grace period.

- Improve logging of autovacuum cancels (Robert Haas)

- Fix log collector so that `log_truncate_on_rotation` works during the very first log rotation after server start (Tom Lane)

- Ensure that a whole-row reference to a subquery doesn't include any extra `GROUP BY` or `ORDER BY` columns (Tom Lane)

- Disallow copying whole-row references in `CHECK` constraints and index definitions during **CREATE TABLE** (Tom Lane)

  This situation can arise in **CREATE TABLE** with `LIKE` or `INHERITS`. The copied whole-row variable was incorrectly labeled with the row type of the original table not the new one. Rejecting the case seems reasonable for `LIKE`, since the row types might well diverge later. For `INHERITS` we should ideally allow it, with an implicit coercion to the parent table's row type; but that will require more work than seems safe to back-patch.

- Fix memory leak in `ARRAY(SELECT ...)` subqueries (Heikki Linnakangas, Tom Lane)

- Fix extraction of common prefixes from regular expressions (Tom Lane)

  The code could get confused by quantified parenthesized subexpressions, such as `^(foo)?bar`. This would lead to incorrect index optimization of searches for such patterns.

- Report errors properly in `contrib/xml2`'s `xslt_process()` (Tom Lane)

- Update time zone data files to tzdata release 2012e for DST law changes in Morocco and Tokelau

E.47. Release 8.3.19

Release Date

2012-06-04

This release contains a variety of fixes from 8.3.18. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.47.1. Migration to Version 8.3.19

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see Section E.49, « Release 8.3.17 ».

E.47.2. Changes

- Fix incorrect password transformation in `contrib/pgcrypto`'s DES `crypt()` function (Solar Designer)

  If a password string contained the byte value `0x80`, the remainder of the password was ignored, causing the password to be much weaker than it appeared. With this fix, the rest of the string is properly included in the DES hash. Any stored password values that are affected by this bug will thus no longer match, so the stored values may need to be updated. (CVE-2012-2143)

- Ignore `SECURITY DEFINER` and `SET` attributes for a procedural language's call handler (Tom Lane)

  Applying such attributes to a call handler could crash the server. (CVE-2012-2655)

- Allow numeric timezone offsets in timestamp input to be up to 16 hours away from UTC (Tom Lane)

  Some historical time zones have offsets larger than 15 hours, the previous limit. This could result in dumped data values being rejected during reload.

- Fix timestamp conversion to cope when the given time is exactly the last DST transition time for the current timezone (Tom Lane)

  This oversight has been there a long time, but was not noticed previously because most DST-using zones are presumed to have an indefinite sequence of future DST transitions.

- Fix text to name and char to name casts to perform string truncation correctly in multibyte encodings (Karl Schnaitter)

- Fix memory copying bug in `to_tsquery()` (Heikki Linnakangas)

- Fix slow session startup when pg_attribute is very large (Tom Lane)

  If pg_attribute exceeds one-fourth of `shared_buffers`, cache rebuilding code that is sometimes needed during session start would trigger the synchronized-scan logic, causing it to take many times longer than normal. The problem was particularly acute if many new sessions were starting at once.

- Ensure sequential scans check for query cancel reasonably often (Merlin Moncure)

  A scan encountering many consecutive pages that contain no live tuples would not respond to interrupts meanwhile.

- Ensure the Windows implementation of `PGSemaphoreLock()` clears `ImmediateInterruptOK` before returning (Tom Lane)

  This oversight meant that a query-cancel interrupt received later in the same query could be accepted at an unsafe time, with unpredictable but not good consequences.

- Show whole-row variables safely when printing views or rules (Abbas Butt, Tom Lane)

  Corner cases involving ambiguous names (that is, the name could be either a table or column name of the query) were printed in an ambiguous way, risking that the view or rule would be interpreted differently after dump and reload. Avoid the ambiguous case by attaching a no-op cast.

- Ensure autovacuum worker processes perform stack depth checking properly (Heikki Linnakangas)

  Previously, infinite recursion in a function invoked by auto-**ANALYZE** could crash worker processes.

- Fix logging collector to not lose log coherency under high load (Andrew Dunstan)

  The collector previously could fail to reassemble large messages if it got too busy.
- Fix logging collector to ensure it will restart file rotation after receiving `SIGHUP` (Tom Lane)
- Fix PL/pgSQL's **GET DIAGNOSTICS** command when the target is the function's first variable (Tom Lane)
- Fix several performance problems in pg_dump when the database contains many objects (Jeff Janes, Tom Lane)

  pg_dump could get very slow if the database contained many schemas, or if many objects are in dependency loops, or if there are many owned sequences.
- Fix `contrib/dblink`'s `dblink_exec()` to not leak temporary database connections upon error (Tom Lane)
- Update time zone data files to tzdata release 2012c for DST law changes in Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria, and Tokelau Islands; also historical corrections for Canada.

E.48. Release 8.3.18

Release Date

2012-02-27

This release contains a variety of fixes from 8.3.17. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.48.1. Migration to Version 8.3.18

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see Section E.49, « Release 8.3.17 ».

E.48.2. Changes

- Require execute permission on the trigger function for **CREATE TRIGGER** (Robert Haas)

  This missing check could allow another user to execute a trigger function with forged input data, by installing it on a table he owns. This is only of significance for trigger functions marked `SECURITY DEFINER`, since otherwise trigger functions run as the table owner anyway. (CVE-2012-0866)
- Convert newlines to spaces in names written in pg_dump comments (Robert Haas)

  pg_dump was incautious about sanitizing object names that are emitted within SQL comments in its output script. A name containing a newline would at least render the script syntactically incorrect. Maliciously crafted object names could present a SQL injection risk when the script is reloaded. (CVE-2012-0868)
- Fix btree index corruption from insertions concurrent with vacuuming (Tom Lane)

  An index page split caused by an insertion could sometimes cause a concurrently-running **VACUUM** to miss removing index entries that it should remove. After the corresponding table rows are removed, the dangling index entries would cause errors (such as « could not read block N in file ... ») or worse, silently wrong query results after unrelated rows are re-inserted at the now-free table locations. This bug has been present since release 8.2, but occurs so infrequently that it was not diagnosed until now. If you have reason to suspect that it has happened in your database, reindexing the affected index will fix things.
- Allow non-existent values for some settings in **ALTER USER/DATABASE SET** (Heikki Linnakangas)

  Allow `default_text_search_config`, `default_tablespace`, and `temp_tablespaces` to be set to names that are not known. This is because they might be known in another database where the setting is intended to be used, or for the tablespace cases because the tablespace might not be created yet. The same issue was previously recognized for `search_path`, and these settings now act like that one.
- Track the OID counter correctly during WAL replay, even when it wraps around (Tom Lane)

  Previously the OID counter would remain stuck at a high value until the system exited replay mode. The practical consequences of that are usually nil, but there are scenarios wherein a standby server that's been promoted to master might take a long time to advance the OID counter to a reasonable value once values are needed.
- Fix regular expression back-references with `*` attached (Tom Lane)

Rather than enforcing an exact string match, the code would effectively accept any string that satisfies the pattern sub-expression referenced by the back-reference symbol.

A similar problem still afflicts back-references that are embedded in a larger quantified expression, rather than being the immediate subject of the quantifier. This will be addressed in a future PostgreSQL™ release.

- Fix recently-introduced memory leak in processing of inet/cidr values (Heikki Linnakangas)

  A patch in the December 2011 releases of PostgreSQL™ caused memory leakage in these operations, which could be significant in scenarios such as building a btree index on such a column.

- Avoid double close of file handle in syslogger on Windows (MauMau)

  Ordinarily this error was invisible, but it would cause an exception when running on a debug version of Windows.

- Fix I/O-conversion-related memory leaks in plpgsql (Andres Freund, Jan Urbanski, Tom Lane)

  Certain operations would leak memory until the end of the current function.

- Improve pg_dump's handling of alled table columns (Tom Lane)

  pg_dump mishandled situations where a child column has a different default expression than its parent column. If the default is textually identical to the parent's default, but not actually the same (for instance, because of schema search path differences) it would not be recognized as different, so that after dump and restore the child would be allowed to all the parent's default. Child columns that are `NOT NULL` where their parent is not could also be restored subtly incorrectly.

- Fix pg_restore's direct-to-database mode for INSERT-style table data (Tom Lane)

  Direct-to-database restores from archive files made with `--inserts` or `--column-inserts` options fail when using pg_restore from a release dated September or December 2011, as a result of an oversight in a fix for another problem. The archive file itself is not at fault, and text-mode output is okay.

- Fix error in `contrib/intarray`'s `int[] & int[]` operator (Guillaume Lelarge)

  If the smallest integer the two input arrays have in common is 1, and there are smaller values in either array, then 1 would be incorrectly omitted from the result.

- Fix error detection in `contrib/pgcrypto`'s `encrypt_iv()` and `decrypt_iv()` (Marko Kreen)

  These functions failed to report certain types of invalid-input errors, and would instead return random garbage values for incorrect input.

- Fix one-byte buffer overrun in `contrib/test_parser` (Paul Guyot)

  The code would try to read one more byte than it should, which would crash in corner cases. Since `contrib/test_parser` is only example code, this is not a security issue in itself, but bad example code is still bad.

- Use `__sync_lock_test_and_set()` for spinlocks on ARM, if available (Martin Pitt)

  This function replaces our previous use of the `SWPB` instruction, which is deprecated and not available on ARMv6 and later. Reports suggest that the old code doesn't fail in an obvious way on recent ARM boards, but simply doesn't interlock concurrent accesses, leading to bizarre failures in multiprocess operation.

- Use `-fexcess-precision=standard` option when building with gcc versions that accept it (Andrew Dunstan)

  This prevents assorted scenarios wherein recent versions of gcc will produce creative results.

- Allow use of threaded Python on FreeBSD (Chris Rees)

  Our configure script previously believed that this combination wouldn't work; but FreeBSD fixed the problem, so remove that error check.

E.49. Release 8.3.17

Release Date

2011-12-05

This release contains a variety of fixes from 8.3.16. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.49.1. Migration to Version 8.3.17

A dump/restore is not required for those running 8.3.X.

However, a longstanding error was discovered in the definition of the `information_schema.referential_constraints` view. If you rely on correct results from that view, you should replace its definition as explained in the first changelog item below.

Also, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.49.2. Changes

- Fix bugs in `information_schema.referential_constraints` view (Tom Lane)

  This view was being insufficiently careful about matching the foreign-key constraint to the depended-on primary or unique key constraint. That could result in failure to show a foreign key constraint at all, or showing it multiple times, or claiming that it depends on a different constraint than the one it really does.

  Since the view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can (as a superuser) drop the `information_schema` schema then re-create it by sourcing *`SHAREDIR/`*`information_schema.sql`. (Run `pg_config --sharedir` if you're uncertain where *`SHAREDIR`* is.) This must be repeated in each database to be fixed.

- Fix TOAST-related data corruption during `CREATE TABLE dest AS SELECT * FROM src` or `INSERT INTO dest SELECT * FROM src` (Tom Lane)

  If a table has been modified by **ALTER TABLE ADD COLUMN**, attempts to copy its data verbatim to another table could produce corrupt results in certain corner cases. The problem can only manifest in this precise form in 8.4 and later, but we patched earlier versions as well in case there are other code paths that could trigger the same bug.

- Fix race condition during toast table access from stale syscache entries (Tom Lane)

  The typical symptom was transient errors like « missing chunk number 0 for toast value NNNNN in pg_toast_2619 », where the cited toast table would always belong to a system catalog.

- Make `DatumGetInetP()` unpack inet datums that have a 1-byte header, and add a new macro, `DatumGetInetPP()`, that does not (Heikki Linnakangas)

  This change affects no core code, but might prevent crashes in add-on code that expects `DatumGetInetP()` to produce an unpacked datum as per usual convention.

- Improve locale support in money type's input and output (Tom Lane)

  Aside from not supporting all standard `lc_monetary` formatting options, the input and output functions were inconsistent, meaning there were locales in which dumped money values could not be re-read.

- Don't let `transform_null_equals` affect `CASE foo WHEN NULL ...` constructs (Heikki Linnakangas)

  `transform_null_equals` is only supposed to affect `foo = NULL` expressions written directly by the user, not equality checks generated internally by this form of `CASE`.

- Change foreign-key trigger creation order to better support self-referential foreign keys (Tom Lane)

  For a cascading foreign key that references its own table, a row update will fire both the `ON UPDATE` trigger and the `CHECK` trigger as one event. The `ON UPDATE` trigger must execute first, else the `CHECK` will check a non-final state of the row and possibly throw an inappropriate error. However, the firing order of these triggers is determined by their names, which generally sort in creation order since the triggers have auto-generated names following the convention « RI_ConstraintTrigger_NNNN ». A proper fix would require modifying that convention, which we will do in 9.2, but it seems risky to change it in existing releases. So this patch just changes the creation order of the triggers. Users encountering this type of error should drop and re-create the foreign key constraint to get its triggers into the right order.

- Avoid floating-point underflow while tracking buffer allocation rate (Greg Matthews)

  While harmless in itself, on certain platforms this would result in annoying kernel log messages.

- Preserve blank lines within commands in psql's command history (Robert Haas)

  The former behavior could cause problems if an empty line was removed from within a string literal, for example.

- Fix pg_dump to dump user-defined casts between auto-generated types, such as table rowtypes (Tom Lane)

- Use the preferred version of xsubpp to build PL/Perl, not necessarily the operating system's main copy (David Wheeler and

Alex Hunsaker)

- Fix incorrect coding in `contrib/dict_int` and `contrib/dict_xsyn` (Tom Lane)

  Some functions incorrectly assumed that memory returned by `palloc()` is guaranteed zeroed.
- Honor query cancel interrupts promptly in `pgstatindex()` (Robert Haas)
- Ensure VPATH builds properly install all server header files (Peter Eisentraut)
- Shorten file names reported in verbose error messages (Peter Eisentraut)

  Regular builds have always reported just the name of the C file containing the error message call, but VPATH builds formerly reported an absolute path name.
- Fix interpretation of Windows timezone names for Central America (Tom Lane)

  Map « Central America Standard Time » to `CST6`, not `CST6CDT`, because DST is generally not observed anywhere in Central America.
- Update time zone data files to tzdata release 2011n for DST law changes in Brazil, Cuba, Fiji, Palestine, Russia, and Samoa; also historical corrections for Alaska and British East Africa.

E.50. Release 8.3.16

Release Date

2011-09-26

This release contains a variety of fixes from 8.3.15. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.50.1. Migration to Version 8.3.16

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.50.2. Changes

- Fix bugs in indexing of in-doubt HOT-updated tuples (Tom Lane)

  These bugs could result in index corruption after reindexing a system catalog. They are not believed to affect user indexes.
- Fix multiple bugs in GiST index page split processing (Heikki Linnakangas)

  The probability of occurrence was low, but these could lead to index corruption.
- Fix possible buffer overrun in `tsvector_concat()` (Tom Lane)

  The function could underestimate the amount of memory needed for its result, leading to server crashes.
- Fix crash in `xml_recv` when processing a « standalone » parameter (Tom Lane)
- Avoid possibly accessing off the end of memory in **ANALYZE** and in SJIS-2004 encoding conversion (Noah Misch)

  This fixes some very-low-probability server crash scenarios.
- Fix race condition in relcache init file invalidation (Tom Lane)

  There was a window wherein a new backend process could read a stale init file but miss the inval messages that would tell it the data is stale. The result would be bizarre failures in catalog accesses, typically « could not read block 0 in file ... » later during startup.
- Fix memory leak at end of a GiST index scan (Tom Lane)

  Commands that perform many separate GiST index scans, such as verification of a new GiST-based exclusion constraint on a table already containing many rows, could transiently require large amounts of memory due to this leak.
- Fix performance problem when constructing a large, lossy bitmap (Tom Lane)
- Fix array- and path-creating functions to ensure padding bytes are zeroes (Tom Lane)

This avoids some situations where the planner will think that semantically-equal constants are not equal, resulting in poor optimization.

- Work around gcc 4.6.0 bug that breaks WAL replay (Tom Lane)

  This could lead to loss of committed transactions after a server crash.

- Fix dump bug for `VALUES` in a view (Tom Lane)
- Disallow `SELECT FOR UPDATE/SHARE` on sequences (Tom Lane)

  This operation doesn't work as expected and can lead to failures.

- Defend against integer overflow when computing size of a hash table (Tom Lane)
- Fix cases where **CLUSTER** might attempt to access already-removed TOAST data (Tom Lane)
- Fix portability bugs in use of credentials control messages for « peer » authentication (Tom Lane)
- Fix SSPI login when multiple roundtrips are required (Ahmed Shinwari, Magnus Hagander)

  The typical symptom of this problem was « The function requested is not supported » errors during SSPI login.

- Fix typo in `pg_srand48` seed initialization (Andres Freund)

  This led to failure to use all bits of the provided seed. This function is not used on most platforms (only those without `srandom`), and the potential security exposure from a less-random-than-expected seed seems minimal in any case.

- Avoid integer overflow when the sum of `LIMIT` and `OFFSET` values exceeds 2^63 (Heikki Linnakangas)
- Add overflow checks to int4 and int8 versions of `generate_series()` (Robert Haas)
- Fix trailing-zero removal in `to_char()` (Marti Raudsepp)

  In a format with `FM` and no digit positions after the decimal point, zeroes to the left of the decimal point could be removed incorrectly.

- Fix `pg_size_pretty()` to avoid overflow for inputs close to 2^63 (Tom Lane)
- In pg_ctl, support silent mode for service registrations on Windows (MauMau)
- Fix psql's counting of script file line numbers during `COPY` from a different file (Tom Lane)
- Fix pg_restore's direct-to-database mode for `standard_conforming_strings` (Tom Lane)

  pg_restore could emit incorrect commands when restoring directly to a database server from an archive file that had been made with `standard_conforming_strings` set to `on`.

- Fix write-past-buffer-end and memory leak in libpq's LDAP service lookup code (Albe Laurenz)
- In libpq, avoid failures when using nonblocking I/O and an SSL connection (Martin Pihlak, Tom Lane)
- Improve libpq's handling of failures during connection startup (Tom Lane)

  In particular, the response to a server report of `fork()` failure during SSL connection startup is now saner.

- Improve libpq's error reporting for SSL failures (Tom Lane)
- Make ecpglib write double values with 15 digits precision (Akira Kurosawa)
- In ecpglib, be sure `LC_NUMERIC` setting is restored after an error (Michael Meskes)
- Apply upstream fix for blowfish signed-character bug (CVE-2011-2483) (Tom Lane)

  `contrib/pg_crypto`'s blowfish encryption code could give wrong results on platforms where char is signed (which is most), leading to encrypted passwords being weaker than they should be.

- Fix memory leak in `contrib/seg` (Heikki Linnakangas)
- Fix `pgstatindex()` to give consistent results for empty indexes (Tom Lane)
- Allow building with perl 5.14 (Alex Hunsaker)
- Update configure script's method for probing existence of system functions (Tom Lane)

  The version of autoconf we used in 8.3 and 8.2 could be fooled by compilers that perform link-time optimization.

- Fix assorted issues with build and install file paths containing spaces (Tom Lane)

- Update time zone data files to tzdata release 2011i for DST law changes in Canada, Egypt, Russia, Samoa, and South Sudan.

E.51. Release 8.3.15

Release Date

2011-04-18

This release contains a variety of fixes from 8.3.14. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.51.1. Migration to Version 8.3.15

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.51.2. Changes

- Disallow including a composite type in itself (Tom Lane)

  This prevents scenarios wherein the server could recurse infinitely while processing the composite type. While there are some possible uses for such a structure, they don't seem compelling enough to justify the effort required to make sure it always works safely.

- Avoid potential deadlock during catalog cache initialization (Nikhil Sontakke)

  In some cases the cache loading code would acquire share lock on a system index before locking the index's catalog. This could deadlock against processes trying to acquire exclusive locks in the other, more standard order.

- Fix dangling-pointer problem in `BEFORE ROW UPDATE` trigger handling when there was a concurrent update to the target tuple (Tom Lane)

  This bug has been observed to result in intermittent « cannot extract system attribute from virtual tuple » failures while trying to do `UPDATE RETURNING ctid`. There is a very small probability of more serious errors, such as generating incorrect index entries for the updated tuple.

- Disallow **DROP TABLE** when there are pending deferred trigger events for the table (Tom Lane)

  Formerly the **DROP** would go through, leading to « could not open relation with OID nnn » errors when the triggers were eventually fired.

- Fix PL/Python memory leak involving array slices (Daniel Popowich)
- Fix pg_restore to cope with long lines (over 1KB) in TOC files (Tom Lane)
- Put in more safeguards against crashing due to division-by-zero with overly enthusiastic compiler optimization (Aurelien Jarno)
- Support use of dlopen() in FreeBSD and OpenBSD on MIPS (Tom Lane)

  There was a hard-wired assumption that this system function was not available on MIPS hardware on these systems. Use a compile-time test instead, since more recent versions have it.

- Fix compilation failures on HP-UX (Heikki Linnakangas)
- Fix version-incompatibility problem with libintl on Windows (Hiroshi Inoue)
- Fix usage of xcopy in Windows build scripts to work correctly under Windows 7 (Andrew Dunstan)

  This affects the build scripts only, not installation or usage.

- Fix path separator used by pg_regress on Cygwin (Andrew Dunstan)
- Update time zone data files to tzdata release 2011f for DST law changes in Chile, Cuba, Falkland Islands, Morocco, Samoa, and Turkey; also historical corrections for South Australia, Alaska, and Hawaii.

E.52. Release 8.3.14

Release Date

2011-01-31

This release contains a variety of fixes from 8.3.13. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.52.1. Migration to Version 8.3.14

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.52.2. Changes

- Avoid failures when **EXPLAIN** tries to display a simple-form `CASE` expression (Tom Lane)

  If the `CASE`'s test expression was a constant, the planner could simplify the `CASE` into a form that confused the expression-display code, resulting in « unexpected CASE WHEN clause » errors.

- Fix assignment to an array slice that is before the existing range of subscripts (Tom Lane)

  If there was a gap between the newly added subscripts and the first pre-existing subscript, the code miscalculated how many entries needed to be copied from the old array's null bitmap, potentially leading to data corruption or crash.

- Avoid unexpected conversion overflow in planner for very distant date values (Tom Lane)

  The date type supports a wider range of dates than can be represented by the timestamp types, but the planner assumed it could always convert a date to timestamp with impunity.

- Fix pg_restore's text output for large objects (BLOBs) when `standard_conforming_strings` is on (Tom Lane)

  Although restoring directly to a database worked correctly, string escaping was incorrect if pg_restore was asked for SQL text output and `standard_conforming_strings` had been enabled in the source database.

- Fix erroneous parsing of tsquery values containing `... & !(subexpression) | ...` (Tom Lane)

  Queries containing this combination of operators were not executed correctly. The same error existed in `contrib/intarray`'s query_int type and `contrib/ltree`'s ltxtquery type.

- Fix buffer overrun in `contrib/intarray`'s input function for the query_int type (Apple)

  This bug is a security risk since the function's return address could be overwritten. Thanks to Apple Inc's security team for reporting this issue and supplying the fix. (CVE-2010-4015)

- Fix bug in `contrib/seg`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a seg column. If you have such an index, consider **REINDEX**ing it after installing this update. (This is identical to the bug that was fixed in `contrib/cube` in the previous update.)

E.53. Release 8.3.13

Release Date

2010-12-16

This release contains a variety of fixes from 8.3.12. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.53.1. Migration to Version 8.3.13

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.53.2. Changes

- Force the default `wal_sync_method` to be `fdatasync` on Linux (Tom Lane, Marti Raudsepp)

  The default on Linux has actually been `fdatasync` for many years, but recent kernel changes caused PostgreSQL™ to choose `open_datasync` instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably `ext4` with the `data=journal` mount option.

- Fix assorted bugs in WAL replay logic for GIN indexes (Tom Lane)

  This could result in « bad buffer id: 0 » failures or corruption of index contents during replication.

- Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)

- Fix persistent slowdown of autovacuum workers when multiple workers remain active for a long time (Tom Lane)

  The effective `vacuum_cost_limit` for an autovacuum worker could drop to nearly zero if it processed enough tables, causing it to run extremely slowly.

- Add support for detecting register-stack overrun on `IA64` (Tom Lane)

  The `IA64` architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.

- Add a check for stack overflow in `copyObject()` (Tom Lane)

  Certain code paths could crash due to stack overflow given a sufficiently complex query.

- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)

  It is possible to have a « concurrent » page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.

- Avoid memory leakage while **ANALYZE**'ing complex index expressions (Tom Lane)

- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)

  An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.

- Do not « inline » a SQL function with multiple `OUT` parameters (Tom Lane)

  This avoids a possible crash due to loss of information about the expected result rowtype.

- Behave correctly if `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `WITH` is attached to the `VALUES` part of `INSERT ... VALUES` (Tom Lane)

- Fix constant-folding of `COALESCE()` expressions (Tom Lane)

  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.

- Fix postmaster crash when connection acceptance (`accept()` or one of the calls made immediately after it) fails, and the postmaster was compiled with GSSAPI support (Alexander Chernikov)

- Fix missed unlink of temporary files when `log_temp_files` is active (Tom Lane)

  If an error occurred while attempting to emit the log message, the unlink was not done, resulting in accumulation of temp files.

- Add print functionality for InhRelation nodes (Tom Lane)

  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.

- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)

  This bug affected several different geometric distance-measurement operators.

- Fix PL/pgSQL's handling of « simple » expressions to not fail in recursion or error-recovery cases (Tom Lane)

- Fix PL/Python's handling of set-returning functions (Jan Urbanski)

  Attempts to call SPI functions within the iterator generating a set result would fail.

- Fix bug in `contrib/cube`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a cube column. If you have such an index, consider **REINDEX**ing it after installing this update.

- Don't emit « identifier will be truncated » notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)
- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)
- Fix memory leak in `contrib/xml2`'s XPath query functions (Tom Lane)
- Update time zone data files to tzdata release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.

E.54. Release 8.3.12

Release Date

2010-10-04

This release contains a variety of fixes from 8.3.11. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.54.1. Migration to Version 8.3.12

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.54.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).
- Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it's intended to be used with (Heikki Linnakangas, Tom Lane)
- Treat exit code 128 (`ERROR_WAIT_NO_CHILDREN`) as non-fatal on Windows (Magnus Hagander)

  Under high load, Windows processes will sometimes fail at startup with this error code. Formerly the postmaster treated this as a panic condition and restarted the whole database, but that seems to be an overreaction.
- Fix incorrect usage of non-strict OR joinclauses in Append indexscans (Tom Lane)

  This is a back-patch of an 8.4 fix that was missed in the 8.3 branch. This corrects an error introduced in 8.3.8 that could cause incorrect results for outer joins when the inner relation is an allance tree or `UNION ALL` subquery.
- Fix possible duplicate scans of `UNION ALL` member relations (Tom Lane)
- Fix « cannot handle unplanned sub-select » error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.
- Fix failure to mark cached plans as transient (Tom Lane)

  If a plan is prepared while **CREATE INDEX CONCURRENTLY** is in progress for one of the referenced tables, it is supposed to be re-planned once the index is ready for use. This was not happening reliably.
- Reduce PANIC to ERROR in some occasionally-reported btree failure cases, and provide additional detail in the resulting error messages (Tom Lane)

This should improve the system's robustness with corrupted indexes.

- Prevent show_session_authorization() from crashing within autovacuum processes (Tom Lane)
- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)
- Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)
- Improve merge join's handling of NULLs in the join columns (Tom Lane)

  A merge join can now stop entirely upon reaching the first NULL, if the sort order is such that NULLs sort high.
- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.
- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.
- Avoid holding open old WAL segments in the walwriter process (Magnus Hagander, Heikki Linnakangas)

  The previous coding would prevent removal of no-longer-needed segments.
- Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)
- Fix possible data corruption in **ALTER TABLE ... SET TABLESPACE** when archiving is enabled (Jeff Davis)
- Allow **CREATE DATABASE** and **ALTER DATABASE ... SET TABLESPACE** to be interrupted by query-cancel (Guillaume Lelarge)
- Fix **REASSIGN OWNED** to handle operator classes and families (Asko Tiidumaa)
- Fix possible core dump when comparing two empty tsquery values (Tom Lane)
- Fix `LIKE`'s handling of patterns containing `%` followed by `_` (Tom Lane)

  We've fixed this before, but there were still some incorrectly-handled cases.
- In PL/Python, defend against null pointer results from `PyCObject_AsVoidPtr` and `PyCObject_FromVoidPtr` (Peter Eisentraut)
- Make psql recognize **DISCARD ALL** as a command that should not be encased in a transaction block in autocommit-off mode (Itagaki Takahiro)
- Fix ecpg to process data from `RETURNING` clauses correctly (Michael Meskes)
- Improve `contrib/dblink`'s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after « duplicate connection name » errors in `contrib/dblink` (Itagaki Takahiro)
- Fix `contrib/dblink` to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
- Add `hstore(text, text)` function to `contrib/hstore` (Robert Haas)

  This function is the recommended substitute for the now-deprecated `=>` operator. It was back-patched so that future-proofed code can be used with older server versions. Note that the patch will be effective only after `contrib/hstore` is installed or reinstalled in a particular database. Users might prefer to execute the **CREATE FUNCTION** command by hand, instead.
- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)
- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.
- Make Windows' « N. Central Asia Standard Time » timezone map to Asia/Novosibirsk, not Asia/Almaty (Magnus Hagander)

  Microsoft changed the DST behavior of this zone in the timezone update from KB976098. Asia/Novosibirsk is a better match to its new behavior.

E.55. Release 8.3.11

Release Date

2010-05-17

This release contains a variety of fixes from 8.3.10. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.55.1. Migration to Version 8.3.11

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.55.2. Changes

- Enforce restrictions in `plperl` using an opmask applied to the whole interpreter, instead of using `Safe.pm` (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that `Safe.pm` is too insecure to rely on for making `plperl` trustable. This change removes use of `Safe.pm` altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's `strict` pragma in a natural way in `plperl`, and that Perl's `$a` and `$b` variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted « normal » Tcl interpreter unless we are really going to execute a `pltclu` function. (CVE-2010-1170)

- Fix possible crash if a cache reset message is received during rebuild of a relcache entry (Heikki)

  This error was introduced in 8.3.10 while fixing a related failure.

- Apply per-function GUC settings while running the language validator for the function (Itagaki Takahiro)

  This avoids failures if the function's code is invalid without the setting; an example is that SQL functions may not parse if the `search_path` is not correct.

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran `ALTER USER ... RESET ALL` for himself, or `ALTER DATABASE ... RESET ALL` for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the **ALTER** will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a `CONTEXT` addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Ensure the archiver process responds to changes in `archive_command` as soon as possible (Tom)
- Update pl/perl's `ppport.h` for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)
- Fix psql's `\copy` to not add spaces around a dot within `\copy (select ...)` (Tom)

  Addition of spaces around the decimal point in a numeric literal would result in a syntax error.

- Fix unnecessary « GIN indexes do not support whole-index scans » errors for unsatisfiable queries using `contrib/intarray` operators (Tom)
- Ensure that `contrib/pgstattuple` functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that `shmget()` returns `EINVAL` for an existing shared memory segment

(Tom)

This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

- Avoid possible crashes in syslogger process on Windows (Heikki)
- Deal more robustly with incomplete time zone information in the Windows registry (Magnus)
- Update the set of known Windows time zone names (Magnus)
- Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

  Also, add `PKST` (Pakistan Summer Time) to the default set of timezone abbreviations.

E.56. Release 8.3.10

Release Date

2010-03-15

This release contains a variety of fixes from 8.3.9. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.56.1. Migration to Version 8.3.10

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.56.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for an SSL connection (Magnus)

  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.
- Fix possible deadlock during backend startup (Tom)
- Fix possible crashes due to not handling errors during relcache reload cleanly (Tom)
- Fix possible crash due to use of dangling pointer to a cached plan (Tatsuo)
- Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)
- Fix server memory leak associated with use of savepoints and a client encoding different from server's encoding (Tom)
- Fix incorrect WAL data emitted during end-of-recovery cleanup of a GIST index page split (Yoichi Hirai)

  This would result in index corruption, or even more likely an error during WAL replay, if we were unlucky enough to crash during end-of-recovery cleanup after having completed an incomplete GIST insertion.
- Make `substring()` for bit types treat any negative length as meaning « all the rest of the string » (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix assorted crashes in xml processing caused by sloppy memory management (Tom)

  This is a back-patch of changes first applied in 8.4. The 8.3 code was known buggy, but the new code was sufficiently different to not want to back-patch it until it had gotten some field testing.
- Fix bug with trying to update a field of an element of a composite-type array column (Tom)
- Fix the `STOP WAL LOCATION` entry in backup history files to report the next WAL segment's name when the end location

is exactly at a segment boundary (Itagaki Takahiro)

- Fix some more cases of temporary-file leakage (Heikki)

  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function's exception handler.

- Improve constraint exclusion processing of boolean-variable cases, in particular make it possible to exclude a partition that has a « bool_column = false » constraint (Tom)
- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.

- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)
- Fix possible infinite loop if `SSL_read` or `SSL_write` fails without setting `errno` (Tom)

  This is reportedly possible with some Windows versions of openssl.

- Disallow GSSAPI authentication on local connections, since it requires a hostname to function correctly (Magnus)
- Make ecpg report the proper SQLSTATE if the connection disappears (Michael)
- Fix psql's `numericlocale` option to not format strings it shouldn't in latex and troff output formats (Heikki)
- Make psql return the correct exit status (3) when `ON_ERROR_STOP` and `--single-transaction` are both specified and an error occurs during the implied **COMMIT** (Bruce)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Fix possible failure when calling PL/Perl functions from PL/PerlU or vice versa (Tim Bunce)
- Add `volatile` markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.

- Prevent crash in `contrib/dblink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)
- Allow zero-dimensional arrays in `contrib/ltree` operations (Tom)

  This case was formerly rejected as an error, but it's more convenient to treat it the same as a zero-element array. In particular this avoids unnecessary failures when an ltree operation is applied to the result of `ARRAY(SELECT ...)` and the sub-select returns no rows.

- Fix assorted crashes in `contrib/xml2` caused by sloppy memory management (Tom)
- Make building of `contrib/xml2` more robust on Windows (Andrew)
- Fix race condition in Windows signal handling (Radu Ilie)

  One known symptom of this bug is that rows in pg_listener could be dropped under heavy load.

- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.57. Release 8.3.9

Release Date

2009-12-14

This release contains a variety of fixes from 8.3.8. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.57.1. Migration to Version 8.3.9

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see Section E.58, « Release 8.3.8 ».

E.57.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

  This change prevents allegedly-immutable index functions from possibly subverting a superuser's session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).
- Fix possible crash during backend-startup-time cache initialization (Tom)
- Avoid crash on empty thesaurus dictionary (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)

  This fix prevents a PANIC if a `VACUUM FULL` is canceled after it's already committed its tuple movements, as well as transient errors if a plain `VACUUM` is interrupted after having truncated the table.
- Fix possible crash due to integer overflow in hash table size calculation (Tom)

  This could occur with extremely large planner estimates for the size of a hashjoin's result.
- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix memory leak in syslogger process when rotating to a new CSV logfile (Tom)
- Fix Windows permission-downgrade logic (Jesse Morris)

  This fixes some cases where the database failed to start on Windows, often with misleading error messages such as « could not locate matching postgres executable ».
- Fix incorrect logic for GiST index page splits, when the split depends on a non-first column of the index (Paul Ramsey)
- Don't error out if recycling or removing an old WAL file fails at the end of checkpoint (Heikki)

  It's better to treat the problem as non-fatal and allow the checkpoint to complete. Future checkpoints will retry the removal. Such problems are not expected in normal operation, but have been seen to be caused by misdesigned Windows anti-virus and backup software.
- Ensure WAL files aren't repeatedly archived on Windows (Heikki)

  This is another symptom that could happen if some other process interfered with deletion of a no-longer-needed file.
- Fix PAM password processing to be more robust (Tom)

  The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Raise the maximum authentication token (Kerberos ticket) size in GSSAPI and SSPI authentication methods (Ian Turner)

  While the old 2000-byte limit was more than enough for Unix Kerberos implementations, tickets issued by Windows Domain Controllers can be much larger.
- Re-enable collection of access statistics for sequences (Akira Kurosawa)

  This used to work but was broken in 8.3.
- Fix processing of ownership dependencies during `CREATE OR REPLACE FUNCTION` (Tom)
- Fix incorrect handling of `WHERE` *x*=*x* conditions (Tom)

  In some cases these could get ignored as redundant, but they aren't -- they're equivalent to *x* `IS NOT NULL`.
- Make text search parser accept underscores in XML attributes (Peter)
- Fix encoding handling in xml binary input (Heikki)

If the XML header doesn't specify an encoding, we now assume UTF-8 by default; the previous handling was inconsistent.

- Fix bug with calling `plperl` from `plperlu` or vice versa (Tom)

  An error exit from the inner function could result in crashes due to failure to re-select the correct Perl interpreter for the outer function.

- Fix session-lifespan memory leak when a PL/Perl function is redefined (Tom)
- Ensure that Perl arrays are properly converted to PostgreSQL™ arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)

  This worked correctly already for non-set-returning functions.

- Fix rare crash in exception processing in PL/Python (Peter)
- In `contrib/pg_standby`, disable triggering failover with a signal on Windows (Fujii Masao)

  This never did anything useful, because Windows doesn't have Unix-style signals, but recent changes made it actually crash.

- Ensure psql's flex module is compiled with the correct system header definitions (Tom)

  This fixes build failures on platforms where `--enable-largefile` causes incompatible changes in the generated code.

- Make the postmaster ignore any `application_name` parameter in connection request packets, to improve compatibility with future libpq versions (Tom)
- Update the timezone abbreviation files to match current reality (Joachim Wieland)

  This includes adding `IDT` and `SGT` to the default timezone abbreviation set.

- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.

E.58. Release 8.3.8

Release Date

2009-09-09

This release contains a variety of fixes from 8.3.7. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.58.1. Migration to Version 8.3.8

A dump/restore is not required for those running 8.3.X. However, if you have any hash indexes on interval columns, you must **REINDEX** them after updating to 8.3.8. Also, if you are upgrading from a version earlier than 8.3.5, see Section E.61, « Release 8.3.5 ».

E.58.2. Changes

- Fix Windows shared-memory allocation code (Tsutomu Yamada, Magnus)

  This bug led to the often-reported « could not reattach to shared memory » error message.

- Force WAL segment switch during `pg_start_backup()` (Heikki)

  This avoids corner cases that could render a base backup unusable.

- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)

- Make **LOAD** of an already-loaded loadable module into a no-op (Tom)

  Formerly, **LOAD** would attempt to unload and re-load the module, but this is unsafe and not all that useful.

- Disallow empty passwords during LDAP authentication (Magnus)

- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
- Fix bugs associated with fetching a whole-row value from the output of a Sort or Materialize plan node (Tom)
- Prevent `synchronize_seqscans` from changing the results of scrollable and `WITH HOLD` cursors (Tom)
- Revert planner change that disabled partial-index and constraint exclusion optimizations when there were more than 100 clauses in an AND or OR list (Tom)
- Fix hash calculation for data type interval (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must **REINDEX** them after updating.
- Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with `'HH'/'HH12'` (Heikki)

  It was previously handled as `'th'` (lowercase).
- Fix overflow for `INTERVAL 'x ms'` when *x* is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.
- Fix money data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)
- Fix `LIKE` for case where pattern contains `%_` (Tom)
- Properly round datetime input like `00:12:57.999999999999999999999999999` (Tom)
- Fix memory leaks in XML operations (Tom)
- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)
- Ensure that a « fast shutdown » request will forcibly terminate open sessions, even if a « smart shutdown » was already in progress (Fujii Masao)
- Avoid performance degradation in bulk inserts into GIN indexes when the input values are (nearly) in sorted order (Tom)
- Correctly enforce NOT NULL domain constraints in some contexts in PL/pgSQL (Tom)
- Fix portability issues in plperl initialization (Andrew Dunstan)
- Fix pg_ctl to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)
- Improve pg_dump's efficiency when there are many large objects (Tamas Vincze)
- Use `SIGUSR1`, not `SIGQUIT`, as the failover signal for pg_standby (Heikki)
- Make pg_standby's `maxretries` option behave as documented (Fujii Masao)
- Make `contrib/hstore` throw an error when a key or value is too long to fit in its data structure, rather than silently truncating it (Andrew Gierth)
- Fix `contrib/xml2`'s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)
- Improve robustness of libpq's code to recover from errors during **COPY FROM STDIN** (Tom)
- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)
- Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.59. Release 8.3.7

Release Date

2009-03-16

This release contains a variety of fixes from 8.3.6. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.59.1. Migration to Version 8.3.7

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.5, see Section E.61, « Release 8.3.5 ».

E.59.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)

  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that *any* error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)

- Disallow **CREATE CONVERSION** with the wrong encodings for the specified conversion function (Heikki)

  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.

- Fix `xpath()` to not modify the path expression unless necessary, and to make a saner attempt at it when necessary (Andrew)

  The SQL standard suggests that `xpath` should work on data that is a document fragment, but libxml doesn't support that, and indeed it's not clear that this is sensible according to the XPath standard. `xpath` attempted to work around this mismatch by modifying both the data and the path expression, but the modification was buggy and could cause valid searches to fail. Now, `xpath` checks whether the data is in fact a well-formed document, and if so invokes libxml with no change to the data or path expression. Otherwise, a different modification method that is somewhat less likely to fail is used.

  > **Note**
  >
  > The new modification method is still not 100% satisfactory, and it seems likely that no real solution is possible. This patch should therefore be viewed as a band-aid to keep from breaking existing applications unnecessarily. It is likely that PostgreSQL™ 8.4 will simply reject use of `xpath` on data that is not a well-formed document.

- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)
- Fix possible failure in text search when C locale is used with a multi-byte encoding (Teodor)

  Crashes were possible on platforms where wchar_t is narrower than int; Windows in particular.

- Fix extreme inefficiency in text search parser's handling of an email-like string containing multiple `@` characters (Heikki)
- Fix planner problem with sub-**SELECT** in the output list of a larger subquery (Tom)

  The known symptom of this bug is a « failed to locate grouping columns » error that is dependent on the datatype involved; but there could be other issues as well.

- Fix decompilation of `CASE WHEN` with an implicit coercion (Tom)

  This mistake could lead to Assert failures in an Assert-enabled build, or an « unexpected CASE WHEN clause » error message in other cases, when trying to examine or dump a view.

- Fix possible misassignment of the owner of a TOAST table's rowtype (Tom)

  If **CLUSTER** or a rewriting variant of **ALTER TABLE** were executed by someone other than the table owner, the pg_type entry for the table's TOAST table would end up marked as owned by that someone. This caused no immediate problems, since the permissions on the TOAST rowtype aren't examined by any ordinary database operation. However, it could lead to unexpected failures if one later tried to drop the role that issued the command (in 8.1 or 8.2), or « owner of data type appears to be invalid » warnings from pg_dump after having done so (in 8.3).

- Change **UNLISTEN** to exit quickly if the current session has never executed any **LISTEN** command (Tom)

  Most of the time this is not a particularly useful optimization, but since **DISCARD ALL** invokes **UNLISTEN**, the previous coding caused a substantial performance problem for applications that made heavy use of **DISCARD ALL**.

- Fix PL/pgSQL to not treat `INTO` after **INSERT** as an INTO-variables clause anywhere in the string, not only at the start; in particular, don't fail for **INSERT INTO** within **CREATE RULE** (Tom)
- Clean up PL/pgSQL error status variables fully at block exit (Ashesh Vashi and Dave Page)

  This is not a problem for PL/pgSQL itself, but the omission could cause the PL/pgSQL Debugger to crash while examining the state of a function.

- Retry failed calls to `CallNamedPipe()` on Windows (Steve Marshall, Magnus)

  It appears that this function can sometimes fail transiently; we previously treated any failure as a hard error, which could confuse **LISTEN**/**NOTIFY** as well as other operations.
- Add `MUST` (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

E.60. Release 8.3.6

Release Date

2009-02-02

This release contains a variety of fixes from 8.3.5. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.60.1. Migration to Version 8.3.6

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.5, see Section E.61, « Release 8.3.5 ».

E.60.2. Changes

- Make **DISCARD ALL** release advisory locks, in addition to everything it already did (Tom)

  This was decided to be the most appropriate behavior. This could affect existing applications, however.
- Fix whole-index GiST scans to work correctly (Teodor)

  This error could cause rows to be lost if a table is clustered on a GiST index.
- Fix crash of `xmlconcat(NULL)` (Peter)
- Fix possible crash in `ispell` dictionary if high-bit-set characters are used as flags (Teodor)

  This is known to be done by one widely available Norwegian dictionary, and the same condition may exist in others.
- Fix misordering of pg_dump output for composite types (Tom)

  The most likely problem was for user-defined operator classes to be dumped after indexes or views that needed them.
- Improve handling of URLs in `headline()` function (Teodor)
- Improve handling of overlength headlines in `headline()` function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Fix possible Assert failure if a statement executed in PL/pgSQL is rewritten into another kind of statement, for example if an **INSERT** is rewritten into an **UPDATE** (Heikki)
- Ensure that a snapshot is available to datatype input functions (Tom)

  This primarily affects domains that are declared with `CHECK` constraints involving user-defined stable or immutable functions. Such functions typically fail if no snapshot has been set.
- Make it safer for SPI-using functions to be used within datatype I/O; in particular, to be used in domain check constraints (Tom)
- Avoid unnecessary locking of small tables in **VACUUM** (Heikki)
- Fix a problem that sometimes kept **ALTER TABLE ENABLE/DISABLE RULE** from being recognized by active sessions (Tom)
- Fix a problem that made `UPDATE RETURNING tableoid` return zero instead of the correct OID (Tom)
- Allow functions declared as taking ANYARRAY to work on the pg_statistic columns of that type (Tom)

  This used to work, but was unintentionally broken in 8.3.
- Fix planner misestimation of selectivity when transitive equality is applied to an outer-join clause (Tom)

This could result in bad plans for queries like `... from a left join b on a.a1 = b.b1 where a.a1 = 42 ...`

- Improve optimizer's handling of long `IN` lists (Tom)

  This change avoids wasting large amounts of time on such lists when constraint exclusion is enabled.

- Prevent synchronous scan during GIN index build (Tom)

  Because GIN is optimized for inserting tuples in increasing TID order, choosing to use a synchronous scan could slow the build by a factor of three or more.

- Ensure that the contents of a holdable cursor don't depend on the contents of TOAST tables (Tom)

  Previously, large field values in a cursor result might be represented as TOAST pointers, which would fail if the referenced table got dropped before the cursor is read, or if the large value is deleted and then vacuumed away. This cannot happen with an ordinary cursor, but it could with a cursor that is held past its creating transaction.

- Fix memory leak when a set-returning function is terminated without reading its whole result (Tom)
- Fix encoding conversion problems in XML functions when the database encoding isn't UTF-8 (Tom)
- Fix `contrib/dblink`'s `dblink_get_result(text,bool)` function (Joe)
- Fix possible garbage output from `contrib/sslinfo` functions (Tom)
- Fix incorrect behavior of `contrib/tsearch2` compatibility trigger when it's fired more than once in a command (Teodor)
- Fix possible mis-signaling in autovacuum (Heikki)
- Support running as a service on Windows 7 beta (Dave and Magnus)
- Fix ecpg's handling of varchar structs (Michael)
- Fix configure script to properly report failure when unable to obtain linkage information for PL/Perl (Andrew)
- Make all documentation reference `pgsql-bugs` and/or `pgsql-hackers` as appropriate, instead of the now-decommissioned `pgsql-ports` and `pgsql-patches` mailing lists (Tom)
- Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections in Switzerland, Cuba)

E.61. Release 8.3.5

Release Date

2008-11-03

This release contains a variety of fixes from 8.3.4. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.61.1. Migration to Version 8.3.5

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see Section E.65, « Release 8.3.1 ». Also, if you were running a previous 8.3.X release, it is recommended to **REINDEX** all GiST indexes after the upgrade.

E.61.2. Changes

- Fix GiST index corruption due to marking the wrong index entry « dead » after a deletion (Teodor)

  This would result in index searches failing to find rows they should have found. Corrupted indexes can be fixed with **REINDEX**.

- Fix backend crash when the client encoding cannot represent a localized error message (Tom)

  We have addressed similar issues before, but it would still fail if the « character has no equivalent » message itself couldn't be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.

- Fix possible crash in bytea-to-XML mapping (Michael McMaster)

- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Improve optimization of *expression* `IN` (*expression-list*) queries (Tom, per an idea from Robert Haas)

  Cases in which there are query variables on the right-hand side had been handled less efficiently in 8.2.x and 8.3.x than in prior versions. The fix restores 8.1 behavior for such cases.
- Fix mis-expansion of rule queries when a sub-`SELECT` appears in a function call in `FROM`, a multi-row `VALUES` list, or a `RETURNING` list (Tom)

  The usual symptom of this problem is an « unrecognized node type » error.
- Fix Assert failure during rescan of an `IS NULL` search of a GiST index (Teodor)
- Fix memory leak during rescan of a hashed aggregation plan (Neil)
- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Force a checkpoint before **CREATE DATABASE** starts to copy files (Heikki)

  This prevents a possible failure if files had recently been deleted in the source database.
- Prevent possible collision of *relfilenode* numbers when moving a table to another tablespace with **ALTER SET TABLESPACE** (Heikki)

  The command tried to re-use the existing filename, instead of picking one that is known unused in the destination directory.
- Fix incorrect text search headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an `--enable-integer-datetimes` build (Ron Mayer)
- Make `ILIKE` compare characters case-insensitively even when they're escaped (Andrew)
- Ensure **DISCARD** is handled properly by statement logging (Tom)
- Fix incorrect logging of last-completed-transaction time during PITR recovery (Tom)
- Ensure `SPI_getvalue` and `SPI_getbinval` behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)

  This situation is normal when a table has had columns added or removed, but these two functions didn't handle it properly. The only likely consequence is an incorrect error indication.
- Mark `SessionReplicationRole` as `PGDLLIMPORT` so it can be used by Slony on Windows (Magnus)
- Fix small memory leak when using libpq's `gsslib` parameter (Magnus)

  The space used by the parameter string was not freed at connection close.
- Ensure libgssapi is linked into libpq if needed (Markus Schaaf)
- Fix ecpg's parsing of **CREATE ROLE** (Michael)
- Fix recent breakage of `pg_ctl restart` (Tom)
- Ensure `pg_control` is opened in binary mode (Itagaki Takahiro)

  pg_controldata and pg_resetxlog did this incorrectly, and so could fail on Windows.
- Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil, Mauritius, Syria)

E.62. Release 8.3.4

Release Date

2008-09-22

This release contains a variety of fixes from 8.3.3. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.62.1. Migration to Version 8.3.4

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see Section E.65, « Release 8.3.1 ».

E.62.2. Changes

- Fix bug in btree WAL recovery code (Heikki)

  Recovery failed if the WAL ended partway through a page split operation.

- Fix potential use of wrong cutoff XID for HOT page pruning (Alvaro)

  This error created a risk of corruption in system catalogs that are consulted by **VACUUM**: dead tuple versions might be removed too soon. The impact of this on actual database operations would be minimal, since the system doesn't follow MVCC rules while examining catalogs, but it might result in transiently wrong output from pg_dump or other client programs.

- Fix potential miscalculation of *`datfrozenxid`* (Alvaro)

  This error may explain some recent reports of failure to remove old pg_clog data.

- Fix incorrect HOT updates after pg_class is reindexed (Tom)

  Corruption of pg_class could occur if `REINDEX TABLE pg_class` was followed in the same session by an `ALTER TABLE RENAME` or `ALTER TABLE SET SCHEMA` command.

- Fix missed « combo cid » case (Karl Schnaitter)

  This error made rows incorrectly invisible to a transaction in which they had been deleted by multiple subtransactions that all aborted.

- Prevent autovacuum from crashing if the table it's currently checking is deleted at just the wrong time (Alvaro)

- Widen local lock counters from 32 to 64 bits (Tom)

  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected « lock is already held » errors.

- Fix possible duplicate output of tuples during a GiST index scan (Teodor)

- Regenerate foreign key checking queries from scratch when either table is modified (Tom)

  Previously, 8.3 would attempt to replan the query, but would work from previously generated query text. This led to failures if a table or column was renamed.

- Fix missed permissions checks when a view contains a simple `UNION ALL` construct (Heikki)

  Permissions for the referenced tables were checked properly, but not permissions for the view itself.

- Add checks in executor startup to ensure that the tuples produced by an **INSERT** or **UPDATE** will match the target table's current rowtype (Tom)

  This situation is believed to be impossible in 8.3, but it can happen in prior releases, so a check seems prudent.

- Fix possible repeated drops during **DROP OWNED** (Tom)

  This would typically result in strange errors such as « cache lookup failed for relation NNN ».

- Fix several memory leaks in XML operations (Kris Jurka, Tom)

- Fix `xmlserialize()` to raise error properly for unacceptable target data type (Tom)

- Fix a couple of places that mis-handled multibyte characters in text search configuration file parsing (Tom)

  Certain characters occurring in configuration files would always cause « invalid byte sequence for encoding » failures.

- Provide file name and line number location for all errors reported in text search configuration files (Tom)

- Fix `AT TIME ZONE` to first try to interpret its timezone argument as a timezone abbreviation, and only try it as a full timezone name if that fails, rather than the other way around as formerly (Tom)

  The timestamp input functions have always resolved ambiguous zone names in this order. Making `AT TIME ZONE` do so as well improves consistency, and fixes a compatibility bug introduced in 8.1: in ambiguous cases we now behave the same as 8.0 and before did, since in the older versions `AT TIME ZONE` accepted *only* abbreviations.

- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)

- Prevent integer overflows during units conversion when displaying a configuration parameter that has units (Tom)

- Improve performance of writing very long log messages to syslog (Tom)
- Allow spaces in the suffix part of an LDAP URL in `pg_hba.conf` (Tom)
- Fix bug in backwards scanning of a cursor on a `SELECT DISTINCT ON` query (Tom)
- Fix planner bug that could improperly push down `IS NULL` tests below an outer join (Tom)

  This was triggered by occurrence of `IS NULL` tests for the same relation in all arms of an upper `OR` clause.
- Fix planner bug with nested sub-select expressions (Tom)

  If the outer sub-select has no direct dependency on the parent query, but the inner one does, the outer value might not get re-calculated for new parent query rows.
- Fix planner to estimate that `GROUP BY` expressions yielding boolean results always result in two groups, regardless of the expressions' contents (Tom)

  This is very substantially more accurate than the regular `GROUP BY` estimate for certain boolean tests like *`col`* `IS NULL`.
- Fix PL/pgSQL to not fail when a `FOR` loop's target variable is a record containing composite-type fields (Tom)
- Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)
- Improve performance of `PQescapeBytea()` (Rudolf Leitgeb)
- On Windows, work around a Microsoft bug by preventing libpq from trying to send more than 64kB per system call (Magnus)
- Fix ecpg to handle variables properly in **SET** commands (Michael)
- Improve pg_dump and pg_restore's error reporting after failure to send a SQL command (Tom)
- Fix pg_ctl to properly preserve postmaster command-line arguments across a `restart` (Bruce)
- Fix erroneous WAL file cutoff point calculation in pg_standby (Simon)
- Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.63. Release 8.3.3

Release Date

2008-06-12

This release contains one serious and one minor bug fix over 8.3.2. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.63.1. Migration to Version 8.3.3

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see Section E.65, « Release 8.3.1 ».

E.63.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)

  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL™ to reject what had been a valid **SELECT DISTINCT** view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.
- Make **ALTER AGGREGATE ... OWNER TO** update pg_shdepend (Tom)

  This oversight could lead to problems if the aggregate was later involved in a **DROP OWNED** or **REASSIGN OWNED** operation.

E.64. Release 8.3.2

Release Date

never released

This release contains a variety of fixes from 8.3.1. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.64.1. Migration to Version 8.3.2

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see Section E.65, « Release 8.3.1 ».

E.64.2. Changes

- Fix `ERRORDATA_STACK_SIZE exceeded` crash that occurred on Windows when using UTF-8 database encoding and a different client encoding (Tom)
- Fix incorrect archive truncation point calculation for the `%r` macro in `recovery_command` parameters (Simon)

  This could lead to data loss if a warm-standby script relied on `%r` to decide when to throw away WAL segment files.
- Fix **ALTER TABLE ADD COLUMN ... PRIMARY KEY** so that the new column is correctly checked to see if it's been initialized to all non-nulls (Brendan Jurd)

  Previous versions neglected to check this requirement at all.
- Fix **REASSIGN OWNED** so that it works on procedural languages too (Alvaro)
- Fix problems with **SELECT FOR UPDATE/SHARE** occurring as a subquery in a query with a non-**SELECT** top-level operation (Tom)
- Fix possible **CREATE TABLE** failure when alling the « same » constraint from multiple parent relations that alled that constraint from a common ancestor (Tom)
- Fix `pg_get_ruledef()` to show the alias, if any, attached to the target table of an **UPDATE** or **DELETE** (Tom)
- Restore the pre-8.3 behavior that an out-of-range block number in a TID being used in a TidScan plan results in silently not matching any rows (Tom)

  8.3.0 and 8.3.1 threw an error instead.
- Fix GIN bug that could result in a `too many LWLocks taken` failure (Teodor)
- Fix broken GiST comparison function for tsquery (Teodor)
- Fix `tsvector_update_trigger()` and `ts_stat()` to accept domains over the types they expect to work with (Tom)
- Fix failure to support enum data types as foreign keys (Tom)
- Avoid possible crash when decompressing corrupted data (Zdenek Kotala)
- Fix race conditions between delayed unlinks and **DROP DATABASE** (Heikki)

  In the worst case this could result in deleting a newly created table in a new database that happened to get the same OID as the recently-dropped one; but of course that is an extremely low-probability scenario.
- Repair two places where SIGTERM exit of a backend could leave corrupted state in shared memory (Tom)

  Neither case is very important if SIGTERM is used to shut down the whole database cluster together, but there was a problem if someone tried to SIGTERM individual backends.
- Fix possible crash due to incorrect plan generated for an `x IN (SELECT y FROM ...)` clause when *x* and *y* have different data types; and make sure the behavior is semantically correct when the conversion from *y*'s type to *x*'s type is lossy (Tom)
- Fix oversight that prevented the planner from substituting known Param values as if they were constants (Tom)

  This mistake partially disabled optimization of unnamed extended-Query statements in 8.3.0 and 8.3.1: in particular the LIKE-to-indexscan optimization would never be applied if the LIKE pattern was passed as a parameter, and constraint exclusion depending on a parameter value didn't work either.
- Fix planner failure when an indexable `MIN` or `MAX` aggregate is used with `DISTINCT` or `ORDER BY` (Tom)

- Fix planner to ensure it never uses a « physical tlist » for a plan node that is feeding a Sort node (Tom)

  This led to the sort having to push around more data than it really needed to, since unused column values were included in the sorted data.
- Avoid unnecessary copying of query strings (Tom)

  This fixes a performance problem introduced in 8.3.0 when a very large number of commands are submitted as a single query string.
- Make `TransactionIdIsCurrentTransactionId()` use binary search instead of linear search when checking child-transaction XIDs (Heikki)

  This fixes some cases in which 8.3.0 was significantly slower than earlier releases.
- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic « Yo » characters (`e` and `E` with two dots) (Sergey Burladyan)
- Fix several datatype input functions, notably `array_in()`, that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched `ORDER BY` and `DISTINCT` expressions.
- Fix a corner case in regular-expression substring matching (`substring(`*`string`*` from `*`pattern`*`)`) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn't got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since `(bar)` isn't matched, but it was mistakenly returning the whole-pattern match instead (ie, `foo`).
- Prevent cancellation of an auto-vacuum that was launched to prevent XID wraparound (Alvaro)
- Improve **ANALYZE**'s handling of in-doubt tuples (those inserted or deleted by a not-yet-committed transaction) so that the counts it reports to the stats collector are more likely to be correct (Pavan Deolasee)
- Fix initdb to reject a relative path for its `--xlogdir` (`-X`) option (Tom)
- Make psql print tab characters as an appropriate number of spaces, rather than `\x09` as was done in 8.3.0 and 8.3.1 (Bruce)
- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, and Argentina/San_Luis)
- Add `ECPGget_PGconn()` function to ecpglib (Michael)
- Fix incorrect result from ecpg's `PGTYPEStimestamp_sub()` function (Michael)
- Fix handling of continuation line markers in ecpg (Michael)
- Fix possible crashes in `contrib/cube` functions (Tom)
- Fix core dump in `contrib/xml2`'s `xpath_table()` function when the input query returns a NULL value (Tom)
- Fix `contrib/xml2`'s makefile to not override `CFLAGS`, and make it auto-configure properly for libxslt present or not (Tom)

E.65. Release 8.3.1

> **Release Date**
>
> 2008-03-17

This release contains a variety of fixes from 8.3.0. For information about new features in the 8.3 major release, see Section E.66, « Release 8.3 ».

E.65.1. Migration to Version 8.3.1

A dump/restore is not required for those running 8.3.X. However, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the Windows locale issue described below.

E.65.2. Changes

- Fix character string comparison for Windows locales that consider different character combinations as equal (Tom)

  This fix applies only on Windows and only when using UTF-8 database encoding. The same fix was made for all other cases over two years ago, but Windows with UTF-8 uses a separate code path that was not updated. If you are using a locale that considers some non-identical strings as equal, you may need to **REINDEX** to fix existing indexes on textual columns.

- Repair corner-case bugs in **VACUUM FULL** (Tom)

  A potential deadlock between concurrent **VACUUM FULL** operations on different system catalogs was introduced in 8.2. This has now been corrected. 8.3 made this worse because the deadlock could occur within a critical code section, making it a PANIC rather than just ERROR condition.

  Also, a **VACUUM FULL** that failed partway through vacuuming a system catalog could result in cache corruption in concurrent database sessions.

  Another **VACUUM FULL** bug introduced in 8.3 could result in a crash or out-of-memory report when dealing with pages containing no live tuples.

- Fix misbehavior of foreign key checks involving character or bit columns (Tom)

  If the referencing column were of a different but compatible type (for instance varchar), the constraint was enforced incorrectly.

- Avoid needless deadlock failures in no-op foreign-key checks (Stephan Szabo, Tom)

- Fix possible core dump when re-planning a prepared query (Tom)

  This bug affected only protocol-level prepare operations, not SQL **PREPARE**, and so tended to be seen only with JDBC, DBI, and other client-side drivers that use prepared statements heavily.

- Fix possible failure when re-planning a query that calls an SPI-using function (Tom)

- Fix failure in row-wise comparisons involving columns of different datatypes (Tom)

- Fix longstanding **LISTEN**/**NOTIFY** race condition (Tom)

  In rare cases a session that had just executed a **LISTEN** might not get a notification, even though one would be expected because the concurrent transaction executing **NOTIFY** was observed to commit later.

  A side effect of the fix is that a transaction that has executed a not-yet-committed **LISTEN** command will not see any row in pg_listener for the **LISTEN**, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Disallow **LISTEN** and **UNLISTEN** within a prepared transaction (Tom)

  This was formerly allowed but trying to do it had various unpleasant consequences, notably that the originating backend could not exit as long as an **UNLISTEN** remained uncommitted.

- Disallow dropping a temporary table within a prepared transaction (Heikki)

  This was correctly disallowed by 8.1, but the check was inadvertently broken in 8.2 and 8.3.

- Fix rare crash when an error occurs during a query using a hash index (Heikki)

- Fix incorrect comparison of tsquery values (Teodor)

- Fix incorrect behavior of `LIKE` with non-ASCII characters in single-byte encodings (Rolf Jentsch)

- Disable `xmlvalidate` (Tom)

  This function should have been removed before 8.3 release, but was inadvertently left in the source code. It poses a small security risk since unprivileged users could use it to read the first few characters of any file accessible to the server.

- Fix memory leaks in certain usages of set-returning functions (Neil)

- Make `encode(`*`bytea`*`, 'escape')` convert all high-bit-set byte values into `\`*`nnn`* octal escape sequences (Tom)

  This is necessary to avoid encoding problems when the database encoding is multi-byte. This change could pose compatibility issues for applications that are expecting specific results from `encode`.

- Fix input of datetime values for February 29 in years BC (Tom)

  The former coding was mistaken about which years were leap years.

- Fix « unrecognized node type » error in some variants of **ALTER OWNER** (Tom)

- Avoid tablespace permissions errors in **CREATE TABLE LIKE INCLUDING INDEXES** (Tom)
- Ensure pg_stat_activity.*`waiting`* flag is cleared when a lock wait is aborted (Tom)
- Fix handling of process permissions on Windows Vista (Dave, Magnus)

  In particular, this fix allows starting the server as the Administrator user.
- Update time zone data files to tzdata release 2008a (in particular, recent Chile changes); adjust timezone abbreviation `VET` (Venezuela) to mean UTC-4:30, not UTC-4:00 (Tom)
- Fix ecpg problems with arrays (Michael)
- Fix pg_ctl to correctly extract the postmaster's port number from command-line options (Itagaki Takahiro, Tom)

  Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.
- Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)

  This is known to be necessary when building PostgreSQL™ with gcc 4.3 or later.
- Enable building `contrib/uuid-ossp` with MSVC (Hiroshi Saito)

E.66. Release 8.3

Release Date

2008-02-04

E.66.1. Overview

With significant new functionality and performance enhancements, this release represents a major leap forward for PostgreSQL™. This was made possible by a growing community that has dramatically accelerated the pace of development. This release adds the following major features:

- Full text search is integrated into the core database system
- Support for the SQL/XML standard, including new operators and an XML data type
- Enumerated data types (ENUM)
- Arrays of composite types
- Universally Unique Identifier (UUID) data type
- Add control over whether `NULL`s sort first or last
- Updatable cursors
- Server configuration parameters can now be set on a per-function basis
- User-defined types can now have type modifiers
- Automatically re-plan cached queries when table definitions change or statistics are updated
- Numerous improvements in logging and statistics collection
- Support Security Service Provider Interface (SSPI) for authentication on Windows
- Support multiple concurrent autovacuum processes, and other autovacuum improvements
- Allow the whole PostgreSQL™ distribution to be compiled with Microsoft Visual C++™

Major performance improvements are listed below. Most of these enhancements are automatic and do not require user changes or tuning:

- Asynchronous commit delays writes to WAL during transaction commit
- Checkpoint writes can be spread over a longer time period to smooth the I/O spike during each checkpoint
- Heap-Only Tuples (HOT) accelerate space reuse for most **UPDATE**s and **DELETE**s

- Just-in-time background writer strategy improves disk write efficiency
- Using non-persistent transaction IDs for read-only transactions reduces overhead and **VACUUM** requirements
- Per-field and per-row storage overhead has been reduced
- Large sequential scans no longer force out frequently used cached pages
- Concurrent large sequential scans can now share disk reads
- `ORDER BY ... LIMIT` can be done without sorting

The above items are explained in more detail in the sections below.

E.66.2. Migration to Version 8.3

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

E.66.2.1. General

- Non-character data types are no longer automatically cast to TEXT (Peter, Tom)

  Previously, if a non-character value was supplied to an operator or function that requires text input, it was automatically cast to text, for most (though not all) built-in data types. This no longer happens: an explicit cast to text is now required for all non-character-string types. For example, these expressions formerly worked:

  ```
  substr(current_date, 1, 4)
  23 LIKE '2%'
  ```

  but will now draw « function does not exist » and « operator does not exist » errors respectively. Use an explicit cast instead:

  ```
  substr(current_date::text, 1, 4)
  23::text LIKE '2%'
  ```

  (Of course, you can use the more verbose `CAST()` syntax too.) The reason for the change is that these automatic casts too often caused surprising behavior. An example is that in previous releases, this expression was accepted but did not do what was expected:

  ```
  current_date < 2017-11-17
  ```

  This is actually comparing a date to an integer, which should be (and now is) rejected -- but in the presence of automatic casts both sides were cast to text and a textual comparison was done, because the `text < text` operator was able to match the expression when no other `<` operator could.

  Types char(*n*) and varchar(*n*) still cast to text automatically. Also, automatic casting to text still works for inputs to the concatenation (`||`) operator, so long as least one input is a character-string type.

- Full text search features from `contrib/tsearch2` have been moved into the core server, with some minor syntax changes

  `contrib/tsearch2` now contains a compatibility interface.

- `ARRAY(SELECT ...)`, where the **SELECT** returns no rows, now returns an empty array, rather than NULL (Tom)
- The array type name for a base data type is no longer always the base type's name with an underscore prefix

  The old naming convention is still honored when possible, but application code should no longer depend on it. Instead use the new `pg_type.typarray` column to identify the array data type associated with a given type.

- `ORDER BY ... USING` *`operator`* must now use a less-than or greater-than *`operator`* that is defined in a btree operator class

  This restriction was added to prevent inconsistent results.

- **SET LOCAL** changes now persist until the end of the outermost transaction, unless rolled back (Tom)

  Previously **SET LOCAL**'s effects were lost after subtransaction commit (**RELEASE SAVEPOINT** or exit from a PL/pgSQL exception block).

- Commands rejected in transaction blocks are now also rejected in multiple-statement query strings (Tom)

For example, `"BEGIN; DROP DATABASE; COMMIT"` will now be rejected even if submitted as a single query message.

- **ROLLBACK** outside a transaction block now issues `NOTICE` instead of `WARNING` (Bruce)
- Prevent **NOTIFY**/**LISTEN**/**UNLISTEN** from accepting schema-qualified names (Bruce)

  Formerly, these commands accepted `schema.relation` but ignored the schema part, which was confusing.
- **ALTER SEQUENCE** no longer affects the sequence's `currval()` state (Tom)
- Foreign keys now must match indexable conditions for cross-data-type references (Tom)

  This improves semantic consistency and helps avoid performance problems.
- Restrict object size functions to users who have reasonable permissions to view such information (Tom)

  For example, `pg_database_size()` now requires `CONNECT` permission, which is granted to everyone by default. `pg_tablespace_size()` requires `CREATE` permission in the tablespace, or is allowed if the tablespace is the default tablespace for the database.
- Remove the undocumented `!!=` (not in) operator (Tom)

  `NOT IN (SELECT ...)` is the proper way to perform this operation.
- Internal hashing functions are now more uniformly-distributed (Tom)

  If application code was computing and storing hash values using internal PostgreSQL™ hashing functions, the hash values must be regenerated.
- C-code conventions for handling variable-length data values have changed (Greg Stark, Tom)

  The new `SET_VARSIZE()` macro *must* be used to set the length of generated varlena values. Also, it might be necessary to expand (« de-TOAST ») input values in more cases.
- Continuous archiving no longer reports each successful archive operation to the server logs unless `DEBUG` level is used (Simon)

E.66.2.2. Configuration Parameters

- Numerous changes in administrative server parameters

  `bgwriter_lru_percent`, `bgwriter_all_percent`, `bgwriter_all_maxpages`, `stats_start_collector`, and `stats_reset_on_server_start` are removed. `redirect_stderr` is renamed to `logging_collector`. `stats_command_string` is renamed to `track_activities`. `stats_block_level` and `stats_row_level` are merged into `track_counts`. A new boolean configuration parameter, `archive_mode`, controls archiving. Autovacuum's default settings have changed.
- Remove `stats_start_collector` parameter (Tom)

  We now always start the collector process, unless UDP socket creation fails.
- Remove `stats_reset_on_server_start` parameter (Tom)

  This was removed because `pg_stat_reset()` can be used for this purpose.
- Commenting out a parameter in `postgresql.conf` now causes it to revert to its default value (Joachim Wieland)

  Previously, commenting out an entry left the parameter's value unchanged until the next server restart.

E.66.2.3. Character Encodings

- Add more checks for invalidly-encoded data (Andrew)

  This change plugs some holes that existed in literal backslash escape string processing and **COPY** escape processing. Now the de-escaped string is rechecked to see if the result created an invalid multi-byte character.
- Disallow database encodings that are inconsistent with the server's locale setting (Tom)

  On most platforms, `C` locale is the only locale that will work with any database encoding. Other locale settings imply a specific encoding and will misbehave if the database encoding is something different. (Typical symptoms include bogus textual sort order and wrong results from `upper()` or `lower()`.) The server now rejects attempts to create databases that have an incompatible encoding.

- Ensure that `chr()` cannot create invalidly-encoded values (Andrew)

  In UTF8-encoded databases the argument of `chr()` is now treated as a Unicode code point. In other multi-byte encodings `chr()`'s argument must designate a 7-bit ASCII character. Zero is no longer accepted. `ascii()` has been adjusted to match.

- Adjust `convert()` behavior to ensure encoding validity (Andrew)

  The two argument form of `convert()` has been removed. The three argument form now takes a bytea first argument and returns a bytea. To cover the loss of functionality, three new functions have been added:

  - `convert_from(bytea, name)` returns text -- converts the first argument from the named encoding to the database encoding
  - `convert_to(text, name)` returns bytea -- converts the first argument from the database encoding to the named encoding
  - `length(bytea, name)` returns integer -- gives the length of the first argument in characters in the named encoding

- Remove `convert(argument USING conversion_name)` (Andrew)

  Its behavior did not match the SQL standard.

- Make JOHAB encoding client-only (Tatsuo)

  JOHAB is not safe as a server-side encoding.

E.66.3. Changes

Below you will find a detailed account of the changes between PostgreSQL™ 8.3 and the previous major release.

E.66.3.1. Performance

- Asynchronous commit delays writes to WAL during transaction commit (Simon)

  This feature dramatically increases performance for short data-modifying transactions. The disadvantage is that because disk writes are delayed, if the database or operating system crashes before data is written to the disk, committed data will be lost. This feature is useful for applications that can accept some data loss. Unlike turning off `fsync`, using asynchronous commit does not put database consistency at risk; the worst case is that after a crash the last few reportedly-committed transactions might not be committed after all. This feature is enabled by turning off `synchronous_commit` (which can be done per-session or per-transaction, if some transactions are critical and others are not). `wal_writer_delay` can be adjusted to control the maximum delay before transactions actually reach disk.

- Checkpoint writes can be spread over a longer time period to smooth the I/O spike during each checkpoint (Itagaki Takahiro and Heikki Linnakangas)

  Previously all modified buffers were forced to disk as quickly as possible during a checkpoint, causing an I/O spike that decreased server performance. This new approach spreads out disk writes during checkpoints, reducing peak I/O usage. (User-requested and shutdown checkpoints are still written as quickly as possible.)

- Heap-Only Tuples (HOT) accelerate space reuse for most **UPDATE**s and **DELETE**s (Pavan Deolasee, with ideas from many others)

  UPDATEs and **DELETE**s leave dead tuples behind, as do failed **INSERT**s. Previously only **VACUUM** could reclaim space taken by dead tuples. With HOT dead tuple space can be automatically reclaimed at the time of **INSERT** or **UPDATE** if no changes are made to indexed columns. This allows for more consistent performance. Also, HOT avoids adding duplicate index entries.

- Just-in-time background writer strategy improves disk write efficiency (Greg Smith, Itagaki Takahiro)

  This greatly reduces the need for manual tuning of the background writer.

- Per-field and per-row storage overhead have been reduced (Greg Stark, Heikki Linnakangas)

  Variable-length data types with data values less than 128 bytes long will see a storage decrease of 3 to 6 bytes. For example, two adjacent char(1) fields now use 4 bytes instead of 16. Row headers are also 4 bytes shorter than before.

- Using non-persistent transaction IDs for read-only transactions reduces overhead and **VACUUM** requirements (Florian Pflug)

  Non-persistent transaction IDs do not increment the global transaction counter. Therefore, they reduce the load on pg_clog and increase the time between forced vacuums to prevent transaction ID wraparound. Other performance improvements were also

made that should improve concurrency.

- Avoid incrementing the command counter after a read-only command (Tom)

  There was formerly a hard limit of 2^{32} (4 billion) commands per transaction. Now only commands that actually changed the database count, so while this limit still exists, it should be significantly less annoying.

- Create a dedicated WAL writer process to off-load work from backends (Simon)
- Skip unnecessary WAL writes for **CLUSTER** and **COPY** (Simon)

  Unless WAL archiving is enabled, the system now avoids WAL writes for **CLUSTER** and just `fsync()`s the table at the end of the command. It also does the same for **COPY** if the table was created in the same transaction.

- Large sequential scans no longer force out frequently used cached pages (Simon, Heikki, Tom)
- Concurrent large sequential scans can now share disk reads (Jeff Davis)

  This is accomplished by starting the new sequential scan in the middle of the table (where another sequential scan is already in-progress) and wrapping around to the beginning to finish. This can affect the order of returned rows in a query that does not specify `ORDER BY`. The `synchronize_seqscans` configuration parameter can be used to disable this if necessary.

- `ORDER BY ... LIMIT` can be done without sorting (Greg Stark)

  This is done by sequentially scanning the table and tracking just the « top N » candidate rows, rather than performing a full sort of the entire table. This is useful when there is no matching index and the `LIMIT` is not large.

- Put a rate limit on messages sent to the statistics collector by backends (Tom)

  This reduces overhead for short transactions, but might sometimes increase the delay before statistics are tallied.

- Improve hash join performance for cases with many NULLs (Tom)
- Speed up operator lookup for cases with non-exact datatype matches (Tom)

E.66.3.2. Server

- Autovacuum is now enabled by default (Alvaro)

  Several changes were made to eliminate disadvantages of having autovacuum enabled, thereby justifying the change in default. Several other autovacuum parameter defaults were also modified.

- Support multiple concurrent autovacuum processes (Alvaro, Itagaki Takahiro)

  This allows multiple vacuums to run concurrently. This prevents vacuuming of a large table from delaying vacuuming of smaller tables.

- Automatically re-plan cached queries when table definitions change or statistics are updated (Tom)

  Previously PL/pgSQL functions that referenced temporary tables would fail if the temporary table was dropped and recreated between function invocations, unless `EXECUTE` was used. This improvement fixes that problem and many related issues.

- Add a `temp_tablespaces` parameter to control the tablespaces for temporary tables and files (Jaime Casanova, Albert Cervera, Bernd Helmle)

  This parameter defines a list of tablespaces to be used. This enables spreading the I/O load across multiple tablespaces. A random tablespace is chosen each time a temporary object is created. Temporary files are no longer stored in per-database `pgsql_tmp/` directories but in per-tablespace directories.

- Place temporary tables' TOAST tables in special schemas named `pg_toast_temp_`*`nnn`* (Tom)

  This allows low-level code to recognize these tables as temporary, which enables various optimizations such as not WAL-logging changes and using local rather than shared buffers for access. This also fixes a bug wherein backends unexpectedly held open file references to temporary TOAST tables.

- Fix problem that a constant flow of new connection requests could indefinitely delay the postmaster from completing a shutdown or a crash restart (Tom)
- Guard against a very-low-probability data loss scenario by preventing re-use of a deleted table's relfilenode until after the next checkpoint (Heikki)
- Fix **CREATE CONSTRAINT TRIGGER** to convert old-style foreign key trigger definitions into regular foreign key constraints (Tom)

This will ease porting of foreign key constraints carried forward from pre-7.3 databases, if they were never converted using `contrib/adddepend`.

- Fix `DEFAULT NULL` to override alled defaults (Tom)

  `DEFAULT NULL` was formerly considered a noise phrase, but it should (and now does) override non-null defaults that would otherwise be alled from a parent table or domain.

- Add new encodings EUC_JIS_2004 and SHIFT_JIS_2004 (Tatsuo)

  These new encodings can be converted to and from UTF-8.

- Change server startup log message from « database system is ready » to « database system is ready to accept connections », and adjust its timing

  The message now appears only when the postmaster is really ready to accept connections.

E.66.3.3. Monitoring

- Add `log_autovacuum_min_duration` parameter to support configurable logging of autovacuum activity (Simon, Alvaro)
- Add `log_lock_waits` parameter to log lock waiting (Simon)
- Add `log_temp_files` parameter to log temporary file usage (Bill Moran)
- Add `log_checkpoints` parameter to improve logging of checkpoints (Greg Smith, Heikki)
- `log_line_prefix` now supports `%s` and `%c` escapes in all processes (Andrew)

  Previously these escapes worked only for user sessions, not for background database processes.

- Add `log_restartpoints` to control logging of point-in-time recovery restart points (Simon)
- Last transaction end time is now logged at end of recovery and at each logged restart point (Simon)
- Autovacuum now reports its activity start time in `pg_stat_activity` (Tom)
- Allow server log output in comma-separated value (CSV) format (Arul Shaji, Greg Smith, Andrew Dunstan)

  CSV-format log files can easily be loaded into a database table for subsequent analysis.

- Use PostgreSQL-supplied timezone support for formatting timestamps displayed in the server log (Tom)

  This avoids Windows-specific problems with localized time zone names that are in the wrong encoding. There is a new `log_timezone` parameter that controls the timezone used in log messages, independently of the client-visible `timezone` parameter.

- New system view `pg_stat_bgwriter` displays statistics about background writer activity (Magnus)
- Add new columns for database-wide tuple statistics to `pg_stat_database` (Magnus)
- Add an `xact_start` (transaction start time) column to `pg_stat_activity` (Neil)

  This makes it easier to identify long-running transactions.

- Add `n_live_tuples` and `n_dead_tuples` columns to `pg_stat_all_tables` and related views (Glen Parker)
- Merge `stats_block_level` and `stats_row_level` parameters into a single parameter `track_counts`, which controls all messages sent to the statistics collector process (Tom)
- Rename `stats_command_string` parameter to `track_activities` (Tom)
- Fix statistical counting of live and dead tuples to recognize that committed and aborted transactions have different effects (Tom)

E.66.3.4. Authentication

- Support Security Service Provider Interface (SSPI) for authentication on Windows (Magnus)
- Support GSSAPI authentication (Henry Hotz, Magnus)

  This should be preferred to native Kerberos authentication because GSSAPI is an industry standard.

- Support a global SSL configuration file (Victor Wagner)
- Add `ssl_ciphers` parameter to control accepted SSL ciphers (Victor Wagner)
- Add a Kerberos realm parameter, `krb_realm` (Magnus)

E.66.3.5. Write-Ahead Log (WAL) and Continuous Archiving

- Change the timestamps recorded in transaction WAL records from time_t to TimestampTz representation (Tom)

  This provides sub-second resolution in WAL, which can be useful for point-in-time recovery.
- Reduce WAL disk space needed by warm standby servers (Simon)

  This change allows a warm standby server to pass the name of the earliest still-needed WAL file to the recovery script, allowing automatic removal of no-longer-needed WAL files. This is done using `%r` in the `restore_command` parameter of `recovery.conf`.
- New boolean configuration parameter, `archive_mode`, controls archiving (Simon)

  Previously setting `archive_command` to an empty string turned off archiving. Now `archive_mode` turns archiving on and off, independently of `archive_command`. This is useful for stopping archiving temporarily.

E.66.3.6. Queries

- Full text search is integrated into the core database system (Teodor, Oleg)

  Text search has been improved, moved into the core code, and is now installed by default. `contrib/tsearch2` now contains a compatibility interface.
- Add control over whether `NULL`s sort first or last (Teodor, Tom)

  The syntax is `ORDER BY ... NULLS FIRST/LAST`.
- Allow per-column ascending/descending (`ASC`/`DESC`) ordering options for indexes (Teodor, Tom)

  Previously a query using `ORDER BY` with mixed `ASC`/`DESC` specifiers could not fully use an index. Now an index can be fully used in such cases if the index was created with matching `ASC`/`DESC` specifications. `NULL` sort order within an index can be controlled, too.
- Allow `col IS NULL` to use an index (Teodor)
- Updatable cursors (Arul Shaji, Tom)

  This eliminates the need to reference a primary key to **UPDATE** or **DELETE** rows returned by a cursor. The syntax is `UPDATE/DELETE WHERE CURRENT OF`.
- Allow `FOR UPDATE` in cursors (Arul Shaji, Tom)
- Create a general mechanism that supports casts to and from the standard string types (TEXT, VARCHAR, CHAR) for *every* datatype, by invoking the datatype's I/O functions (Tom)

  Previously, such casts were available only for types that had specialized function(s) for the purpose. These new casts are assignment-only in the to-string direction, explicit-only in the other direction, and therefore should create no surprising behavior.
- Allow `UNION` and related constructs to return a domain type, when all inputs are of that domain type (Tom)

  Formerly, the output would be considered to be of the domain's base type.
- Allow limited hashing when using two different data types (Tom)

  This allows hash joins, hash indexes, hashed subplans, and hash aggregation to be used in situations involving cross-data-type comparisons, if the data types have compatible hash functions. Currently, cross-data-type hashing support exists for smallint/integer/bigint, and for float4/float8.
- Improve optimizer logic for detecting when variables are equal in a `WHERE` clause (Tom)

  This allows mergejoins to work with descending sort orders, and improves recognition of redundant sort columns.
- Improve performance when planning large allance trees in cases where most tables are excluded by constraints (Tom)

E.66.3.7. Object Manipulation

- Arrays of composite types (David Fetter, Andrew, Tom)

  In addition to arrays of explicitly-declared composite types, arrays of the rowtypes of regular tables and views are now supported, except for rowtypes of system catalogs, sequences, and TOAST tables.

- Server configuration parameters can now be set on a per-function basis (Tom)

  For example, functions can now set their own `search_path` to prevent unexpected behavior if a different `search_path` exists at run-time. Security definer functions should set `search_path` to avoid security loopholes.

- **CREATE/ALTER FUNCTION** now supports `COST` and `ROWS` options (Tom)

  `COST` allows specification of the cost of a function call. `ROWS` allows specification of the average number or rows returned by a set-returning function. These values are used by the optimizer in choosing the best plan.

- Implement **CREATE TABLE LIKE ... INCLUDING INDEXES** (Trevor Hardcastle, Nikhil Sontakke, Neil)
- Allow **CREATE INDEX CONCURRENTLY** to ignore transactions in other databases (Simon)
- Add **ALTER VIEW ... RENAME TO** and **ALTER SEQUENCE ... RENAME TO** (David Fetter, Neil)

  Previously this could only be done via **ALTER TABLE ... RENAME TO**.

- Make **CREATE/DROP/RENAME DATABASE** wait briefly for conflicting backends to exit before failing (Tom)

  This increases the likelihood that these commands will succeed.

- Allow triggers and rules to be deactivated in groups using a configuration parameter, for replication purposes (Jan)

  This allows replication systems to disable triggers and rewrite rules as a group without modifying the system catalogs directly. The behavior is controlled by **ALTER TABLE** and a new parameter `session_replication_role`.

- User-defined types can now have type modifiers (Teodor, Tom)

  This allows a user-defined type to take a modifier, like ssnum(7). Previously only built-in data types could have modifiers.

E.66.3.8. Utility Commands

- Non-superuser database owners now are able to add trusted procedural languages to their databases by default (Jeremy Drake)

  While this is reasonably safe, some administrators might wish to revoke the privilege. It is controlled by pg_pltemplate.*`tmpldbacreate`*.

- Allow a session's current parameter setting to be used as the default for future sessions (Tom)

  This is done with `SET ... FROM CURRENT` in **CREATE/ALTER FUNCTION**, **ALTER DATABASE**, or **ALTER ROLE**.

- Implement new commands **DISCARD ALL**, **DISCARD PLANS**, **DISCARD TEMPORARY**, **CLOSE ALL**, and **DEALLOCATE ALL** (Marko Kreen, Neil)

  These commands simplify resetting a database session to its initial state, and are particularly useful for connection-pooling software.

- Make **CLUSTER** MVCC-safe (Heikki Linnakangas)

  Formerly, **CLUSTER** would discard all tuples that were committed dead, even if there were still transactions that should be able to see them under MVCC visibility rules.

- Add new **CLUSTER** syntax: `CLUSTER` *`table`* `USING` *`index`* (Holger Schurig)

  The old **CLUSTER** syntax is still supported, but the new form is considered more logical.

- Fix **EXPLAIN** so it can show complex plans more accurately (Tom)

  References to subplan outputs are now always shown correctly, instead of using `?column`*`N`*`?` for complicated cases.

- Limit the amount of information reported when a user is dropped (Alvaro)

  Previously, dropping (or attempting to drop) a user who owned many objects could result in large `NOTICE` or `ERROR` messages listing all these objects; this caused problems for some client applications. The length of the message is now limited, although a full list is still sent to the server log.

E.66.3.9. Data Types

- Support for the SQL/XML standard, including new operators and an XML data type (Nikolay Samokhvalov, Pavel Stehule, Peter)
- Enumerated data types (ENUM) (Tom Dunstan)

  This feature provides convenient support for fields that have a small, fixed set of allowed values. An example of creating an `ENUM` type is `CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')`.
- Universally Unique Identifier (UUID) data type (Gevik Babakhani, Neil)

  This closely matches RFC 4122.
- Widen the MONEY data type to 64 bits (D'Arcy Cain)

  This greatly increases the range of supported MONEY values.
- Fix float4/float8 to handle `Infinity` and `NAN` (Not A Number) consistently (Bruce)

  The code formerly was not consistent about distinguishing `Infinity` from overflow conditions.
- Allow leading and trailing whitespace during input of boolean values (Neil)
- Prevent **COPY** from using digits and lowercase letters as delimiters (Tom)

E.66.3.10. Functions

- Add new regular expression functions `regexp_matches()`, `regexp_split_to_array()`, and `regexp_split_to_table()` (Jeremy Drake, Neil)

  These functions provide extraction of regular expression subexpressions and allow splitting a string using a POSIX regular expression.
- Add `lo_truncate()` for large object truncation (Kris Jurka)
- Implement `width_bucket()` for the float8 data type (Neil)
- Add `pg_stat_clear_snapshot()` to discard statistics snapshots collected during the current transaction (Tom)

  The first request for statistics in a transaction takes a statistics snapshot that does not change during the transaction. This function allows the snapshot to be discarded and a new snapshot loaded during the next statistics query. This is particularly useful for PL/pgSQL functions, which are confined to a single transaction.
- Add `isodow` option to `EXTRACT()` and `date_part()` (Bruce)

  This returns the day of the week, with Sunday as seven. (`dow` returns Sunday as zero.)
- Add `ID` (ISO day of week) and `IDDD` (ISO day of year) format codes for `to_char()`, `to_date()`, and `to_timestamp()` (Brendan Jurd)
- Make `to_timestamp()` and `to_date()` assume `TM` (trim) option for potentially variable-width fields (Bruce)

  This matches Oracle™'s behavior.
- Fix off-by-one conversion error in `to_date()`/`to_timestamp()` `D` (non-ISO day of week) fields (Bruce)
- Make `setseed()` return void, rather than a useless integer value (Neil)
- Add a hash function for NUMERIC (Neil)

  This allows hash indexes and hash-based plans to be used with NUMERIC columns.
- Improve efficiency of `LIKE`/`ILIKE`, especially for multi-byte character sets like UTF-8 (Andrew, Itagaki Takahiro)
- Make `currtid()` functions require `SELECT` privileges on the target table (Tom)
- Add several `txid_*()` functions to query active transaction IDs (Jan)

  This is useful for various replication solutions.

E.66.3.11. PL/pgSQL Server-Side Language

- Add scrollable cursor support, including directional control in **FETCH** (Pavel Stehule)
- Allow `IN` as an alternative to `FROM` in PL/pgSQL's **FETCH** statement, for consistency with the backend's **FETCH** command

(Pavel Stehule)

- Add **MOVE** to PL/pgSQL (Magnus, Pavel Stehule, Neil)
- Implement **RETURN QUERY** (Pavel Stehule, Neil)

  This adds convenient syntax for PL/pgSQL set-returning functions that want to return the result of a query. **RETURN QUERY** is easier and more efficient than a loop around **RETURN NEXT**.
- Allow function parameter names to be qualified with the function's name (Tom)

  For example, `myfunc.myvar`. This is particularly useful for specifying variables in a query where the variable name might match a column name.
- Make qualification of variables with block labels work properly (Tom)

  Formerly, outer-level block labels could unexpectedly interfere with recognition of inner-level record or row references.
- Tighten requirements for `FOR` loop `STEP` values (Tom)

  Prevent non-positive `STEP` values, and handle loop overflows.
- Improve accuracy when reporting syntax error locations (Tom)

E.66.3.12. Other Server-Side Languages

- Allow type-name arguments to PL/Perl `spi_prepare()` to be data type aliases in addition to names found in `pg_type` (Andrew)
- Allow type-name arguments to PL/Python `plpy.prepare()` to be data type aliases in addition to names found in `pg_type` (Andrew)
- Allow type-name arguments to PL/Tcl `spi_prepare` to be data type aliases in addition to names found in `pg_type` (Andrew)
- Enable PL/PythonU to compile on Python 2.5 (Marko Kreen)
- Support a true PL/Python boolean type in compatible Python versions (Python 2.3 and later) (Marko Kreen)
- Fix PL/Tcl problems with thread-enabled `libtcl` spawning multiple threads within the backend (Steve Marshall, Paul Bayer, Doug Knight)

  This caused all sorts of unpleasantness.

E.66.3.13. psql

- List disabled triggers separately in `\d` output (Brendan Jurd)
- In `\d` patterns, always match `$` literally (Tom)
- Show aggregate return types in `\da` output (Greg Sabino Mullane)
- Add the function's volatility status to the output of `\df+` (Neil)
- Add `\prompt` capability (Chad Wagner)
- Allow `\pset`, `\t`, and `\x` to specify `on` or `off`, rather than just toggling (Chad Wagner)
- Add `\sleep` capability (Jan)
- Enable `\timing` output for `\copy` (Andrew)
- Improve `\timing` resolution on Windows (Itagaki Takahiro)
- Flush `\o` output after each backslash command (Tom)
- Correctly detect and report errors while reading a `-f` input file (Peter)
- Remove `-u` option (this option has long been deprecated) (Tom)

E.66.3.14. pg_dump

- Add `--tablespaces-only` and `--roles-only` options to pg_dumpall (Dave Page)
- Add an output file option to pg_dumpall (Dave Page)

  This is primarily useful on Windows, where output redirection of child pg_dump processes does not work.
- Allow pg_dumpall to accept an initial-connection database name rather than the default `template1` (Dave Page)
- In `-n` and `-t` switches, always match `$` literally (Tom)
- Improve performance when a database has thousands of objects (Tom)
- Remove `-u` option (this option has long been deprecated) (Tom)

E.66.3.15. Other Client Applications

- In initdb, allow the location of the `pg_xlog` directory to be specified (Euler Taveira de Oliveira)
- Enable server core dump generation in pg_regress on supported operating systems (Andrew)
- Add a `-t` (timeout) parameter to pg_ctl (Bruce)

  This controls how long pg_ctl will wait when waiting for server startup or shutdown. Formerly the timeout was hard-wired as 60 seconds.
- Add a pg_ctl option to control generation of server core dumps (Andrew)
- Allow Control-C to cancel clusterdb, reindexdb, and vacuumdb (Itagaki Takahiro, Magnus)
- Suppress command tag output for createdb, createuser, dropdb, and dropuser (Peter)

  The `--quiet` option is ignored and will be removed in 8.4. Progress messages when acting on all databases now go to stdout instead of stderr because they are not actually errors.

E.66.3.16. libpq

- Interpret the `dbName` parameter of `PQsetdbLogin()` as a `conninfo` string if it contains an equals sign (Andrew)

  This allows use of `conninfo` strings in client programs that still use `PQsetdbLogin()`.
- Support a global SSL configuration file (Victor Wagner)
- Add environment variable `PGSSLKEY` to control SSL hardware keys (Victor Wagner)
- Add `lo_truncate()` for large object truncation (Kris Jurka)
- Add `PQconnectionNeedsPassword()` that returns true if the server required a password but none was supplied (Joe Conway, Tom)

  If this returns true after a failed connection attempt, a client application should prompt the user for a password. In the past applications have had to check for a specific error message string to decide whether a password is needed; that approach is now deprecated.
- Add `PQconnectionUsedPassword()` that returns true if the supplied password was actually used (Joe Conway, Tom)

  This is useful in some security contexts where it is important to know whether a user-supplied password is actually valid.

E.66.3.17. ecpg

- Use V3 frontend/backend protocol (Michael)

  This adds support for server-side prepared statements.
- Use native threads, instead of pthreads, on Windows (Magnus)
- Improve thread-safety of ecpglib (Itagaki Takahiro)
- Make the ecpg libraries export only necessary API symbols (Michael)

E.66.3.18. Windows Port

- Allow the whole PostgreSQL™ distribution to be compiled with Microsoft Visual C++™ (Magnus and others)

  This allows Windows-based developers to use familiar development and debugging tools. Windows executables made with Visual C++ might also have better stability and performance than those made with other tool sets. The client-only Visual C++ build scripts have been removed.
- Drastically reduce postmaster's memory usage when it has many child processes (Magnus)
- Allow regression tests to be started by an administrative user (Magnus)
- Add native shared memory implementation (Magnus)

E.66.3.19. Server Programming Interface (SPI)

- Add cursor-related functionality in SPI (Pavel Stehule)

  Allow access to the cursor-related planning options, and add **FETCH**/**MOVE** routines.
- Allow execution of cursor commands through `SPI_execute` (Tom)

  The macro `SPI_ERROR_CURSOR` still exists but will never be returned.
- SPI plan pointers are now declared as `SPIPlanPtr` instead of `void *` (Tom)

  This does not break application code, but switching is recommended to help catch simple programming mistakes.

E.66.3.20. Build Options

- Add configure option `--enable-profiling` to enable code profiling (works only with gcc) (Korry Douglas and Nikhil Sontakke)
- Add configure option `--with-system-tzdata` to use the operating system's time zone database (Peter)
- Fix PGXS so extensions can be built against PostgreSQL installations whose pg_config program does not appear first in the `PATH` (Tom)
- Support **gmake draft** when building the SGML documentation (Bruce)

  Unless `draft` is used, the documentation build will now be repeated if necessary to ensure the index is up-to-date.

E.66.3.21. Source Code

- Rename macro `DLLIMPORT` to `PGDLLIMPORT` to avoid conflicting with third party includes (like Tcl) that define `DLLIMPORT` (Magnus)
- Create « operator families » to improve planning of queries involving cross-data-type comparisons (Tom)
- Update GIN `extractQuery()` API to allow signalling that nothing can satisfy the query (Teodor)
- Move `NAMEDATALEN` definition from `postgres_ext.h` to `pg_config_manual.h` (Peter)
- Provide `strlcpy()` and `strlcat()` on all platforms, and replace error-prone uses of `strncpy()`, `strncat()`, etc (Peter)
- Create hooks to let an external plugin monitor (or even replace) the planner and create plans for hypothetical situations (Gurjeet Singh, Tom)
- Create a function variable `join_search_hook` to let plugins override the join search order portion of the planner (Julius Stroffek)
- Add `tas()` support for Renesas' M32R processor (Kazuhiro Inaoka)
- `quote_identifier()` and pg_dump no longer quote keywords that are unreserved according to the grammar (Tom)
- Change the on-disk representation of the NUMERIC data type so that the *`sign_dscale`* word comes before the weight (Tom)
- Use SYSV semaphores rather than POSIX on Darwin >= 6.0, i.e., OS X 10.2 and up (Chris Marcellino)
- Add acronym and NFS documentation sections (Bruce)
- "Postgres" is now documented as an accepted alias for "PostgreSQL" (Peter)

- Add documentation about preventing database server spoofing when the server is down (Bruce)

E.66.3.22. Contrib

- Move `contrib README` content into the main PostgreSQL™ documentation (Albert Cervera i Areny)
- Add `contrib/pageinspect` module for low-level page inspection (Simon, Heikki)
- Add `contrib/pg_standby` module for controlling warm standby operation (Simon)
- Add `contrib/uuid-ossp` module for generating UUID values using the OSSP UUID library (Peter)

  Use configure `--with-ossp-uuid` to activate. This takes advantage of the new UUID builtin type.
- Add `contrib/dict_int`, `contrib/dict_xsyn`, and `contrib/test_parser` modules to provide sample add-on text search dictionary templates and parsers (Sergey Karpov)
- Allow contrib/pgbench to set the fillfactor (Pavan Deolasee)
- Add timestamps to contrib/pgbench `-l` (Greg Smith)
- Add usage count statistics to `contrib/pgbuffercache` (Greg Smith)
- Add GIN support for `contrib/hstore` (Teodor)
- Add GIN support for `contrib/pg_trgm` (Guillaume Smet, Teodor)
- Update OS/X startup scripts in `contrib/start-scripts` (Mark Cotner, David Fetter)
- Restrict `pgrowlocks()` and `dblink_get_pkey()` to users who have `SELECT` privilege on the target table (Tom)
- Restrict `contrib/pgstattuple` functions to superusers (Tom)
- `contrib/xml2` is deprecated and planned for removal in 8.4 (Peter)

  The new XML support in core PostgreSQL supersedes this module.

E.67. Release 8.2.23

Release Date

2011-12-05

This release contains a variety of fixes from 8.2.22. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

This is expected to be the last PostgreSQL™ release in the 8.2.X series. Users are encouraged to update to a newer release branch soon.

E.67.1. Migration to Version 8.2.23

A dump/restore is not required for those running 8.2.X.

However, a longstanding error was discovered in the definition of the `information_schema.referential_constraints` view. If you rely on correct results from that view, you should replace its definition as explained in the first changelog item below.

Also, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.67.2. Changes

- Fix bugs in `information_schema.referential_constraints` view (Tom Lane)

  This view was being insufficiently careful about matching the foreign-key constraint to the depended-on primary or unique key constraint. That could result in failure to show a foreign key constraint at all, or showing it multiple times, or claiming that it depends on a different constraint than the one it really does.

  Since the view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can (as a superuser) drop the `information_schema` schema then re-create it by sourcing *`SHAREDIR/`*

`information_schema.sql`. (Run `pg_config --sharedir` if you're uncertain where *`SHAREDIR`* is.) This must be repeated in each database to be fixed.

- Fix TOAST-related data corruption during `CREATE TABLE dest AS SELECT * FROM src` or `INSERT INTO dest SELECT * FROM src` (Tom Lane)

  If a table has been modified by **ALTER TABLE ADD COLUMN**, attempts to copy its data verbatim to another table could produce corrupt results in certain corner cases. The problem can only manifest in this precise form in 8.4 and later, but we patched earlier versions as well in case there are other code paths that could trigger the same bug.

- Fix race condition during toast table access from stale syscache entries (Tom Lane)

  The typical symptom was transient errors like « missing chunk number 0 for toast value NNNNN in pg_toast_2619 », where the cited toast table would always belong to a system catalog.

- Improve locale support in money type's input and output (Tom Lane)

  Aside from not supporting all standard `lc_monetary` formatting options, the input and output functions were inconsistent, meaning there were locales in which dumped money values could not be re-read.

- Don't let `transform_null_equals` affect `CASE foo WHEN NULL ...` constructs (Heikki Linnakangas)

  `transform_null_equals` is only supposed to affect `foo = NULL` expressions written directly by the user, not equality checks generated internally by this form of `CASE`.

- Change foreign-key trigger creation order to better support self-referential foreign keys (Tom Lane)

  For a cascading foreign key that references its own table, a row update will fire both the `ON UPDATE` trigger and the `CHECK` trigger as one event. The `ON UPDATE` trigger must execute first, else the `CHECK` will check a non-final state of the row and possibly throw an inappropriate error. However, the firing order of these triggers is determined by their names, which generally sort in creation order since the triggers have auto-generated names following the convention « RI_ConstraintTrigger_NNNN ». A proper fix would require modifying that convention, which we will do in 9.2, but it seems risky to change it in existing releases. So this patch just changes the creation order of the triggers. Users encountering this type of error should drop and re-create the foreign key constraint to get its triggers into the right order.

- Preserve blank lines within commands in psql's command history (Robert Haas)

  The former behavior could cause problems if an empty line was removed from within a string literal, for example.

- Use the preferred version of xsubpp to build PL/Perl, not necessarily the operating system's main copy (David Wheeler and Alex Hunsaker)

- Honor query cancel interrupts promptly in `pgstatindex()` (Robert Haas)

- Ensure VPATH builds properly install all server header files (Peter Eisentraut)

- Shorten file names reported in verbose error messages (Peter Eisentraut)

  Regular builds have always reported just the name of the C file containing the error message call, but VPATH builds formerly reported an absolute path name.

- Fix interpretation of Windows timezone names for Central America (Tom Lane)

  Map « Central America Standard Time » to `CST6`, not `CST6CDT`, because DST is generally not observed anywhere in Central America.

- Update time zone data files to tzdata release 2011n for DST law changes in Brazil, Cuba, Fiji, Palestine, Russia, and Samoa; also historical corrections for Alaska and British East Africa.

E.68. Release 8.2.22

Release Date

2011-09-26

This release contains a variety of fixes from 8.2.21. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

The PostgreSQL™ community will stop releasing updates for the 8.2.X release series in December 2011. Users are encouraged to update to a newer release branch soon.

E.68.1. Migration to Version 8.2.22

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.68.2. Changes

- Fix multiple bugs in GiST index page split processing (Heikki Linnakangas)

  The probability of occurrence was low, but these could lead to index corruption.
- Avoid possibly accessing off the end of memory in **ANALYZE** (Noah Misch)

  This fixes a very-low-probability server crash scenario.
- Fix race condition in relcache init file invalidation (Tom Lane)

  There was a window wherein a new backend process could read a stale init file but miss the inval messages that would tell it the data is stale. The result would be bizarre failures in catalog accesses, typically « could not read block 0 in file ... » later during startup.
- Fix memory leak at end of a GiST index scan (Tom Lane)

  Commands that perform many separate GiST index scans, such as verification of a new GiST-based exclusion constraint on a table already containing many rows, could transiently require large amounts of memory due to this leak.
- Fix performance problem when constructing a large, lossy bitmap (Tom Lane)
- Fix array- and path-creating functions to ensure padding bytes are zeroes (Tom Lane)

  This avoids some situations where the planner will think that semantically-equal constants are not equal, resulting in poor optimization.
- Work around gcc 4.6.0 bug that breaks WAL replay (Tom Lane)

  This could lead to loss of committed transactions after a server crash.
- Fix dump bug for `VALUES` in a view (Tom Lane)
- Disallow `SELECT FOR UPDATE/SHARE` on sequences (Tom Lane)

  This operation doesn't work as expected and can lead to failures.
- Defend against integer overflow when computing size of a hash table (Tom Lane)
- Fix portability bugs in use of credentials control messages for « peer » authentication (Tom Lane)
- Fix typo in `pg_srand48` seed initialization (Andres Freund)

  This led to failure to use all bits of the provided seed. This function is not used on most platforms (only those without `srandom`), and the potential security exposure from a less-random-than-expected seed seems minimal in any case.
- Avoid integer overflow when the sum of `LIMIT` and `OFFSET` values exceeds 2^63 (Heikki Linnakangas)
- Add overflow checks to int4 and int8 versions of `generate_series()` (Robert Haas)
- Fix trailing-zero removal in `to_char()` (Marti Raudsepp)

  In a format with `FM` and no digit positions after the decimal point, zeroes to the left of the decimal point could be removed incorrectly.
- Fix `pg_size_pretty()` to avoid overflow for inputs close to 2^63 (Tom Lane)
- Fix psql's counting of script file line numbers during `COPY` from a different file (Tom Lane)
- Fix pg_restore's direct-to-database mode for `standard_conforming_strings` (Tom Lane)

  pg_restore could emit incorrect commands when restoring directly to a database server from an archive file that had been made with `standard_conforming_strings` set to `on`.
- Fix write-past-buffer-end and memory leak in libpq's LDAP service lookup code (Albe Laurenz)
- In libpq, avoid failures when using nonblocking I/O and an SSL connection (Martin Pihlak, Tom Lane)
- Improve libpq's handling of failures during connection startup (Tom Lane)

  In particular, the response to a server report of `fork()` failure during SSL connection startup is now saner.

- Make ecpglib write double values with 15 digits precision (Akira Kurosawa)
- Apply upstream fix for blowfish signed-character bug (CVE-2011-2483) (Tom Lane)

  `contrib/pg_crypto`'s blowfish encryption code could give wrong results on platforms where char is signed (which is most), leading to encrypted passwords being weaker than they should be.
- Fix memory leak in `contrib/seg` (Heikki Linnakangas)
- Fix `pgstatindex()` to give consistent results for empty indexes (Tom Lane)
- Allow building with perl 5.14 (Alex Hunsaker)
- Update configure script's method for probing existence of system functions (Tom Lane)

  The version of autoconf we used in 8.3 and 8.2 could be fooled by compilers that perform link-time optimization.
- Fix assorted issues with build and install file paths containing spaces (Tom Lane)
- Update time zone data files to tzdata release 2011i for DST law changes in Canada, Egypt, Russia, Samoa, and South Sudan.

E.69. Release 8.2.21

Release Date

2011-04-18

This release contains a variety of fixes from 8.2.20. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.69.1. Migration to Version 8.2.21

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.69.2. Changes

- Avoid potential deadlock during catalog cache initialization (Nikhil Sontakke)

  In some cases the cache loading code would acquire share lock on a system index before locking the index's catalog. This could deadlock against processes trying to acquire exclusive locks in the other, more standard order.
- Fix dangling-pointer problem in `BEFORE ROW UPDATE` trigger handling when there was a concurrent update to the target tuple (Tom Lane)

  This bug has been observed to result in intermittent « cannot extract system attribute from virtual tuple » failures while trying to do `UPDATE RETURNING ctid`. There is a very small probability of more serious errors, such as generating incorrect index entries for the updated tuple.
- Disallow **DROP TABLE** when there are pending deferred trigger events for the table (Tom Lane)

  Formerly the **DROP** would go through, leading to « could not open relation with OID nnn » errors when the triggers were eventually fired.
- Fix PL/Python memory leak involving array slices (Daniel Popowich)
- Fix pg_restore to cope with long lines (over 1KB) in TOC files (Tom Lane)
- Put in more safeguards against crashing due to division-by-zero with overly enthusiastic compiler optimization (Aurelien Jarno)
- Support use of dlopen() in FreeBSD and OpenBSD on MIPS (Tom Lane)

  There was a hard-wired assumption that this system function was not available on MIPS hardware on these systems. Use a compile-time test instead, since more recent versions have it.
- Fix compilation failures on HP-UX (Heikki Linnakangas)
- Fix path separator used by pg_regress on Cygwin (Andrew Dunstan)
- Update time zone data files to tzdata release 2011f for DST law changes in Chile, Cuba, Falkland Islands, Morocco, Samoa,

and Turkey; also historical corrections for South Australia, Alaska, and Hawaii.

E.70. Release 8.2.20

Release Date

2011-01-31

This release contains a variety of fixes from 8.2.19. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.70.1. Migration to Version 8.2.20

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.70.2. Changes

- Avoid failures when **EXPLAIN** tries to display a simple-form `CASE` expression (Tom Lane)

  If the `CASE`'s test expression was a constant, the planner could simplify the `CASE` into a form that confused the expression-display code, resulting in « unexpected CASE WHEN clause » errors.

- Fix assignment to an array slice that is before the existing range of subscripts (Tom Lane)

  If there was a gap between the newly added subscripts and the first pre-existing subscript, the code miscalculated how many entries needed to be copied from the old array's null bitmap, potentially leading to data corruption or crash.

- Avoid unexpected conversion overflow in planner for very distant date values (Tom Lane)

  The date type supports a wider range of dates than can be represented by the timestamp types, but the planner assumed it could always convert a date to timestamp with impunity.

- Fix pg_restore's text output for large objects (BLOBs) when `standard_conforming_strings` is on (Tom Lane)

  Although restoring directly to a database worked correctly, string escaping was incorrect if pg_restore was asked for SQL text output and `standard_conforming_strings` had been enabled in the source database.

- Fix erroneous parsing of tsquery values containing `... & !(subexpression) | ...` (Tom Lane)

  Queries containing this combination of operators were not executed correctly. The same error existed in `contrib/intarray`'s query_int type and `contrib/ltree`'s ltxtquery type.

- Fix buffer overrun in `contrib/intarray`'s input function for the query_int type (Apple)

  This bug is a security risk since the function's return address could be overwritten. Thanks to Apple Inc's security team for reporting this issue and supplying the fix. (CVE-2010-4015)

- Fix bug in `contrib/seg`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a seg column. If you have such an index, consider **REINDEX**ing it after installing this update. (This is identical to the bug that was fixed in `contrib/cube` in the previous update.)

E.71. Release 8.2.19

Release Date

2010-12-16

This release contains a variety of fixes from 8.2.18. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.71.1. Migration to Version 8.2.19

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.71.2. Changes

- Force the default `wal_sync_method` to be `fdatasync` on Linux (Tom Lane, Marti Raudsepp)

  The default on Linux has actually been `fdatasync` for many years, but recent kernel changes caused PostgreSQL™ to choose `open_datasync` instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably `ext4` with the `data=journal` mount option.

- Fix assorted bugs in WAL replay logic for GIN indexes (Tom Lane)

  This could result in « bad buffer id: 0 » failures or corruption of index contents during replication.

- Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)

- Add support for detecting register-stack overrun on `IA64` (Tom Lane)

  The `IA64` architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.

- Add a check for stack overflow in `copyObject()` (Tom Lane)

  Certain code paths could crash due to stack overflow given a sufficiently complex query.

- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)

  It is possible to have a « concurrent » page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.

- Avoid memory leakage while **ANALYZE**'ing complex index expressions (Tom Lane)

- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)

  An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.

- Do not « inline » a SQL function with multiple `OUT` parameters (Tom Lane)

  This avoids a possible crash due to loss of information about the expected result rowtype.

- Behave correctly if `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `WITH` is attached to the `VALUES` part of `INSERT ... VALUES` (Tom Lane)

- Fix constant-folding of `COALESCE()` expressions (Tom Lane)

  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.

- Add print functionality for InhRelation nodes (Tom Lane)

  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.

- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)

  This bug affected several different geometric distance-measurement operators.

- Fix PL/pgSQL's handling of « simple » expressions to not fail in recursion or error-recovery cases (Tom Lane)

- Fix PL/Python's handling of set-returning functions (Jan Urbanski)

  Attempts to call SPI functions within the iterator generating a set result would fail.

- Fix bug in `contrib/cube`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a cube column. If you have such an index, consider **REINDEX**ing it after installing this update.

- Don't emit « identifier will be truncated » notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)

- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)

- Fix memory leak in `contrib/xml2`'s XPath query functions (Tom Lane)

- Update time zone data files to tzdata release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.

E.72. Release 8.2.18

Release Date

2010-10-04

This release contains a variety of fixes from 8.2.17. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.72.1. Migration to Version 8.2.18

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.72.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).
- Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it's intended to be used with (Heikki Linnakangas, Tom Lane)
- Fix Windows shared-memory allocation code (Tsutomu Yamada, Magnus Hagander)

  This bug led to the often-reported « could not reattach to shared memory » error message. This is a back-patch of a fix that was applied to newer branches some time ago.
- Treat exit code 128 (`ERROR_WAIT_NO_CHILDREN`) as non-fatal on Windows (Magnus Hagander)

  Under high load, Windows processes will sometimes fail at startup with this error code. Formerly the postmaster treated this as a panic condition and restarted the whole database, but that seems to be an overreaction.
- Fix possible duplicate scans of `UNION ALL` member relations (Tom Lane)
- Fix « cannot handle unplanned sub-select » error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.
- Reduce PANIC to ERROR in some occasionally-reported btree failure cases, and provide additional detail in the resulting error messages (Tom Lane)

  This should improve the system's robustness with corrupted indexes.
- Prevent show_session_authorization() from crashing within autovacuum processes (Tom Lane)
- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)
- Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)
- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in

turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.

- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.

- Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)
- Fix possible data corruption in **ALTER TABLE ... SET TABLESPACE** when archiving is enabled (Jeff Davis)
- Allow **CREATE DATABASE** and **ALTER DATABASE ... SET TABLESPACE** to be interrupted by query-cancel (Guillaume Lelarge)
- In PL/Python, defend against null pointer results from `PyCObject_AsVoidPtr` and `PyCObject_FromVoidPtr` (Peter Eisentraut)
- Improve `contrib/dblink`'s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after « duplicate connection name » errors in `contrib/dblink` (Itagaki Takahiro)
- Fix `contrib/dblink` to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
- Add `hstore(text, text)` function to `contrib/hstore` (Robert Haas)

  This function is the recommended substitute for the now-deprecated `=>` operator. It was back-patched so that future-proofed code can be used with older server versions. Note that the patch will be effective only after `contrib/hstore` is installed or reinstalled in a particular database. Users might prefer to execute the **CREATE FUNCTION** command by hand, instead.

- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)
- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.

- Make Windows' « N. Central Asia Standard Time » timezone map to Asia/Novosibirsk, not Asia/Almaty (Magnus Hagander)

  Microsoft changed the DST behavior of this zone in the timezone update from KB976098. Asia/Novosibirsk is a better match to its new behavior.

E.73. Release 8.2.17

Release Date

2010-05-17

This release contains a variety of fixes from 8.2.16. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.73.1. Migration to Version 8.2.17

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.73.2. Changes

- Enforce restrictions in `plperl` using an opmask applied to the whole interpreter, instead of using `Safe.pm` (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that `Safe.pm` is too insecure to rely on for making `plperl` trustable. This change removes use of `Safe.pm` altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's `strict` pragma in a natural way in `plperl`, and that Perl's `$a` and `$b` variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted « normal » Tcl interpreter unless we are really going to execute a `pltclu` function. (CVE-2010-1170)

- Fix possible crash if a cache reset message is received during rebuild of a relcache entry (Heikki)

  This error was introduced in 8.2.16 while fixing a related failure.

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran `ALTER USER ... RESET ALL` for himself, or `ALTER DATABASE ... RESET ALL` for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the **ALTER** will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a `CONTEXT` addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Update pl/perl's `ppport.h` for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)
- Fix psql's `\copy` to not add spaces around a dot within `\copy (select ...)` (Tom)

  Addition of spaces around the decimal point in a numeric literal would result in a syntax error.

- Ensure that `contrib/pgstattuple` functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that `shmget()` returns `EINVAL` for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

- Avoid possible crashes in syslogger process on Windows (Heikki)
- Deal more robustly with incomplete time zone information in the Windows registry (Magnus)
- Update the set of known Windows time zone names (Magnus)
- Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

  Also, add `PKST` (Pakistan Summer Time) to the default set of timezone abbreviations.

E.74. Release 8.2.16

Release Date

2010-03-15

This release contains a variety of fixes from 8.2.15. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.74.1. Migration to Version 8.2.16

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.74.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for

an SSL connection (Magnus)

This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

- Fix possible deadlock during backend startup (Tom)
- Fix possible crashes due to not handling errors during relcache reload cleanly (Tom)
- Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)
- Fix server memory leak associated with use of savepoints and a client encoding different from server's encoding (Tom)
- Fix incorrect WAL data emitted during end-of-recovery cleanup of a GIST index page split (Yoichi Hirai)

  This would result in index corruption, or even more likely an error during WAL replay, if we were unlucky enough to crash during end-of-recovery cleanup after having completed an incomplete GIST insertion.
- Make `substring()` for bit types treat any negative length as meaning « all the rest of the string » (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix the `STOP WAL LOCATION` entry in backup history files to report the next WAL segment's name when the end location is exactly at a segment boundary (Itagaki Takahiro)
- Fix some more cases of temporary-file leakage (Heikki)

  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function's exception handler.
- Improve constraint exclusion processing of boolean-variable cases, in particular make it possible to exclude a partition that has a « bool_column = false » constraint (Tom)
- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.
- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)
- Fix possible infinite loop if `SSL_read` or `SSL_write` fails without setting `errno` (Tom)

  This is reportedly possible with some Windows versions of openssl.
- Fix psql's `numericlocale` option to not format strings it shouldn't in latex and troff output formats (Heikki)
- Make psql return the correct exit status (3) when `ON_ERROR_STOP` and `--single-transaction` are both specified and an error occurs during the implied **COMMIT** (Bruce)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Fix possible failure when calling PL/Perl functions from PL/PerlU or vice versa (Tim Bunce)
- Add `volatile` markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.
- Prevent crash in `contrib/dblink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)
- Fix assorted crashes in `contrib/xml2` caused by sloppy memory management (Tom)
- Make building of `contrib/xml2` more robust on Windows (Andrew)
- Fix race condition in Windows signal handling (Radu Ilie)

One known symptom of this bug is that rows in pg_listener could be dropped under heavy load.

- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.75. Release 8.2.15

Release Date

2009-12-14

This release contains a variety of fixes from 8.2.14. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.75.1. Migration to Version 8.2.15

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see Section E.76, « Release 8.2.14 ».

E.75.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

  This change prevents allegedly-immutable index functions from possibly subverting a superuser's session (CVE-2009-4136).

- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).

- Fix possible crash during backend-startup-time cache initialization (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)

  This fix prevents a PANIC if a `VACUUM FULL` is canceled after it's already committed its tuple movements, as well as transient errors if a plain `VACUUM` is interrupted after having truncated the table.

- Fix possible crash due to integer overflow in hash table size calculation (Tom)

  This could occur with extremely large planner estimates for the size of a hashjoin's result.

- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix incorrect logic for GiST index page splits, when the split depends on a non-first column of the index (Paul Ramsey)
- Don't error out if recycling or removing an old WAL file fails at the end of checkpoint (Heikki)

  It's better to treat the problem as non-fatal and allow the checkpoint to complete. Future checkpoints will retry the removal. Such problems are not expected in normal operation, but have been seen to be caused by misdesigned Windows anti-virus and backup software.

- Ensure WAL files aren't repeatedly archived on Windows (Heikki)

  This is another symptom that could happen if some other process interfered with deletion of a no-longer-needed file.

- Fix PAM password processing to be more robust (Tom)

  The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.

- Fix processing of ownership dependencies during `CREATE OR REPLACE FUNCTION` (Tom)
- Fix bug with calling `plperl` from `plperlu` or vice versa (Tom)

  An error exit from the inner function could result in crashes due to failure to re-select the correct Perl interpreter for the outer function.

- Fix session-lifespan memory leak when a PL/Perl function is redefined (Tom)
- Ensure that Perl arrays are properly converted to PostgreSQL™ arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)

  This worked correctly already for non-set-returning functions.
- Fix rare crash in exception processing in PL/Python (Peter)
- Ensure psql's flex module is compiled with the correct system header definitions (Tom)

  This fixes build failures on platforms where `--enable-largefile` causes incompatible changes in the generated code.
- Make the postmaster ignore any `application_name` parameter in connection request packets, to improve compatibility with future libpq versions (Tom)
- Update the timezone abbreviation files to match current reality (Joachim Wieland)

  This includes adding `IDT` and `SGT` to the default timezone abbreviation set.
- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.

E.76. Release 8.2.14

Release Date

2009-09-09

This release contains a variety of fixes from 8.2.13. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.76.1. Migration to Version 8.2.14

A dump/restore is not required for those running 8.2.X. However, if you have any hash indexes on interval columns, you must **REINDEX** them after updating to 8.2.14. Also, if you are upgrading from a version earlier than 8.2.11, see Section E.79, « Release 8.2.11 ».

E.76.2. Changes

- Force WAL segment switch during `pg_start_backup()` (Heikki)

  This avoids corner cases that could render a base backup unusable.
- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)
- Make **LOAD** of an already-loaded loadable module into a no-op (Tom)

  Formerly, **LOAD** would attempt to unload and re-load the module, but this is unsafe and not all that useful.
- Disallow empty passwords during LDAP authentication (Magnus)
- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
- Fix bugs associated with fetching a whole-row value from the output of a Sort or Materialize plan node (Tom)
- Revert planner change that disabled partial-index and constraint exclusion optimizations when there were more than 100 clauses in an AND or OR list (Tom)
- Fix hash calculation for data type interval (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must **REINDEX** them after updating.
- Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with `'HH'`/`'HH12'` (Heikki)

  It was previously handled as `'th'` (lowercase).

- Fix overflow for `INTERVAL 'x ms'` when *x* is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.
- Fix money data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)
- Properly round datetime input like `00:12:57.999999999999999999999999999` (Tom)
- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)
- Avoid performance degradation in bulk inserts into GIN indexes when the input values are (nearly) in sorted order (Tom)
- Correctly enforce NOT NULL domain constraints in some contexts in PL/pgSQL (Tom)
- Fix portability issues in plperl initialization (Andrew Dunstan)
- Fix pg_ctl to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)
- Make `contrib/hstore` throw an error when a key or value is too long to fit in its data structure, rather than silently truncating it (Andrew Gierth)
- Fix `contrib/xml2`'s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)
- Improve robustness of libpq's code to recover from errors during **COPY FROM STDIN** (Tom)
- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)
- Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.77. Release 8.2.13

Release Date

2009-03-16

This release contains a variety of fixes from 8.2.12. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.77.1. Migration to Version 8.2.13

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.11, see Section E.79, « Release 8.2.11 ».

E.77.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)

  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that *any* error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)
- Disallow **CREATE CONVERSION** with the wrong encodings for the specified conversion function (Heikki)

  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.
- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)
- Fix possible failure in `contrib/tsearch2` when C locale is used with a multi-byte encoding (Teodor)

  Crashes were possible on platforms where wchar_t is narrower than int; Windows in particular.
- Fix extreme inefficiency in `contrib/tsearch2` parser's handling of an email-like string containing multiple `@` characters (Heikki)
- Fix decompilation of `CASE WHEN` with an implicit coercion (Tom)

This mistake could lead to Assert failures in an Assert-enabled build, or an « unexpected CASE WHEN clause » error message in other cases, when trying to examine or dump a view.

- Fix possible misassignment of the owner of a TOAST table's rowtype (Tom)

  If **CLUSTER** or a rewriting variant of **ALTER TABLE** were executed by someone other than the table owner, the pg_type entry for the table's TOAST table would end up marked as owned by that someone. This caused no immediate problems, since the permissions on the TOAST rowtype aren't examined by any ordinary database operation. However, it could lead to unexpected failures if one later tried to drop the role that issued the command (in 8.1 or 8.2), or « owner of data type appears to be invalid » warnings from pg_dump after having done so (in 8.3).

- Fix PL/pgSQL to not treat `INTO` after **INSERT** as an INTO-variables clause anywhere in the string, not only at the start; in particular, don't fail for **INSERT INTO** within **CREATE RULE** (Tom)

- Clean up PL/pgSQL error status variables fully at block exit (Ashesh Vashi and Dave Page)

  This is not a problem for PL/pgSQL itself, but the omission could cause the PL/pgSQL Debugger to crash while examining the state of a function.

- Retry failed calls to `CallNamedPipe()` on Windows (Steve Marshall, Magnus)

  It appears that this function can sometimes fail transiently; we previously treated any failure as a hard error, which could confuse **LISTEN**/**NOTIFY** as well as other operations.

- Add `MUST` (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

E.78. Release 8.2.12

Release Date

2009-02-02

This release contains a variety of fixes from 8.2.11. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.78.1. Migration to Version 8.2.12

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.11, see Section E.79, « Release 8.2.11 ».

E.78.2. Changes

- Improve handling of URLs in `headline()` function (Teodor)
- Improve handling of overlength headlines in `headline()` function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Fix possible Assert failure if a statement executed in PL/pgSQL is rewritten into another kind of statement, for example if an **INSERT** is rewritten into an **UPDATE** (Heikki)
- Ensure that a snapshot is available to datatype input functions (Tom)

  This primarily affects domains that are declared with `CHECK` constraints involving user-defined stable or immutable functions. Such functions typically fail if no snapshot has been set.

- Make it safer for SPI-using functions to be used within datatype I/O; in particular, to be used in domain check constraints (Tom)
- Avoid unnecessary locking of small tables in **VACUUM** (Heikki)
- Fix a problem that made `UPDATE RETURNING tableoid` return zero instead of the correct OID (Tom)
- Fix planner misestimation of selectivity when transitive equality is applied to an outer-join clause (Tom)

  This could result in bad plans for queries like `... from a left join b on a.a1 = b.b1 where a.a1 = 42 ...`

- Improve optimizer's handling of long `IN` lists (Tom)

  This change avoids wasting large amounts of time on such lists when constraint exclusion is enabled.
- Ensure that the contents of a holdable cursor don't depend on the contents of TOAST tables (Tom)

  Previously, large field values in a cursor result might be represented as TOAST pointers, which would fail if the referenced table got dropped before the cursor is read, or if the large value is deleted and then vacuumed away. This cannot happen with an ordinary cursor, but it could with a cursor that is held past its creating transaction.
- Fix memory leak when a set-returning function is terminated without reading its whole result (Tom)
- Fix `contrib/dblink`'s `dblink_get_result(text,bool)` function (Joe)
- Fix possible garbage output from `contrib/sslinfo` functions (Tom)
- Fix configure script to properly report failure when unable to obtain linkage information for PL/Perl (Andrew)
- Make all documentation reference `pgsql-bugs` and/or `pgsql-hackers` as appropriate, instead of the now-decommissioned `pgsql-ports` and `pgsql-patches` mailing lists (Tom)
- Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections in Switzerland, Cuba)

E.79. Release 8.2.11

Release Date

2008-11-03

This release contains a variety of fixes from 8.2.10. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.79.1. Migration to Version 8.2.11

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see Section E.83, « Release 8.2.7 ». Also, if you were running a previous 8.2.X release, it is recommended to **REINDEX** all GiST indexes after the upgrade.

E.79.2. Changes

- Fix GiST index corruption due to marking the wrong index entry « dead » after a deletion (Teodor)

  This would result in index searches failing to find rows they should have found. Corrupted indexes can be fixed with **REINDEX**.
- Fix backend crash when the client encoding cannot represent a localized error message (Tom)

  We have addressed similar issues before, but it would still fail if the « character has no equivalent » message itself couldn't be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Improve optimization of *`expression`* `IN` (*`expression-list`*) queries (Tom, per an idea from Robert Haas)

  Cases in which there are query variables on the right-hand side had been handled less efficiently in 8.2.x and 8.3.x than in prior versions. The fix restores 8.1 behavior for such cases.
- Fix mis-expansion of rule queries when a sub-`SELECT` appears in a function call in `FROM`, a multi-row `VALUES` list, or a `RETURNING` list (Tom)

  The usual symptom of this problem is an « unrecognized node type » error.
- Fix memory leak during rescan of a hashed aggregation plan (Neil)
- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Prevent possible collision of *`relfilenode`* numbers when moving a table to another tablespace with **ALTER SET TABLESPACE** (Heikki)

  The command tried to re-use the existing filename, instead of picking one that is known unused in the destination directory.

- Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an `--enable-integer-datetimes` build (Ron Mayer)
- Ensure `SPI_getvalue` and `SPI_getbinval` behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)

  This situation is normal when a table has had columns added or removed, but these two functions didn't handle it properly. The only likely consequence is an incorrect error indication.
- Fix ecpg's parsing of **CREATE ROLE** (Michael)
- Fix recent breakage of `pg_ctl restart` (Tom)
- Ensure `pg_control` is opened in binary mode (Itagaki Takahiro)

  pg_controldata and pg_resetxlog did this incorrectly, and so could fail on Windows.
- Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil, Mauritius, Syria)

E.80. Release 8.2.10

Release Date

2008-09-22

This release contains a variety of fixes from 8.2.9. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.80.1. Migration to Version 8.2.10

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see Section E.83, « Release 8.2.7 ».

E.80.2. Changes

- Fix bug in btree WAL recovery code (Heikki)

  Recovery failed if the WAL ended partway through a page split operation.
- Fix potential miscalculation of *`datfrozenxid`* (Alvaro)

  This error may explain some recent reports of failure to remove old pg_clog data.
- Widen local lock counters from 32 to 64 bits (Tom)

  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected « lock is already held » errors.
- Fix possible duplicate output of tuples during a GiST index scan (Teodor)
- Fix missed permissions checks when a view contains a simple `UNION ALL` construct (Heikki)

  Permissions for the referenced tables were checked properly, but not permissions for the view itself.
- Add checks in executor startup to ensure that the tuples produced by an **INSERT** or **UPDATE** will match the target table's current rowtype (Tom)

  ALTER COLUMN TYPE, followed by re-use of a previously cached plan, could produce this type of situation. The check protects against data corruption and/or crashes that could ensue.
- Fix possible repeated drops during **DROP OWNED** (Tom)

  This would typically result in strange errors such as « cache lookup failed for relation NNN ».
- Fix `AT TIME ZONE` to first try to interpret its timezone argument as a timezone abbreviation, and only try it as a full timezone name if that fails, rather than the other way around as formerly (Tom)

  The timestamp input functions have always resolved ambiguous zone names in this order. Making `AT TIME ZONE` do so as well improves consistency, and fixes a compatibility bug introduced in 8.1: in ambiguous cases we now behave the same as

8.0 and before did, since in the older versions `AT TIME ZONE` accepted *only* abbreviations.

- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Prevent integer overflows during units conversion when displaying a configuration parameter that has units (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Allow spaces in the suffix part of an LDAP URL in `pg_hba.conf` (Tom)
- Fix bug in backwards scanning of a cursor on a `SELECT DISTINCT ON` query (Tom)
- Fix planner bug with nested sub-select expressions (Tom)

  If the outer sub-select has no direct dependency on the parent query, but the inner one does, the outer value might not get re-calculated for new parent query rows.
- Fix planner to estimate that `GROUP BY` expressions yielding boolean results always result in two groups, regardless of the expressions' contents (Tom)

  This is very substantially more accurate than the regular `GROUP BY` estimate for certain boolean tests like *`col`* `IS NULL`.
- Fix PL/pgSQL to not fail when a `FOR` loop's target variable is a record containing composite-type fields (Tom)
- Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)
- On Windows, work around a Microsoft bug by preventing libpq from trying to send more than 64kB per system call (Magnus)
- Improve pg_dump and pg_restore's error reporting after failure to send a SQL command (Tom)
- Fix pg_ctl to properly preserve postmaster command-line arguments across a `restart` (Bruce)
- Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.81. Release 8.2.9

Release Date

2008-06-12

This release contains one serious and one minor bug fix over 8.2.8. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.81.1. Migration to Version 8.2.9

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see Section E.83, « Release 8.2.7 ».

E.81.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)

  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL™ to reject what had been a valid **SELECT DISTINCT** view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.
- Make **ALTER AGGREGATE ... OWNER TO** update pg_shdepend (Tom)

  This oversight could lead to problems if the aggregate was later involved in a **DROP OWNED** or **REASSIGN OWNED** operation.

E.82. Release 8.2.8

Release Date

never released

This release contains a variety of fixes from 8.2.7. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.82.1. Migration to Version 8.2.8

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see Section E.83, « Release 8.2.7 ».

E.82.2. Changes

- Fix `ERRORDATA_STACK_SIZE exceeded` crash that occurred on Windows when using UTF-8 database encoding and a different client encoding (Tom)
- Fix **ALTER TABLE ADD COLUMN ... PRIMARY KEY** so that the new column is correctly checked to see if it's been initialized to all non-nulls (Brendan Jurd)

  Previous versions neglected to check this requirement at all.
- Fix possible **CREATE TABLE** failure when alling the « same » constraint from multiple parent relations that alled that constraint from a common ancestor (Tom)
- Fix `pg_get_ruledef()` to show the alias, if any, attached to the target table of an **UPDATE** or **DELETE** (Tom)
- Fix GIN bug that could result in a `too many LWLocks taken` failure (Teodor)
- Avoid possible crash when decompressing corrupted data (Zdenek Kotala)
- Repair two places where SIGTERM exit of a backend could leave corrupted state in shared memory (Tom)

  Neither case is very important if SIGTERM is used to shut down the whole database cluster together, but there was a problem if someone tried to SIGTERM individual backends.
- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic « Yo » characters (`e` and `E` with two dots) (Sergey Burladyan)
- Fix several datatype input functions, notably `array_in()`, that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched `ORDER BY` and `DISTINCT` expressions.
- Fix a corner case in regular-expression substring matching (`substring(`*`string`*` from `*`pattern`*`)`) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn't got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since `(bar)` isn't matched, but it was mistakenly returning the whole-pattern match instead (ie, `foo`).
- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, and Argentina/San_Luis)
- Fix incorrect result from ecpg's `PGTYPEStimestamp_sub()` function (Michael)
- Fix broken GiST comparison function for `contrib/tsearch2`'s tsquery type (Teodor)
- Fix possible crashes in `contrib/cube` functions (Tom)
- Fix core dump in `contrib/xml2`'s `xpath_table()` function when the input query returns a NULL value (Tom)
- Fix `contrib/xml2`'s makefile to not override `CFLAGS` (Tom)
- Fix `DatumGetBool` macro to not fail with gcc 4.3 (Tom)

  This problem affects « old style » (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.

E.83. Release 8.2.7

Release Date

2008-03-17

This release contains a variety of fixes from 8.2.6. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.83.1. Migration to Version 8.2.7

A dump/restore is not required for those running 8.2.X. However, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the Windows locale issue described below.

E.83.2. Changes

- Fix character string comparison for Windows locales that consider different character combinations as equal (Tom)

  This fix applies only on Windows and only when using UTF-8 database encoding. The same fix was made for all other cases over two years ago, but Windows with UTF-8 uses a separate code path that was not updated. If you are using a locale that considers some non-identical strings as equal, you may need to **REINDEX** to fix existing indexes on textual columns.

- Repair potential deadlock between concurrent **VACUUM FULL** operations on different system catalogs (Tom)

- Fix longstanding **LISTEN**/**NOTIFY** race condition (Tom)

  In rare cases a session that had just executed a **LISTEN** might not get a notification, even though one would be expected because the concurrent transaction executing **NOTIFY** was observed to commit later.

  A side effect of the fix is that a transaction that has executed a not-yet-committed **LISTEN** command will not see any row in pg_listener for the **LISTEN**, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Disallow **LISTEN** and **UNLISTEN** within a prepared transaction (Tom)

  This was formerly allowed but trying to do it had various unpleasant consequences, notably that the originating backend could not exit as long as an **UNLISTEN** remained uncommitted.

- Disallow dropping a temporary table within a prepared transaction (Heikki)

  This was correctly disallowed by 8.1, but the check was inadvertently broken in 8.2.

- Fix rare crash when an error occurs during a query using a hash index (Heikki)

- Fix memory leaks in certain usages of set-returning functions (Neil)

- Fix input of datetime values for February 29 in years BC (Tom)

  The former coding was mistaken about which years were leap years.

- Fix « unrecognized node type » error in some variants of **ALTER OWNER** (Tom)

- Ensure pg_stat_activity.*`waiting`* flag is cleared when a lock wait is aborted (Tom)

- Fix handling of process permissions on Windows Vista (Dave, Magnus)

  In particular, this fix allows starting the server as the Administrator user.

- Update time zone data files to tzdata release 2008a (in particular, recent Chile changes); adjust timezone abbreviation `VET` (Venezuela) to mean UTC-4:30, not UTC-4:00 (Tom)

- Fix pg_ctl to correctly extract the postmaster's port number from command-line options (Itagaki Takahiro, Tom)

  Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.

- Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)

  This is known to be necessary when building PostgreSQL™ with gcc 4.3 or later.

- Correctly enforce `statement_timeout` values longer than `INT_MAX` microseconds (about 35 minutes) (Tom)

  This bug affects only builds with `--enable-integer-datetimes`.

- Fix « unexpected PARAM_SUBLINK ID » planner error when constant-folding simplifies a sub-select (Tom)

- Fix logical errors in constraint-exclusion handling of `IS NULL` and `NOT` expressions (Tom)

  The planner would sometimes exclude partitions that should not have been excluded because of the possibility of NULL results.

- Fix another cause of « failed to build any N-way joins » planner errors (Tom)

  This could happen in cases where a clauseless join needed to be forced before a join clause could be exploited.

- Fix incorrect constant propagation in outer-join planning (Tom)

  The planner could sometimes incorrectly conclude that a variable could be constrained to be equal to a constant, leading to wrong query results.

- Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)

  An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.

- Fix libpq to handle NOTICE messages correctly during COPY OUT (Tom)

  This failure has only been observed to occur when a user-defined datatype's output routine issues a NOTICE, but there is no guarantee it couldn't happen due to other causes.

E.84. Release 8.2.6

Release Date

2008-01-07

This release contains a variety of fixes from 8.2.5, including fixes for significant security issues. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.84.1. Migration to Version 8.2.6

A dump/restore is not required for those running 8.2.X.

E.84.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running **VACUUM**, **ANALYZE**, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as **VACUUM FULL**, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including **VACUUM**, **ANALYZE**, **REINDEX**, and **CLUSTER**) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of **SET SESSION AUTHORIZATION** and **SET ROLE** is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 8.2.5 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Fix bugs in WAL replay for GIN indexes (Teodor)

- Fix GIN index build to work properly when `maintenance_work_mem` is 4GB or more (Tom)

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)

- Improve planner's handling of LIKE/regex estimation in non-C locales (Tom)
- Fix planning-speed problem for deep outer-join nests, as well as possible poor choice of join order (Tom)
- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)
- Make **CREATE TABLE ... SERIAL** and **ALTER SEQUENCE ... OWNED BY** not change the `currval()` state of the sequence (Tom)
- Preserve the tablespace and storage parameters of indexes that are rebuilt by **ALTER TABLE ... ALTER COLUMN TYPE** (Tom)
- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.
- Make **VACUUM** not use all of `maintenance_work_mem` when the table is too small for it to be useful (Alvaro)
- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)
- Make `corr()` return the correct result for negative correlation values (Neil)
- Fix overflow in `extract(epoch from interval)` for intervals exceeding 68 years (Tom)
- Fix PL/Perl to not fail when a UTF-8 regular expression is used in a trusted function (Andrew)
- Fix PL/Perl to cope when platform's Perl defines type `bool` as `int` rather than `char` (Tom)

  While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X™ 10.5.
- Fix PL/Python to work correctly with Python 2.5 on 64-bit machines (Marko Kreen)
- Fix PL/Python to not crash on long exception messages (Alvaro)
- Fix pg_dump to correctly handle allance child tables that have default expressions different from their parent's (Tom)
- Fix libpq crash when `PGPASSFILE` refers to a file that is not a plain file (Martin Pitt)
- ecpg parser fixes (Michael)
- Make `contrib/pgcrypto` defend against OpenSSL libraries that fail on keys longer than 128 bits; which is the case at least on some Solaris versions (Marko Kreen)
- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of `to_tsvector()` on huge input strings (Teodor)
- Require a specific version of Autoconf™ to be used when re-generating the **configure** script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf™ and PostgreSQL™ versions. You can remove the version check if you really want to use a different Autoconf™ version, but it's your responsibility whether the result works or not.
- Update `gettimeofday` configuration check so that PostgreSQL™ can be built on newer versions of MinGW™ (Magnus)

E.85. Release 8.2.5

Release Date

2007-09-17

This release contains a variety of fixes from 8.2.4. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.85.1. Migration to Version 8.2.5

A dump/restore is not required for those running 8.2.X.

E.85.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent **VACUUM** on the same table (Tom)
- Fix `ALTER DOMAIN ADD CONSTRAINT` for cases involving domains over domains (Tom)
- Make **CREATE DOMAIN ... DEFAULT NULL** work properly (Tom)
- Fix some planner problems with outer joins, notably poor size estimation for `t1 LEFT JOIN t2 WHERE t2.col IS NULL` (Tom)
- Allow the interval data type to accept input consisting only of milliseconds or microseconds (Neil)
- Allow timezone name to appear before the year in timestamp input (Tom)
- Fixes for GIN indexes used by `/contrib/tsearch2` (Teodor)
- Speed up rtree index insertion (Teodor)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)
- Fix `stddev_pop(numeric)` and `var_pop(numeric)` (Tom)
- Prevent **REINDEX** and **CLUSTER** from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand's upcoming changes (Tom)
- Windows socket and semaphore improvements (Magnus)
- Make **pg_ctl -w** work properly in Windows service mode (Dave Page)
- Fix memory allocation bug when using MIT Kerberos on Windows (Magnus)
- Suppress timezone name (`%Z`) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)
- Restrict `/contrib/pgstattuple` functions to superusers, for security reasons (Tom)
- Do not let `/contrib/intarray` try to make its GIN opclass the default (this caused problems at dump/restore) (Tom)

E.86. Release 8.2.4

Release Date

2007-04-23

This release contains a variety of fixes from 8.2.3, including a security fix. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.86.1. Migration to Version 8.2.4

A dump/restore is not required for those running 8.2.X.

E.86.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See **CREATE FUNCTION** for more information.
- Fix `shared_preload_libraries` for Windows by forcing reload in each backend (Korry Douglas)
- Fix `to_char()` so it properly upper/lower cases localized day or month names (Pavel Stehule)

- `/contrib/tsearch2` crash fixes (Teodor)
- Require **COMMIT PREPARED** to be executed in the same database as the transaction was prepared in (Heikki)
- Allow **pg_dump** to do binary backups larger than two gigabytes on Windows (Magnus)
- New traditional (Taiwan) Chinese FAQ (Zhou Daojing)
- Prevent the statistics collector from writing to disk too frequently (Tom)
- Fix potential-data-corruption bug in how **VACUUM FULL** handles **UPDATE** chains (Tom, Pavan Deolasee)
- Fix bug in domains that use array types (Tom)
- Fix **pg_dump** so it can dump a serial column's sequence using `-t` when not also dumping the owning table (Tom)
- Planner fixes, including improving outer join and bitmap scan selection logic (Tom)
- Fix possible wrong answers or crash when a PL/pgSQL function tries to `RETURN` from within an `EXCEPTION` block (Tom)
- Fix PANIC during enlargement of a hash index (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.87. Release 8.2.3

Release Date

2007-02-07

This release contains two fixes from 8.2.2. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.87.1. Migration to Version 8.2.3

A dump/restore is not required for those running 8.2.X.

E.87.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)
- Fix optimization so MIN/MAX in subqueries can again use indexes (Tom)

E.88. Release 8.2.2

Release Date

2007-02-05

This release contains a variety of fixes from 8.2.1, including a security fix. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.88.1. Migration to Version 8.2.2

A dump/restore is not required for those running 8.2.X.

E.88.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)

  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it's declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix not-so-rare-anymore bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linna-

kangas)

- Fix Borland C compile scripts (L Bayuk)
- Properly handle `to_char('CC')` for years ending in `00` (Tom)

  Year 2000 is in the twentieth century, not the twenty-first.
- `/contrib/tsearch2` localization improvements (Tatsuo, Teodor)
- Fix incorrect permission check in `information_schema.key_column_usage` view (Tom)

  The symptom is « relation with OID nnnnn does not exist » errors. To get this fix without using **initdb**, use **CREATE OR REPLACE VIEW** to install the corrected definition found in `share/information_schema.sql`. Note you will need to do this in each database.
- Improve **VACUUM** performance for databases with many tables (Tom)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Fix potentially incorrect results from index searches using `ROW` inequality conditions (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)
- Fix bogus « permission denied » failures occurring on Windows due to attempts to fsync already-deleted files (Magnus, Tom)
- Fix bug that could cause the statistics collector to hang on Windows (Magnus)

  This would in turn lead to autovacuum not working.
- Fix possible crashes when an already-in-use PL/pgSQL function is updated (Tom)
- Improve PL/pgSQL handling of domain types (Sergiy Vyshnevetskiy, Tom)
- Fix possible errors in processing PL/pgSQL exception blocks (Tom)

E.89. Release 8.2.1

Release Date

2007-01-08

This release contains a variety of fixes from 8.2. For information about new features in the 8.2 major release, see Section E.90, « Release 8.2 ».

E.89.1. Migration to Version 8.2.1

A dump/restore is not required for those running 8.2.

E.89.2. Changes

- Fix crash with `SELECT ... LIMIT ALL` (also `LIMIT NULL`) (Tom)
- `Several /contrib/tsearch2` fixes (Teodor)
- On Windows, make log messages coming from the operating system use ASCII encoding (Hiroshi Saito)

  This fixes a conversion problem when there is a mismatch between the encoding of the operating system and database server.
- Fix Windows linking of pg_dump using `win32.mak` (Hiroshi Saito)
- Fix planner mistakes for outer join queries (Tom)
- Fix several problems in queries involving sub-SELECTs (Tom)
- Fix potential crash in SPI during subtransaction abort (Tom)

  This affects all PL functions since they all use SPI.
- Improve build speed of PDF documentation (Peter)
- Re-add JST (Japan) timezone abbreviation (Tom)

- Improve optimization decisions related to index scans (Tom)
- Have psql print multi-byte combining characters as before, rather than output as `\u` (Tom)
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.
- Make pg_dumpall assume that databases have public `CONNECT` privilege, when dumping from a pre-8.2 server (Tom)

  This preserves the previous behavior that anyone can connect to a database if allowed by `pg_hba.conf`.

E.90. Release 8.2

Release Date

2006-12-05

E.90.1. Overview

This release adds many functionality and performance improvements that were requested by users, including:

- Query language enhancements including **INSERT/UPDATE/DELETE RETURNING**, multirow `VALUES` lists, and optional target-table alias in **UPDATE/DELETE**
- Index creation without blocking concurrent **INSERT/UPDATE/DELETE** operations
- Many query optimization improvements, including support for reordering outer joins
- Improved sorting performance with lower memory usage
- More efficient locking with better concurrency
- More efficient vacuuming
- Easier administration of warm standby servers
- New `FILLFACTOR` support for tables and indexes
- Monitoring, logging, and performance tuning additions
- More control over creating and dropping objects
- Table allance relationships can be defined for and removed from pre-existing tables
- **COPY TO** can copy the output of an arbitrary **SELECT** statement
- Array improvements, including nulls in arrays
- Aggregate-function improvements, including multiple-input aggregates and SQL:2003 statistical functions
- Many `contrib/` improvements

E.90.2. Migration to Version 8.2

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- Set `escape_string_warning` to `on` by default (Bruce)

  This issues a warning if backslash escapes are used in non-escape (non-`E''`) strings.
- Change the row constructor syntax (`ROW(...)`) so that list elements `foo.*` will be expanded to a list of their member fields, rather than creating a nested row type field as formerly (Tom)

  The new behavior is substantially more useful since it allows, for example, triggers to check for data changes with `IF row(new.*) IS DISTINCT FROM row(old.*)`. The old behavior is still available by omitting `.*`.
- Make row comparisons follow SQL standard semantics and allow them to be used in index scans (Tom)

Previously, row = and <> comparisons followed the standard but < <= > >= did not. A row comparison can now be used as an index constraint for a multicolumn index matching the row value.

- Make row `IS [NOT] NULL` tests follow SQL standard semantics (Tom)

  The former behavior conformed to the standard for simple cases with `IS NULL`, but `IS NOT NULL` would return true if any row field was non-null, whereas the standard says it should return true only when all fields are non-null.

- Make **SET CONSTRAINT** affect only one constraint (Kris Jurka)

  In previous releases, **SET CONSTRAINT** modified all constraints with a matching name. In this release, the schema search path is used to modify only the first matching constraint. A schema specification is also supported. This more nearly conforms to the SQL standard.

- Remove `RULE` permission for tables, for security reasons (Tom)

  As of this release, only a table's owner can create or modify rules for the table. For backwards compatibility, **GRANT**/**REVOKE RULE** is still accepted, but it does nothing.

- Array comparison improvements (Tom)

  Now array dimensions are also compared.

- Change array concatenation to match documented behavior (Tom)

  This changes the previous behavior where concatenation would modify the array lower bound.

- Make command-line options of postmaster and postgres identical (Peter)

  This allows the postmaster to pass arguments to each backend without using `-o`. Note that some options are now only available as long-form options, because there were conflicting single-letter options.

- Deprecate use of postmaster symbolic link (Peter)

  postmaster and postgres commands now act identically, with the behavior determined by command-line options. The postmaster symbolic link is kept for compatibility, but is not really needed.

- Change `log_duration` to output even if the query is not output (Tom)

  In prior releases, `log_duration` only printed if the query appeared earlier in the log.

- Make `to_char(time)` and `to_char(interval)` treat `HH` and `HH12` as 12-hour intervals

  Most applications should use `HH24` unless they want a 12-hour display.

- Zero unmasked bits in conversion from INET to CIDR (Tom)

  This ensures that the converted value is actually valid for CIDR.

- Remove `australian_timezones` configuration variable (Joachim Wieland)

  This variable has been superseded by a more general facility for configuring timezone abbreviations.

- Improve cost estimation for nested-loop index scans (Tom)

  This might eliminate the need to set unrealistically small values of `random_page_cost`. If you have been using a very small `random_page_cost`, please recheck your test cases.

- Change behavior of **pg_dump** `-n` and `-t` options. (Greg Sabino Mullane)

  See the **pg_dump** manual page for details.

- Change libpq `PQdsplen()` to return a useful value (Martijn van Oosterhout)
- Declare libpq `PQgetssl()` as returning `void *`, rather than `SSL *` (Martijn van Oosterhout)

  This allows applications to use the function without including the OpenSSL headers.

- C-language loadable modules must now include a `PG_MODULE_MAGIC` macro call for version compatibility checking (Martijn van Oosterhout)
- For security's sake, modules used by a PL/PerlU function are no longer available to PL/Perl functions (Andrew)

Note

This also implies that data can no longer be shared between a PL/Perl function and a PL/PerlU function. Some

> Perl installations have not been compiled with the correct flags to allow multiple interpreters to exist within a single process. In this situation PL/Perl and PL/PerlU cannot both be used in a single backend. The solution is to get a Perl installation which supports multiple interpreters.

- In `contrib/xml2/`, rename `xml_valid()` to `xml_is_well_formed()` (Tom)

  `xml_valid()` will remain for backward compatibility, but its behavior will change to do schema checking in a future release.
- Remove `contrib/ora2pg/`, now at *http://www.samse.fr/gpl/ora2pg*
- Remove contrib modules that have been migrated to PgFoundry: `adddepend`, `dbase`, `dbmirror`, `fulltextindex`, `mac`, `userlock`
- Remove abandoned contrib modules: `mSQL-interface`, `tips`
- Remove QNX and BEOS ports (Bruce)

  These ports no longer had active maintainers.

E.90.3. Changes

Below you will find a detailed account of the changes between PostgreSQL™ 8.2 and the previous major release.

E.90.3.1. Performance Improvements

- Allow the planner to reorder outer joins in some circumstances (Tom)

  In previous releases, outer joins would always be evaluated in the order written in the query. This change allows the query optimizer to consider reordering outer joins, in cases where it can determine that the join order can be changed without altering the meaning of the query. This can make a considerable performance difference for queries involving multiple outer joins or mixed inner and outer joins.
- Improve efficiency of `IN` (list-of-expressions) clauses (Tom)
- Improve sorting speed and reduce memory usage (Simon, Tom)
- Improve subtransaction performance (Alvaro, Itagaki Takahiro, Tom)
- Add `FILLFACTOR` to table and index creation (ITAGAKI Takahiro)

  This leaves extra free space in each table or index page, allowing improved performance as the database grows. This is particularly valuable to maintain clustering.
- Increase default values for `shared_buffers` and `max_fsm_pages` (Andrew)
- Improve locking performance by breaking the lock manager tables into sections (Tom)

  This allows locking to be more fine-grained, reducing contention.
- Reduce locking requirements of sequential scans (Qingqing Zhou)
- Reduce locking required for database creation and destruction (Tom)
- Improve the optimizer's selectivity estimates for `LIKE`, `ILIKE`, and regular expression operations (Tom)
- Improve planning of joins to inherited tables and `UNION ALL` views (Tom)
- Allow constraint exclusion to be applied to inherited **UPDATE** and **DELETE** queries (Tom)

  SELECT already honored constraint exclusion.
- Improve planning of constant `WHERE` clauses, such as a condition that depends only on variables alled from an outer query level (Tom)
- Protocol-level unnamed prepared statements are re-planned for each set of `BIND` values (Tom)

  This improves performance because the exact parameter values can be used in the plan.
- Speed up vacuuming of B-Tree indexes (Heikki Linnakangas, Tom)
- Avoid extra scan of tables without indexes during **VACUUM** (Greg Stark)

- Improve multicolumn GiST indexing (Oleg, Teodor)
- Remove dead index entries before B-Tree page split (Junji Teramoto)

E.90.3.2. Server Changes

- Allow a forced switch to a new transaction log file (Simon, Tom)

  This is valuable for keeping warm standby slave servers in sync with the master. Transaction log file switching now also happens automatically during `pg_stop_backup()`. This ensures that all transaction log files needed for recovery can be archived immediately.

- Add WAL informational functions (Simon)

  Add functions for interrogating the current transaction log insertion point and determining WAL filenames from the hex WAL locations displayed by `pg_stop_backup()` and related functions.

- Improve recovery from a crash during WAL replay (Simon)

  The server now does periodic checkpoints during WAL recovery, so if there is a crash, future WAL recovery is shortened. This also eliminates the need for warm standby servers to replay the entire log since the base backup if they crash.

- Improve reliability of long-term WAL replay (Heikki, Simon, Tom)

  Formerly, trying to roll forward through more than 2 billion transactions would not work due to XID wraparound. This meant warm standby servers had to be reloaded from fresh base backups periodically.

- Add `archive_timeout` to force transaction log file switches at a given interval (Simon)

  This enforces a maximum replication delay for warm standby servers.

- Add native LDAP authentication (Magnus Hagander)

  This is particularly useful for platforms that do not support PAM, such as Windows.

- Add `GRANT CONNECT ON DATABASE` (Gevik Babakhani)

  This gives SQL-level control over database access. It works as an additional filter on top of the existing `pg_hba.conf` controls.

- Add support for SSL Certificate Revocation List (CRL) files (Libor Hohoš)

  The server and libpq both recognize CRL files now.

- GiST indexes are now clusterable (Teodor)
- Remove routine autovacuum server log entries (Bruce)

  `pg_stat_activity` now shows autovacuum activity.

- Track maximum XID age within individual tables, instead of whole databases (Alvaro)

  This reduces the overhead involved in preventing transaction ID wraparound, by avoiding unnecessary VACUUMs.

- Add last vacuum and analyze timestamp columns to the stats collector (Larry Rosenman)

  These values now appear in the `pg_stat_*_tables` system views.

- Improve performance of statistics monitoring, especially `stats_command_string` (Tom, Bruce)

  This release enables `stats_command_string` by default, now that its overhead is minimal. This means `pg_stat_activity` will now show all active queries by default.

- Add a `waiting` column to `pg_stat_activity` (Tom)

  This allows pg_stat_activity to show all the information included in the ps display.

- Add configuration parameter `update_process_title` to control whether the ps display is updated for every command (Bruce)

  On platforms where it is expensive to update the ps display, it might be worthwhile to turn this off and rely solely on pg_stat_activity for status information.

- Allow units to be specified in configuration settings (Peter)

  For example, you can now set `shared_buffers` to `32MB` rather than mentally converting sizes.

- Add support for include directives in `postgresql.conf` (Joachim Wieland)
- Improve logging of protocol-level prepare/bind/execute messages (Bruce, Tom)

  Such logging now shows statement names, bind parameter values, and the text of the query being executed. Also, the query text is properly included in logged error messages when enabled by `log_min_error_statement`.
- Prevent `max_stack_depth` from being set to unsafe values

  On platforms where we can determine the actual kernel stack depth limit (which is most), make sure that the initial default value of `max_stack_depth` is safe, and reject attempts to set it to unsafely large values.
- Enable highlighting of error location in query in more cases (Tom)

  The server is now able to report a specific error location for some semantic errors (such as unrecognized column name), rather than just for basic syntax errors as before.
- Fix « failed to re-find parent key » errors in **VACUUM** (Tom)
- Clean out `pg_internal.init` cache files during server restart (Simon)

  This avoids a hazard that the cache files might contain stale data after PITR recovery.
- Fix race condition for truncation of a large relation across a gigabyte boundary by **VACUUM** (Tom)
- Fix bug causing needless deadlock errors on row-level locks (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Each backend process is now its own process group leader (Tom)

  This allows query cancel to abort subprocesses invoked from a backend or archive/recovery process.

E.90.3.3. Query Changes

- Add **INSERT**/**UPDATE**/**DELETE** `RETURNING` (Jonah Harris, Tom)

  This allows these commands to return values, such as the computed serial key for a new row. In the **UPDATE** case, values from the updated version of the row are returned.
- Add support for multiple-row `VALUES` clauses, per SQL standard (Joe, Tom)

  This allows **INSERT** to insert multiple rows of constants, or queries to generate result sets using constants. For example, `INSERT ... VALUES (...), (...), ....`, and `SELECT * FROM (VALUES (...), (...), ....) AS alias(f1, ...)`.
- Allow **UPDATE** and **DELETE** to use an alias for the target table (Atsushi Ogawa)

  The SQL standard does not permit an alias in these commands, but many database systems allow one anyway for notational convenience.
- Allow **UPDATE** to set multiple columns with a list of values (Susanne Ebrecht)

  This is basically a short-hand for assigning the columns and values in pairs. The syntax is `UPDATE tab SET (column, ...) = (val, ...)`.
- Make row comparisons work per standard (Tom)

  The forms <, <=, >, >= now compare rows lexicographically, that is, compare the first elements, if equal compare the second elements, and so on. Formerly they expanded to an AND condition across all the elements, which was neither standard nor very useful.
- Add `CASCADE` option to **TRUNCATE** (Joachim Wieland)

  This causes **TRUNCATE** to automatically include all tables that reference the specified table(s) via foreign keys. While convenient, this is a dangerous tool -- use with caution!
- Support `FOR UPDATE` and `FOR SHARE` in the same `SELECT` command (Tom)
- Add `IS NOT DISTINCT FROM` (Pavel Stehule)

  This operator is similar to equality (`=`), but evaluates to true when both left and right operands are `NULL`, and to false when just one is, rather than yielding `NULL` in these cases.
- Improve the length output used by `UNION`/`INTERSECT`/`EXCEPT` (Tom)

When all corresponding columns are of the same defined length, that length is used for the result, rather than a generic length.

- Allow `ILIKE` to work for multi-byte encodings (Tom)

  Internally, `ILIKE` now calls `lower()` and then uses `LIKE`. Locale-specific regular expression patterns still do not work in these encodings.

- Enable `standard_conforming_strings` to be turned `on` (Kevin Grittner)

  This allows backslash escaping in strings to be disabled, making PostgreSQL™ more standards-compliant. The default is `off` for backwards compatibility, but future releases will default this to `on`.

- Do not flatten subqueries that contain `volatile` functions in their target lists (Jaime Casanova)

  This prevents surprising behavior due to multiple evaluation of a `volatile` function (such as `random()` or `nextval()`). It might cause performance degradation in the presence of functions that are unnecessarily marked as `volatile`.

- Add system views `pg_prepared_statements` and `pg_cursors` to show prepared statements and open cursors (Joachim Wieland, Neil)

  These are very useful in pooled connection setups.

- Support portal parameters in **EXPLAIN** and **EXECUTE** (Tom)

  This allows, for example, JDBC `?` parameters to work in these commands.

- If SQL-level **PREPARE** parameters are unspecified, infer their types from the content of the query (Neil)

  Protocol-level **PREPARE** already did this.

- Allow `LIMIT` and `OFFSET` to exceed two billion (Dhanaraj M)

E.90.3.4. Object Manipulation Changes

- Add `TABLESPACE` clause to **CREATE TABLE AS** (Neil)

  This allows a tablespace to be specified for the new table.

- Add `ON COMMIT` clause to **CREATE TABLE AS** (Neil)

  This allows temporary tables to be truncated or dropped on transaction commit. The default behavior is for the table to remain until the session ends.

- Add `INCLUDING CONSTRAINTS` to **CREATE TABLE LIKE** (Greg Stark)

  This allows easy copying of `CHECK` constraints to a new table.

- Allow the creation of placeholder (shell) types (Martijn van Oosterhout)

  A shell type declaration creates a type name, without specifying any of the details of the type. Making a shell type is useful because it allows cleaner declaration of the type's input/output functions, which must exist before the type can be defined « for real ». The syntax is **CREATE TYPE *`typename`***.

- Aggregate functions now support multiple input parameters (Sergey Koposov, Tom)

- Add new aggregate creation syntax (Tom)

  The new syntax is **CREATE AGGREGATE *`aggname`* (*`input_type`*) (*`parameter_list`*)**. This more naturally supports the new multi-parameter aggregate functionality. The previous syntax is still supported.

- Add **ALTER ROLE PASSWORD NULL** to remove a previously set role password (Peter)

- Add **DROP** object `IF EXISTS` for many object types (Andrew)

  This allows **DROP** operations on non-existent objects without generating an error.

- Add `DROP OWNED` to drop all objects owned by a role (Alvaro)

- Add `REASSIGN OWNED` to reassign ownership of all objects owned by a role (Alvaro)

  This, and `DROP OWNED` above, facilitate dropping roles.

- Add **GRANT ON SEQUENCE** syntax (Bruce)

  This was added for setting sequence-specific permissions. `GRANT ON TABLE` for sequences is still supported for backward

compatibility.

- Add `USAGE` permission for sequences that allows only `currval()` and `nextval()`, not `setval()` (Bruce)

  `USAGE` permission allows more fine-grained control over sequence access. Granting `USAGE` allows users to increment a sequence, but prevents them from setting the sequence to an arbitrary value using `setval()`.

- Add `ALTER TABLE [ NO ] INHERIT` (Greg Stark)

  This allows allance to be adjusted dynamically, rather than just at table creation and destruction. This is very valuable when using allance to implement table partitioning.

- Allow comments on global objects to be stored globally (Kris Jurka)

  Previously, comments attached to databases were stored in individual databases, making them ineffective, and there was no provision at all for comments on roles or tablespaces. This change adds a new shared catalog pg_shdescription and stores comments on databases, roles, and tablespaces therein.

E.90.3.5. Utility Command Changes

- Add option to allow indexes to be created without blocking concurrent writes to the table (Greg Stark, Tom)

  The new syntax is **CREATE INDEX CONCURRENTLY**. The default behavior is still to block table modification while a index is being created.

- Provide advisory locking functionality (Abhijit Menon-Sen, Tom)

  This is a new locking API designed to replace what used to be in /contrib/userlock. The userlock code is now on pgfoundry.

- Allow **COPY** to dump a **SELECT** query (Zoltan Boszormenyi, Karel Zak)

  This allows **COPY** to dump arbitrary SQL queries. The syntax is `COPY (SELECT ...) TO`.

- Make the **COPY** command return a command tag that includes the number of rows copied (Volkan YAZICI)
- Allow **VACUUM** to expire rows without being affected by other concurrent **VACUUM** operations (Hannu Krossing, Alvaro, Tom)
- Make initdb detect the operating system locale and set the default `DateStyle` accordingly (Peter)

  This makes it more likely that the installed `postgresql.conf` `DateStyle` value will be as desired.

- Reduce number of progress messages displayed by initdb (Tom)

E.90.3.6. Date/Time Changes

- Allow full timezone names in timestamp input values (Joachim Wieland)

  For example, `'2006-05-24 21:11 America/New_York'::timestamptz`.

- Support configurable timezone abbreviations (Joachim Wieland)

  A desired set of timezone abbreviations can be chosen via the configuration parameter `timezone_abbreviations`.

- Add `pg_timezone_abbrevs` and `pg_timezone_names` views to show supported timezones (Magnus Hagander)
- Add `clock_timestamp()`, `statement_timestamp()`, and `transaction_timestamp()` (Bruce)

  `clock_timestamp()` is the current wall-clock time, `statement_timestamp()` is the time the current statement arrived at the server, and `transaction_timestamp()` is an alias for `now()`.

- Allow `to_char()` to print localized month and day names (Euler Taveira de Oliveira)
- Allow `to_char(time)` and `to_char(interval)` to output AM/PM specifications (Bruce)

  Intervals and times are treated as 24-hour periods, e.g. `25 hours` is considered AM.

- Add new function `justify_interval()` to adjust interval units (Mark Dilger)
- Allow timezone offsets up to 14:59 away from GMT

  Kiribati uses GMT+14, so we'd better accept that.

- Interval computation improvements (Michael Glaesemann, Bruce)

E.90.3.7. Other Data Type and Function Changes

- Allow arrays to contain `NULL` elements (Tom)
- Allow assignment to array elements not contiguous with the existing entries (Tom)

  The intervening array positions will be filled with nulls. This is per SQL standard.
- New built-in operators for array-subset comparisons (`@>`, `<@`, `&&`) (Teodor, Tom)

  These operators can be indexed for many data types using GiST or GIN indexes.
- Add convenient arithmetic operations on INET/CIDR values (Stephen R. van den Berg)

  The new operators are `&` (and), `|` (or), `~` (not), inet `+` int8, inet `-` int8, and inet `-` inet.
- Add new aggregate functions from SQL:2003 (Neil)

  The new functions are `var_pop()`, `var_samp()`, `stddev_pop()`, and `stddev_samp()`. `var_samp()` and `stddev_samp()` are merely renamings of the existing aggregates `variance()` and `stddev()`. The latter names remain available for backward compatibility.
- Add SQL:2003 statistical aggregates (Sergey Koposov)

  New functions: `regr_intercept()`, `regr_slope()`, `regr_r2()`, `corr()`, `covar_samp()`, `covar_pop()`, `regr_avgx()`, `regr_avgy()`, `regr_sxy()`, `regr_sxx()`, `regr_syy()`, `regr_count()`.
- Allow domains to be based on other domains (Tom)
- Properly enforce domain `CHECK` constraints everywhere (Neil, Tom)

  For example, the result of a user-defined function that is declared to return a domain type is now checked against the domain's constraints. This closes a significant hole in the domain implementation.
- Fix problems with dumping renamed SERIAL columns (Tom)

  The fix is to dump a SERIAL column by explicitly specifying its `DEFAULT` and sequence elements, and reconstructing the SERIAL column on reload using a new **ALTER SEQUENCE OWNED BY** command. This also allows dropping a SERIAL column specification.
- Add a server-side sleep function `pg_sleep()` (Joachim Wieland)
- Add all comparison operators for the tid (tuple id) data type (Mark Kirkwood, Greg Stark, Tom)

E.90.3.8. PL/pgSQL Server-Side Language Changes

- Add `TG_table_name` and `TG_table_schema` to trigger parameters (Andrew)

  `TG_relname` is now deprecated. Comparable changes have been made in the trigger parameters for the other PLs as well.
- Allow `FOR` statements to return values to scalars as well as records and row types (Pavel Stehule)
- Add a `BY` clause to the `FOR` loop, to control the iteration increment (Jaime Casanova)
- Add `STRICT` to **SELECT INTO** (Matt Miller)

  `STRICT` mode throws an exception if more or less than one row is returned by the **SELECT**, for Oracle PL/SQL™ compatibility.

E.90.3.9. PL/Perl Server-Side Language Changes

- Add `table_name` and `table_schema` to trigger parameters (Adam Sjøgren)
- Add prepared queries (Dmitry Karasik)
- Make `$_TD` trigger data a global variable (Andrew)

  Previously, it was lexical, which caused unexpected sharing violations.
- Run PL/Perl and PL/PerlU in separate interpreters, for security reasons (Andrew)

  In consequence, they can no longer share data nor loaded modules. Also, if Perl has not been compiled with the requisite flags to allow multiple interpreters, only one of these languages can be used in any given backend process.

E.90.3.10. PL/Python Server-Side Language Changes

- Named parameters are passed as ordinary variables, as well as in the `args[]` array (Sven Suursoho)
- Add `table_name` and `table_schema` to trigger parameters (Andrew)
- Allow returning of composite types and result sets (Sven Suursoho)
- Return result-set as `list`, `iterator`, or `generator` (Sven Suursoho)
- Allow functions to return `void` (Neil)
- Python 2.5 is now supported (Tom)

E.90.3.11. psql Changes

- Add new command `\password` for changing role password with client-side password encryption (Peter)
- Allow `\c` to connect to a new host and port number (David, Volkan YAZICI)
- Add tablespace display to `\l+` (Philip Yarra)
- Improve `\df` slash command to include the argument names and modes (`OUT` or `INOUT`) of the function (David Fetter)
- Support binary **COPY** (Andreas Pflug)
- Add option to run the entire session in a single transaction (Simon)

  Use option `-1` or `--single-transaction`.
- Support for automatically retrieving **SELECT** results in batches using a cursor (Chris Mair)

  This is enabled using **\set FETCH_COUNT *n***. This feature allows large result sets to be retrieved in psql without attempting to buffer the entire result set in memory.
- Make multi-line values align in the proper column (Martijn van Oosterhout)

  Field values containing newlines are now displayed in a more readable fashion.
- Save multi-line statements as a single entry, rather than one line at a time (Sergey E. Koposov)

  This makes up-arrow recall of queries easier. (This is not available on Windows, because that platform uses the native command-line editing present in the operating system.)
- Make the line counter 64-bit so it can handle files with more than two billion lines (David Fetter)
- Report both the returned data and the command status tag for **INSERT**/**UPDATE**/**DELETE RETURNING** (Tom)

E.90.3.12. pg_dump Changes

- Allow complex selection of objects to be included or excluded by pg_dump (Greg Sabino Mullane)

  pg_dump now supports multiple `-n` (schema) and `-t` (table) options, and adds `-N` and `-T` options to exclude objects. Also, the arguments of these switches can now be wild-card expressions rather than single object names, for example `-t 'foo*'`, and a schema can be part of a `-t` or `-T` switch, for example `-t schema1.table1`.
- Add pg_restore `--no-data-for-failed-tables` option to suppress loading data if table creation failed (i.e., the table already exists) (Martin Pitt)
- Add pg_restore option to run the entire session in a single transaction (Simon)

  Use option `-1` or `--single-transaction`.

E.90.3.13. libpq Changes

- Add `PQencryptPassword()` to encrypt passwords (Tom)

  This allows passwords to be sent pre-encrypted for commands like **ALTER ROLE ... PASSWORD**.
- Add function `PQisthreadsafe()` (Bruce)

This allows applications to query the thread-safety status of the library.

- Add `PQdescribePrepared()`, `PQdescribePortal()`, and related functions to return information about previously prepared statements and open cursors (Volkan YAZICI)
- Allow LDAP lookups from `pg_service.conf` (Laurenz Albe)
- Allow a hostname in `~/.pgpass` to match the default socket directory (Bruce)

  A blank hostname continues to match any Unix-socket connection, but this addition allows entries that are specific to one of several postmasters on the machine.

E.90.3.14. ecpg Changes

- Allow **SHOW** to put its result into a variable (Joachim Wieland)
- Add **COPY TO STDOUT** (Joachim Wieland)
- Add regression tests (Joachim Wieland, Michael)
- Major source code cleanups (Joachim Wieland, Michael)

E.90.3.15. Windows Port

- Allow MSVC to compile the PostgreSQL™ server (Magnus, Hiroshi Saito)
- Add MSVC support for utility commands and pg_dump (Hiroshi Saito)
- Add support for Windows code pages `1253`, `1254`, `1255`, and `1257` (Kris Jurka)
- Drop privileges on startup, so that the server can be started from an administrative account (Magnus)
- Stability fixes (Qingqing Zhou, Magnus)
- Add native semaphore implementation (Qingqing Zhou)

  The previous code mimicked SysV semaphores.

E.90.3.16. Source Code Changes

- Add GIN (Generalized Inverted iNdex) index access method (Teodor, Oleg)
- Remove R-tree indexing (Tom)

  Rtree has been re-implemented using GiST. Among other differences, this means that rtree indexes now have support for crash recovery via write-ahead logging (WAL).
- Reduce libraries needlessly linked into the backend (Martijn van Oosterhout, Tom)
- Add a configure flag to allow libedit to be preferred over GNU readline (Bruce)

  Use configure `--with-libedit-preferred`.
- Allow installation into directories containing spaces (Peter)
- Improve ability to relocate installation directories (Tom)
- Add support for Solaris x86_64™ using the Solaris™ compiler (Pierre Girard, Theo Schlossnagle, Bruce)
- Add DTrace support (Robert Lor)
- Add `PG_VERSION_NUM` for use by third-party applications wanting to test the backend version in C using > and < comparisons (Bruce)
- Add `XLOG_BLCKSZ` as independent from `BLCKSZ` (Mark Wong)
- Add `LWLOCK_STATS` define to report locking activity (Tom)
- Emit warnings for unknown configure options (Martijn van Oosterhout)
- Add server support for « plugin » libraries that can be used for add-on tasks such as debugging and performance measurement (Korry Douglas)

This consists of two features: a table of « rendezvous variables » that allows separately-loaded shared libraries to communicate, and a new configuration parameter `local_preload_libraries` that allows libraries to be loaded into specific sessions without explicit cooperation from the client application. This allows external add-ons to implement features such as a PL/pgSQL debugger.

- Rename existing configuration parameter `preload_libraries` to `shared_preload_libraries` (Tom)

  This was done for clarity in comparison to `local_preload_libraries`.

- Add new configuration parameter `server_version_num` (Greg Sabino Mullane)

  This is like `server_version`, but is an integer, e.g. `80200`. This allows applications to make version checks more easily.

- Add a configuration parameter `seq_page_cost` (Tom)
- Re-implement the regression test script as a C program (Magnus, Tom)
- Allow loadable modules to allocate shared memory and lightweight locks (Marc Munro)
- Add automatic initialization and finalization of dynamically loaded libraries (Ralf Engelschall, Tom)

  New functions `_PG_init()` and `_PG_fini()` are called if the library defines such symbols. Hence we no longer need to specify an initialization function in `shared_preload_libraries`; we can assume that the library used the `_PG_init()` convention instead.

- Add `PG_MODULE_MAGIC` header block to all shared object files (Martijn van Oosterhout)

  The magic block prevents version mismatches between loadable object files and servers.

- Add shared library support for AIX (Laurenz Albe)
- New XML documentation section (Bruce)

E.90.3.17. Contrib Changes

- Major tsearch2 improvements (Oleg, Teodor)
  - multibyte encoding support, including UTF8
  - query rewriting support
  - improved ranking functions
  - thesaurus dictionary support
  - Ispell dictionaries now recognize MySpell format, used by OpenOffice
  - GIN support
- Add adminpack module containing Pgadmin administration functions (Dave)

  These functions provide additional file system access routines not present in the default PostgreSQL™ server.

- Add sslinfo module (Victor Wagner)

  Reports information about the current connection's SSL certificate.

- Add pgrowlocks module (Tatsuo)

  This shows row locking information for a specified table.

- Add hstore module (Oleg, Teodor)
- Add isn module, replacing isbn_issn (Jeremy Kronuz)

  This new implementation supports EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serials).

- Add index information functions to pgstattuple (ITAGAKI Takahiro, Satoshi Nagayasu)
- Add pg_freespacemap module to display free space map information (Mark Kirkwood)
- pgcrypto now has all planned functionality (Marko Kreen)
  - Include iMath library in pgcrypto to have the public-key encryption functions always available.

  - Add SHA224 algorithm that was missing in OpenBSD code.
  - Activate builtin code for SHA224/256/384/512 hashes on older OpenSSL to have those algorithms always available.
  - New function gen_random_bytes() that returns cryptographically strong randomness. Useful for generating encryption keys.
  - Remove digest_exists(), hmac_exists() and cipher_exists() functions.

- Improvements to cube module (Joshua Reich)

  New functions are `cube(float[])`, `cube(float[], float[])`, and `cube_subset(cube, int4[])`.
- Add async query capability to dblink (Kai Londenberg, Joe Conway)
- New operators for array-subset comparisons (`@>`, `<@`, `&&`) (Tom)

  Various contrib packages already had these operators for their datatypes, but the naming wasn't consistent. We have now added consistently named array-subset comparison operators to the core code and all the contrib packages that have such functionality. (The old names remain available, but are deprecated.)
- Add uninstall scripts for all contrib packages that have install scripts (David, Josh Drake)

E.91. Release 8.1.23

Release Date

2010-12-16

This release contains a variety of fixes from 8.1.22. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

This is expected to be the last PostgreSQL™ release in the 8.1.X series. Users are encouraged to update to a newer release branch soon.

E.91.1. Migration to Version 8.1.23

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see Section E.96, « Release 8.1.18 ».

E.91.2. Changes

- Force the default `wal_sync_method` to be `fdatasync` on Linux (Tom Lane, Marti Raudsepp)

  The default on Linux has actually been `fdatasync` for many years, but recent kernel changes caused PostgreSQL™ to choose `open_datasync` instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably `ext4` with the `data=journal` mount option.
- Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)
- Add support for detecting register-stack overrun on `IA64` (Tom Lane)

  The `IA64` architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.
- Add a check for stack overflow in `copyObject()` (Tom Lane)

  Certain code paths could crash due to stack overflow given a sufficiently complex query.
- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)

  It is possible to have a « concurrent » page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.
- Avoid memory leakage while **ANALYZE**'ing complex index expressions (Tom Lane)
- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)

An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.

- Do not « inline » a SQL function with multiple `OUT` parameters (Tom Lane)

  This avoids a possible crash due to loss of information about the expected result rowtype.

- Fix constant-folding of `COALESCE()` expressions (Tom Lane)

  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.

- Add print functionality for InhRelation nodes (Tom Lane)

  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.

- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)

  This bug affected several different geometric distance-measurement operators.

- Fix PL/pgSQL's handling of « simple » expressions to not fail in recursion or error-recovery cases (Tom Lane)
- Fix bug in `contrib/cube`'s GiST picksplit algorithm (Alexander Korotkov)

  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a cube column. If you have such an index, consider **REINDEX**ing it after installing this update.

- Don't emit « identifier will be truncated » notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)
- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)
- Fix memory leak in `contrib/xml2`'s XPath query functions (Tom Lane)
- Update time zone data files to tzdata release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.

E.92. Release 8.1.22

Release Date

2010-10-04

This release contains a variety of fixes from 8.1.21. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

The PostgreSQL™ community will stop releasing updates for the 8.1.X release series in November 2010. Users are encouraged to update to a newer release branch soon.

E.92.1. Migration to Version 8.1.22

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see Section E.96, « Release 8.1.18 ».

E.92.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise

contacting the authors of any PL you are depending on for security-critical purposes.

Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

- Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it's intended to be used with (Heikki Linnakangas, Tom Lane)
- Fix « cannot handle unplanned sub-select » error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.
- Prevent show_session_authorization() from crashing within autovacuum processes (Tom Lane)
- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)
- Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)
- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.
- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.
- Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)
- Fix possible data corruption in **ALTER TABLE ... SET TABLESPACE** when archiving is enabled (Jeff Davis)
- Allow **CREATE DATABASE** and **ALTER DATABASE ... SET TABLESPACE** to be interrupted by query-cancel (Guillaume Lelarge)
- In PL/Python, defend against null pointer results from `PyCObject_AsVoidPtr` and `PyCObject_FromVoidPtr` (Peter Eisentraut)
- Improve `contrib/dblink`'s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after « duplicate connection name » errors in `contrib/dblink` (Itagaki Takahiro)
- Fix `contrib/dblink` to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)
- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.

E.93. Release 8.1.21

Release Date

2010-05-17

This release contains a variety of fixes from 8.1.20. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.93.1. Migration to Version 8.1.21

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see Section E.96, « Release 8.1.18 ».

E.93.2. Changes

- Enforce restrictions in `plperl` using an opmask applied to the whole interpreter, instead of using `Safe.pm` (Tim Bunce, Andrew Dunstan)

Recent developments have convinced us that `Safe.pm` is too insecure to rely on for making `plperl` trustable. This change removes use of `Safe.pm` altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's `strict` pragma in a natural way in `plperl`, and that Perl's `$a` and `$b` variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted « normal » Tcl interpreter unless we are really going to execute a `pltclu` function. (CVE-2010-1170)

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran `ALTER USER ... RESET ALL` for himself, or `ALTER DATABASE ... RESET ALL` for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the **ALTER** will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a `CONTEXT` addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Update pl/perl's `ppport.h` for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)
- Ensure that `contrib/pgstattuple` functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that `shmget()` returns `EINVAL` for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

- Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

E.94. Release 8.1.20

Release Date

2010-03-15

This release contains a variety of fixes from 8.1.19. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.94.1. Migration to Version 8.1.20

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see Section E.96, « Release 8.1.18 ».

E.94.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for an SSL connection (Magnus)

  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

- Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)

- Fix server memory leak associated with use of savepoints and a client encoding different from server's encoding (Tom)
- Make `substring()` for bit types treat any negative length as meaning « all the rest of the string » (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix the `STOP WAL LOCATION` entry in backup history files to report the next WAL segment's name when the end location is exactly at a segment boundary (Itagaki Takahiro)
- Fix some more cases of temporary-file leakage (Heikki)

  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function's exception handler.
- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.
- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)
- Fix psql's `numericlocale` option to not format strings it shouldn't in latex and troff output formats (Heikki)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Add `volatile` markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.
- Prevent crash in `contrib/dblink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)
- Fix assorted crashes in `contrib/xml2` caused by sloppy memory management (Tom)
- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.95. Release 8.1.19

Release Date

2009-12-14

This release contains a variety of fixes from 8.1.18. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.95.1. Migration to Version 8.1.19

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see Section E.96, « Release 8.1.18 ».

E.95.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

  This change prevents allegedly-immutable index functions from possibly subverting a superuser's session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).

- Fix possible crash during backend-startup-time cache initialization (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)

  This fix prevents a PANIC if a `VACUUM FULL` is canceled after it's already committed its tuple movements, as well as transient errors if a plain `VACUUM` is interrupted after having truncated the table.
- Fix possible crash due to integer overflow in hash table size calculation (Tom)

  This could occur with extremely large planner estimates for the size of a hashjoin's result.
- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix PAM password processing to be more robust (Tom)

  The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Fix processing of ownership dependencies during `CREATE OR REPLACE FUNCTION` (Tom)
- Ensure that Perl arrays are properly converted to PostgreSQL™ arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)

  This worked correctly already for non-set-returning functions.
- Fix rare crash in exception processing in PL/Python (Peter)
- Ensure psql's flex module is compiled with the correct system header definitions (Tom)

  This fixes build failures on platforms where `--enable-largefile` causes incompatible changes in the generated code.
- Make the postmaster ignore any `application_name` parameter in connection request packets, to improve compatibility with future libpq versions (Tom)
- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.

E.96. Release 8.1.18

Release Date

2009-09-09

This release contains a variety of fixes from 8.1.17. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.96.1. Migration to Version 8.1.18

A dump/restore is not required for those running 8.1.X. However, if you have any hash indexes on interval columns, you must **REINDEX** them after updating to 8.1.18. Also, if you are upgrading from a version earlier than 8.1.15, see Section E.99, « Release 8.1.15 ».

E.96.2. Changes

- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)
- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
- Fix hash calculation for data type interval (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must **REINDEX** them after updating.

- Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with `'HH'`/`'HH12'` (Heikki)

  It was previously handled as `'th'` (lowercase).
- Fix overflow for `INTERVAL 'x ms'` when *x* is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.
- Fix money data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)
- Properly round datetime input like `00:12:57.999999999999999999999999999` (Tom)
- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)
- Fix portability issues in plperl initialization (Andrew Dunstan)
- Fix pg_ctl to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)
- Fix `contrib/xml2`'s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)
- Improve robustness of libpq's code to recover from errors during **COPY FROM STDIN** (Tom)
- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)
- Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.97. Release 8.1.17

Release Date

2009-03-16

This release contains a variety of fixes from 8.1.16. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.97.1. Migration to Version 8.1.17

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.15, see Section E.99, « Release 8.1.15 ».

E.97.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)

  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that *any* error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)
- Disallow **CREATE CONVERSION** with the wrong encodings for the specified conversion function (Heikki)

  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.
- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)
- Fix decompilation of `CASE WHEN` with an implicit coercion (Tom)

  This mistake could lead to Assert failures in an Assert-enabled build, or an « unexpected CASE WHEN clause » error message in other cases, when trying to examine or dump a view.
- Fix possible misassignment of the owner of a TOAST table's rowtype (Tom)

  If **CLUSTER** or a rewriting variant of **ALTER TABLE** were executed by someone other than the table owner, the pg_type entry for the table's TOAST table would end up marked as owned by that someone. This caused no immediate problems, since the permissions on the TOAST rowtype aren't examined by any ordinary database operation. However, it could lead to unex-

pected failures if one later tried to drop the role that issued the command (in 8.1 or 8.2), or « owner of data type appears to be invalid » warnings from pg_dump after having done so (in 8.3).

- Clean up PL/pgSQL error status variables fully at block exit (Ashesh Vashi and Dave Page)

  This is not a problem for PL/pgSQL itself, but the omission could cause the PL/pgSQL Debugger to crash while examining the state of a function.

- Add `MUST` (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

E.98. Release 8.1.16

Release Date

2009-02-02

This release contains a variety of fixes from 8.1.15. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.98.1. Migration to Version 8.1.16

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.15, see Section E.99, « Release 8.1.15 ».

E.98.2. Changes

- Fix crash in autovacuum (Alvaro)

  The crash occurs only after vacuuming a whole database for anti-transaction-wraparound purposes, which means that it occurs infrequently and is hard to track down.

- Improve handling of URLs in `headline()` function (Teodor)
- Improve handling of overlength headlines in `headline()` function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Avoid unnecessary locking of small tables in **VACUUM** (Heikki)
- Ensure that the contents of a holdable cursor don't depend on the contents of TOAST tables (Tom)

  Previously, large field values in a cursor result might be represented as TOAST pointers, which would fail if the referenced table got dropped before the cursor is read, or if the large value is deleted and then vacuumed away. This cannot happen with an ordinary cursor, but it could with a cursor that is held past its creating transaction.

- Fix uninitialized variables in `contrib/tsearch2`'s `get_covers()` function (Teodor)
- Fix configure script to properly report failure when unable to obtain linkage information for PL/Perl (Andrew)
- Make all documentation reference `pgsql-bugs` and/or `pgsql-hackers` as appropriate, instead of the now-decommissioned `pgsql-ports` and `pgsql-patches` mailing lists (Tom)
- Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections in Switzerland, Cuba)

E.99. Release 8.1.15

Release Date

2008-11-03

This release contains a variety of fixes from 8.1.14. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.99.1. Migration to Version 8.1.15

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ». Also, if you were running a previous 8.1.X release, it is recommended to **REINDEX** all GiST indexes after the upgrade.

E.99.2. Changes

- Fix GiST index corruption due to marking the wrong index entry « dead » after a deletion (Teodor)

  This would result in index searches failing to find rows they should have found. Corrupted indexes can be fixed with **REINDEX**.

- Fix backend crash when the client encoding cannot represent a localized error message (Tom)

  We have addressed similar issues before, but it would still fail if the « character has no equivalent » message itself couldn't be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.

- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Fix mis-expansion of rule queries when a sub-`SELECT` appears in a function call in `FROM`, a multi-row `VALUES` list, or a `RETURNING` list (Tom)

  The usual symptom of this problem is an « unrecognized node type » error.

- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Prevent possible collision of *`relfilenode`* numbers when moving a table to another tablespace with **ALTER SET TABLESPACE** (Heikki)

  The command tried to re-use the existing filename, instead of picking one that is known unused in the destination directory.

- Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an `--enable-integer-datetimes` build (Ron Mayer)
- Ensure `SPI_getvalue` and `SPI_getbinval` behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)

  This situation is normal when a table has had columns added or removed, but these two functions didn't handle it properly. The only likely consequence is an incorrect error indication.

- Fix ecpg's parsing of **CREATE ROLE** (Michael)
- Fix recent breakage of `pg_ctl restart` (Tom)
- Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil, Mauritius, Syria)

E.100. Release 8.1.14

Release Date

2008-09-22

This release contains a variety of fixes from 8.1.13. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.100.1. Migration to Version 8.1.14

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.100.2. Changes

- Widen local lock counters from 32 to 64 bits (Tom)

  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected « lock is already held » errors.

- Fix possible duplicate output of tuples during a GiST index scan (Teodor)
- Add checks in executor startup to ensure that the tuples produced by an **INSERT** or **UPDATE** will match the target table's current rowtype (Tom)

  ALTER COLUMN TYPE, followed by re-use of a previously cached plan, could produce this type of situation. The check protects against data corruption and/or crashes that could ensue.
- Fix `AT TIME ZONE` to first try to interpret its timezone argument as a timezone abbreviation, and only try it as a full time-zone name if that fails, rather than the other way around as formerly (Tom)

  The timestamp input functions have always resolved ambiguous zone names in this order. Making `AT TIME ZONE` do so as well improves consistency, and fixes a compatibility bug introduced in 8.1: in ambiguous cases we now behave the same as 8.0 and before did, since in the older versions `AT TIME ZONE` accepted *only* abbreviations.
- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Fix bug in backwards scanning of a cursor on a `SELECT DISTINCT ON` query (Tom)
- Fix planner bug with nested sub-select expressions (Tom)

  If the outer sub-select has no direct dependency on the parent query, but the inner one does, the outer value might not get re-calculated for new parent query rows.
- Fix planner to estimate that `GROUP BY` expressions yielding boolean results always result in two groups, regardless of the expressions' contents (Tom)

  This is very substantially more accurate than the regular `GROUP BY` estimate for certain boolean tests like *`col`* `IS NULL`.
- Fix PL/pgSQL to not fail when a `FOR` loop's target variable is a record containing composite-type fields (Tom)
- Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)
- Fix PL/Python to work with Python 2.5

  This is a back-port of fixes made during the 8.2 development cycle.
- Improve pg_dump and pg_restore's error reporting after failure to send a SQL command (Tom)
- Fix pg_ctl to properly preserve postmaster command-line arguments across a `restart` (Bruce)
- Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.101. Release 8.1.13

Release Date

2008-06-12

This release contains one serious and one minor bug fix over 8.1.12. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.101.1. Migration to Version 8.1.13

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.101.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)

  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL™ to reject what had been a valid **SELECT DISTINCT** view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

- Make **ALTER AGGREGATE ... OWNER TO** update pg_shdepend (Tom)

  This oversight could lead to problems if the aggregate was later involved in a **DROP OWNED** or **REASSIGN OWNED** operation.

E.102. Release 8.1.12

Release Date

never released

This release contains a variety of fixes from 8.1.11. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.102.1. Migration to Version 8.1.12

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.102.2. Changes

- Fix **ALTER TABLE ADD COLUMN ... PRIMARY KEY** so that the new column is correctly checked to see if it's been initialized to all non-nulls (Brendan Jurd)

  Previous versions neglected to check this requirement at all.

- Fix possible **CREATE TABLE** failure when alling the « same » constraint from multiple parent relations that alled that constraint from a common ancestor (Tom)

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic « Yo » characters (`e` and `E` with two dots) (Sergey Burladyan)

- Fix a few datatype input functions that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched `ORDER BY` and `DISTINCT` expressions.

- Fix a corner case in regular-expression substring matching (`substring(`*`string`*` from `*`pattern`*`)`) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn't got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since `(bar)` isn't matched, but it was mistakenly returning the whole-pattern match instead (ie, `foo`).

- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, Argentina/San_Luis, and Chile)

- Fix incorrect result from ecpg's `PGTYPEStimestamp_sub()` function (Michael)

- Fix core dump in `contrib/xml2`'s `xpath_table()` function when the input query returns a NULL value (Tom)

- Fix `contrib/xml2`'s makefile to not override `CFLAGS` (Tom)

- Fix `DatumGetBool` macro to not fail with gcc 4.3 (Tom)

  This problem affects « old style » (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.

- Fix longstanding **LISTEN**/**NOTIFY** race condition (Tom)

  In rare cases a session that had just executed a **LISTEN** might not get a notification, even though one would be expected because the concurrent transaction executing **NOTIFY** was observed to commit later.

  A side effect of the fix is that a transaction that has executed a not-yet-committed **LISTEN** command will not see any row in pg_listener for the **LISTEN**, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Disallow **LISTEN** and **UNLISTEN** within a prepared transaction (Tom)

This was formerly allowed but trying to do it had various unpleasant consequences, notably that the originating backend could not exit as long as an **UNLISTEN** remained uncommitted.

- Fix rare crash when an error occurs during a query using a hash index (Heikki)
- Fix input of datetime values for February 29 in years BC (Tom)

  The former coding was mistaken about which years were leap years.
- Fix « unrecognized node type » error in some variants of **ALTER OWNER** (Tom)
- Fix pg_ctl to correctly extract the postmaster's port number from command-line options (Itagaki Takahiro, Tom)

  Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.
- Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)

  This is known to be necessary when building PostgreSQL™ with gcc 4.3 or later.
- Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)

  An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.
- Fix libpq to handle NOTICE messages correctly during COPY OUT (Tom)

  This failure has only been observed to occur when a user-defined datatype's output routine issues a NOTICE, but there is no guarantee it couldn't happen due to other causes.

E.103. Release 8.1.11

Release Date

2008-01-07

This release contains a variety of fixes from 8.1.10, including fixes for significant security issues. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

This is the last 8.1.X release for which the PostgreSQL™ community will produce binary packages for Windows™. Windows users are encouraged to move to 8.2.X or later, since there are Windows-specific fixes in 8.2.X that are impractical to back-port. 8.1.X will continue to be supported on other platforms.

E.103.1. Migration to Version 8.1.11

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.103.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running **VACUUM**, **ANALYZE**, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as **VACUUM FULL**, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including **VACUUM**, **ANALYZE**, **REINDEX**, and **CLUSTER**) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of **SET SESSION AUTHORIZATION** and **SET ROLE** is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)
- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy

sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 8.1.10 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)
- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)
- Improve planner's handling of LIKE/regex estimation in non-C locales (Tom)
- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)
- Preserve the tablespace of indexes that are rebuilt by **ALTER TABLE ... ALTER COLUMN TYPE** (Tom)
- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.
- Make **VACUUM** not use all of `maintenance_work_mem` when the table is too small for it to be useful (Alvaro)
- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)
- Fix overflow in `extract(epoch from interval)` for intervals exceeding 68 years (Tom)
- Fix PL/Perl to not fail when a UTF-8 regular expression is used in a trusted function (Andrew)
- Fix PL/Perl to cope when platform's Perl defines type `bool` as `int` rather than `char` (Tom)

  While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X™ 10.5.
- Fix PL/Python to not crash on long exception messages (Alvaro)
- Fix pg_dump to correctly handle allance child tables that have default expressions different from their parent's (Tom)
- Fix libpq crash when `PGPASSFILE` refers to a file that is not a plain file (Martin Pitt)
- ecpg parser fixes (Michael)
- Make `contrib/pgcrypto` defend against OpenSSL libraries that fail on keys longer than 128 bits; which is the case at least on some Solaris versions (Marko Kreen)
- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of `to_tsvector()` on huge input strings (Teodor)
- Require a specific version of Autoconf™ to be used when re-generating the **configure** script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf™ and PostgreSQL™ versions. You can remove the version check if you really want to use a different Autoconf™ version, but it's your responsibility whether the result works or not.

E.104. Release 8.1.10

Release Date

2007-09-17

This release contains a variety of fixes from 8.1.9. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.104.1. Migration to Version 8.1.10

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.104.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent **VACUUM** on the same table (Tom)
- Make **CREATE DOMAIN ... DEFAULT NULL** work properly (Tom)
- Allow the interval data type to accept input consisting only of milliseconds or microseconds (Neil)
- Speed up rtree index insertion (Teodor)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)
- Prevent **REINDEX** and **CLUSTER** from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand's upcoming changes (Tom)
- Windows socket improvements (Magnus)
- Suppress timezone name (`%Z`) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.105. Release 8.1.9

Release Date

2007-04-23

This release contains a variety of fixes from 8.1.8, including a security fix. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.105.1. Migration to Version 8.1.9

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.105.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See **CREATE FUNCTION** for more information.
- `/contrib/tsearch2` crash fixes (Teodor)
- Require **COMMIT PREPARED** to be executed in the same database as the transaction was prepared in (Heikki)
- Fix potential-data-corruption bug in how **VACUUM FULL** handles **UPDATE** chains (Tom, Pavan Deolasee)
- Planner fixes, including improving outer join and bitmap scan selection logic (Tom)
- Fix PANIC during enlargement of a hash index (bug introduced in 8.1.6) (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.106. Release 8.1.8

Release Date

2007-02-07

This release contains one fix from 8.1.7. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.106.1. Migration to Version 8.1.8

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.106.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)

E.107. Release 8.1.7

Release Date

2007-02-05

This release contains a variety of fixes from 8.1.6, including a security fix. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.107.1. Migration to Version 8.1.7

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.107.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)

  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it's declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Improve **VACUUM** performance for databases with many tables (Tom)
- Fix autovacuum to avoid leaving non-permanent transaction IDs in non-connectable databases (Alvaro)

  This bug affects the 8.1 branch only.
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)
- Fix bogus « permission denied » failures occurring on Windows due to attempts to fsync already-deleted files (Magnus, Tom)
- Fix possible crashes when an already-in-use PL/pgSQL function is updated (Tom)

E.108. Release 8.1.6

Release Date

2007-01-08

This release contains a variety of fixes from 8.1.5. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.108.1. Migration to Version 8.1.6

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.108.2. Changes

- Improve handling of `getaddrinfo()` on AIX (Tom)

  This fixes a problem with starting the statistics collector, among other things.
- Fix pg_restore to handle a tar-format backup that contains large objects (blobs) with comments (Tom)
- Fix « failed to re-find parent key » errors in **VACUUM** (Tom)
- Clean out `pg_internal.init` cache files during server restart (Simon)

  This avoids a hazard that the cache files might contain stale data after PITR recovery.
- Fix race condition for truncation of a large relation across a gigabyte boundary by **VACUUM** (Tom)
- Fix bug causing needless deadlock errors on row-level locks (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix possible deadlock in Windows signal handling (Teodor)
- Fix error when constructing an `ARRAY[]` made up of multiple empty elements (Tom)
- Fix ecpg memory leak during connection (Michael)
- Fix for Darwin (OS X) compilation (Tom)
- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.
- Update timezone database

  This affects Australian and Canadian daylight-savings rules in particular.

E.109. Release 8.1.5

> **Release Date**
>
> 2006-10-16

This release contains a variety of fixes from 8.1.4. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.109.1. Migration to Version 8.1.5

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.109.2. Changes

- Disallow aggregate functions in **UPDATE** commands, except within sub-SELECTs (Tom)

  The behavior of such an aggregate was unpredictable, and in 8.1.X could cause a crash, so it has been disabled. The SQL standard does not allow this either.
- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix core dump in duration logging for extended query protocol when a **COMMIT** or **ROLLBACK** is executed
- Fix mishandling of AFTER triggers when query contains a SQL function returning multiple rows (Tom)
- Fix **ALTER TABLE ... TYPE** to recheck `NOT NULL` for `USING` clause (Tom)
- Fix `string_to_array()` to handle overlapping matches for the separator string

  For example, `string_to_array('123xx456xxx789', 'xx')`.

- Fix `to_timestamp()` for `AM`/`PM` formats (Bruce)
- Fix autovacuum's calculation that decides whether **ANALYZE** is needed (Alvaro)
- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Numerous robustness fixes in ecpg (Joachim Wieland)
- Fix backslash escaping in /contrib/dbmirror
- Minor fixes in /contrib/dblink and /contrib/tsearch2
- Efficiency improvements in hash tables and bitmap index scans (Tom)
- Fix instability of statistics collection on Windows (Tom, Andrew)
- Fix `statement_timeout` to use the proper units on Win32 (Bruce)

  In previous Win32 8.1.X versions, the delay was off by a factor of 100.
- Fixes for MSVC and Borland C++™ compilers (Hiroshi Saito)
- Fixes for `AIX` and Intel™ compilers (Tom)
- Fix rare bug in continuous archiving (Tom)

E.110. Release 8.1.4

Release Date

2006-05-23

This release contains a variety of fixes from 8.1.3, including patches for extremely serious security issues. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.110.1. Migration to Version 8.1.4

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.110.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL™ has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.
- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.
- Modify libpq's string-escaping routines to be aware of encoding considerations and `standard_conforming_strings`

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple

™ connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping « by hand » should be modified to rely on library routines instead.

- Fix weak key selection in pgcrypto (Marko Kreen)

  Errors in fortuna PRNG reseeding logic could cause a predictable session key to be selected by `pgp_sym_encrypt()` in some cases. This only affects non-OpenSSL-using builds.

- Fix some incorrect encoding conversion functions

  `win1251_to_iso`, `win866_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.

- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)
- Make autovacuum visible in pg_stat_activity (Alvaro)
- Disable `full_page_writes` (Tom)

  In certain cases, having `full_page_writes` off would cause crash recovery to fail. A proper fix will appear in 8.2; for now it's just disabled.

- Various planner fixes, particularly for bitmap index scans and MIN/MAX optimization (Tom)
- Fix incorrect optimization in merge join (Tom)

  Outer joins could sometimes emit multiple copies of unmatched rows.

- Fix crash from using and modifying a plpgsql function in the same transaction
- Fix WAL replay for case where a B-Tree index has been truncated
- Fix `SIMILAR TO` for patterns involving `|` (Tom)
- Fix **SELECT INTO** and **CREATE TABLE AS** to create tables in the default tablespace, not the base directory (Kris Jurka)
- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
- Improve qsort performance (Dann Corbit)

  Currently this code is only used on Solaris.

- Fix for OS/X Bonjour on x86 systems (Ashley Clark)
- Fix various minor memory leaks
- Fix problem with password prompting on some Win32 systems (Robert Kinberg)
- Improve pg_dump's handling of default values for domains
- Fix pg_dumpall to handle identically-named users and groups reasonably (only possible when dumping from a pre-8.1 server) (Tom)

  The user and group will be merged into a single role with `LOGIN` permission. Formerly the merged role wouldn't have `LOGIN` permission, making it unusable as a user.

- Fix pg_restore `-n` to work as documented (Tom)

E.111. Release 8.1.3

Release Date

2006-02-14

This release contains a variety of fixes from 8.1.2, including one very serious security issue. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.111.1. Migration to Version 8.1.3

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see Section E.112, « Release 8.1.2 ».

E.111.2. Changes

- Fix bug that allowed any logged-in user to **SET ROLE** to any other database user id (CVE-2006-0553)

  Due to inadequate validity checking, a user could exploit the special case that **SET ROLE** normally uses to restore the previous role setting after an error. This allowed ordinary users to acquire superuser status, for example. The escalation-of-privilege risk exists only in 8.1.0-8.1.2. However, in all releases back to 7.3 there is a related bug in **SET SESSION AUTHORIZATION** that allows unprivileged users to crash the server, if it has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 8.0.4, 7.4.9, and 7.3.11 releases.
- Fix race condition that could lead to « file already exists » errors during pg_clog and pg_subtrans file creation (Tom)
- Fix cases that could lead to crashes if a cache-invalidation message arrives at just the wrong time (Tom)
- Properly check `DOMAIN` constraints for `UNKNOWN` parameters in prepared statements (Neil)
- Ensure **ALTER COLUMN TYPE** will process `FOREIGN KEY`, `UNIQUE`, and `PRIMARY KEY` constraints in the proper order (Nakano Yoshihisa)
- Fixes to allow restoring dumps that have cross-schema references to custom operators or operator classes (Tom)
- Allow pg_restore to continue properly after a **COPY** failure; formerly it tried to treat the remaining **COPY** data as SQL commands (Stephen Frost)
- Fix pg_ctl `unregister` crash when the data directory is not specified (Magnus)
- Fix libpq `PQprint` HTML tags (Christoph Zwerschke)
- Fix ecpg crash on AMD64 and PPC (Neil)
- Allow `SETOF` and `%TYPE` to be used together in function result type declarations
- Recover properly if error occurs during argument passing in PL/python (Neil)
- Fix memory leak in `plperl_return_next` (Neil)
- Fix PL/perl's handling of locales on Win32 to match the backend (Andrew)
- Various optimizer fixes (Tom)
- Fix crash when `log_min_messages` is set to `DEBUG3` or above in `postgresql.conf` on Win32 (Bruce)
- Fix pgxs `-L` library path specification for Win32, Cygwin, OS X, AIX (Bruce)
- Check that SID is enabled while checking for Win32 admin privileges (Magnus)
- Properly reject out-of-range date inputs (Kris Jurka)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)
- Improve speed of **COPY IN** via libpq, by avoiding a kernel call per data line (Alon Goldshuv)
- Improve speed of `/contrib/tsearch2` index creation (Tom)

E.112. Release 8.1.2

Release Date

2006-01-09

This release contains a variety of fixes from 8.1.1. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.112.1. Migration to Version 8.1.2

A dump/restore is not required for those running 8.1.X. However, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.112.2. Changes

- Fix Windows code so that postmaster will continue rather than exit if there is no more room in ShmemBackendArray (Magnus)

  The previous behavior could lead to a denial-of-service situation if too many connection requests arrive close together. This applies *only* to the Windows port.
- Fix bug introduced in 8.0 that could allow ReadBuffer to return an already-used page as new, potentially causing loss of recently-committed data (Tom)
- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require **REINDEX** to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need **REINDEX** to fix existing indexes on textual columns if this has happened to you.
- Allow more flexible relocation of installation directories (Tom)

  Previous releases supported relocation only if all installation directory paths were the same except for the last component.
- Prevent crashes caused by the use of `ISO-8859-5` and `ISO-8859-9` encodings (Tatsuo)
- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Fix bug where COPY CSV mode considered any `\.` to terminate the copy data

  The new code requires `\.` to appear alone on a line, as per documentation.
- Make COPY CSV mode quote a literal data value of `\.` to ensure it cannot be interpreted as the end-of-data marker (Bruce)
- Various fixes for functions returning `RECORD`s (Tom)
- Fix processing of `postgresql.conf` so a final line with no newline is processed properly (Tom)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.
- Fix autovacuum crash when processing expression indexes
- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.113. Release 8.1.1

Release Date

2005-12-12

This release contains a variety of fixes from 8.1.0. For information about new features in the 8.1 major release, see Section E.114, « Release 8.1 ».

E.113.1. Migration to Version 8.1.1

A dump/restore is not required for those running 8.1.X.

E.113.2. Changes

- Fix incorrect optimizations of outer-join conditions (Tom)
- Fix problems with wrong reported column names in cases involving sub-selects flattened by the optimizer (Tom)
- Fix update failures in scenarios involving CHECK constraints, toasted columns, *and* indexes (Tom)

- Fix bgwriter problems after recovering from errors (Tom)

  The background writer was found to leak buffer pins after write errors. While not fatal in itself, this might lead to mysterious blockages of later VACUUM commands.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- `/contrib/tsearch2` and `/contrib/ltree` fixes (Teodor)
- Fix problems with translated error messages in languages that require word reordering, such as Turkish; also problems with unexpected truncation of output strings and wrong display of the smallest possible bigint value (Andrew, Tom)

  These problems only appeared on platforms that were using our `port/snprintf.c` code, which includes BSD variants if `--enable-nls` was given, and perhaps others. In addition, a different form of the translated-error-message problem could appear on Windows depending on which version of `libintl` was used.
- Re-allow `AM`/`PM`, `HH`, `HH12`, and `D` format specifiers for `to_char(time)` and `to_char(interval)`. (`to_char(interval)` should probably use `HH24`.) (Bruce)
- AIX, HPUX, and MSVC compile fixes (Tom, Hiroshi Saito)
- Optimizer improvements (Tom)
- Retry file reads and writes after Windows NO_SYSTEM_RESOURCES error (Qingqing Zhou)
- Prevent autovacuum from crashing during ANALYZE of expression index (Alvaro)
- Fix problems with ON COMMIT DELETE ROWS temp tables
- Fix problems when a trigger alters the output of a SELECT DISTINCT query
- Add 8.1.0 release note item on how to migrate invalid `UTF-8` byte sequences (Paul Lindner)

E.114. Release 8.1

Release Date

2005-11-08

E.114.1. Overview

Major changes in this release:

Improve concurrent access to the shared buffer cache (Tom)
: Access to the shared buffer cache was identified as a significant scalability problem, particularly on multi-CPU systems. In this release, the way that locking is done in the buffer manager has been overhauled to reduce lock contention and improve scalability. The buffer manager has also been changed to use a « clock sweep » replacement policy.

Allow index scans to use an intermediate in-memory bitmap (Tom)
: In previous releases, only a single index could be used to do lookups on a table. With this feature, if a query has **WHERE tab.col1 = 4 and tab.col2 = 9**, and there is no multicolumn index on `col1` and `col2`, but there is an index on `col1` and another on `col2`, it is possible to search both indexes and combine the results in memory, then do heap fetches for only the rows matching both the `col1` and `col2` restrictions. This is very useful in environments that have a lot of unstructured queries where it is impossible to create indexes that match all possible access conditions. Bitmap scans are useful even with a single index, as they reduce the amount of random access needed; a bitmap index scan is efficient for retrieving fairly large fractions of the complete table, whereas plain index scans are not.

Add two-phase commit (Heikki Linnakangas, Alvaro, Tom)
: Two-phase commit allows transactions to be "prepared" on several computers, and once all computers have successfully prepared their transactions (none failed), all transactions can be committed. Even if a machine crashes after a prepare, the prepared transaction can be committed after the machine is restarted. New syntax includes **PREPARE TRANSACTION** and **COMMIT/ROLLBACK PREPARED**. A new system view `pg_prepared_xacts` has also been added.

Create a new role system that replaces users and groups (Stephen Frost)
: Roles are a combination of users and groups. Like users, they can have login capability, and like groups, a role can have other roles as members. Roles basically remove the distinction between users and groups. For example, a role can:

  - Have login capability (optionally)

- Own objects
- Hold access permissions for database objects
- Inherit permissions from other roles it is a member of

Once a user logs into a role, she obtains capabilities of the login role plus any alled roles, and can use **SET ROLE** to switch to other roles she is a member of. This feature is a generalization of the SQL standard's concept of roles. This change also replaces pg_shadow and pg_group by new role-capable catalogs pg_authid and pg_auth_members. The old tables are redefined as read-only views on the new role tables.

Automatically use indexes for `MIN()` and `MAX()` (Tom)
In previous releases, the only way to use an index for `MIN()` or `MAX()` was to rewrite the query as **SELECT col FROM tab ORDER BY col LIMIT 1**. Index usage now happens automatically.

Move `/contrib/pg_autovacuum` into the main server (Alvaro)
Integrating autovacuum into the server allows it to be automatically started and stopped in sync with the database server, and allows autovacuum to be configured from `postgresql.conf`.

Add shared row level locks using **SELECT ... FOR SHARE** (Alvaro)
While PostgreSQL™'s MVCC locking allows **SELECT** to never be blocked by writers and therefore does not need shared row locks for typical operations, shared locks are useful for applications that require shared row locking. In particular this reduces the locking requirements imposed by referential integrity checks.

Add dependencies on shared objects, specifically roles (Alvaro)
This extension of the dependency mechanism prevents roles from being dropped while there are still database objects they own. Formerly it was possible to accidentally « orphan » objects by deleting their owner. While this could be recovered from, it was messy and unpleasant.

Improve performance for partitioned tables (Simon)
The new `constraint_exclusion` configuration parameter avoids lookups on child tables where constraints indicate that no matching rows exist in the child table.

This allows for a basic type of table partitioning. If child tables store separate key ranges and this is enforced using appropriate **CHECK** constraints, the optimizer will skip child table accesses when the constraint guarantees no matching rows exist in the child table.

E.114.2. Migration to Version 8.1

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

The 8.0 release announced that the `to_char()` function for intervals would be removed in 8.1. However, since no better API has been suggested, `to_char(interval)` has been enhanced in 8.1 and will remain in the server.

Observe the following incompatibilities:

- `add_missing_from` is now false by default (Neil)

  By default, we now generate an error if a table is used in a query without a `FROM` reference. The old behavior is still available, but the parameter must be set to 'true' to obtain it.

  It might be necessary to set `add_missing_from` to true in order to load an existing dump file, if the dump contains any views or rules created using the implicit-`FROM` syntax. This should be a one-time annoyance, because PostgreSQL™ 8.1 will convert such views and rules to standard explicit-`FROM` syntax. Subsequent dumps will therefore not have the problem.

- Cause input of a zero-length string (`''`) for float4/float8/oid to throw an error, rather than treating it as a zero (Neil)

  This change is consistent with the current handling of zero-length strings for integers. The schedule for this change was announced in 8.0.

- `default_with_oids` is now false by default (Neil)

  With this option set to false, user-created tables no longer have an OID column unless **WITH OIDS** is specified in **CREATE TABLE**. Though OIDs have existed in all releases of PostgreSQL™, their use is limited because they are only four bytes long and the counter is shared across all installed databases. The preferred way of uniquely identifying rows is via sequences and the SERIAL type, which have been supported since PostgreSQL™ 6.4.

- Add `E''` syntax so eventually ordinary strings can treat backslashes literally (Bruce)

  Currently PostgreSQL™ processes a backslash in a string literal as introducing a special escape sequence, e.g. `\n` or `\010`.

While this allows easy entry of special values, it is nonstandard and makes porting of applications from other databases more difficult. For this reason, the PostgreSQL™ project is planning to remove the special meaning of backslashes in strings. For backward compatibility and for users who want special backslash processing, a new string syntax has been created. This new string syntax is formed by writing an `E` immediately preceding the single quote that starts the string, e.g. `E'hi\n'`. While this release does not change the handling of backslashes in strings, it does add new configuration parameters to help users migrate applications for future releases:

- `standard_conforming_strings` -- does this release treat backslashes literally in ordinary strings?
- `escape_string_warning` -- warn about backslashes in ordinary (non-E) strings

The `standard_conforming_strings` value is read-only. Applications can retrieve the value to know how backslashes are processed. (Presence of the parameter can also be taken as an indication that `E''` string syntax is supported.) In a future release, `standard_conforming_strings` will be true, meaning backslashes will be treated literally in non-E strings. To prepare for this change, use `E''` strings in places that need special backslash processing, and turn on `escape_string_warning` to find additional strings that need to be converted to use `E''`. Also, use two single-quotes (`''`) to embed a literal single-quote in a string, rather than the PostgreSQL™-supported syntax of backslash single-quote (`\'`). The former is standards-conforming and does not require the use of the `E''` string syntax. You can also use the `$$` string syntax, which does not treat backslashes specially.

- Make **REINDEX DATABASE** reindex all indexes in the database (Tom)

  Formerly, **REINDEX DATABASE** reindexed only system tables. This new behavior seems more intuitive. A new command **REINDEX SYSTEM** provides the old functionality of reindexing just the system tables.

- Read-only large object descriptors now obey MVCC snapshot semantics

  When a large object is opened with `INV_READ` (and not `INV_WRITE`), the data read from the descriptor will now reflect a « snapshot » of the large object's state at the time of the transaction snapshot in use by the query that called `lo_open()`. To obtain the old behavior of always returning the latest committed data, include `INV_WRITE` in the mode flags for `lo_open()`.

- Add proper dependencies for arguments of sequence functions (Tom)

  In previous releases, sequence names passed to `nextval()`, `currval()`, and `setval()` were stored as simple text strings, meaning that renaming or dropping a sequence used in a `DEFAULT` clause made the clause invalid. This release stores all newly-created sequence function arguments as internal OIDs, allowing them to track sequence renaming, and adding dependency information that prevents improper sequence removal. It also makes such `DEFAULT` clauses immune to schema renaming and search path changes.

  Some applications might rely on the old behavior of run-time lookup for sequence names. This can still be done by explicitly casting the argument to text, for example `nextval('myseq'::text)`.

  Pre-8.1 database dumps loaded into 8.1 will use the old text-based representation and therefore will not have the features of OID-stored arguments. However, it is possible to update a database containing text-based `DEFAULT` clauses. First, save this query into a file, such as `fixseq.sql`:

```
SELECT  'ALTER TABLE ' ||
   pg_catalog.quote_ident(n.nspname) || '.' ||
   pg_catalog.quote_ident(c.relname) ||
   ' ALTER COLUMN ' || pg_catalog.quote_ident(a.attname) ||
   ' SET DEFAULT ' ||
   regexp_replace(d.adsrc,
                  $$val\(\(('[^']*')::text\)::regclass$$,
                  $$val(\1$$,
                  'g') ||
   ';'
FROM    pg_namespace n, pg_class c, pg_attribute a, pg_attrdef d
WHERE   n.oid = c.relnamespace AND
   c.oid = a.attrelid AND
   a.attrelid = d.adrelid AND
   a.attnum = d.adnum AND
   d.adsrc ~ $$val\(\('[^']*'::text\)::regclass$$;
```

  Next, run the query against a database to find what adjustments are required, like this for database `db1`:

```
psql -t -f fixseq.sql db1
```

This will show the **ALTER TABLE** commands needed to convert the database to the newer OID-based representation. If the commands look reasonable, run this to update the database:

```
psql -t -f fixseq.sql db1 | psql -e db1
```

This process must be repeated in each database to be updated.

- In psql, treat unquoted `\{digit}+` sequences as octal (Bruce)

  In previous releases, `\{digit}+` sequences were treated as decimal, and only `\0{digit}+` were treated as octal. This change was made for consistency.

- Remove grammar productions for prefix and postfix `%` and `^` operators (Tom)

  These have never been documented and complicated the use of the modulus operator (`%`) with negative numbers.

- Make `&<` and `&>` for polygons consistent with the box "over" operators (Tom)
- **CREATE LANGUAGE** can ignore the provided arguments in favor of information from pg_pltemplate (Tom)

  A new system catalog pg_pltemplate has been defined to carry information about the preferred definitions of procedural languages (such as whether they have validator functions). When an entry exists in this catalog for the language being created, **CREATE LANGUAGE** will ignore all its parameters except the language name and instead use the catalog information. This measure was taken because of increasing problems with obsolete language definitions being loaded by old dump files. As of 8.1, pg_dump will dump procedural language definitions as just **CREATE LANGUAGE** ***`name`***, relying on a template entry to exist at load time. We expect this will be a more future-proof representation.

- Make `pg_cancel_backend(int)` return a boolean rather than an integer (Neil)
- Some users are having problems loading UTF-8 data into 8.1.X. This is because previous versions allowed invalid UTF-8 byte sequences to be entered into the database, and this release properly accepts only valid UTF-8 sequences. One way to correct a dumpfile is to run the command **iconv -c -f UTF-8 -t UTF-8 -o cleanfile.sql dumpfile.sql**. The `-c` option removes invalid character sequences. A diff of the two files will show the sequences that are invalid. **iconv** reads the entire input file into memory so it might be necessary to use split to break up the dump into multiple smaller files for processing.

E.114.3. Additional Changes

Below you will find a detailed account of the additional changes between PostgreSQL™ 8.1 and the previous major release.

E.114.3.1. Performance Improvements

- Improve GiST and R-tree index performance (Neil)
- Improve the optimizer, including auto-resizing of hash joins (Tom)
- Overhaul internal API in several areas
- Change WAL record CRCs from 64-bit to 32-bit (Tom)

  We determined that the extra cost of computing 64-bit CRCs was significant, and the gain in reliability too marginal to justify it.

- Prevent writing large empty gaps in WAL pages (Tom)
- Improve spinlock behavior on SMP machines, particularly Opterons (Tom)
- Allow nonconsecutive index columns to be used in a multicolumn index (Tom)

  For example, this allows an index on columns a,b,c to be used in a query with **WHERE a = 4 and c = 10**.

- Skip WAL logging for **CREATE TABLE AS** / **SELECT INTO** (Simon)

  Since a crash during **CREATE TABLE AS** would cause the table to be dropped during recovery, there is no reason to WAL log as the table is loaded. (Logging still happens if WAL archiving is enabled, however.)

- Allow concurrent GiST index access (Teodor, Oleg)
- Add configuration parameter `full_page_writes` to control writing full pages to WAL (Bruce)

  To prevent partial disk writes from corrupting the database, PostgreSQL™ writes a complete copy of each database disk page to WAL the first time it is modified after a checkpoint. This option turns off that functionality for more speed. This is safe to use with battery-backed disk caches where partial page writes cannot happen.

- Use `O_DIRECT` if available when using `O_SYNC` for `wal_sync_method` (Itagaki Takahiro)

  `O_DIRECT` causes disk writes to bypass the kernel cache, and for WAL writes, this improves performance.
- Improve **COPY FROM** performance (Alon Goldshuv)

  This was accomplished by reading **COPY** input in larger chunks, rather than character by character.
- Improve the performance of `COUNT()`, `SUM`, `AVG()`, `STDDEV()`, and `VARIANCE()` (Neil, Tom)

E.114.3.2. Server Changes

- Prevent problems due to transaction ID (XID) wraparound (Tom)

  The server will now warn when the transaction counter approaches the wraparound point. If the counter becomes too close to wraparound, the server will stop accepting queries. This ensures that data is not lost before needed vacuuming is performed.
- Fix problems with object IDs (OIDs) conflicting with existing system objects after the OID counter has wrapped around (Tom)
- Add warning about the need to increase `max_fsm_relations` and `max_fsm_pages` during **VACUUM** (Ron Mayer)
- Add `temp_buffers` configuration parameter to allow users to determine the size of the local buffer area for temporary table access (Tom)
- Add session start time and client IP address to `pg_stat_activity` (Magnus)
- Adjust `pg_stat` views for bitmap scans (Tom)

  The meanings of some of the fields have changed slightly.
- Enhance `pg_locks` view (Tom)
- Log queries for client-side **PREPARE** and **EXECUTE** (Simon)
- Allow Kerberos name and user name case sensitivity to be specified in `postgresql.conf` (Magnus)
- Add configuration parameter `krb_server_hostname` so that the server host name can be specified as part of service principal (Todd Kover)

  If not set, any service principal matching an entry in the keytab can be used. This is new Kerberos matching behavior in this release.
- Add `log_line_prefix` options for millisecond timestamps (`%m`) and remote host (`%h`) (Ed L.)
- Add WAL logging for GiST indexes (Teodor, Oleg)

  GiST indexes are now safe for crash and point-in-time recovery.
- Remove old `*.backup` files when we do `pg_stop_backup()` (Bruce)

  This prevents a large number of `*.backup` files from existing in `pg_xlog/`.
- Add configuration parameters to control TCP/IP keep-alive times for idle, interval, and count (Oliver Jowett)

  These values can be changed to allow more rapid detection of lost client connections.
- Add per-user and per-database connection limits (Petr Jelinek)

  Using **ALTER USER** and **ALTER DATABASE**, limits can now be enforced on the maximum number of sessions that can concurrently connect as a specific user or to a specific database. Setting the limit to zero disables user or database connections.
- Allow more than two gigabytes of shared memory and per-backend work memory on 64-bit machines (Koichi Suzuki)
- New system catalog pg_pltemplate allows overriding obsolete procedural-language definitions in dump files (Tom)

E.114.3.3. Query Changes

- Add temporary views (Koju Iijima, Neil)
- Fix **HAVING** without any aggregate functions or **GROUP BY** so that the query returns a single group (Tom)

  Previously, such a case would treat the **HAVING** clause the same as a **WHERE** clause. This was not per spec.
- Add **USING** clause to allow additional tables to be specified to **DELETE** (Euler Taveira de Oliveira, Neil)

In prior releases, there was no clear method for specifying additional tables to be used for joins in a **DELETE** statement. **UPDATE** already has a `FROM` clause for this purpose.

- Add support for `\x` hex escapes in backend and ecpg strings (Bruce)

  This is just like the standard C `\x` escape syntax. Octal escapes were already supported.

- Add **BETWEEN SYMMETRIC** query syntax (Pavel Stehule)

  This feature allows **BETWEEN** comparisons without requiring the first value to be less than the second. For example, **2 BETWEEN [ASYMMETRIC] 3 AND 1** returns false, while **2 BETWEEN SYMMETRIC 3 AND 1** returns true. **BETWEEN ASYMMETRIC** was already supported.

- Add **NOWAIT** option to **SELECT ... FOR UPDATE/SHARE** (Hans-Juergen Schoenig)

  While the `statement_timeout` configuration parameter allows a query taking more than a certain amount of time to be canceled, the **NOWAIT** option allows a query to be canceled as soon as a **SELECT ... FOR UPDATE/SHARE** command cannot immediately acquire a row lock.

E.114.3.4. Object Manipulation Changes

- Track dependencies of shared objects (Alvaro)

  PostgreSQL™ allows global tables (users, databases, tablespaces) to reference information in multiple databases. This addition adds dependency information for global tables, so, for example, user ownership can be tracked across databases, so a user who owns something in any database can no longer be removed. Dependency tracking already existed for database-local objects.

- Allow limited **ALTER OWNER** commands to be performed by the object owner (Stephen Frost)

  Prior releases allowed only superusers to change object owners. Now, ownership can be transferred if the user executing the command owns the object and would be able to create it as the new owner (that is, the user is a member of the new owning role and that role has the CREATE permission that would be needed to create the object afresh).

- Add **ALTER** object **SET SCHEMA** capability for some object types (tables, functions, types) (Bernd Helmle)

  This allows objects to be moved to different schemas.

- Add **ALTER TABLE ENABLE/DISABLE TRIGGER** to disable triggers (Satoshi Nagayasu)

E.114.3.5. Utility Command Changes

- Allow **TRUNCATE** to truncate multiple tables in a single command (Alvaro)

  Because of referential integrity checks, it is not allowed to truncate a table that is part of a referential integrity constraint. Using this new functionality, **TRUNCATE** can be used to truncate such tables, if both tables involved in a referential integrity constraint are truncated in a single **TRUNCATE** command.

- Properly process carriage returns and line feeds in **COPY CSV** mode (Andrew)

  In release 8.0, carriage returns and line feeds in **CSV COPY TO** were processed in an inconsistent manner. (This was documented on the TODO list.)

- Add **COPY WITH CSV HEADER** to allow a header line as the first line in **COPY** (Andrew)

  This allows handling of the common **CSV** usage of placing the column names on the first line of the data file. For **COPY TO**, the first line contains the column names, and for **COPY FROM**, the first line is ignored.

- On Windows, display better sub-second precision in **EXPLAIN ANALYZE** (Magnus)

- Add trigger duration display to **EXPLAIN ANALYZE** (Tom)

  Prior releases included trigger execution time as part of the total execution time, but did not show it separately. It is now possible to see how much time is spent in each trigger.

- Add support for `\x` hex escapes in **COPY** (Sergey Ten)

  Previous releases only supported octal escapes.

- Make **SHOW ALL** include variable descriptions (Matthias Schmidt)

SHOW varname still only displays the variable's value and does not include the description.

- Make initdb create a new standard database called `postgres`, and convert utilities to use `postgres` rather than `template1` for standard lookups (Dave)

  In prior releases, `template1` was used both as a default connection for utilities like createuser, and as a template for new databases. This caused **CREATE DATABASE** to sometimes fail, because a new database cannot be created if anyone else is in the template database. With this change, the default connection database is now `postgres`, meaning it is much less likely someone will be using `template1` during **CREATE DATABASE**.

- Create new reindexdb command-line utility by moving `/contrib/reindexdb` into the server (Euler Taveira de Oliveira)

E.114.3.6. Data Type and Function Changes

- Add `MAX()` and `MIN()` aggregates for array types (Koju Iijima)
- Fix `to_date()` and `to_timestamp()` to behave reasonably when `CC` and `YY` fields are both used (Karel Zak)

  If the format specification contains `CC` and a year specification is `YYY` or longer, ignore the `CC`. If the year specification is `YY` or shorter, interpret `CC` as the previous century.

- Add `md5(bytea)` (Abhijit Menon-Sen)

  `md5(text)` already existed.

- Add support for **numeric ^ numeric** based on `power(numeric, numeric)`

  The function already existed, but there was no operator assigned to it.

- Fix NUMERIC modulus by properly truncating the quotient during computation (Bruce)

  In previous releases, modulus for large values sometimes returned negative results due to rounding of the quotient.

- Add a function `lastval()` (Dennis Björklund)

  `lastval()` is a simplified version of `currval()`. It automatically determines the proper sequence name based on the most recent `nextval()` or `setval()` call performed by the current session.

- Add `to_timestamp(DOUBLE PRECISION) (Michael Glaesemann)`

  Converts Unix seconds since 1970 to a TIMESTAMP WITH TIMEZONE.

- Add `pg_postmaster_start_time()` function (Euler Taveira de Oliveira, Matthias Schmidt)
- Allow the full use of time zone names in **AT TIME ZONE**, not just the short list previously available (Magnus)

  Previously, only a predefined list of time zone names were supported by **AT TIME ZONE**. Now any supported time zone name can be used, e.g.:

  ```
  SELECT CURRENT_TIMESTAMP AT TIME ZONE 'Europe/London';
  ```

  In the above query, the time zone used is adjusted based on the daylight saving time rules that were in effect on the supplied date.

- Add `GREATEST()` and `LEAST()` variadic functions (Pavel Stehule)

  These functions take a variable number of arguments and return the greatest or least value among the arguments.

- Add `pg_column_size()` (Mark Kirkwood)

  This returns storage size of a column, which might be compressed.

- Add `regexp_replace()` (Atsushi Ogawa)

  This allows regular expression replacement, like sed. An optional flag argument allows selection of global (replace all) and case-insensitive modes.

- Fix interval division and multiplication (Bruce)

  Previous versions sometimes returned unjustified results, like **'4 months'::interval / 5** returning **'1 mon -6 days'**.

- Fix roundoff behavior in timestamp, time, and interval output (Tom)

  This fixes some cases in which the seconds field would be shown as `60` instead of incrementing the higher-order fields.

- Add a separate day field to type interval so a one day interval can be distinguished from a 24 hour interval (Michael Glaesemann)

  Days that contain a daylight saving time adjustment are not 24 hours long, but typically 23 or 25 hours. This change creates a conceptual distinction between intervals of « so many days » and intervals of « so many hours ». Adding `1 day` to a timestamp now gives the same local time on the next day even if a daylight saving time adjustment occurs between, whereas adding `24 hours` will give a different local time when this happens. For example, under US DST rules:

  ```
  '2005-04-03 00:00:00-05' + '1 day' = '2005-04-04 00:00:00-04'
  '2005-04-03 00:00:00-05' + '24 hours' = '2005-04-04 01:00:00-04'
  ```

- Add `justify_days()` and `justify_hours()` (Michael Glaesemann)

  These functions, respectively, adjust days to an appropriate number of full months and days, and adjust hours to an appropriate number of full days and hours.

- Move `/contrib/dbsize` into the backend, and rename some of the functions (Dave Page, Andreas Pflug)

  - `pg_tablespace_size()`
  - `pg_database_size()`
  - `pg_relation_size()`
  - `pg_total_relation_size()`
  - `pg_size_pretty()`

  `pg_total_relation_size()` includes indexes and TOAST tables.

- Add functions for read-only file access to the cluster directory (Dave Page, Andreas Pflug)

  - `pg_stat_file()`
  - `pg_read_file()`
  - `pg_ls_dir()`

- Add `pg_reload_conf()` to force reloading of the configuration files (Dave Page, Andreas Pflug)
- Add `pg_rotate_logfile()` to force rotation of the server log file (Dave Page, Andreas Pflug)
- Change `pg_stat_*` views to include TOAST tables (Tom)

E.114.3.7. Encoding and Locale Changes

- Rename some encodings to be more consistent and to follow international standards (Bruce)

  - `UNICODE` is now `UTF8`
  - `ALT` is now `WIN866`
  - `WIN` is now `WIN1251`
  - `TCVN` is now `WIN1258`

  The original names still work.

- Add support for `WIN1252` encoding (Roland Volkmann)
- Add support for four-byte `UTF8` characters (John Hansen)

  Previously only one, two, and three-byte `UTF8` characters were supported. This is particularly important for support for some Chinese character sets.

- Allow direct conversion between `EUC_JP` and `SJIS` to improve performance (Atsushi Ogawa)
- Allow the UTF8 encoding to work on Windows (Magnus)

This is done by mapping UTF8 to the Windows-native UTF16 implementation.

E.114.3.8. General Server-Side Language Changes

- Fix **ALTER LANGUAGE RENAME** (Sergey Yatskevich)
- Allow function characteristics, like strictness and volatility, to be modified via **ALTER FUNCTION** (Neil)
- Increase the maximum number of function arguments to 100 (Tom)
- Allow SQL and PL/pgSQL functions to use **OUT** and **INOUT** parameters (Tom)

  OUT is an alternate way for a function to return values. Instead of using **RETURN**, values can be returned by assigning to parameters declared as **OUT** or **INOUT**. This is notationally simpler in some cases, particularly so when multiple values need to be returned. While returning multiple values from a function was possible in previous releases, this greatly simplifies the process. (The feature will be extended to other server-side languages in future releases.)
- Move language handler functions into the `pg_catalog` schema

  This makes it easier to drop the public schema if desired.
- Add `SPI_getnspname()` to SPI (Neil)

E.114.3.9. PL/pgSQL Server-Side Language Changes

- Overhaul the memory management of PL/pgSQL functions (Neil)

  The parsetree of each function is now stored in a separate memory context. This allows this memory to be easily reclaimed when it is no longer needed.
- Check function syntax at **CREATE FUNCTION** time, rather than at runtime (Neil)

  Previously, most syntax errors were reported only when the function was executed.
- Allow **OPEN** to open non-**SELECT** queries like **EXPLAIN** and **SHOW** (Tom)
- No longer require functions to issue a **RETURN** statement (Tom)

  This is a byproduct of the newly added **OUT** and **INOUT** functionality. **RETURN** can be omitted when it is not needed to provide the function's return value.
- Add support for an optional **INTO** clause to PL/pgSQL's **EXECUTE** statement (Pavel Stehule, Neil)
- Make **CREATE TABLE AS** set **ROW_COUNT** (Tom)
- Define `SQLSTATE` and `SQLERRM` to return the `SQLSTATE` and error message of the current exception (Pavel Stehule, Neil)

  These variables are only defined inside exception blocks.
- Allow the parameters to the **RAISE** statement to be expressions (Pavel Stehule, Neil)
- Add a loop **CONTINUE** statement (Pavel Stehule, Neil)
- Allow block and loop labels (Pavel Stehule)

E.114.3.10. PL/Perl Server-Side Language Changes

- Allow large result sets to be returned efficiently (Abhijit Menon-Sen)

  This allows functions to use `return_next()` to avoid building the entire result set in memory.
- Allow one-row-at-a-time retrieval of query results (Abhijit Menon-Sen)

  This allows functions to use `spi_query()` and `spi_fetchrow()` to avoid accumulating the entire result set in memory.
- Force PL/Perl to handle strings as `UTF8` if the server encoding is `UTF8` (David Kamholz)
- Add a validator function for PL/Perl (Andrew)

  This allows syntax errors to be reported at definition time, rather than execution time.
- Allow PL/Perl to return a Perl array when the function returns an array type (Andrew)

This basically maps PostgreSQL™ arrays to Perl arrays.

- Allow Perl nonfatal warnings to generate **NOTICE** messages (Andrew)
- Allow Perl's `strict` mode to be enabled (Andrew)

E.114.3.11. psql Changes

- Add **\set ON_ERROR_ROLLBACK** to allow statements in a transaction to error without affecting the rest of the transaction (Greg Sabino Mullane)

  This is basically implemented by wrapping every statement in a sub-transaction.
- Add support for `\x` hex strings in psql variables (Bruce)

  Octal escapes were already supported.
- Add support for **troff -ms** output format (Roger Leigh)
- Allow the history file location to be controlled by `HISTFILE` (Andreas Seltenreich)

  This allows configuration of per-database history storage.
- Prevent **\x** (expanded mode) from affecting the output of **\d tablename** (Neil)
- Add `-L` option to psql to log sessions (Lorne Sunley)

  This option was added because some operating systems do not have simple command-line activity logging functionality.
- Make **\d** show the tablespaces of indexes (Qingqing Zhou)
- Allow psql help (**\h**) to make a best guess on the proper help information (Greg Sabino Mullane)

  This allows the user to just add **\h** to the front of the syntax error query and get help on the supported syntax. Previously any additional query text beyond the command name had to be removed to use **\h**.
- Add **\pset numericlocale** to allow numbers to be output in a locale-aware format (Eugen Nedelcu)

  For example, using `C` locale `100000` would be output as `100,000.0` while a European locale might output this value as `100.000,0`.
- Make startup banner show both server version number and psql's version number, when they are different (Bruce)

  Also, a warning will be shown if the server and psql are from different major releases.

E.114.3.12. pg_dump Changes

- Add `-n` / `--schema` switch to pg_restore (Richard van den Berg)

  This allows just the objects in a specified schema to be restored.
- Allow pg_dump to dump large objects even in text mode (Tom)

  With this change, large objects are now always dumped; the former `-b` switch is a no-op.
- Allow pg_dump to dump a consistent snapshot of large objects (Tom)
- Dump comments for large objects (Tom)
- Add `--encoding` to pg_dump (Magnus Hagander)

  This allows a database to be dumped in an encoding that is different from the server's encoding. This is valuable when transferring the dump to a machine with a different encoding.
- Rely on pg_pltemplate for procedural languages (Tom)

  If the call handler for a procedural language is in the `pg_catalog` schema, pg_dump does not dump the handler. Instead, it dumps the language using just **CREATE LANGUAGE *name***, relying on the pg_pltemplate catalog to provide the language's creation parameters at load time.

E.114.3.13. libpq Changes

- Add a `PGPASSFILE` environment variable to specify the password file's filename (Andrew)
- Add `lo_create()`, that is similar to `lo_creat()` but allows the OID of the large object to be specified (Tom)
- Make libpq consistently return an error to the client application on `malloc()` failure (Neil)

E.114.3.14. Source Code Changes

- Fix pgxs to support building against a relocated installation
- Add spinlock support for the Itanium processor using Intel compiler (Vikram Kalsi)
- Add Kerberos 5 support for Windows (Magnus)
- Add Chinese FAQ (laser@pgsqldb.com)
- Rename Rendezvous to Bonjour to match OS/X feature renaming (Bruce)
- Add support for `fsync_writethrough` on Darwin (Chris Campbell)
- Streamline the passing of information within the server, the optimizer, and the lock system (Tom)
- Allow pg_config to be compiled using MSVC (Andrew)

  This is required to build DBD::Pg using MSVC.
- Remove support for Kerberos V4 (Magnus)

  Kerberos 4 had security vulnerabilities and is no longer maintained.
- Code cleanups (Coverity static analysis performed by EnterpriseDB)
- Modify `postgresql.conf` to use documentation defaults `on`/`off` rather than `true`/`false` (Bruce)
- Enhance pg_config to be able to report more build-time values (Tom)
- Allow libpq to be built thread-safe on Windows (Dave Page)
- Allow IPv6 connections to be used on Windows (Andrew)
- Add Server Administration documentation about I/O subsystem reliability (Bruce)
- Move private declarations from `gist.h` to `gist_private.h` (Neil)

  In previous releases, `gist.h` contained both the public GiST API (intended for use by authors of GiST index implementations) as well as some private declarations used by the implementation of GiST itself. The latter have been moved to a separate file, `gist_private.h`. Most GiST index implementations should be unaffected.
- Overhaul GiST memory management (Neil)

  GiST methods are now always invoked in a short-lived memory context. Therefore, memory allocated via `palloc()` will be reclaimed automatically, so GiST index implementations do not need to manually release allocated memory via `pfree()`.

E.114.3.15. Contrib Changes

- Add `/contrib/pg_buffercache` contrib module (Mark Kirkwood)

  This displays the contents of the buffer cache, for debugging and performance tuning purposes.
- Remove `/contrib/array` because it is obsolete (Tom)
- Clean up the `/contrib/lo` module (Tom)
- Move `/contrib/findoidjoins` to `/src/tools` (Tom)
- Remove the `<<`, `>>`, `&<`, and `&>` operators from `/contrib/cube`

  These operators were not useful.
- Improve `/contrib/btree_gist` (Janko Richter)
- Improve `/contrib/pgbench` (Tomoaki Sato, Tatsuo)

  There is now a facility for testing with SQL command scripts given by the user, instead of only a hard-wired command sequence.

- Improve `/contrib/pgcrypto` (Marko Kreen)
  - Implementation of OpenPGP symmetric-key and public-key encryption

    Both RSA and Elgamal public-key algorithms are supported.
  - Stand alone build: include SHA256/384/512 hashes, Fortuna PRNG
  - OpenSSL build: support 3DES, use internal AES with OpenSSL < 0.9.7
  - Take build parameters (OpenSSL, zlib) from `configure` result

    There is no need to edit the `Makefile` anymore.
  - Remove support for `libmhash` and `libmcrypt`

E.115. Release 8.0.26

Release Date

2010-10-04

This release contains a variety of fixes from 8.0.25. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

This is expected to be the last PostgreSQL™ release in the 8.0.X series. Users are encouraged to update to a newer release branch soon.

E.115.1. Migration to Version 8.0.26

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see Section E.119, « Release 8.0.22 ».

E.115.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).
- Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it's intended to be used with (Heikki Linnakangas, Tom Lane)
- Fix « cannot handle unplanned sub-select » error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.
- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)
- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.
- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.

- Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)
- Fix possible data corruption in **ALTER TABLE ... SET TABLESPACE** when archiving is enabled (Jeff Davis)
- Allow **CREATE DATABASE** and **ALTER DATABASE ... SET TABLESPACE** to be interrupted by query-cancel (Guillaume Lelarge)
- In PL/Python, defend against null pointer results from `PyCObject_AsVoidPtr` and `PyCObject_FromVoidPtr` (Peter Eisentraut)
- Improve `contrib/dblink`'s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after « duplicate connection name » errors in `contrib/dblink` (Itagaki Takahiro)
- Fix `contrib/dblink` to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)
- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.

E.116. Release 8.0.25

Release Date

2010-05-17

This release contains a variety of fixes from 8.0.24. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

The PostgreSQL™ community will stop releasing updates for the 8.0.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.116.1. Migration to Version 8.0.25

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see Section E.119, « Release 8.0.22 ».

E.116.2. Changes

- Enforce restrictions in `plperl` using an opmask applied to the whole interpreter, instead of using `Safe.pm` (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that `Safe.pm` is too insecure to rely on for making `plperl` trustable. This change removes use of `Safe.pm` altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's `strict` pragma in a natural way in `plperl`, and that Perl's `$a` and `$b` variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted « normal » Tcl interpreter unless we are really going to execute a `pltclu` function. (CVE-2010-1170)

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran `ALTER USER ... RESET ALL` for himself, or `ALTER DATABASE ... RESET ALL` for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the **ALTER** will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a `CONTEXT` addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.
- Update pl/perl's `ppport.h` for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)
- Ensure that `contrib/pgstattuple` functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that `shmget()` returns `EINVAL` for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.
- Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

E.117. Release 8.0.24

Release Date

2010-03-15

This release contains a variety of fixes from 8.0.23. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

The PostgreSQL™ community will stop releasing updates for the 8.0.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.117.1. Migration to Version 8.0.24

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see Section E.119, « Release 8.0.22 ».

E.117.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for an SSL connection (Magnus)

  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.
- Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)
- Fix server memory leak associated with use of savepoints and a client encoding different from server's encoding (Tom)
- Make `substring()` for bit types treat any negative length as meaning « all the rest of the string » (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix the `STOP WAL LOCATION` entry in backup history files to report the next WAL segment's name when the end location is exactly at a segment boundary (Itagaki Takahiro)
- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains

spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.

- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Add `volatile` markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.
- Prevent crash in `contrib/dblink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)
- Fix assorted crashes in `contrib/xml2` caused by sloppy memory management (Tom)
- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.118. Release 8.0.23

Release Date

2009-12-14

This release contains a variety of fixes from 8.0.22. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.118.1. Migration to Version 8.0.23

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see Section E.119, « Release 8.0.22 ».

E.118.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

  This change prevents allegedly-immutable index functions from possibly subverting a superuser's session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).
- Fix possible crash during backend-startup-time cache initialization (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)

  This fix prevents a PANIC if a `VACUUM FULL` is canceled after it's already committed its tuple movements, as well as transient errors if a plain `VACUUM` is interrupted after having truncated the table.
- Fix possible crash due to integer overflow in hash table size calculation (Tom)

  This could occur with extremely large planner estimates for the size of a hashjoin's result.
- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix PAM password processing to be more robust (Tom)

  The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Fix rare crash in exception processing in PL/Python (Peter)
- Ensure psql's flex module is compiled with the correct system header definitions (Tom)

  This fixes build failures on platforms where `--enable-largefile` causes incompatible changes in the generated code.

- Make the postmaster ignore any `application_name` parameter in connection request packets, to improve compatibility with future libpq versions (Tom)
- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.

E.119. Release 8.0.22

Release Date

2009-09-09

This release contains a variety of fixes from 8.0.21. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.119.1. Migration to Version 8.0.22

A dump/restore is not required for those running 8.0.X. However, if you have any hash indexes on interval columns, you must **REINDEX** them after updating to 8.0.22. Also, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.119.2. Changes

- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)
- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
- Fix hash calculation for data type interval (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must **REINDEX** them after updating.
- Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with `'HH'/'HH12'` (Heikki)

  It was previously handled as `'th'` (lowercase).
- Fix overflow for `INTERVAL 'x ms'` when *x* is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.
- Fix money data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)
- Properly round datetime input like `00:12:57.999999999999999999999999999` (Tom)
- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)
- Fix portability issues in plperl initialization (Andrew Dunstan)
- Fix pg_ctl to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)
- Fix `contrib/xml2`'s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)
- Improve robustness of libpq's code to recover from errors during **COPY FROM STDIN** (Tom)
- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)
- Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.120. Release 8.0.21

Release Date

2009-03-16

This release contains a variety of fixes from 8.0.20. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.120.1. Migration to Version 8.0.21

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.120.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)

  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that *any* error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)

- Disallow **CREATE CONVERSION** with the wrong encodings for the specified conversion function (Heikki)

  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.

- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)
- Add `MUST` (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

E.121. Release 8.0.20

Release Date

2009-02-02

This release contains a variety of fixes from 8.0.19. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.121.1. Migration to Version 8.0.20

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.121.2. Changes

- Improve handling of URLs in `headline()` function (Teodor)
- Improve handling of overlength headlines in `headline()` function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Avoid unnecessary locking of small tables in **VACUUM** (Heikki)
- Fix uninitialized variables in `contrib/tsearch2`'s `get_covers()` function (Teodor)
- Make all documentation reference `pgsql-bugs` and/or `pgsql-hackers` as appropriate, instead of the now-decommissioned `pgsql-ports` and `pgsql-patches` mailing lists (Tom)
- Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections in Switzerland, Cuba)

E.122. Release 8.0.19

Release Date

2008-11-03

This release contains a variety of fixes from 8.0.18. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.122.1. Migration to Version 8.0.19

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.122.2. Changes

- Fix backend crash when the client encoding cannot represent a localized error message (Tom)

  We have addressed similar issues before, but it would still fail if the « character has no equivalent » message itself couldn't be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an `--enable-integer-datetimes` build (Ron Mayer)
- Ensure `SPI_getvalue` and `SPI_getbinval` behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)

  This situation is normal when a table has had columns added or removed, but these two functions didn't handle it properly. The only likely consequence is an incorrect error indication.
- Fix ecpg's parsing of **CREATE USER** (Michael)
- Fix recent breakage of `pg_ctl restart` (Tom)
- Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil, Mauritius, Syria)

E.123. Release 8.0.18

Release Date

2008-09-22

This release contains a variety of fixes from 8.0.17. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.123.1. Migration to Version 8.0.18

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.123.2. Changes

- Widen local lock counters from 32 to 64 bits (Tom)

  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected « lock is already held » errors.
- Add checks in executor startup to ensure that the tuples produced by an **INSERT** or **UPDATE** will match the target table's current rowtype (Tom)

  ALTER COLUMN TYPE, followed by re-use of a previously cached plan, could produce this type of situation. The check protects against data corruption and/or crashes that could ensue.

- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Fix bug in backwards scanning of a cursor on a `SELECT DISTINCT ON` query (Tom)
- Fix planner to estimate that `GROUP BY` expressions yielding boolean results always result in two groups, regardless of the expressions' contents (Tom)

  This is very substantially more accurate than the regular `GROUP BY` estimate for certain boolean tests like *`col`* `IS NULL`.
- Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)
- Fix PL/Python to work with Python 2.5

  This is a back-port of fixes made during the 8.2 development cycle.
- Improve pg_dump and pg_restore's error reporting after failure to send a SQL command (Tom)
- Fix pg_ctl to properly preserve postmaster command-line arguments across a `restart` (Bruce)
- Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.124. Release 8.0.17

Release Date

2008-06-12

This release contains one serious bug fix over 8.0.16. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.124.1. Migration to Version 8.0.17

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.124.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)

  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL™ to reject what had been a valid **SELECT DISTINCT** view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

E.125. Release 8.0.16

Release Date

never released

This release contains a variety of fixes from 8.0.15. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.125.1. Migration to Version 8.0.16

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.125.2. Changes

- Fix **ALTER TABLE ADD COLUMN ... PRIMARY KEY** so that the new column is correctly checked to see if it's been initialized to all non-nulls (Brendan Jurd)

  Previous versions neglected to check this requirement at all.
- Fix possible **CREATE TABLE** failure when alling the « same » constraint from multiple parent relations that alled that constraint from a common ancestor (Tom)
- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic « Yo » characters (`e` and `E` with two dots) (Sergey Burladyan)
- Fix a few datatype input functions that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched `ORDER BY` and `DISTINCT` expressions.
- Fix a corner case in regular-expression substring matching (`substring(`*`string`*` from `*`pattern`*`)`) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn't got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since `(bar)` isn't matched, but it was mistakenly returning the whole-pattern match instead (ie, `foo`).
- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, Argentina/San_Luis, and Chile)
- Fix incorrect result from ecpg's `PGTYPEStimestamp_sub()` function (Michael)
- Fix core dump in `contrib/xml2`'s `xpath_table()` function when the input query returns a NULL value (Tom)
- Fix `contrib/xml2`'s makefile to not override `CFLAGS` (Tom)
- Fix `DatumGetBool` macro to not fail with gcc 4.3 (Tom)

  This problem affects « old style » (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.
- Fix longstanding **LISTEN**/**NOTIFY** race condition (Tom)

  In rare cases a session that had just executed a **LISTEN** might not get a notification, even though one would be expected because the concurrent transaction executing **NOTIFY** was observed to commit later.

  A side effect of the fix is that a transaction that has executed a not-yet-committed **LISTEN** command will not see any row in pg_listener for the **LISTEN**, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.
- Fix rare crash when an error occurs during a query using a hash index (Heikki)
- Fix input of datetime values for February 29 in years BC (Tom)

  The former coding was mistaken about which years were leap years.
- Fix « unrecognized node type » error in some variants of **ALTER OWNER** (Tom)
- Fix pg_ctl to correctly extract the postmaster's port number from command-line options (Itagaki Takahiro, Tom)

  Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.
- Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)

  This is known to be necessary when building PostgreSQL™ with gcc 4.3 or later.
- Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)

  An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.
- Fix libpq to handle NOTICE messages correctly during COPY OUT (Tom)

  This failure has only been observed to occur when a user-defined datatype's output routine issues a NOTICE, but there is no guarantee it couldn't happen due to other causes.

E.126. Release 8.0.15

Release Date

2008-01-07

This release contains a variety of fixes from 8.0.14, including fixes for significant security issues. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

This is the last 8.0.X release for which the PostgreSQL™ community will produce binary packages for Windows™. Windows users are encouraged to move to 8.2.X or later, since there are Windows-specific fixes in 8.2.X that are impractical to back-port. 8.0.X will continue to be supported on other platforms.

E.126.1. Migration to Version 8.0.15

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.126.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running **VACUUM**, **ANALYZE**, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as **VACUUM FULL**, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including **VACUUM**, **ANALYZE**, **REINDEX**, and **CLUSTER**) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of **SET SESSION AUTHORIZATION** and **SET ROLE** is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 8.0.14 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)

- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)

- Preserve the tablespace of indexes that are rebuilt by **ALTER TABLE ... ALTER COLUMN TYPE** (Tom)

- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.

- Make **VACUUM** not use all of `maintenance_work_mem` when the table is too small for it to be useful (Alvaro)

- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)

- Fix PL/Perl to cope when platform's Perl defines type `bool` as `int` rather than `char` (Tom)

  While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X™ 10.5.

- Fix PL/Python to not crash on long exception messages (Alvaro)

- Fix pg_dump to correctly handle allance child tables that have default expressions different from their parent's (Tom)

- ecpg parser fixes (Michael)

- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)

- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)

- Fix crash of `to_tsvector()` on huge input strings (Teodor)
- Require a specific version of Autoconf™ to be used when re-generating the **configure** script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf™ and PostgreSQL™ versions. You can remove the version check if you really want to use a different Autoconf™ version, but it's your responsibility whether the result works or not.

E.127. Release 8.0.14

Release Date

2007-09-17

This release contains a variety of fixes from 8.0.13. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.127.1. Migration to Version 8.0.14

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.127.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent **VACUUM** on the same table (Tom)
- Make **CREATE DOMAIN ... DEFAULT NULL** work properly (Tom)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)
- Prevent **CLUSTER** from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand's upcoming changes (Tom)
- Windows socket improvements (Magnus)
- Suppress timezone name (`%Z`) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.128. Release 8.0.13

Release Date

2007-04-23

This release contains a variety of fixes from 8.0.12, including a security fix. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.128.1. Migration to Version 8.0.13

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.128.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and

operators (Tom)

This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See **CREATE FUNCTION** for more information.

- `/contrib/tsearch2` crash fixes (Teodor)
- Fix potential-data-corruption bug in how **VACUUM FULL** handles **UPDATE** chains (Tom, Pavan Deolasee)
- Fix PANIC during enlargement of a hash index (bug introduced in 8.0.10) (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.129. Release 8.0.12

Release Date

2007-02-07

This release contains one fix from 8.0.11. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.129.1. Migration to Version 8.0.12

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.129.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)

E.130. Release 8.0.11

Release Date

2007-02-05

This release contains a variety of fixes from 8.0.10, including a security fix. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.130.1. Migration to Version 8.0.11

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.130.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)

  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it's declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.131. Release 8.0.10

Release Date

2007-01-08

This release contains a variety of fixes from 8.0.9. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.131.1. Migration to Version 8.0.10

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.131.2. Changes

- Improve handling of `getaddrinfo()` on AIX (Tom)

  This fixes a problem with starting the statistics collector, among other things.
- Fix « failed to re-find parent key » errors in **VACUUM** (Tom)
- Fix race condition for truncation of a large relation across a gigabyte boundary by **VACUUM** (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix possible deadlock in Windows signal handling (Teodor)
- Fix error when constructing an `ARRAY[]` made up of multiple empty elements (Tom)
- Fix ecpg memory leak during connection (Michael)
- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.
- Update timezone database

  This affects Australian and Canadian daylight-savings rules in particular.

E.132. Release 8.0.9

Release Date

2006-10-16

This release contains a variety of fixes from 8.0.8. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.132.1. Migration to Version 8.0.9

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.132.2. Changes

- Fix crash when referencing `NEW` row values in rule WHERE expressions (Tom)
- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix mishandling of AFTER triggers when query contains a SQL function returning multiple rows (Tom)
- Fix **ALTER TABLE ... TYPE** to recheck `NOT NULL` for `USING` clause (Tom)
- Fix `string_to_array()` to handle overlapping matches for the separator string

For example, `string_to_array('123xx456xxx789', 'xx')`.

- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Numerous robustness fixes in ecpg (Joachim Wieland)
- Fix backslash escaping in /contrib/dbmirror
- Fix instability of statistics collection on Win32 (Tom, Andrew)
- Fixes for `AIX` and Intel™ compilers (Tom)

E.133. Release 8.0.8

Release Date

2006-05-23

This release contains a variety of fixes from 8.0.7, including patches for extremely serious security issues. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.133.1. Migration to Version 8.0.8

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.133.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL™ has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations and `standard_conforming_strings`

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL™ connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping « by hand » should be modified to rely on library routines instead.

- Fix some incorrect encoding conversion functions

  `win1251_to_iso`, `alt_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.

- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)
- Fix bug that sometimes caused OR'd index scans to miss rows they should have returned

- Fix WAL replay for case where a btree index has been truncated
- Fix `SIMILAR TO` for patterns involving `|` (Tom)
- Fix **SELECT INTO** and **CREATE TABLE AS** to create tables in the default tablespace, not the base directory (Kris Jurka)
- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
- Fix for Bonjour on Intel Macs (Ashley Clark)
- Fix various minor memory leaks
- Fix problem with password prompting on some Win32 systems (Robert Kinberg)

E.134. Release 8.0.7

Release Date

2006-02-14

This release contains a variety of fixes from 8.0.6. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.134.1. Migration to Version 8.0.7

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see Section E.135, « Release 8.0.6 ».

E.134.2. Changes

- Fix potential crash in **SET SESSION AUTHORIZATION** (CVE-2006-0553)

  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 8.0.4, 7.4.9, and 7.3.11 releases.
- Fix race condition that could lead to « file already exists » errors during pg_clog and pg_subtrans file creation (Tom)
- Fix cases that could lead to crashes if a cache-invalidation message arrives at just the wrong time (Tom)
- Properly check `DOMAIN` constraints for `UNKNOWN` parameters in prepared statements (Neil)
- Ensure **ALTER COLUMN TYPE** will process `FOREIGN KEY`, `UNIQUE`, and `PRIMARY KEY` constraints in the proper order (Nakano Yoshihisa)
- Fixes to allow restoring dumps that have cross-schema references to custom operators or operator classes (Tom)
- Allow pg_restore to continue properly after a **COPY** failure; formerly it tried to treat the remaining **COPY** data as SQL commands (Stephen Frost)
- Fix pg_ctl `unregister` crash when the data directory is not specified (Magnus)
- Fix ecpg crash on AMD64 and PPC (Neil)
- Recover properly if error occurs during argument passing in PL/python (Neil)
- Fix PL/perl's handling of locales on Win32 to match the backend (Andrew)
- Fix crash when `log_min_messages` is set to `DEBUG3` or above in `postgresql.conf` on Win32 (Bruce)
- Fix pgxs `-L` library path specification for Win32, Cygwin, OS X, AIX (Bruce)
- Check that SID is enabled while checking for Win32 admin privileges (Magnus)
- Properly reject out-of-range date inputs (Kris Jurka)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)

E.135. Release 8.0.6

Release Date

2006-01-09

This release contains a variety of fixes from 8.0.5. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.135.1. Migration to Version 8.0.6

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see Section E.138, « Release 8.0.3 ». Also, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.135.2. Changes

- Fix Windows code so that postmaster will continue rather than exit if there is no more room in ShmemBackendArray (Magnus)

  The previous behavior could lead to a denial-of-service situation if too many connection requests arrive close together. This applies *only* to the Windows port.
- Fix bug introduced in 8.0 that could allow ReadBuffer to return an already-used page as new, potentially causing loss of recently-committed data (Tom)
- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require **REINDEX** to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need **REINDEX** to fix existing indexes on textual columns if this has happened to you.
- Allow more flexible relocation of installation directories (Tom)

  Previous releases supported relocation only if all installation directory paths were the same except for the last component.
- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Various fixes for functions returning `RECORD`s (Tom)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.
- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.136. Release 8.0.5

Release Date

2005-12-12

This release contains a variety of fixes from 8.0.4. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.136.1. Migration to Version 8.0.5

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see Section E.138, « Release 8.0.3 ».

E.136.2. Changes

- Fix race condition in transaction log management

  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- Fix bgwriter problems after recovering from errors (Tom)

  The background writer was found to leak buffer pins after write errors. While not fatal in itself, this might lead to mysterious blockages of later VACUUM commands.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- `/contrib/ltree` fixes (Teodor)
- AIX and HPUX compile fixes (Tom)
- Retry file reads and writes after Windows NO_SYSTEM_RESOURCES error (Qingqing Zhou)
- Fix intermittent failure when `log_line_prefix` includes `%i`
- Fix psql performance issue with long scripts on Windows (Merlin Moncure)
- Fix missing updates of `pg_group` flat file
- Fix longstanding planning error for outer joins

  This bug sometimes caused a bogus error « RIGHT JOIN is only supported with merge-joinable join conditions ».
- Postpone timezone initialization until after `postmaster.pid` is created

  This avoids confusing startup scripts that expect the pid file to appear quickly.
- Prevent core dump in pg_autovacuum when a table has been dropped
- Fix problems with whole-row references (`foo.*`) to subquery results

E.137. Release 8.0.4

Release Date

2005-10-04

This release contains a variety of fixes from 8.0.3. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.137.1. Migration to Version 8.0.4

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see Section E.138, « Release 8.0.3 ».

E.137.2. Changes

- Fix error that allowed **VACUUM** to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

  This fixes a long-standing problem that could cause crashes in very rare circumstances.
- Fix CHAR() to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of CHAR() was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
- Force a checkpoint before committing **CREATE DATABASE**

  This should fix recent reports of « index is not a btree » failures when a crash occurs shortly after **CREATE DATABASE**.
- Fix the sense of the test for read-only transaction in **COPY**

  The code formerly prohibited **COPY TO**, where it should prohibit **COPY FROM**.
- Handle consecutive embedded newlines in **COPY** CSV-mode input

- Fix `date_trunc(week)` for dates near year end
- Fix planning problem with outer-join ON clauses that reference only the inner-side relation
- Further fixes for `x FULL JOIN y ON true` corner cases
- Fix overenthusiastic optimization of `x IN (SELECT DISTINCT ...)` and related cases
- Fix mis-planning of queries with small `LIMIT` values due to poorly thought out « fuzzy » cost comparison
- Make `array_in` and `array_recv` more paranoid about validating their OID parameter
- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve robustness of datetime parsing
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Improve MIPS and M68K spinlock code
- Don't try to open more than `max_files_per_process` files during postmaster startup
- Various memory leakage fixes
- Various portability improvements
- Update timezone data files
- Improve handling of DLL load failures on Windows
- Improve random-number generation on Windows
- Make `psql -f filename` return a nonzero exit code when opening the file fails
- Change pg_dump to handle alled check constraints more reliably
- Fix password prompting in pg_restore on Windows
- Fix PL/pgSQL to handle `var := var` correctly when the variable is of pass-by-reference type
- Fix PL/Perl `%_SHARED` so it's actually shared
- Fix `contrib/pg_autovacuum` to allow sleep intervals over 2000 sec
- Update `contrib/tsearch2` to use current Snowball code

E.138. Release 8.0.3

Release Date

2005-05-09

This release contains a variety of fixes from 8.0.2, including several security-related issues. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.138.1. Migration to Version 8.0.3

A dump/restore is not required for those running 8.0.X. However, it is one possible way of handling two significant security problems that have been found in the initial contents of 8.0.X system catalogs. A dump/initdb/reload sequence using 8.0.3's initdb will automatically correct these problems.

The larger security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.)

The lesser problem is that the `contrib/tsearch2` module creates several functions that are improperly declared to return internal when they do not accept internal arguments. This breaks type safety for all functions using internal arguments.

It is strongly recommended that all installations repair these errors, either by initdb or by following the manual repair procedure given below. The errors at least allow unprivileged database users to crash their server process, and might allow unprivileged

users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the same manual repair procedures shown in the 7.4.8 release notes.

E.138.2. Changes

- Change encoding function signature to prevent misuse
- Change `contrib/tsearch2` to avoid unsafe use of INTERNAL function results
- Guard against incorrect second parameter to `record_out`
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of TIME WITH TIME ZONE values

  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. NOTE: if you have an index on a TIME WITH TIME ZONE column, it will need to be **REINDEX**ed after installing this update, because the fix corrects the sort order of column values.
- Fix `EXTRACT(EPOCH)` for TIME WITH TIME ZONE values
- Fix mis-display of negative fractional seconds in INTERVAL values

  This error only occurred when the `--enable-integer-datetimes` configuration switch had been used.
- Fix pg_dump to dump trigger names containing `%` correctly (Neil)
- Still more 64-bit fixes for `contrib/intagg`
- Prevent incorrect optimization of functions returning RECORD
- Prevent crash on `COALESCE(NULL,NULL)`
- Fix Borland makefile for libpq
- Fix `contrib/btree_gist` for timetz type (Teodor)
- Make **pg_ctl** check the PID found in `postmaster.pid` to see if it is still a live process
- Fix **pg_dump**/**pg_restore** problems caused by addition of dump timestamps
- Fix interaction between materializing holdable cursors and firing deferred triggers during transaction commit
- Fix memory leak in SQL functions returning pass-by-reference data types

E.139. Release 8.0.2

Release Date

2005-04-07

This release contains a variety of fixes from 8.0.1. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.139.1. Migration to Version 8.0.2

A dump/restore is not required for those running 8.0.*. This release updates the major version number of the PostgreSQL™ libraries, so it might be necessary to re-link some user applications if they cannot find the properly-numbered shared library.

E.139.2. Changes

- Increment the major version number of all interface libraries (Bruce)

  This should have been done in 8.0.0. It is required so 7.4.X versions of PostgreSQL client applications, like psql, can be used on the same machine as 8.0.X applications. This might require re-linking user applications that use these libraries.

- Add Windows-only `wal_sync_method` setting of `fsync_writethrough` (Magnus, Bruce)

  This setting causes PostgreSQL™ to write through any disk-drive write cache when writing to WAL. This behavior was formerly called `fsync`, but was renamed because it acts quite differently from `fsync` on other platforms.

- Enable the `wal_sync_method` setting of `open_datasync` on Windows, and make it the default for that platform (Magnus, Bruce)

  Because the default is no longer `fsync_writethrough`, data loss is possible during a power failure if the disk drive has write caching enabled. To turn off the write cache on Windows, from the Device Manager, choose the drive properties, then `Policies`.

- New cache management algorithm 2Q replaces ARC (Tom)

  This was done to avoid a pending US patent on ARC. The 2Q code might be a few percentage points slower than ARC for some work loads. A better cache management algorithm will appear in 8.1.

- Planner adjustments to improve behavior on freshly-created tables (Tom)
- Allow plpgsql to assign to an element of an array that is initially `NULL` (Tom)

  Formerly the array would remain `NULL`, but now it becomes a single-element array. The main SQL engine was changed to handle **UPDATE** of a null array value this way in 8.0, but the similar case in plpgsql was overlooked.

- Convert `\r\n` and `\r` to `\n` in plpython function bodies (Michael Fuhr)

  This prevents syntax errors when plpython code is written on a Windows or Mac client.

- Allow SPI cursors to handle utility commands that return rows, such as **EXPLAIN** (Tom)
- Fix **CLUSTER** failure after **ALTER TABLE SET WITHOUT OIDS** (Tom)
- Reduce memory usage of **ALTER TABLE ADD COLUMN** (Neil)
- Fix **ALTER LANGUAGE RENAME** (Tom)
- Document the Windows-only `register` and `unregister` options of pg_ctl (Magnus)
- Ensure operations done during backend shutdown are counted by statistics collector

  This is expected to resolve reports of pg_autovacuum not vacuuming the system catalogs often enough -- it was not being told about catalog deletions caused by temporary table removal during backend exit.

- Change the Windows default for configuration parameter `log_destination` to `eventlog` (Magnus)

  By default, a server running on Windows will now send log output to the Windows event logger rather than standard error.

- Make Kerberos authentication work on Windows (Magnus)
- Allow **ALTER DATABASE RENAME** by superusers who aren't flagged as having CREATEDB privilege (Tom)
- Modify WAL log entries for **CREATE** and **DROP DATABASE** to not specify absolute paths (Tom)

  This allows point-in-time recovery on a different machine with possibly different database location. Note that **CREATE TABLESPACE** still poses a hazard in such situations.

- Fix crash from a backend exiting with an open transaction that created a table and opened a cursor on it (Tom)
- Fix `array_map()` so it can call PL functions (Tom)
- Several `contrib/tsearch2` and `contrib/btree_gist` fixes (Teodor)
- Fix crash of some `contrib/pgcrypto` functions on some platforms (Marko Kreen)
- Fix `contrib/intagg` for 64-bit platforms (Tom)
- Fix ecpg bugs in parsing of **CREATE** statement (Michael)
- Work around gcc bug on powerpc and amd64 causing problems in ecpg (Christof Petig)
- Do not use locale-aware versions of `upper()`, `lower()`, and `initcap()` when the locale is `C` (Bruce)

  This allows these functions to work on platforms that generate errors for non-7-bit data when the locale is `C`.

- Fix `quote_ident()` to quote names that match keywords (Tom)
- Fix `to_date()` to behave reasonably when `CC` and `YY` fields are both used (Karel)
- Prevent `to_char(interval)` from failing when given a zero-month interval (Tom)
- Fix wrong week returned by `date_trunc('week')` (Bruce)

  `date_trunc('week')` returned the wrong year for the first few days of January in some years.
- Use the correct default mask length for class `D` addresses in INET data types (Tom)

E.140. Release 8.0.1

Release Date

2005-01-31

This release contains a variety of fixes from 8.0.0, including several security-related issues. For information about new features in the 8.0 major release, see Section E.141, « Release 8.0 ».

E.140.1. Migration to Version 8.0.1

A dump/restore is not required for those running 8.0.0.

E.140.2. Changes

- Disallow **LOAD** to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), **LOAD** can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Check that creator of an aggregate function has the right to execute the specified transition functions

  This oversight made it possible to bypass denial of EXECUTE permission on a function.
- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Make **ALTER TABLE ADD COLUMN** enforce domain constraints in all cases
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Improve planning of grouped aggregate queries
- **ROLLBACK TO *`savepoint`*** closes cursors created since the savepoint
- Fix inadequate backend stack size on Windows
- Avoid SHGetSpecialFolderPath() on Windows (Magnus)
- Fix some problems in running pg_autovacuum as a Windows service (Dave Page)
- Multiple minor bug fixes in pg_dump/pg_restore
- Fix ecpg segfault with named structs used in typedefs (Michael)

E.141. Release 8.0

Release Date

2005-01-19

E.141.1. Overview

Major changes in this release:

Microsoft Windows Native Server
: This is the first PostgreSQL™ release to run natively on Microsoft Windows® as a server. It can run as a Windows™ service. This release supports NT-based Windows releases like Windows 2000 SP4™, Windows XP™, and Windows 2003™. Older releases like Windows 95™, Windows 98™, and Windows ME™ are not supported because these operating systems do not have the infrastructure to support PostgreSQL™. A separate installer project has been created to ease installation on Windows™ -- see *http://www.postgresql.org/ftp/win32/*.

  Although tested throughout our release cycle, the Windows port does not have the benefit of years of use in production environments that PostgreSQL™ has on Unix platforms. Therefore it should be treated with the same level of caution as you would a new product.

  Previous releases required the Unix emulation toolkit Cygwin™ in order to run the server on Windows operating systems. PostgreSQL™ has supported native clients on Windows for many years.

Savepoints
: Savepoints allow specific parts of a transaction to be aborted without affecting the remainder of the transaction. Prior releases had no such capability; there was no way to recover from a statement failure within a transaction except by aborting the whole transaction. This feature is valuable for application writers who require error recovery within a complex transaction.

Point-In-Time Recovery
: In previous releases there was no way to recover from disk drive failure except to restore from a previous backup or use a standby replication server. Point-in-time recovery allows continuous backup of the server. You can recover either to the point of failure or to some transaction in the past.

Tablespaces
: Tablespaces allow administrators to select different file systems for storage of individual tables, indexes, and databases. This improves performance and control over disk space usage. Prior releases used initlocation and manual symlink management for such tasks.

Improved Buffer Management, **CHECKPOINT**, **VACUUM**
: This release has a more intelligent buffer replacement strategy, which will make better use of available shared buffers and improve performance. The performance impact of vacuum and checkpoints is also lessened.

Change Column Types
: A column's data type can now be changed with **ALTER TABLE**.

New Perl Server-Side Language
: A new version of the plperl server-side language now supports a persistent shared storage area, triggers, returning records and arrays of records, and SPI calls to access the database.

Comma-separated-value (CSV) support in **COPY**
: **COPY** can now read and write comma-separated-value files. It has the flexibility to interpret nonstandard quoting and separation characters too.

E.141.2. Migration to Version 8.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than **SELECT**.
- Nondeferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation.
- Server configuration parameters `virtual_host` and `tcpip_socket` have been replaced with a more general parameter `listen_addresses`. Also, the server now listens on `localhost` by default, which eliminates the need for the `-i` post-

master switch in many scenarios.

- Server configuration parameters `SortMem` and `VacuumMem` have been renamed to `work_mem` and `maintenance_work_mem` to better reflect their use. The original names are still supported in **SET** and **SHOW**.
- Server configuration parameters `log_pid`, `log_timestamp`, and `log_source_port` have been replaced with a more general parameter `log_line_prefix`.
- Server configuration parameter `syslog` has been replaced with a more logical `log_destination` variable to control the log output destination.
- Server configuration parameter `log_statement` has been changed so it can selectively log just database modification or data definition statements. Server configuration parameter `log_duration` now prints only when `log_statement` prints the query.
- Server configuration parameter `max_expr_depth` parameter has been replaced with `max_stack_depth` which measures the physical stack size rather than the expression nesting depth. This helps prevent session termination due to stack overflow caused by recursive functions.
- The `length()` function no longer counts trailing spaces in CHAR(n) values.
- Casting an integer to BIT(N) selects the rightmost N bits of the integer, not the leftmost N bits as before.
- Updating an element or slice of a NULL array value now produces a nonnull array result, namely an array containing just the assigned-to positions.
- Syntax checking of array input values has been tightened up considerably. Junk that was previously allowed in odd places with odd results now causes an error. Empty-string element values must now be written as `""`, rather than writing nothing. Also changed behavior with respect to whitespace surrounding array elements: trailing whitespace is now ignored, for symmetry with leading whitespace (which has always been ignored).
- Overflow in integer arithmetic operations is now detected and reported as an error.
- The arithmetic operators associated with the single-byte "char" data type have been removed.
- The `extract()` function (also called `date_part`) now returns the proper year for BC dates. It previously returned one less than the correct year. The function now also returns the proper values for millennium and century.
- CIDR values now must have their nonmasked bits be zero. For example, we no longer allow `204.248.199.1/31` as a CIDR value. Such values should never have been accepted by PostgreSQL™ and will now be rejected.
- **EXECUTE** now returns a completion tag that matches the executed statement.
- psql's **\copy** command now reads or writes to the query's `stdin/stdout`, rather than psql's `stdin/stdout`. The previous behavior can be accessed via new `pstdin`/`pstdout` parameters.
- The JDBC client interface has been removed from the core distribution, and is now hosted at *http://jdbc.postgresql.org*.
- The Tcl client interface has also been removed. There are several Tcl interfaces now hosted at *http://gborg.postgresql.org*.
- The server now uses its own time zone database, rather than the one supplied by the operating system. This will provide consistent behavior across all platforms. In most cases, there should be little noticeable difference in time zone behavior, except that the time zone names used by **SET**/**SHOW** `TimeZone` might be different from what your platform provides.
- Configure's threading option no longer requires users to run tests or edit configuration files; threading options are now detected automatically.
- Now that tablespaces have been implemented, initlocation has been removed.
- The API for user-defined GiST indexes has been changed. The Union and PickSplit methods are now passed a pointer to a special GistEntryVector structure, rather than a bytea.

E.141.3. Deprecated Features

Some aspects of PostgreSQL™'s behavior have been determined to be suboptimal. For the sake of backward compatibility these have not been removed in 8.0, but they are considered deprecated and will be removed in the next major release.

- The 8.1 release will remove the `to_char()` function for intervals.
- The server now warns of empty strings passed to oid/float4/float8 data types, but continues to interpret them as zeroes as before. In the next major release, empty strings will be considered invalid input for these data types.

- By default, tables in PostgreSQL™ 8.0 and earlier are created with OIDs. In the next release, this will *not* be the case: to create a table that contains OIDs, the `WITH OIDS` clause must be specified or the `default_with_oids` configuration parameter must be set. Users are encouraged to explicitly specify `WITH OIDS` if their tables require OIDs for compatibility with future releases of PostgreSQL™.

E.141.4. Changes

Below you will find a detailed account of the changes between release 8.0 and the previous major release.

E.141.4.1. Performance Improvements

- Support cross-data-type index usage (Tom)

  Before this change, many queries would not use an index if the data types did not match exactly. This improvement makes index usage more intuitive and consistent.

- New buffer replacement strategy that improves caching (Jan)

  Prior releases used a least-recently-used (LRU) cache to keep recently referenced pages in memory. The LRU algorithm did not consider the number of times a specific cache entry was accessed, so large table scans could force out useful cache pages. The new cache algorithm uses four separate lists to track most recently used and most frequently used cache pages and dynamically optimize their replacement based on the work load. This should lead to much more efficient use of the shared buffer cache. Administrators who have tested shared buffer sizes in the past should retest with this new cache replacement policy.

- Add subprocess to write dirty buffers periodically to reduce checkpoint writes (Jan)

  In previous releases, the checkpoint process, which runs every few minutes, would write all dirty buffers to the operating system's buffer cache then flush all dirty operating system buffers to disk. This resulted in a periodic spike in disk usage that often hurt performance. The new code uses a background writer to trickle disk writes at a steady pace so checkpoints have far fewer dirty pages to write to disk. Also, the new code does not issue a global `sync()` call, but instead `fsync()`s just the files written since the last checkpoint. This should improve performance and minimize degradation during checkpoints.

- Add ability to prolong vacuum to reduce performance impact (Jan)

  On busy systems, **VACUUM** performs many I/O requests which can hurt performance for other users. This release allows you to slow down **VACUUM** to reduce its impact on other users, though this increases the total duration of **VACUUM**.

- Improve B-tree index performance for duplicate keys (Dmitry Tkach, Tom)

  This improves the way indexes are scanned when many duplicate values exist in the index.

- Use dynamically-generated table size estimates while planning (Tom)

  Formerly the planner estimated table sizes using the values seen by the last **VACUUM** or **ANALYZE**, both as to physical table size (number of pages) and number of rows. Now, the current physical table size is obtained from the kernel, and the number of rows is estimated by multiplying the table size by the row density (rows per page) seen by the last **VACUUM** or **ANALYZE**. This should produce more reliable estimates in cases where the table size has changed significantly since the last housekeeping command.

- Improved index usage with `OR` clauses (Tom)

  This allows the optimizer to use indexes in statements with many OR clauses that would not have been indexed in the past. It can also use multi-column indexes where the first column is specified and the second column is part of an `OR` clause.

- Improve matching of partial index clauses (Tom)

  The server is now smarter about using partial indexes in queries involving complex `WHERE` clauses.

- Improve performance of the GEQO optimizer (Tom)

  The GEQO optimizer is used to plan queries involving many tables (by default, twelve or more). This release speeds up the way queries are analyzed to decrease time spent in optimization.

- Miscellaneous optimizer improvements

  There is not room here to list all the minor improvements made, but numerous special cases work better than in prior releases.

- Improve lookup speed for C functions (Tom)

  This release uses a hash table to lookup information for dynamically loaded C functions. This improves their speed so they perform nearly as quickly as functions that are built into the server executable.

- Add type-specific **ANALYZE** statistics capability (Mark Cave-Ayland)

  This feature allows more flexibility in generating statistics for nonstandard data types.
- **ANALYZE** now collects statistics for expression indexes (Tom)

  Expression indexes (also called functional indexes) allow users to index not just columns but the results of expressions and function calls. With this release, the optimizer can gather and use statistics about the contents of expression indexes. This will greatly improve the quality of planning for queries in which an expression index is relevant.
- New two-stage sampling method for **ANALYZE** (Manfred Koizar)

  This gives better statistics when the density of valid rows is very different in different regions of a table.
- Speed up **TRUNCATE** (Tom)

  This buys back some of the performance loss observed in 7.4, while still keeping **TRUNCATE** transaction-safe.

E.141.4.2. Server Changes

- Add WAL file archiving and point-in-time recovery (Simon Riggs)
- Add tablespaces so admins can control disk layout (Gavin)
- Add a built-in log rotation program (Andreas Pflug)

  It is now possible to log server messages conveniently without relying on either syslog or an external log rotation program.
- Add new read-only server configuration parameters to show server compile-time settings: `block_size`, `integer_datetimes`, `max_function_args`, `max_identifier_length`, `max_index_keys` (Joe)
- Make quoting of `sameuser`, `samegroup`, and `all` remove special meaning of these terms in `pg_hba.conf` (Andrew)
- Use clearer IPv6 name `::1/128` for `localhost` in default `pg_hba.conf` (Andrew)
- Use CIDR format in `pg_hba.conf` examples (Andrew)
- Rename server configuration parameters `SortMem` and `VacuumMem` to `work_mem` and `maintenance_work_mem` (Old names still supported) (Tom)

  This change was made to clarify that bulk operations such as index and foreign key creation use `maintenance_work_mem`, while `work_mem` is for workspaces used during query execution.
- Allow logging of session disconnections using server configuration `log_disconnections` (Andrew)
- Add new server configuration parameter `log_line_prefix` to allow control of information emitted in each log line (Andrew)

  Available information includes user name, database name, remote IP address, and session start time.
- Remove server configuration parameters `log_pid`, `log_timestamp`, `log_source_port`; functionality superseded by `log_line_prefix` (Andrew)
- Replace the `virtual_host` and `tcpip_socket` parameters with a unified `listen_addresses` parameter (Andrew, Tom)

  `virtual_host` could only specify a single IP address to listen on. `listen_addresses` allows multiple addresses to be specified.
- Listen on localhost by default, which eliminates the need for the `-i` postmaster switch in many scenarios (Andrew)

  Listening on localhost (`127.0.0.1`) opens no new security holes but allows configurations like Windows and JDBC, which do not support local sockets, to work without special adjustments.
- Remove `syslog` server configuration parameter, and add more logical `log_destination` variable to control log output location (Magnus)
- Change server configuration parameter `log_statement` to take values `all`, `mod`, `ddl`, or `none` to select which queries are logged (Bruce)

  This allows administrators to log only data definition changes or only data modification statements.
- Some logging-related configuration parameters could formerly be adjusted by ordinary users, but only in the « more verbose » direction. They are now treated more strictly: only superusers can set them. However, a superuser can use **ALTER USER** to

provide per-user settings of these values for non-superusers. Also, it is now possible for superusers to set values of superuser-only configuration parameters via `PGOPTIONS`.

- Allow configuration files to be placed outside the data directory (mlw)

  By default, configuration files are kept in the cluster's top directory. With this addition, configuration files can be placed outside the data directory, easing administration.

- Plan prepared queries only when first executed so constants can be used for statistics (Oliver Jowett)

  Prepared statements plan queries once and execute them many times. While prepared queries avoid the overhead of replanning on each use, the quality of the plan suffers from not knowing the exact parameters to be used in the query. In this release, planning of unnamed prepared statements is delayed until the first execution, and the actual parameter values of that execution are used as optimization hints. This allows use of out-of-line parameter passing without incurring a performance penalty.

- Allow **DECLARE CURSOR** to take parameters (Oliver Jowett)

  It is now useful to issue **DECLARE CURSOR** in a `Parse` message with parameters. The parameter values sent at `Bind` time will be substituted into the execution of the cursor's query.

- Fix hash joins and aggregates of inet and cidr data types (Tom)

  Release 7.4 handled hashing of mixed inet and cidr values incorrectly. (This bug did not exist in prior releases because they wouldn't try to hash either data type.)

- Make `log_duration` print only when `log_statement` prints the query (Ed L.)

E.141.4.3. Query Changes

- Add savepoints (nested transactions) (Alvaro)
- Unsupported isolation levels are now accepted and promoted to the nearest supported level (Peter)

  The SQL specification states that if a database doesn't support a specific isolation level, it should use the next more restrictive level. This change complies with that recommendation.

- Allow **BEGIN WORK** to specify transaction isolation levels like **START TRANSACTION** does (Bruce)
- Fix table permission checking for cases in which rules generate a query type different from the originally submitted query (Tom)
- Implement dollar quoting to simplify single-quote usage (Andrew, Tom, David Fetter)

  In previous releases, because single quotes had to be used to quote a function's body, the use of single quotes inside the function text required use of two single quotes or other error-prone notations. With this release we add the ability to use "dollar quoting" to quote a block of text. The ability to use different quoting delimiters at different nesting levels greatly simplifies the task of quoting correctly, especially in complex functions. Dollar quoting can be used anywhere quoted text is needed.

- Make `CASE val WHEN compval1 THEN ...` evaluate `val` only once (Tom)

  `CASE` no longer evaluates the tested expression multiple times. This has benefits when the expression is complex or is volatile.

- Test `HAVING` before computing target list of an aggregate query (Tom)

  Fixes improper failure of cases such as `SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0`. This should work but formerly could fail with divide-by-zero.

- Replace `max_expr_depth` parameter with `max_stack_depth` parameter, measured in kilobytes of stack size (Tom)

  This gives us a fairly bulletproof defense against crashing due to runaway recursive functions. Instead of measuring the depth of expression nesting, we now directly measure the size of the execution stack.

- Allow arbitrary row expressions (Tom)

  This release allows SQL expressions to contain arbitrary composite types, that is, row values. It also allows functions to more easily take rows as arguments and return row values.

- Allow `LIKE`/`ILIKE` to be used as the operator in row and subselect comparisons (Fabien Coelho)
- Avoid locale-specific case conversion of basic ASCII letters in identifiers and keywords (Tom)

  This solves the « Turkish problem » with mangling of words containing `I` and `i`. Folding of characters outside the 7-bit-ASCII

set is still locale-aware.

- Improve syntax error reporting (Fabien, Tom)

  Syntax error reports are more useful than before.

- Change **EXECUTE** to return a completion tag matching the executed statement (Kris Jurka)

  Previous releases return an **EXECUTE** tag for any **EXECUTE** call. In this release, the tag returned will reflect the command executed.

- Avoid emitting `NATURAL CROSS JOIN` in rule listings (Tom)

  Such a clause makes no logical sense, but in some cases the rule decompiler formerly produced this syntax.

E.141.4.4. Object Manipulation Changes

- Add **COMMENT ON** for casts, conversions, languages, operator classes, and large objects (Christopher)
- Add new server configuration parameter `default_with_oids` to control whether tables are created with OIDs by default (Neil)

  This allows administrators to control whether **CREATE TABLE** commands create tables with or without OID columns by default. (Note: the current factory default setting for `default_with_oids` is `TRUE`, but the default will become `FALSE` in future releases.)

- Add `WITH` / `WITHOUT OIDS` clause to **CREATE TABLE AS** (Neil)
- Allow **ALTER TABLE DROP COLUMN** to drop an OID column (**ALTER TABLE SET WITHOUT OIDS** still works) (Tom)
- Allow composite types as table columns (Tom)
- Allow **ALTER ... ADD COLUMN** with defaults and `NOT NULL` constraints; works per SQL spec (Rod)

  It is now possible for `ADD COLUMN` to create a column that is not initially filled with NULLs, but with a specified default value.

- Add **ALTER COLUMN TYPE** to change column's type (Rod)

  It is now possible to alter a column's data type without dropping and re-adding the column.

- Allow multiple **ALTER** actions in a single **ALTER TABLE** command (Rod)

  This is particularly useful for **ALTER** commands that rewrite the table (which include `ALTER COLUMN TYPE` and `ADD COLUMN` with a default). By grouping **ALTER** commands together, the table need be rewritten only once.

- Allow **ALTER TABLE** to add SERIAL columns (Tom)

  This falls out from the new capability of specifying defaults for new columns.

- Allow changing the owners of aggregates, conversions, databases, functions, operators, operator classes, schemas, types, and tablespaces (Christopher, Euler Taveira de Oliveira)

  Previously this required modifying the system tables directly.

- Allow temporary object creation to be limited to `SECURITY DEFINER` functions (Sean Chittenden)
- Add `ALTER TABLE ... SET WITHOUT CLUSTER` (Christopher)

  Prior to this release, there was no way to clear an auto-cluster specification except to modify the system tables.

- Constraint/Index/SERIAL names are now *`table_column_type`* with numbers appended to guarantee uniqueness within the schema (Tom)

  The SQL specification states that such names should be unique within a schema.

- Add `pg_get_serial_sequence()` to return a SERIAL column's sequence name (Christopher)

  This allows automated scripts to reliably find the SERIAL sequence name.

- Warn when primary/foreign key data type mismatch requires costly lookup
- New **ALTER INDEX** command to allow moving of indexes between tablespaces (Gavin)
- Make **ALTER TABLE OWNER** change dependent sequence ownership too (Alvaro)

E.141.4.5. Utility Command Changes

- Allow **CREATE SCHEMA** to create triggers, indexes, and sequences (Neil)
- Add `ALSO` keyword to **CREATE RULE** (Fabien Coelho)

  This allows `ALSO` to be added to rule creation to contrast it with `INSTEAD` rules.
- Add `NOWAIT` option to **LOCK** (Tatsuo)

  This allows the **LOCK** command to fail if it would have to wait for the requested lock.
- Allow **COPY** to read and write comma-separated-value (CSV) files (Andrew, Bruce)
- Generate error if the **COPY** delimiter and NULL string conflict (Bruce)
- **GRANT**/**REVOKE** behavior follows the SQL spec more closely
- Avoid locking conflict between **CREATE INDEX** and **CHECKPOINT** (Tom)

  In 7.3 and 7.4, a long-running B-tree index build could block concurrent **CHECKPOINT**s from completing, thereby causing WAL bloat because the WAL log could not be recycled.
- Database-wide **ANALYZE** does not hold locks across tables (Tom)

  This reduces the potential for deadlocks against other backends that want exclusive locks on tables. To get the benefit of this change, do not execute database-wide **ANALYZE** inside a transaction block (**BEGIN** block); it must be able to commit and start a new transaction for each table.
- **REINDEX** does not exclusively lock the index's parent table anymore

  The index itself is still exclusively locked, but readers of the table can continue if they are not using the particular index being rebuilt.
- Erase MD5 user passwords when a user is renamed (Bruce)

  PostgreSQL™ uses the user name as salt when encrypting passwords via MD5. When a user's name is changed, the salt will no longer match the stored MD5 password, so the stored password becomes useless. In this release a notice is generated and the password is cleared. A new password must then be assigned if the user is to be able to log in with a password.
- New pg_ctl `kill` option for Windows (Andrew)

  Windows does not have a `kill` command to send signals to backends so this capability was added to pg_ctl.
- Information schema improvements
- Add `--pwfile` option to initdb so the initial password can be set by GUI tools (Magnus)
- Detect locale/encoding mismatch in initdb (Peter)
- Add `register` command to pg_ctl to register Windows operating system service (Dave Page)

E.141.4.6. Data Type and Function Changes

- More complete support for composite types (row types) (Tom)

  Composite values can be used in many places where only scalar values worked before.
- Reject nonrectangular array values as erroneous (Joe)

  Formerly, `array_in` would silently build a surprising result.
- Overflow in integer arithmetic operations is now detected (Tom)
- The arithmetic operators associated with the single-byte "char" data type have been removed.

  Formerly, the parser would select these operators in many situations where an « unable to select an operator » error would be more appropriate, such as `null * null`. If you actually want to do arithmetic on a "char" column, you can cast it to integer explicitly.
- Syntax checking of array input values considerably tightened up (Joe)

  Junk that was previously allowed in odd places with odd results now causes an `ERROR`, for example, non-whitespace after the closing right brace.

- Empty-string array element values must now be written as `""`, rather than writing nothing (Joe)

  Formerly, both ways of writing an empty-string element value were allowed, but now a quoted empty string is required. The case where nothing at all appears will probably be considered to be a NULL element value in some future release.
- Array element trailing whitespace is now ignored (Joe)

  Formerly leading whitespace was ignored, but trailing whitespace between an element value and the delimiter or right brace was significant. Now trailing whitespace is also ignored.
- Emit array values with explicit array bounds when lower bound is not one (Joe)
- Accept `YYYY-monthname-DD` as a date string (Tom)
- Make `netmask` and `hostmask` functions return maximum-length mask length (Tom)
- Change factorial function to return numeric (Gavin)

  Returning numeric allows the factorial function to work for a wider range of input values.
- `to_char`/`to_date()` date conversion improvements (Kurt Roeckx, Fabien Coelho)
- Make `length()` disregard trailing spaces in CHAR(n) (Gavin)

  This change was made to improve consistency: trailing spaces are semantically insignificant in CHAR(n) data, so they should not be counted by `length()`.
- Warn about empty string being passed to OID/float4/float8 data types (Neil)

  8.1 will throw an error instead.
- Allow leading or trailing whitespace in int2/int4/int8/float4/float8 input routines (Neil)
- Better support for IEEE `Infinity` and `NaN` values in float4/float8 (Neil)

  These should now work on all platforms that support IEEE-compliant floating point arithmetic.
- Add `week` option to `date_trunc()` (Robert Creager)
- Fix `to_char` for `1 BC` (previously it returned `1 AD`) (Bruce)
- Fix `date_part(year)` for BC dates (previously it returned one less than the correct year) (Bruce)
- Fix `date_part()` to return the proper millennium and century (Fabien Coelho)

  In previous versions, the century and millennium results had a wrong number and started in the wrong year, as compared to standard reckoning of such things.
- Add `ceiling()` as an alias for `ceil()`, and `power()` as an alias for `pow()` for standards compliance (Neil)
- Change `ln()`, `log()`, `power()`, and `sqrt()` to emit the correct `SQLSTATE` error codes for certain error conditions, as specified by SQL:2003 (Neil)
- Add `width_bucket()` function as defined by SQL:2003 (Neil)
- Add `generate_series()` functions to simplify working with numeric sets (Joe)
- Fix `upper/lower/initcap()` functions to work with multibyte encodings (Tom)
- Add boolean and bitwise integer `AND`/`OR` aggregates (Fabien Coelho)
- New session information functions to return network addresses for client and server (Sean Chittenden)
- Add function to determine the area of a closed path (Sean Chittenden)
- Add function to send cancel request to other backends (Magnus)
- Add interval plus datetime operators (Tom)

  The reverse ordering, datetime plus interval, was already supported, but both are required by the SQL standard.
- Casting an integer to BIT(N) selects the rightmost N bits of the integer (Tom)

  In prior releases, the leftmost N bits were selected, but this was deemed unhelpful, not to mention inconsistent with casting from bit to int.
- Require CIDR values to have all nonmasked bits be zero (Kevin Brintnall)

E.141.4.7. Server-Side Language Changes

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than **SELECT**. There is a considerable performance gain from declaring a function `STABLE` or `IMMUTABLE` rather than `VOLATILE`.
- Nondeferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation. For example, if a function inserts a new row into a table, any nondeferred foreign key checks occur before proceeding with the function.
- Allow function parameters to be declared with names (Dennis Björklund)

  This allows better documentation of functions. Whether the names actually do anything depends on the specific function language being used.
- Allow PL/pgSQL parameter names to be referenced in the function (Dennis Björklund)

  This basically creates an automatic alias for each named parameter.
- Do minimal syntax checking of PL/pgSQL functions at creation time (Tom)

  This allows us to catch simple syntax errors sooner.
- More support for composite types (row and record variables) in PL/pgSQL

  For example, it now works to pass a rowtype variable to another function as a single variable.
- Default values for PL/pgSQL variables can now reference previously declared variables
- Improve parsing of PL/pgSQL FOR loops (Tom)

  Parsing is now driven by presence of `".."` rather than data type of `FOR` variable. This makes no difference for correct functions, but should result in more understandable error messages when a mistake is made.
- Major overhaul of PL/Perl server-side language (Command Prompt, Andrew Dunstan)
- In PL/Tcl, SPI commands are now run in subtransactions. If an error occurs, the subtransaction is cleaned up and the error is reported as an ordinary Tcl error, which can be trapped with `catch`. Formerly, it was not possible to catch such errors.
- Accept **ELSEIF** in PL/pgSQL (Neil)

  Previously PL/pgSQL only allowed **ELSIF**, but many people are accustomed to spelling this keyword **ELSEIF**.

E.141.4.8. psql Changes

- Improve psql information display about database objects (Christopher)
- Allow psql to display group membership in **\du** and **\dg** (Markus Bertheau)
- Prevent psql **\dn** from showing temporary schemas (Bruce)
- Allow psql to handle tilde user expansion for file names (Zach Irmen)
- Allow psql to display fancy prompts, including color, via readline (Reece Hart, Chet Ramey)
- Make psql **\copy** match **COPY** command syntax fully (Tom)
- Show the location of syntax errors (Fabien Coelho, Tom)
- Add **CLUSTER** information to psql **\d** display (Bruce)
- Change psql **\copy stdin/stdout** to read from command input/output (Bruce)
- Add `pstdin`/`pstdout` to read from psql's `stdin`/`stdout` (Mark Feit)
- Add global psql configuration file, `psqlrc.sample` (Bruce)

  This allows a central file where global psql startup commands can be stored.

- Have psql **\d+** indicate if the table has an OID column (Neil)
- On Windows, use binary mode in psql when reading files so control-Z is not seen as end-of-file
- Have **\dn+** show permissions and description for schemas (Dennis Björklund)
- Improve tab completion support (Stefan Kaltenbrunn, Greg Sabino Mullane)
- Allow boolean settings to be set using upper or lower case (Michael Paesold)

E.141.4.9. pg_dump Changes

- Use dependency information to improve the reliability of pg_dump (Tom)

  This should solve the longstanding problems with related objects sometimes being dumped in the wrong order.
- Have pg_dump output objects in alphabetical order if possible (Tom)

  This should make it easier to identify changes between dump files.
- Allow pg_restore to ignore some SQL errors (Fabien Coelho)

  This makes pg_restore's behavior similar to the results of feeding a pg_dump output script to psql. In most cases, ignoring errors and plowing ahead is the most useful thing to do. Also added was a pg_restore option to give the old behavior of exiting on an error.
- pg_restore `-l` display now includes objects' schema names
- New begin/end markers in pg_dump text output (Bruce)
- Add start/stop times for pg_dump/pg_dumpall in verbose mode (Bruce)
- Allow most pg_dump options in pg_dumpall (Christopher)
- Have pg_dump use **ALTER OWNER** rather than **SET SESSION AUTHORIZATION** by default (Christopher)

E.141.4.10. libpq Changes

- Make libpq's `SIGPIPE` handling thread-safe (Bruce)
- Add `PQmbdsplen()` which returns the display length of a character (Tatsuo)
- Add thread locking to SSL and Kerberos connections (Manfred Spraul)
- Allow `PQoidValue()`, `PQcmdTuples()`, and `PQoidStatus()` to work on **EXECUTE** commands (Neil)
- Add `PQserverVersion()` to provide more convenient access to the server version number (Greg Sabino Mullane)
- Add `PQprepare/PQsendPrepared()` functions to support preparing statements without necessarily specifying the data types of their parameters (Abhijit Menon-Sen)
- Many ECPG improvements, including **SET DESCRIPTOR** (Michael)

E.141.4.11. Source Code Changes

- Allow the database server to run natively on Windows (Claudio, Magnus, Andrew)
- Shell script commands converted to C versions for Windows support (Andrew)
- Create an extension makefile framework (Fabien Coelho, Peter)

  This simplifies the task of building extensions outside the original source tree.
- Support relocatable installations (Bruce)

  Directory paths for installed files (such as the `/share` directory) are now computed relative to the actual location of the executables, so that an installation tree can be moved to another place without reconfiguring and rebuilding.
- Use `--with-docdir` to choose installation location of documentation; also allow `--infodir` (Peter)
- Add `--without-docdir` to prevent installation of documentation (Peter)
- Upgrade to DocBook V4.2 SGML (Peter)

- New `PostgreSQL` CVS tag (Marc)

  This was done to make it easier for organizations to manage their own copies of the PostgreSQL™ CVS repository. File version stamps from the master repository will not get munged by checking into or out of a copied repository.
- Clarify locking code (Manfred Koizar)
- Buffer manager cleanup (Neil)
- Decouple platform tests from CPU spinlock code (Bruce, Tom)
- Add inlined test-and-set code on PA-RISC for gcc (ViSolve, Tom)
- Improve i386 spinlock code (Manfred Spraul)
- Clean up spinlock assembly code to avoid warnings from newer gcc releases (Tom)
- Remove JDBC from source tree; now a separate project
- Remove the libpgtcl client interface; now a separate project
- More accurately estimate memory and file descriptor usage (Tom)
- Improvements to the Mac OS X startup scripts (Ray A.)
- New `fsync()` test program (Bruce)
- Major documentation improvements (Neil, Peter)
- Remove pg_encoding; not needed anymore
- Remove pg_id; not needed anymore
- Remove initlocation; not needed anymore
- Auto-detect thread flags (no more manual testing) (Bruce)
- Use Olson's public domain timezone library (Magnus)
- With threading enabled, use thread flags on Unixware for backend executables too (Bruce)

  Unixware cannot mix threaded and nonthreaded object files in the same executable, so everything must be compiled as threaded.
- psql now uses a flex-generated lexical analyzer to process command strings
- Reimplement the linked list data structure used throughout the backend (Neil)

  This improves performance by allowing list append and length operations to be more efficient.
- Allow dynamically loaded modules to create their own server configuration parameters (Thomas Hallgren)
- New Brazilian version of FAQ (Euler Taveira de Oliveira)
- Add French FAQ (Guillaume Lelarge)
- New pgevent for Windows logging
- Make libpq and ECPG build as proper shared libraries on OS X (Tom)

E.141.4.12. Contrib Changes

- Overhaul of `contrib/dblink` (Joe)
- `contrib/dbmirror` improvements (Steven Singer)
- New `contrib/xml2` (John Gray, Torchbox)
- Updated `contrib/mysql`
- New version of `contrib/btree_gist` (Teodor)
- New `contrib/trgm`, trigram matching for PostgreSQL™ (Teodor)
- Many `contrib/tsearch2` improvements (Teodor)
- Add double metaphone to `contrib/fuzzystrmatch` (Andrew)

- Allow `contrib/pg_autovacuum` to run as a Windows service (Dave Page)
- Add functions to `contrib/dbsize` (Andreas Pflug)
- Removed `contrib/pg_logger`: obsoleted by integrated logging subprocess
- Removed `contrib/rserv`: obsoleted by various separate projects

E.142. Release 7.4.30

Release Date

2010-10-04

This release contains a variety of fixes from 7.4.29. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

This is expected to be the last PostgreSQL™ release in the 7.4.X series. Users are encouraged to update to a newer release branch soon.

E.142.1. Migration to Version 7.4.30

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see Section E.146, « Release 7.4.26 ».

E.142.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function's owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).
- Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it's intended to be used with (Heikki Linnakangas, Tom Lane)
- Fix « cannot handle unplanned sub-select » error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.
- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.
- Improve `contrib/dblink`'s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after « duplicate connection name » errors in `contrib/dblink` (Itagaki Takahiro)
- Update build infrastructure and documentation to reflect the source code repository's move from CVS to Git (Magnus Hagander and others)

E.143. Release 7.4.29

Release Date

2010-05-17

This release contains a variety of fixes from 7.4.28. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

The PostgreSQL™ community will stop releasing updates for the 7.4.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.143.1. Migration to Version 7.4.29

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see Section E.146, « Release 7.4.26 ».

E.143.2. Changes

- Enforce restrictions in `plperl` using an opmask applied to the whole interpreter, instead of using `Safe.pm` (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that `Safe.pm` is too insecure to rely on for making `plperl` trustable. This change removes use of `Safe.pm` altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's `strict` pragma in a natural way in `plperl`, and that Perl's `$a` and `$b` variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted « normal » Tcl interpreter unless we are really going to execute a `pltclu` function. (CVE-2010-1170)

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran `ALTER USER ... RESET ALL` for himself, or `ALTER DATABASE ... RESET ALL` for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the **ALTER** will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a `CONTEXT` addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Update pl/perl's `ppport.h` for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Ensure that `contrib/pgstattuple` functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that `shmget()` returns `EINVAL` for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

E.144. Release 7.4.28

Release Date

2010-03-15

This release contains a variety of fixes from 7.4.27. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

The PostgreSQL™ community will stop releasing updates for the 7.4.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.144.1. Migration to Version 7.4.28

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see Section E.146, « Release 7.4.26 ».

E.144.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for an SSL connection (Magnus)

  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.
- Make `substring()` for bit types treat any negative length as meaning « all the rest of the string » (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
- Fix some cases of pathologically slow regular expression matching (Tom)
- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.
- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.
- Prevent crash in `contrib/dblink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)

E.145. Release 7.4.27

Release Date

2009-12-14

This release contains a variety of fixes from 7.4.26. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.145.1. Migration to Version 7.4.27

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see Section E.146, « Release 7.4.26 ».

E.145.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

  This change prevents allegedly-immutable index functions from possibly subverting a superuser's session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).
- Fix possible crash during backend-startup-time cache initialization (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)

This fix prevents a PANIC if a `VACUUM FULL` is canceled after it's already committed its tuple movements, as well as transient errors if a plain `VACUUM` is interrupted after having truncated the table.

- Fix possible crash due to integer overflow in hash table size calculation (Tom)

  This could occur with extremely large planner estimates for the size of a hashjoin's result.

- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Fix PAM password processing to be more robust (Tom)

  The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.

- Make the postmaster ignore any `application_name` parameter in connection request packets, to improve compatibility with future libpq versions (Tom)

E.146. Release 7.4.26

Release Date

2009-09-09

This release contains a variety of fixes from 7.4.25. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.146.1. Migration to Version 7.4.26

A dump/restore is not required for those running 7.4.X. However, if you have any hash indexes on interval columns, you must **REINDEX** them after updating to 7.4.26. Also, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.146.2. Changes

- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)

- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
- Fix hash calculation for data type interval (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must **REINDEX** them after updating.

- Fix overflow for `INTERVAL 'x ms'` when *x* is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.

- Fix money data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)
- Properly round datetime input like `00:12:57.999999999999999999999999999` (Tom)
- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)
- Fix portability issues in plperl initialization (Andrew Dunstan)
- Improve robustness of libpq's code to recover from errors during **COPY FROM STDIN** (Tom)
- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)

E.147. Release 7.4.25

Release Date

2009-03-16

This release contains a variety of fixes from 7.4.24. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.147.1. Migration to Version 7.4.25

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.147.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)

  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that *any* error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)
- Disallow **CREATE CONVERSION** with the wrong encodings for the specified conversion function (Heikki)

  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.
- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)
- Add `MUST` (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

E.148. Release 7.4.24

Release Date

2009-02-02

This release contains a variety of fixes from 7.4.23. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.148.1. Migration to Version 7.4.24

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.148.2. Changes

- Improve handling of URLs in `headline()` function (Teodor)
- Improve handling of overlength headlines in `headline()` function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Avoid unnecessary locking of small tables in **VACUUM** (Heikki)
- Fix uninitialized variables in `contrib/tsearch2`'s `get_covers()` function (Teodor)
- Fix bug in `to_char()`'s handling of `TH` format codes (Andreas Scherbaum)
- Make all documentation reference `pgsql-bugs` and/or `pgsql-hackers` as appropriate, instead of the now-decommissioned `pgsql-ports` and `pgsql-patches` mailing lists (Tom)

E.149. Release 7.4.23

Release Date

2008-11-03

This release contains a variety of fixes from 7.4.22. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.149.1. Migration to Version 7.4.23

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.149.2. Changes

- Fix backend crash when the client encoding cannot represent a localized error message (Tom)

  We have addressed similar issues before, but it would still fail if the « character has no equivalent » message itself couldn't be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an `--enable-integer-datetimes` build (Ron Mayer)
- Ensure `SPI_getvalue` and `SPI_getbinval` behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)

  This situation is normal when a table has had columns added or removed, but these two functions didn't handle it properly. The only likely consequence is an incorrect error indication.
- Fix ecpg's parsing of **CREATE USER** (Michael)

E.150. Release 7.4.22

Release Date

2008-09-22

This release contains a variety of fixes from 7.4.21. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.150.1. Migration to Version 7.4.22

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.150.2. Changes

- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Fix bug in backwards scanning of a cursor on a `SELECT DISTINCT ON` query (Tom)
- Fix planner to estimate that `GROUP BY` expressions yielding boolean results always result in two groups, regardless of the expressions' contents (Tom)

  This is very substantially more accurate than the regular `GROUP BY` estimate for certain boolean tests like *`col`* `IS NULL`.
- Improve pg_dump and pg_restore's error reporting after failure to send a SQL command (Tom)

E.151. Release 7.4.21

Release Date

2008-06-12

This release contains one serious bug fix over 7.4.20. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.151.1. Migration to Version 7.4.21

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.151.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)

  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL™ to reject what had been a valid **SELECT DISTINCT** view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

E.152. Release 7.4.20

Release Date

never released

This release contains a variety of fixes from 7.4.19. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.152.1. Migration to Version 7.4.20

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.152.2. Changes

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic « Yo » characters (`e` and `E` with two dots) (Sergey Burladyan)
- Fix a few datatype input functions that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched `ORDER BY` and `DISTINCT` expressions.
- Fix a corner case in regular-expression substring matching (`substring(`*`string`*` from `*`pattern`*`)`) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn't got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since `(bar)` isn't matched, but it was mistakenly returning the whole-pattern match instead (ie, `foo`).
- Fix incorrect result from ecpg's `PGTYPEStimestamp_sub()` function (Michael)
- Fix `DatumGetBool` macro to not fail with gcc 4.3 (Tom)

  This problem affects « old style » (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.
- Fix longstanding **LISTEN**/**NOTIFY** race condition (Tom)

  In rare cases a session that had just executed a **LISTEN** might not get a notification, even though one would be expected because the concurrent transaction executing **NOTIFY** was observed to commit later.

A side effect of the fix is that a transaction that has executed a not-yet-committed **LISTEN** command will not see any row in pg_listener for the **LISTEN**, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)

  An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.

- Fix libpq to handle NOTICE messages correctly during COPY OUT (Tom)

  This failure has only been observed to occur when a user-defined datatype's output routine issues a NOTICE, but there is no guarantee it couldn't happen due to other causes.

E.153. Release 7.4.19

Release Date

2008-01-07

This release contains a variety of fixes from 7.4.18, including fixes for significant security issues. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.153.1. Migration to Version 7.4.19

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.153.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running **VACUUM**, **ANALYZE**, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as **VACUUM FULL**, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including **VACUUM**, **ANALYZE**, **REINDEX**, and **CLUSTER**) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of **SET SESSION AUTHORIZATION** and **SET ROLE** is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 7.4.18 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Fix planner failure in some cases of `WHERE false AND var IN (SELECT ...)` (Tom)
- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)
- Fix PL/Python to not crash on long exception messages (Alvaro)
- ecpg parser fixes (Michael)
- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of `to_tsvector()` on huge input strings (Teodor)

- Require a specific version of Autoconf™ to be used when re-generating the **configure** script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf™ and PostgreSQL™ versions. You can remove the version check if you really want to use a different Autoconf™ version, but it's your responsibility whether the result works or not.

E.154. Release 7.4.18

Release Date

2007-09-17

This release contains fixes from 7.4.17. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.154.1. Migration to Version 7.4.18

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.154.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent **VACUUM** on the same table (Tom)
- Make **CREATE DOMAIN ... DEFAULT NULL** work properly (Tom)
- Fix excessive logging of SSL error messages (Tom)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Prevent **CLUSTER** from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.155. Release 7.4.17

Release Date

2007-04-23

This release contains fixes from 7.4.16, including a security fix. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.155.1. Migration to Version 7.4.17

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.155.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See **CREATE FUNCTION** for more information.
- `/contrib/tsearch2` crash fixes (Teodor)
- Fix potential-data-corruption bug in how **VACUUM FULL** handles **UPDATE** chains (Tom, Pavan Deolasee)
- Fix PANIC during enlargement of a hash index (bug introduced in 7.4.15) (Tom)

E.156. Release 7.4.16

Release Date

2007-02-05

This release contains a variety of fixes from 7.4.15, including a security fix. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.156.1. Migration to Version 7.4.16

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.156.2. Changes

- Remove security vulnerability that allowed connected users to read backend memory (Tom)

  The vulnerability involves suppressing the normal check that a SQL function returns the data type it's declared to, or changing the data type of a table column used in a SQL function (CVE-2007-0555). This error can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.157. Release 7.4.15

Release Date

2007-01-08

This release contains a variety of fixes from 7.4.14. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.157.1. Migration to Version 7.4.15

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.157.2. Changes

- Improve handling of `getaddrinfo()` on AIX (Tom)

  This fixes a problem with starting the statistics collector, among other things.
- Fix « failed to re-find parent key » errors in **VACUUM** (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix error when constructing an `ARRAY[]` made up of multiple empty elements (Tom)
- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.

E.158. Release 7.4.14

Release Date

2006-10-16

This release contains a variety of fixes from 7.4.13. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.158.1. Migration to Version 7.4.14

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.158.2. Changes

- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix `string_to_array()` to handle overlapping matches for the separator string

  For example, `string_to_array('123xx456xxx789', 'xx')`.
- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Fix backslash escaping in /contrib/dbmirror
- Adjust regression tests for recent changes in US DST laws

E.159. Release 7.4.13

Release Date

2006-05-23

This release contains a variety of fixes from 7.4.12, including patches for extremely serious security issues. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.159.1. Migration to Version 7.4.13

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.159.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL™ has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.
- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations and `standard_conforming_strings`

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL™ connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping « by hand » should be modified to rely on library routines instead.
- Fix some incorrect encoding conversion functions

  `win1251_to_iso`, `alt_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.
- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)
- Fix bug that sometimes caused OR'd index scans to miss rows they should have returned
- Fix WAL replay for case where a btree index has been truncated
- Fix `SIMILAR TO` for patterns involving `|` (Tom)
- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
- Fix for Bonjour on Intel Macs (Ashley Clark)
- Fix various minor memory leaks

E.160. Release 7.4.12

Release Date

2006-02-14

This release contains a variety of fixes from 7.4.11. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.160.1. Migration to Version 7.4.12

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see Section E.161, « Release 7.4.11 ».

E.160.2. Changes

- Fix potential crash in **SET SESSION AUTHORIZATION** (CVE-2006-0553)

  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 7.4.9 and 7.3.11 releases.
- Fix race condition that could lead to « file already exists » errors during pg_clog file creation (Tom)
- Properly check `DOMAIN` constraints for `UNKNOWN` parameters in prepared statements (Neil)
- Fix to allow restoring dumps that have cross-schema references to custom operators (Tom)
- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)

E.161. Release 7.4.11

Release Date

2006-01-09

This release contains a variety of fixes from 7.4.10. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.161.1. Migration to Version 7.4.11

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see Section E.164, « Release 7.4.8 ». Also, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.161.2. Changes

- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require **REINDEX** to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need **REINDEX** to fix existing indexes on textual columns if this has happened to you.
- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.
- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.162. Release 7.4.10

Release Date

2005-12-12

This release contains a variety of fixes from 7.4.9. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.162.1. Migration to Version 7.4.10

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see Section E.164, « Release 7.4.8 ».

E.162.2. Changes

- Fix race condition in transaction log management

  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- `/contrib/ltree` fixes (Teodor)
- AIX and HPUX compile fixes (Tom)
- Fix longstanding planning error for outer joins

  This bug sometimes caused a bogus error « RIGHT JOIN is only supported with merge-joinable join conditions ».
- Prevent core dump in pg_autovacuum when a table has been dropped

E.163. Release 7.4.9

Release Date

2005-10-04

This release contains a variety of fixes from 7.4.8. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.163.1. Migration to Version 7.4.9

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see Section E.164, « Release 7.4.8 ».

E.163.2. Changes

- Fix error that allowed **VACUUM** to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

  This fixes a long-standing problem that could cause crashes in very rare circumstances.
- Fix CHAR() to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of CHAR() was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
- Fix the sense of the test for read-only transaction in **COPY**

  The code formerly prohibited **COPY TO**, where it should prohibit **COPY FROM**.
- Fix planning problem with outer-join ON clauses that reference only the inner-side relation
- Further fixes for `x FULL JOIN y ON true` corner cases
- Make `array_in` and `array_recv` more paranoid about validating their OID parameter
- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve robustness of datetime parsing
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Don't try to open more than `max_files_per_process` files during postmaster startup
- Various memory leakage fixes
- Various portability improvements
- Fix PL/pgSQL to handle `var := var` correctly when the variable is of pass-by-reference type
- Update `contrib/tsearch2` to use current Snowball code

E.164. Release 7.4.8

Release Date

2005-05-09

This release contains a variety of fixes from 7.4.7, including several security-related issues. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.164.1. Migration to Version 7.4.8

A dump/restore is not required for those running 7.4.X. However, it is one possible way of handling two significant security problems that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.8's initdb will automatically correct these problems.

The larger security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.)

The lesser problem is that the `contrib/tsearch2` module creates several functions that are misdeclared to return internal when they do not accept internal arguments. This breaks type safety for all functions using internal arguments.

It is strongly recommended that all installations repair these errors, either by initdb or by following the manual repair procedures given below. The errors at least allow unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the following procedures instead. As the database superuser, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

Next, if you have installed `contrib/tsearch2`, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN (
   'dex_init(text)'::regprocedure,
   'snb_en_init(text)'::regprocedure,
   'snb_ru_init(text)'::regprocedure,
   'spell_init(text)'::regprocedure,
   'syn_init(text)'::regprocedure
);
-- The command should report having updated 5 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

If this command fails with a message like « function "dex_init(text)" does not exist », then either `tsearch2` is not installed in this database, or you already did the update.

The above procedures must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedures. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.164.2. Changes

- Change encoding function signature to prevent misuse
- Change `contrib/tsearch2` to avoid unsafe use of INTERNAL function results
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low

probability. There are no known cases of it having caused more than an Assert failure.

- Fix comparisons of TIME WITH TIME ZONE values

  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. NOTE: if you have an index on a TIME WITH TIME ZONE column, it will need to be **REINDEX**ed after installing this update, because the fix corrects the sort order of column values.

- Fix `EXTRACT(EPOCH)` for TIME WITH TIME ZONE values
- Fix mis-display of negative fractional seconds in INTERVAL values

  This error only occurred when the `--enable-integer-datetimes` configuration switch had been used.

- Ensure operations done during backend shutdown are counted by statistics collector

  This is expected to resolve reports of pg_autovacuum not vacuuming the system catalogs often enough -- it was not being told about catalog deletions caused by temporary table removal during backend exit.

- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump trigger names containing `%` correctly (Neil)
- Fix `contrib/pgcrypto` for newer OpenSSL builds (Marko Kreen)
- Still more 64-bit fixes for `contrib/intagg`
- Prevent incorrect optimization of functions returning RECORD
- Prevent `to_char(interval)` from dumping core for month-related formats
- Prevent crash on `COALESCE(NULL,NULL)`
- Fix `array_map` to call PL functions correctly
- Fix permission checking in **ALTER DATABASE RENAME**
- Fix **ALTER LANGUAGE RENAME**
- Make `RemoveFromWaitQueue` clean up after itself

  This fixes a lock management error that would only be visible if a transaction was kicked out of a wait for a lock (typically by query cancel) and then the holder of the lock released it within a very narrow window.

- Fix problem with untyped parameter appearing in **INSERT ... SELECT**
- Fix **CLUSTER** failure after **ALTER TABLE SET WITHOUT OIDS**

E.165. Release 7.4.7

Release Date

2005-01-31

This release contains a variety of fixes from 7.4.6, including several security-related issues. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.165.1. Migration to Version 7.4.7

A dump/restore is not required for those running 7.4.X.

E.165.2. Changes

- Disallow **LOAD** to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), **LOAD** can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Check that creator of an aggregate function has the right to execute the specified transition functions

  This oversight made it possible to bypass denial of EXECUTE permission on a function.

- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles
- Make age(timestamptz) do calculation in local timezone not GMT

E.166. Release 7.4.6

Release Date

2004-10-22

This release contains a variety of fixes from 7.4.5. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.166.1. Migration to Version 7.4.6

A dump/restore is not required for those running 7.4.X.

E.166.2. Changes

- Repair possible failure to update hint bits on disk

  Under rare circumstances this oversight could lead to « could not access transaction status » failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples

  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg_ctl as root

  This is to guard against any possible security issues.
- Avoid using temp files in `/tmp` in **make_oidjoins_check**

  This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.
- Prevent forced backend shutdown from re-emitting prior command result

  In rare cases, a client might think that its last command had succeeded when it really had been aborted by forced database shutdown.
- Repair bug in `pg_stat_get_backend_idset`

  This could lead to misbehavior in some of the system-statistics views.
- Fix small memory leak in postmaster
- Fix « expected both swapped tables to have TOAST tables » bug

  This could arise in cases such as CLUSTER after ALTER TABLE DROP COLUMN.
- Prevent `pg_ctl restart` from adding `-D` multiple times
- Fix problem with NULL values in GiST indexes
- `::` is no longer interpreted as a variable in an ECPG prepare statement

E.167. Release 7.4.5

Release Date

2004-08-18

This release contains one serious bug fix over 7.4.4. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.167.1. Migration to Version 7.4.5

A dump/restore is not required for those running 7.4.X.

E.167.2. Changes

- Repair possible crash during concurrent B-tree index insertions

  This patch fixes a rare case in which concurrent insertions into a B-tree index could result in a server panic. No permanent damage would result, but it's still worth a re-release. The bug does not exist in pre-7.4 releases.

E.168. Release 7.4.4

Release Date

2004-08-16

This release contains a variety of fixes from 7.4.3. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.168.1. Migration to Version 7.4.4

A dump/restore is not required for those running 7.4.X.

E.168.2. Changes

- Prevent possible loss of committed transactions during crash

  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL™ 7.1.
- Check HAVING restriction before evaluating result list of an aggregate plan
- Avoid crash when session's current user ID is deleted
- Fix hashed crosstab for zero-rows case (Joe)
- Force cache update after renaming a column in a foreign key
- Pretty-print UNION queries correctly
- Make psql handle `\r\n` newlines properly in COPY IN
- pg_dump handled ACLs with grant options incorrectly
- Fix thread support for OS X and Solaris
- Updated JDBC driver (build 215) with various fixes
- ECPG fixes
- Translation updates (various contributors)

E.169. Release 7.4.3

Release Date

2004-06-14

This release contains a variety of fixes from 7.4.2. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.169.1. Migration to Version 7.4.3

A dump/restore is not required for those running 7.4.X.

E.169.2. Changes

- Fix temporary memory leak when using non-hashed aggregates (Tom)
- ECPG fixes, including some for Informix compatibility (Michael)
- Fixes for compiling with thread-safety, particularly Solaris (Bruce)
- Fix error in COPY IN termination when using the old network protocol (ljb)
- Several important fixes in pg_autovacuum, including fixes for large tables, unsigned oids, stability, temp tables, and debug mode (Matthew T. O'Connor)
- Fix problem with reading tar-format dumps on NetBSD and BSD/OS (Bruce)
- Several JDBC fixes
- Fix ALTER SEQUENCE RESTART where last_value equals the restart value (Tom)
- Repair failure to recalculate nested sub-selects (Tom)
- Fix problems with non-constant expressions in LIMIT/OFFSET
- Support FULL JOIN with no join clause, such as X FULL JOIN Y ON TRUE (Tom)
- Fix another zero-column table bug (Tom)
- Improve handling of non-qualified identifiers in GROUP BY clauses in sub-selects (Tom)

  Select-list aliases within the sub-select will now take precedence over names from outer query levels.
- Do not generate « NATURAL CROSS JOIN » when decompiling rules (Tom)
- Add checks for invalid field length in binary COPY (Tom)

  This fixes a difficult-to-exploit security hole.
- Avoid locking conflict between **ANALYZE** and **LISTEN/NOTIFY**
- Numerous translation updates (various contributors)

E.170. Release 7.4.2

Release Date

2004-03-08

This release contains a variety of fixes from 7.4.1. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.170.1. Migration to Version 7.4.2

A dump/restore is not required for those running 7.4.X. However, it might be advisable as the easiest method of incorporating fixes for two errors that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.2's initdb will automatically correct these problems.

The more severe of the two errors is that data type anyarray has the wrong alignment label; this is a problem because the pg_statistic system catalog uses anyarray columns. The mislabeling can cause planner misestimations and even crashes when

planning queries that involve `WHERE` clauses on double-aligned columns (such as float8 and timestamp). It is strongly recommended that all installations repair this error, either by initdb or by following the manual repair procedure given below.

The lesser error is that the system view pg_settings ought to be marked as having public update access, to allow `UPDATE pg_settings` to be used as a substitute for **SET**. This can also be fixed either by initdb or manually, but it is not necessary to fix unless you want to use `UPDATE pg_settings`.

If you wish not to do an initdb, the following procedure will work for fixing pg_statistic. As the database superuser, do:

```
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
--
-- repopulate pg_statistic:
ANALYZE;
```

This can be done in a live database, but beware that all backends running in the altered database must be restarted before it is safe to repopulate pg_statistic.

To repair the pg_settings error, simply do:

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

The above procedures must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedures. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.170.2. Changes

Release 7.4.2 incorporates all the fixes included in release 7.3.6, plus the following fixes:

- Fix pg_statistics alignment bug that could crash optimizer

  See above for details about this problem.
- Allow non-super users to update pg_settings
- Fix several optimizer bugs, most of which led to « variable not found in subplan target lists » errors
- Avoid out-of-memory failure during startup of large multiple index scan
- Fix multibyte problem that could lead to « out of memory » error during **COPY IN**
- Fix problems with **SELECT INTO** / **CREATE TABLE AS** from tables without OIDs
- Fix problems with `alter_table` regression test during parallel testing
- Fix problems with hitting open file limit, especially on OS X (Tom)
- Partial fix for Turkish-locale issues

  initdb will succeed now in Turkish locale, but there are still some inconveniences associated with the `i/I` problem.

- Make pg_dump set client encoding on restore
- Other minor pg_dump fixes
- Allow ecpg to again use C keywords as column names (Michael)
- Added ecpg `WHENEVER NOT_FOUND` to `SELECT/INSERT/UPDATE/DELETE` (Michael)
- Fix ecpg crash for queries calling set-returning functions (Michael)
- Various other ecpg fixes (Michael)
- Fixes for Borland compiler
- Thread build improvements (Bruce)
- Various other build fixes
- Various JDBC fixes

E.171. Release 7.4.1

Release Date

2003-12-22

This release contains a variety of fixes from 7.4. For information about new features in the 7.4 major release, see Section E.172, « Release 7.4 ».

E.171.1. Migration to Version 7.4.1

A dump/restore is *not* required for those running 7.4.

If you want to install the fixes in the information schema you need to reload it into the database. This is either accomplished by initializing a new cluster by running **initdb**, or by running the following sequence of SQL commands in each database (ideally including `template1`) as a superuser in psql, after installing the new release:

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

Substitute your installation path in the second command.

E.171.2. Changes

- Fixed bug in **CREATE SCHEMA** parsing in ECPG (Michael)
- Fix compile error when `--enable-thread-safety` and `--with-perl` are used together (Peter)
- Fix for subqueries that used hash joins (Tom)

  Certain subqueries that used hash joins would crash because of improperly shared structures.
- Fix free space map compaction bug (Tom)

  This fixes a bug where compaction of the free space map could lead to a database server shutdown.
- Fix for Borland compiler build of libpq (Bruce)
- Fix `netmask()` and `hostmask()` to return the maximum-length masklen (Tom)

  Fix these functions to return values consistent with pre-7.4 releases.
- Several `contrib/pg_autovacuum` fixes

  Fixes include improper variable initialization, missing vacuum after **TRUNCATE**, and duration computation overflow for long vacuums.
- Allow compile of `contrib/cube` under Cygwin (Jason Tishler)
- Fix Solaris use of password file when no passwords are defined (Tom)

Fix crash on Solaris caused by use of any type of password authentication when no passwords were defined.

- JDBC fix for thread problems, other fixes
- Fix for bytea index lookups (Joe)
- Fix information schema for bit data types (Peter)
- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of « variable not in subplan target lists »
- Make `PQescapeBytea` and `byteaout` consistent with each other (Joe)
- Escape bytea output for bytes > 0x7e(Joe)

  If different client encodings are used for bytea output and input, it is possible for bytea values to be corrupted by the differing encodings. This fix escapes all bytes that might be affected.
- Added missing `SPI_finish()` calls to dblink's `get_tuple_of_interest()` (Joe)
- New Czech FAQ
- Fix information schema view `constraint_column_usage` for foreign keys (Peter)
- ECPG fixes (Michael)
- Fix bug with multiple `IN` subqueries and joins in the subqueries (Tom)
- Allow `COUNT('x')` to work (Tom)
- Install ECPG include files for Informix compatibility into separate directory (Peter)

  Some names of ECPG include files for Informix compatibility conflicted with operating system include files. By installing them in their own directory, name conflicts have been reduced.
- Fix SSL memory leak (Neil)

  This release fixes a bug in 7.4 where SSL didn't free all memory it allocated.
- Prevent `pg_service.conf` from using service name as default dbname (Bruce)
- Fix local ident authentication on FreeBSD (Tom)

E.172. Release 7.4

Release Date

2003-11-17

E.172.1. Overview

Major changes in this release:

`IN` / `NOT IN` subqueries are now much more efficient
: In previous releases, `IN`/`NOT IN` subqueries were joined to the upper query by sequentially scanning the subquery looking for a match. The 7.4 code uses the same sophisticated techniques used by ordinary joins and so is much faster. An `IN` will now usually be as fast as or faster than an equivalent `EXISTS` subquery; this reverses the conventional wisdom that applied to previous releases.

Improved `GROUP BY` processing by using hash buckets
: In previous releases, rows to be grouped had to be sorted first. The 7.4 code can do `GROUP BY` without sorting, by accumulating results into a hash table with one entry per group. It will still use the sort technique, however, if the hash table is estimated to be too large to fit in `sort_mem`.

New multikey hash join capability
: In previous releases, hash joins could only occur on single keys. This release allows multicolumn hash joins.

Queries using the explicit `JOIN` syntax are now better optimized
: Prior releases evaluated queries using the explicit `JOIN` syntax only in the order implied by the syntax. 7.4 allows full optimi-

zation of these queries, meaning the optimizer considers all possible join orderings and chooses the most efficient. Outer joins, however, must still follow the declared ordering.

Faster and more powerful regular expression code
: The entire regular expression module has been replaced with a new version by Henry Spencer, originally written for Tcl. The code greatly improves performance and supports several flavors of regular expressions.

Function-inlining for simple SQL functions
: Simple SQL functions can now be inlined by including their SQL in the main query. This improves performance by eliminating per-call overhead. That means simple SQL functions now behave like macros.

Full support for IPv6 connections and IPv6 address data types
: Previous releases allowed only IPv4 connections, and the IP data types only supported IPv4 addresses. This release adds full IPv6 support in both of these areas.

Major improvements in SSL performance and reliability
: Several people very familiar with the SSL API have overhauled our SSL code to improve SSL key negotiation and error recovery.

Make free space map efficiently reuse empty index pages, and other free space management improvements
: In previous releases, B-tree index pages that were left empty because of deleted rows could only be reused by rows with index values similar to the rows originally indexed on that page. In 7.4, **VACUUM** records empty index pages and allows them to be reused for any future index rows.

SQL-standard information schema
: The information schema provides a standardized and stable way to access information about the schema objects defined in a database.

Cursors conform more closely to the SQL standard
: The commands **FETCH** and **MOVE** have been overhauled to conform more closely to the SQL standard.

Cursors can exist outside transactions
: These cursors are also called holdable cursors.

New client-to-server protocol
: The new protocol adds error codes, more status information, faster startup, better support for binary data transmission, parameter values separated from SQL commands, prepared statements available at the protocol level, and cleaner recovery from **COPY** failures. The older protocol is still supported by both server and clients.

libpq and ECPG applications are now fully thread-safe
: While previous libpq releases already supported threads, this release improves thread safety by fixing some non-thread-safe code that was used during database connection startup. The **configure** option `--enable-thread-safety` must be used to enable this feature.

New version of full-text indexing
: A new full-text indexing suite is available in `contrib/tsearch2`.

New autovacuum tool
: The new autovacuum tool in `contrib/autovacuum` monitors the database statistics tables for **INSERT**/**UPDATE**/**DELETE** activity and automatically vacuums tables when needed.

Array handling has been improved and moved into the server core
: Many array limitations have been removed, and arrays behave more like fully-supported data types.

E.172.2. Migration to Version 7.4

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The server-side autocommit setting was removed and reimplemented in client applications and languages. Server-side autocommit was causing too many problems with languages and applications that wanted to control their own autocommit behavior, so autocommit was removed from the server and added to individual client APIs as appropriate.
- Error message wording has changed substantially in this release. Significant effort was invested to make the messages more consistent and user-oriented. If your applications try to detect different error conditions by parsing the error message, you are strongly encouraged to use the new error code facility instead.
- Inner joins using the explicit `JOIN` syntax might behave differently because they are now better optimized.

- A number of server configuration parameters have been renamed for clarity, primarily those related to logging.
- `FETCH 0` or `MOVE 0` now does nothing. In prior releases, `FETCH 0` would fetch all remaining rows, and `MOVE 0` would move to the end of the cursor.
- **FETCH** and **MOVE** now return the actual number of rows fetched/moved, or zero if at the beginning/end of the cursor. Prior releases would return the row count passed to the command, not the number of rows actually fetched or moved.
- **COPY** now can process files that use carriage-return or carriage-return/line-feed end-of-line sequences. Literal carriage-returns and line-feeds are no longer accepted in data values; use `\r` and `\n` instead.
- Trailing spaces are now trimmed when converting from type char(*n*) to varchar(*n*) or text. This is what most people always expected to happen anyway.
- The data type float(*p*) now measures *p* in binary digits, not decimal digits. The new behavior follows the SQL standard.
- Ambiguous date values now must match the ordering specified by the `datestyle` setting. In prior releases, a date specification of `10/20/03` was interpreted as a date in October even if `datestyle` specified that the day should be first. 7.4 will throw an error if a date specification is invalid for the current setting of `datestyle`.
- The functions `oidrand`, `oidsrand`, and `userfntest` have been removed. These functions were determined to be no longer useful.
- String literals specifying time-varying date/time values, such as `'now'` or `'today'` will no longer work as expected in column default expressions; they now cause the time of the table creation to be the default, not the time of the insertion. Functions such as `now()`, `current_timestamp`, or `current_date` should be used instead.

  In previous releases, there was special code so that strings such as `'now'` were interpreted at **INSERT** time and not at table creation time, but this work around didn't cover all cases. Release 7.4 now requires that defaults be defined properly using functions such as `now()` or `current_timestamp`. These will work in all situations.
- The dollar sign ($) is no longer allowed in operator names. It can instead be a non-first character in identifiers. This was done to improve compatibility with other database systems, and to avoid syntax problems when parameter placeholders ($*n*) are written adjacent to operators.

E.172.3. Changes

Below you will find a detailed account of the changes between release 7.4 and the previous major release.

E.172.3.1. Server Operation Changes

- Allow IPv6 server connections (Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
- Fix SSL to handle errors cleanly (Nathan Mueller)

  In prior releases, certain SSL API error reports were not handled correctly. This release fixes those problems.
- SSL protocol security and performance improvements (Sean Chittenden)

  SSL key renegotiation was happening too frequently, causing poor SSL performance. Also, initial key handling was improved.
- Print lock information when a deadlock is detected (Tom)

  This allows easier debugging of deadlock situations.
- Update `/tmp` socket modification times regularly to avoid their removal (Tom)

  This should help prevent `/tmp` directory cleaner administration scripts from removing server socket files.
- Enable PAM for Mac OS X (Aaron Hillegass)
- Make B-tree indexes fully WAL-safe (Tom)

  In prior releases, under certain rare cases, a server crash could cause B-tree indexes to become corrupt. This release removes those last few rare cases.
- Allow B-tree index compaction and empty page reuse (Tom)
- Fix inconsistent index lookups during split of first root page (Tom)

  In prior releases, when a single-page index split into two pages, there was a brief period when another database session could miss seeing an index entry. This release fixes that rare failure case.

- Improve free space map allocation logic (Tom)
- Preserve free space information between server restarts (Tom)

  In prior releases, the free space map was not saved when the postmaster was stopped, so newly started servers had no free space information. This release saves the free space map, and reloads it when the server is restarted.
- Add start time to `pg_stat_activity` (Neil)
- New code to detect corrupt disk pages; erase with `zero_damaged_pages` (Tom)
- New client/server protocol: faster, no username length limit, allow clean exit from **COPY** (Tom)
- Add transaction status, table ID, column ID to client/server protocol (Tom)
- Add binary I/O to client/server protocol (Tom)
- Remove autocommit server setting; move to client applications (Tom)
- New error message wording, error codes, and three levels of error detail (Tom, Joe, Peter)

E.172.3.2. Performance Improvements

- Add hashing for `GROUP BY` aggregates (Tom)
- Make nested-loop joins be smarter about multicolumn indexes (Tom)
- Allow multikey hash joins (Tom)
- Improve constant folding (Tom)
- Add ability to inline simple SQL functions (Tom)
- Reduce memory usage for queries using complex functions (Tom)

  In prior releases, functions returning allocated memory would not free it until the query completed. This release allows the freeing of function-allocated memory when the function call completes, reducing the total memory used by functions.
- Improve GEQO optimizer performance (Tom)

  This release fixes several inefficiencies in the way the GEQO optimizer manages potential query paths.
- Allow `IN`/`NOT IN` to be handled via hash tables (Tom)
- Improve `NOT IN (`*`subquery`*`)` performance (Tom)
- Allow most `IN` subqueries to be processed as joins (Tom)
- Pattern matching operations can use indexes regardless of locale (Peter)

  There is no way for non-ASCII locales to use the standard indexes for `LIKE` comparisons. This release adds a way to create a special index for `LIKE`.
- Allow the postmaster to preload libraries using `preload_libraries` (Joe)

  For shared libraries that require a long time to load, this option is available so the library can be preloaded in the postmaster and alled by all database sessions.
- Improve optimizer cost computations, particularly for subqueries (Tom)
- Avoid sort when subquery `ORDER BY` matches upper query (Tom)
- Deduce that `WHERE a.x = b.y AND b.y = 42` also means `a.x = 42` (Tom)
- Allow hash/merge joins on complex joins (Tom)
- Allow hash joins for more data types (Tom)
- Allow join optimization of explicit inner joins, disable with `join_collapse_limit` (Tom)
- Add parameter `from_collapse_limit` to control conversion of subqueries to joins (Tom)
- Use faster and more powerful regular expression code from Tcl (Henry Spencer, Tom)
- Use bit-mapped relation sets in the optimizer (Tom)
- Improve connection startup time (Tom)

The new client/server protocol requires fewer network packets to start a database session.

- Improve trigger/constraint performance (Stephan)
- Improve speed of `col IN (const, const, const, ...)` (Tom)
- Fix hash indexes which were broken in rare cases (Tom)
- Improve hash index concurrency and speed (Tom)

  Prior releases suffered from poor hash index performance, particularly for high concurrency situations. This release fixes that, and the development group is interested in reports comparing B-tree and hash index performance.
- Align shared buffers on 32-byte boundary for copy speed improvement (Manfred Spraul)

  Certain CPU's perform faster data copies when addresses are 32-byte aligned.
- Data type numeric reimplemented for better performance (Tom)

  numeric used to be stored in base 100. The new code uses base 10000, for significantly better performance.

E.172.3.3. Server Configuration Changes

- Rename server parameter `server_min_messages` to `log_min_messages` (Bruce)

  This was done so most parameters that control the server logs begin with `log_`.
- Rename `show_*_stats` to `log_*_stats` (Bruce)
- Rename `show_source_port` to `log_source_port` (Bruce)
- Rename `hostname_lookup` to `log_hostname` (Bruce)
- Add `checkpoint_warning` to warn of excessive checkpointing (Bruce)

  In prior releases, it was difficult to determine if checkpoint was happening too frequently. This feature adds a warning to the server logs when excessive checkpointing happens.
- New read-only server parameters for localization (Tom)
- Change debug server log messages to output as `DEBUG` rather than `LOG` (Bruce)
- Prevent server log variables from being turned off by non-superusers (Bruce)

  This is a security feature so non-superusers cannot disable logging that was enabled by the administrator.
- `log_min_messages`/`client_min_messages` now controls `debug_*` output (Bruce)

  This centralizes client debug information so all debug output can be sent to either the client or server logs.
- Add Mac OS X Rendezvous server support (Chris Campbell)

  This allows Mac OS X hosts to query the network for available PostgreSQL™ servers.
- Add ability to print only slow statements using `log_min_duration_statement` (Christopher)

  This is an often requested debugging feature that allows administrators to see only slow queries in their server logs.
- Allow `pg_hba.conf` to accept netmasks in CIDR format (Andrew Dunstan)

  This allows administrators to merge the host IP address and netmask fields into a single CIDR field in `pg_hba.conf`.
- New read-only parameter `is_superuser` (Tom)
- New parameter `log_error_verbosity` to control error detail (Tom)

  This works with the new error reporting feature to supply additional error information like hints, file names and line numbers.
- `postgres --describe-config` now dumps server config variables (Aizaz Ahmed, Peter)

  This option is useful for administration tools that need to know the configuration variable names and their minimums, maximums, defaults, and descriptions.
- Add new columns in `pg_settings`: `context`, `type`, `source`, `min_val`, `max_val` (Joe)
- Make default `shared_buffers` 1000 and `max_connections` 100, if possible (Tom)

Prior versions defaulted to 64 shared buffers so PostgreSQL™ would start on even very old systems. This release tests the amount of shared memory allowed by the platform and selects more reasonable default values if possible. Of course, users are still encouraged to evaluate their resource load and size `shared_buffers` accordingly.

- New `pg_hba.conf` record type `hostnossl` to prevent SSL connections (Jon Jensen)

  In prior releases, there was no way to prevent SSL connections if both the client and server supported SSL. This option allows that capability.

- Remove parameter `geqo_random_seed` (Tom)
- Add server parameter `regex_flavor` to control regular expression processing (Tom)
- Make **pg_ctl** better handle nonstandard ports (Greg)

E.172.3.4. Query Changes

- New SQL-standard information schema (Peter)
- Add read-only transactions (Peter)
- Print key name and value in foreign-key violation messages (Dmitry Tkach)
- Allow users to see their own queries in `pg_stat_activity` (Kevin Brown)

  In prior releases, only the superuser could see query strings using `pg_stat_activity`. Now ordinary users can see their own query strings.

- Fix aggregates in subqueries to match SQL standard (Tom)

  The SQL standard says that an aggregate function appearing within a nested subquery belongs to the outer query if its argument contains only outer-query variables. Prior PostgreSQL™ releases did not handle this fine point correctly.

- Add option to prevent auto-addition of tables referenced in query (Nigel J. Andrews)

  By default, tables mentioned in the query are automatically added to the `FROM` clause if they are not already there. This is compatible with historic POSTGRES™ behavior but is contrary to the SQL standard. This option allows selecting standard-compatible behavior.

- Allow `UPDATE ... SET col = DEFAULT` (Rod)

  This allows **UPDATE** to set a column to its declared default value.

- Allow expressions to be used in `LIMIT`/`OFFSET` (Tom)

  In prior releases, `LIMIT`/`OFFSET` could only use constants, not expressions.

- Implement `CREATE TABLE AS EXECUTE` (Neil, Peter)

E.172.3.5. Object Manipulation Changes

- Make **CREATE SEQUENCE** grammar more conforming to SQL:2003 (Neil)
- Add statement-level triggers (Neil)

  While this allows a trigger to fire at the end of a statement, it does not allow the trigger to access all rows modified by the statement. This capability is planned for a future release.

- Add check constraints for domains (Rod)

  This greatly increases the usefulness of domains by allowing them to use check constraints.

- Add **ALTER DOMAIN** (Rod)

  This allows manipulation of existing domains.

- Fix several zero-column table bugs (Tom)

  PostgreSQL™ supports zero-column tables. This fixes various bugs that occur when using such tables.

- Have `ALTER TABLE ... ADD PRIMARY KEY` add not-null constraint (Rod)

  In prior releases, `ALTER TABLE ... ADD PRIMARY` would add a unique index, but not a not-null constraint. That is

fixed in this release.

- Add `ALTER TABLE ... WITHOUT OIDS` (Rod)

  This allows control over whether new and updated rows will have an OID column. This is most useful for saving storage space.

- Add `ALTER SEQUENCE` to modify minimum, maximum, increment, cache, cycle values (Rod)
- Add `ALTER TABLE ... CLUSTER ON` (Alvaro Herrera)

  This command is used by **pg_dump** to record the cluster column for each table previously clustered. This information is used by database-wide cluster to cluster all previously clustered tables.

- Improve automatic type casting for domains (Rod, Tom)
- Allow dollar signs in identifiers, except as first character (Tom)
- Disallow dollar signs in operator names, so `x=$1` works (Tom)
- Allow copying table schema using `LIKE` *`subtable`*, also SQL:2003 feature `INCLUDING DEFAULTS` (Rod)
- Add `WITH GRANT OPTION` clause to **GRANT** (Peter)

  This enabled **GRANT** to give other users the ability to grant privileges on a object.

E.172.3.6. Utility Command Changes

- Add `ON COMMIT` clause to **CREATE TABLE** for temporary tables (Gavin)

  This adds the ability for a table to be dropped or all rows deleted on transaction commit.

- Allow cursors outside transactions using `WITH HOLD` (Neil)

  In previous releases, cursors were removed at the end of the transaction that created them. Cursors can now be created with the `WITH HOLD` option, which allows them to continue to be accessed after the creating transaction has committed.

- `FETCH 0` and `MOVE 0` now do nothing (Bruce)

  In previous releases, `FETCH 0` fetched all remaining rows, and `MOVE 0` moved to the end of the cursor.

- Cause **FETCH** and **MOVE** to return the number of rows fetched/moved, or zero if at the beginning/end of cursor, per SQL standard (Bruce)

  In prior releases, the row count returned by **FETCH** and **MOVE** did not accurately reflect the number of rows processed.

- Properly handle `SCROLL` with cursors, or report an error (Neil)

  Allowing random access (both forward and backward scrolling) to some kinds of queries cannot be done without some additional work. If `SCROLL` is specified when the cursor is created, this additional work will be performed. Furthermore, if the cursor has been created with `NO SCROLL`, no random access is allowed.

- Implement SQL-compatible options `FIRST`, `LAST`, `ABSOLUTE` *`n`*, `RELATIVE` *`n`* for **FETCH** and **MOVE** (Tom)
- Allow **EXPLAIN** on **DECLARE CURSOR** (Tom)
- Allow **CLUSTER** to use index marked as pre-clustered by default (Alvaro Herrera)
- Allow **CLUSTER** to cluster all tables (Alvaro Herrera)

  This allows all previously clustered tables in a database to be reclustered with a single command.

- Prevent **CLUSTER** on partial indexes (Tom)
- Allow DOS and Mac line-endings in **COPY** files (Bruce)
- Disallow literal carriage return as a data value, backslash-carriage-return and `\r` are still allowed (Bruce)
- **COPY** changes (binary, `\.`) (Tom)
- Recover from **COPY** failure cleanly (Tom)
- Prevent possible memory leaks in **COPY** (Tom)
- Make **TRUNCATE** transaction-safe (Rod)

  TRUNCATE can now be used inside a transaction. If the transaction aborts, the changes made by the **TRUNCATE** are auto-

matically rolled back.

- Allow prepare/bind of utility commands like **FETCH** and **EXPLAIN** (Tom)
- Add **EXPLAIN EXECUTE** (Neil)
- Improve **VACUUM** performance on indexes by reducing WAL traffic (Tom)
- Functional indexes have been generalized into indexes on expressions (Tom)

  In prior releases, functional indexes only supported a simple function applied to one or more column names. This release allows any type of scalar expression.
- Have **SHOW TRANSACTION ISOLATION** match input to **SET TRANSACTION ISOLATION** (Tom)
- Have **COMMENT ON DATABASE** on nonlocal database generate a warning, rather than an error (Rod)

  Database comments are stored in database-local tables so comments on a database have to be stored in each database.
- Improve reliability of **LISTEN**/**NOTIFY** (Tom)
- Allow **REINDEX** to reliably reindex nonshared system catalog indexes (Tom)

  This allows system tables to be reindexed without the requirement of a standalone session, which was necessary in previous releases. The only tables that now require a standalone session for reindexing are the global system tables `pg_database`, `pg_shadow`, and `pg_group`.

E.172.3.7. Data Type and Function Changes

- New server parameter `extra_float_digits` to control precision display of floating-point numbers (Pedro Ferreira, Tom)

  This controls output precision which was causing regression testing problems.
- Allow `+1300` as a numeric time-zone specifier, for FJST (Tom)
- Remove rarely used functions `oidrand`, `oidsrand`, and `userfntest` functions (Neil)
- Add `md5()` function to main server, already in `contrib/pgcrypto` (Joe)

  An MD5 function was frequently requested. For more complex encryption capabilities, use `contrib/pgcrypto`.
- Increase date range of timestamp (John Cochran)
- Change `EXTRACT(EPOCH FROM timestamp)` so timestamp without time zone is assumed to be in local time, not GMT (Tom)
- Trap division by zero in case the operating system doesn't prevent it (Tom)
- Change the numeric data type internally to base 10000 (Tom)
- New `hostmask()` function (Greg Wickham)
- Fixes for `to_char()` and `to_timestamp()` (Karel)
- Allow functions that can take any argument data type and return any data type, using anyelement and anyarray (Joe)

  This allows the creation of functions that can work with any data type.
- Arrays can now be specified as `ARRAY[1,2,3]`, `ARRAY[['a','b'],['c','d']]`, or `ARRAY[ARRAY[ARRAY[2]]]` (Joe)
- Allow proper comparisons for arrays, including `ORDER BY` and `DISTINCT` support (Joe)
- Allow indexes on array columns (Joe)
- Allow array concatenation with `||` (Joe)
- Allow `WHERE` qualification *`expr op`* `ANY/SOME/ALL (`*`array_expr`*`)` (Joe)

  This allows arrays to behave like a list of values, for purposes like `SELECT * FROM tab WHERE col IN (array_val)`.
- New array functions `array_append`, `array_cat`, `array_lower`, `array_prepend`, `array_to_string`, `array_upper`, `string_to_array` (Joe)
- Allow user defined aggregates to use polymorphic functions (Joe)

- Allow assignments to empty arrays (Joe)
- Allow 60 in seconds fields of time, timestamp, and interval input values (Tom)

  Sixty-second values are needed for leap seconds.
- Allow cidr data type to be cast to text (Tom)
- Disallow invalid time zone names in SET TIMEZONE
- Trim trailing spaces when char is cast to varchar or text (Tom)
- Make float(*p*) measure the precision *p* in binary digits, not decimal digits (Tom)
- Add IPv6 support to the inet and cidr data types (Michael Graff)
- Add `family()` function to report whether address is IPv4 or IPv6 (Michael Graff)
- Have `SHOW datestyle` generate output similar to that used by `SET datestyle` (Tom)
- Make `EXTRACT(TIMEZONE)` and `SET/SHOW TIME ZONE` follow the SQL convention for the sign of time zone offsets, i.e., positive is east from UTC (Tom)
- Fix `date_trunc('quarter', ...)` (Böjthe Zoltán)

  Prior releases returned an incorrect value for this function call.
- Make `initcap()` more compatible with Oracle (Mike Nolan)

  `initcap()` now uppercases a letter appearing after any non-alphanumeric character, rather than only after whitespace.
- Allow only `datestyle` field order for date values not in ISO-8601 format (Greg)
- Add new `datestyle` values `MDY`, `DMY`, and `YMD` to set input field order; honor `US` and `European` for backward compatibility (Tom)
- String literals like `'now'` or `'today'` will no longer work as a column default. Use functions such as `now()`, `current_timestamp` instead. (change required for prepared statements) (Tom)
- Treat NaN as larger than any other value in `min()`/`max()` (Tom)

  NaN was already sorted after ordinary numeric values for most purposes, but `min()` and `max()` didn't get this right.
- Prevent interval from suppressing `:00` seconds display
- New functions `pg_get_triggerdef(prettyprint)` and `pg_conversion_is_visible()` (Christopher)
- Allow time to be specified as `040506` or `0405` (Tom)
- Input date order must now be `YYYY-MM-DD` (with 4-digit year) or match `datestyle`
- Make `pg_get_constraintdef` support unique, primary-key, and check constraints (Christopher)

E.172.3.8. Server-Side Language Changes

- Prevent PL/pgSQL crash when `RETURN NEXT` is used on a zero-row record variable (Tom)
- Make PL/Python's `spi_execute` interface handle null values properly (Andrew Bosma)
- Allow PL/pgSQL to declare variables of composite types without `%ROWTYPE` (Tom)
- Fix PL/Python's `_quote()` function to handle big integers
- Make PL/Python an untrusted language, now called `plpythonu` (Kevin Jacobs, Tom)

  The Python language no longer supports a restricted execution environment, so the trusted version of PL/Python was removed. If this situation changes, a version of PL/Python that can be used by non-superusers will be readded.
- Allow polymorphic PL/pgSQL functions (Joe, Tom)
- Allow polymorphic SQL functions (Joe)
- Improved compiled function caching mechanism in PL/pgSQL with full support for polymorphism (Joe)
- Add new parameter `$0` in PL/pgSQL representing the function's actual return type (Joe)
- Allow PL/Tcl and PL/Python to use the same trigger on multiple tables (Tom)

- Fixed PL/Tcl's `spi_prepare` to accept fully qualified type names in the parameter type list (Jan)

E.172.3.9. psql Changes

- Add `\pset pager always` to always use pager (Greg)

  This forces the pager to be used even if the number of rows is less than the screen height. This is valuable for rows that wrap across several screen rows.
- Improve tab completion (Rod, Ross Reedstrom, Ian Barwick)
- Reorder `\?` help into groupings (Harald Armin Massa, Bruce)
- Add backslash commands for listing schemas, casts, and conversions (Christopher)
- **\encoding** now changes based on the server parameter `client_encoding` (Tom)

  In previous versions, **\encoding** was not aware of encoding changes made using `SET client_encoding`.
- Save editor buffer into readline history (Ross)

  When **\e** is used to edit a query, the result is saved in the readline history for retrieval using the up arrow.
- Improve **\d** display (Christopher)
- Enhance HTML mode to be more standards-conforming (Greg)
- New **\set AUTOCOMMIT off** capability (Tom)

  This takes the place of the removed server parameter `autocommit`.
- New **\set VERBOSITY** to control error detail (Tom)

  This controls the new error reporting details.
- New prompt escape sequence `%x` to show transaction status (Tom)
- Long options for psql are now available on all platforms

E.172.3.10. pg_dump Changes

- Multiple pg_dump fixes, including tar format and large objects
- Allow pg_dump to dump specific schemas (Neil)
- Make pg_dump preserve column storage characteristics (Christopher)

  This preserves `ALTER TABLE ... SET STORAGE` information.
- Make pg_dump preserve **CLUSTER** characteristics (Christopher)
- Have pg_dumpall use **GRANT**/**REVOKE** to dump database-level privileges (Tom)
- Allow pg_dumpall to support the options `-a`, `-s`, `-x` of pg_dump (Tom)
- Prevent pg_dump from lowercasing identifiers specified on the command line (Tom)
- pg_dump options `--use-set-session-authorization` and `--no-reconnect` now do nothing, all dumps use **SET SESSION AUTHORIZATION**

  pg_dump no longer reconnects to switch users, but instead always uses **SET SESSION AUTHORIZATION**. This will reduce password prompting during restores.
- Long options for pg_dump are now available on all platforms

  PostgreSQL™ now includes its own long-option processing routines.

E.172.3.11. libpq Changes

- Add function `PQfreemem` for freeing memory on Windows, suggested for **NOTIFY** (Bruce)

  Windows requires that memory allocated in a library be freed by a function in the same library, hence `free()` doesn't work for freeing memory allocated by libpq. `PQfreemem` is the proper way to free libpq memory, especially on Windows, and is

recommended for other platforms as well.

- Document service capability, and add sample file (Bruce)

  This allows clients to look up connection information in a central file on the client machine.
- Make `PQsetdbLogin` have the same defaults as `PQconnectdb` (Tom)
- Allow libpq to cleanly fail when result sets are too large (Tom)
- Improve performance of function `PQunescapeBytea` (Ben Lamb)
- Allow thread-safe libpq with `configure` option `--enable-thread-safety` (Lee Kindness, Philip Yarra)
- Allow function `pqInternalNotice` to accept a format string and arguments instead of just a preformatted message (Tom, Sean Chittenden)
- Control SSL negotiation with `sslmode` values `disable`, `allow`, `prefer`, and `require` (Jon Jensen)
- Allow new error codes and levels of text (Tom)
- Allow access to the underlying table and column of a query result (Tom)

  This is helpful for query-builder applications that want to know the underlying table and column names associated with a specific result set.
- Allow access to the current transaction status (Tom)
- Add ability to pass binary data directly to the server (Tom)
- Add function `PQexecPrepared` and `PQsendQueryPrepared` functions which perform bind/execute of previously prepared statements (Tom)

E.172.3.12. JDBC Changes

- Allow `setNull` on updateable result sets
- Allow `executeBatch` on a prepared statement (Barry)
- Support SSL connections (Barry)
- Handle schema names in result sets (Paul Sorenson)
- Add refcursor support (Nic Ferrier)

E.172.3.13. Miscellaneous Interface Changes

- Prevent possible memory leak or core dump during libpgtcl shutdown (Tom)
- Add Informix compatibility to ECPG (Michael)

  This allows ECPG to process embedded C programs that were written using certain Informix extensions.
- Add type decimal to ECPG that is fixed length, for Informix (Michael)
- Allow thread-safe embedded SQL programs with `configure` option `--enable-thread-safety` (Lee Kindness, Bruce)

  This allows multiple threads to access the database at the same time.
- Moved Python client PyGreSQL to *http://www.pygresql.org* (Marc)

E.172.3.14. Source Code Changes

- Prevent need for separate platform geometry regression result files (Tom)
- Improved PPC locking primitive (Reinhard Max)
- New function `palloc0` to allocate and clear memory (Bruce)
- Fix locking code for s390x CPU (64-bit) (Tom)
- Allow OpenBSD to use local ident credentials (William Ahern)

- Make query plan trees read-only to executor (Tom)
- Add Darwin startup scripts (David Wheeler)
- Allow libpq to compile with Borland C++ compiler (Lester Godwin, Karl Waclawek)
- Use our own version of `getopt_long()` if needed (Peter)
- Convert administration scripts to C (Peter)
- Bison >= 1.85 is now required to build the PostgreSQL™ grammar, if building from CVS
- Merge documentation into one book (Peter)
- Add Windows compatibility functions (Bruce)
- Allow client interfaces to compile under MinGW (Bruce)
- New `ereport()` function for error reporting (Tom)
- Support Intel compiler on Linux (Peter)
- Improve Linux startup scripts (Slawomir Sudnik, Darko Prenosil)
- Add support for AMD Opteron and Itanium (Jeffrey W. Baker, Bruce)
- Remove `--enable-recode` option from **configure**

  This was no longer needed now that we have **CREATE CONVERSION**.
- Generate a compile error if spinlock code is not found (Bruce)

  Platforms without spinlock code will now fail to compile, rather than silently using semaphores. This failure can be disabled with a new **configure** option.

E.172.3.15. Contrib Changes

- Change dbmirror license to BSD
- Improve earthdistance (Bruno Wolff III)
- Portability improvements to pgcrypto (Marko Kreen)
- Prevent crash in xml (John Gray, Michael Richards)
- Update oracle
- Update mysql
- Update cube (Bruno Wolff III)
- Update earthdistance to use cube (Bruno Wolff III)
- Update btree_gist (Oleg)
- New tsearch2 full-text search module (Oleg, Teodor)
- Add hash-based crosstab function to tablefuncs (Joe)
- Add serial column to order `connectby()` siblings in tablefuncs (Nabil Sayegh,Joe)
- Add named persistent connections to dblink (Shridhar Daithanka)
- New pg_autovacuum allows automatic **VACUUM** (Matthew T. O'Connor)
- Make pgbench honor environment variables `PGHOST`, `PGPORT`, `PGUSER` (Tatsuo)
- Improve intarray (Teodor Sigaev)
- Improve pgstattuple (Rod)
- Fix bug in `metaphone()` in fuzzystrmatch
- Improve adddepend (Rod)
- Update spi/timetravel (Böjthe Zoltán)
- Fix dbase `-s` option and improve non-ASCII handling (Thomas Behr, Márcio Smiderle)

- Remove array module because features now included by default (Joe)

E.173. Release 7.3.21

Release Date

2008-01-07

This release contains a variety of fixes from 7.3.20, including fixes for significant security issues.

This is expected to be the last PostgreSQL™ release in the 7.3.X series. Users are encouraged to update to a newer release branch soon.

E.173.1. Migration to Version 7.3.21

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.173.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running **VACUUM**, **ANALYZE**, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as **VACUUM FULL**, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including **VACUUM**, **ANALYZE**, **REINDEX**, and **CLUSTER**) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of **SET SESSION AUTHORIZATION** and **SET ROLE** is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)

- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 7.3.20 was incomplete, as it plugged the hole for only some `dblink` functions. (CVE-2007-6601, CVE-2007-3278)

- Fix potential crash in `translate()` when using a multibyte database encoding (Tom)

- Make `contrib/tablefunc`'s `crosstab()` handle NULL rowid as a category in its own right, rather than crashing (Joe)

- Require a specific version of Autoconf™ to be used when re-generating the **configure** script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf™ and PostgreSQL™ versions. You can remove the version check if you really want to use a different Autoconf™ version, but it's your responsibility whether the result works or not.

E.174. Release 7.3.20

Release Date

2007-09-17

This release contains fixes from 7.3.19.

E.174.1. Migration to Version 7.3.20

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.174.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent **VACUUM** on the same table (Tom)
- Make **CREATE DOMAIN ... DEFAULT NULL** work properly (Tom)
- Fix crash when `log_min_error_statement` logging runs out of memory (Tom)
- Require non-superusers who use `/contrib/dblink` to use only password authentication, as a security measure (Joe)

E.175. Release 7.3.19

Release Date

2007-04-23

This release contains fixes from 7.3.18, including a security fix.

E.175.1. Migration to Version 7.3.19

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.175.2. Changes

- Support explicit placement of the temporary-table schema within `search_path`, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of `search_path`. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See **CREATE FUNCTION** for more information.
- Fix potential-data-corruption bug in how **VACUUM FULL** handles **UPDATE** chains (Tom, Pavan Deolasee)

E.176. Release 7.3.18

Release Date

2007-02-05

This release contains a variety of fixes from 7.3.17, including a security fix.

E.176.1. Migration to Version 7.3.18

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.176.2. Changes

- Remove security vulnerability that allowed connected users to read backend memory (Tom)

  The vulnerability involves changing the data type of a table column used in a SQL function (CVE-2007-0555). This error can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.177. Release 7.3.17

Release Date

2007-01-08

This release contains a variety of fixes from 7.3.16.

E.177.1. Migration to Version 7.3.17

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.177.2. Changes

- `to_number()` and `to_char(numeric)` are now `STABLE`, not `IMMUTABLE`, for new initdb installs (Tom)

  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)

  This improves psql `\d` performance also.

E.178. Release 7.3.16

Release Date

2006-10-16

This release contains a variety of fixes from 7.3.15.

E.178.1. Migration to Version 7.3.16

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.178.2. Changes

- Fix corner cases in pattern matching for psql's `\d` commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Back-port 7.4 spinlock code to improve performance and support 64-bit architectures better
- Fix SSL-related memory leak in libpq
- Fix backslash escaping in /contrib/dbmirror
- Adjust regression tests for recent changes in US DST laws

E.179. Release 7.3.15

Release Date

2006-05-23

This release contains a variety of fixes from 7.3.14, including patches for extremely serious security issues.

E.179.1. Migration to Version 7.3.15

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines

provided by libraries or drivers (such as libpq's `PQescapeStringConn()`) to perform string escaping, rather than relying on *ad hoc* code to do it.

E.179.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)

  While PostgreSQL™ has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

- Reject unsafe uses of `\'` in string literals

  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts `''` and not `\'` as a representation of ASCII single quote in SQL string literals. By default, `\'` is rejected only when `client_encoding` is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter `backslash_quote` is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of `backslash_quote` is in part to make it obvious that insecure clients are insecure.

- Modify libpq's string-escaping routines to be aware of encoding considerations

  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314. Applications that use multiple PostgreSQL™ connections concurrently should migrate to `PQescapeStringConn()` and `PQescapeByteaConn()` to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping « by hand » should be modified to rely on library routines instead.

- Fix some incorrect encoding conversion functions

  `win1251_to_iso`, `alt_to_iso`, `euc_tw_to_big5`, `euc_tw_to_mic`, `mic_to_euc_tw` were all broken to varying extents.

- Clean up stray remaining uses of `\'` in strings (Bruce, Jan)
- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
- Fix various minor memory leaks

E.180. Release 7.3.14

Release Date

2006-02-14

This release contains a variety of fixes from 7.3.13.

E.180.1. Migration to Version 7.3.14

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see Section E.181, « Release 7.3.13 ».

E.180.2. Changes

- Fix potential crash in **SET SESSION AUTHORIZATION** (CVE-2006-0553)

  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.

- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 7.3.11 release.

- Fix race condition that could lead to « file already exists » errors during pg_clog file creation (Tom)
- Fix to allow restoring dumps that have cross-schema references to custom operators (Tom)

- Portability fix for testing presence of `finite` and `isinf` during configure (Tom)

E.181. Release 7.3.13

Release Date

2006-01-09

This release contains a variety of fixes from 7.3.12.

E.181.1. Migration to Version 7.3.13

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see Section E.184, « Release 7.3.10 ». Also, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.181.2. Changes

- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

  This might require **REINDEX** to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won't change the locale later

  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need **REINDEX** to fix existing indexes on textual columns if this has happened to you.
- Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Fix bug in `/contrib/pgcrypto` gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

  Salts for Blowfish and standard DES are unaffected.
- Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what's actually returned by the query (Joe)

E.182. Release 7.3.12

Release Date

2005-12-12

This release contains a variety of fixes from 7.3.11.

E.182.1. Migration to Version 7.3.12

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see Section E.184, « Release 7.3.10 ».

E.182.2. Changes

- Fix race condition in transaction log management

  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- `/contrib/ltree` fixes (Teodor)
- Fix longstanding planning error for outer joins

  This bug sometimes caused a bogus error « RIGHT JOIN is only supported with merge-joinable join conditions ».

- Prevent core dump in pg_autovacuum when a table has been dropped

E.183. Release 7.3.11

Release Date

2005-10-04

This release contains a variety of fixes from 7.3.10.

E.183.1. Migration to Version 7.3.11

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see Section E.184, « Release 7.3.10 ».

E.183.2. Changes

- Fix error that allowed **VACUUM** to remove `ctid` chains too soon, and add more checking in code that follows `ctid` links

  This fixes a long-standing problem that could cause crashes in very rare circumstances.
- Fix CHAR() to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)

  In prior releases, the padding of CHAR() was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
- Fix missing rows in queries like `UPDATE a=... WHERE a...` with GiST index on column `a`
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Various memory leakage fixes
- Various portability improvements
- Fix PL/pgSQL to handle `var := var` correctly when the variable is of pass-by-reference type

E.184. Release 7.3.10

Release Date

2005-05-09

This release contains a variety of fixes from 7.3.9, including several security-related issues.

E.184.1. Migration to Version 7.3.10

A dump/restore is not required for those running 7.3.X. However, it is one possible way of handling a significant security problem that has been found in the initial contents of 7.3.X system catalogs. A dump/initdb/reload sequence using 7.3.10's initdb will automatically correct this problem.

The security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.) It is strongly recommended that all installations repair this error, either by initdb or by following the manual repair procedure given below. The error at least allows unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the following procedure instead. As the database superuser, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
```

```
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

The above procedure must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same error. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedure. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.184.2. Changes

- Change encoding function signature to prevent misuse
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of TIME WITH TIME ZONE values

  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. NOTE: if you have an index on a TIME WITH TIME ZONE column, it will need to be **REINDEX**ed after installing this update, because the fix corrects the sort order of column values.
- Fix `EXTRACT(EPOCH)` for TIME WITH TIME ZONE values
- Fix mis-display of negative fractional seconds in INTERVAL values

  This error only occurred when the `--enable-integer-datetimes` configuration switch had been used.
- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump trigger names containing `%` correctly (Neil)
- Prevent `to_char(interval)` from dumping core for month-related formats
- Fix `contrib/pgcrypto` for newer OpenSSL builds (Marko Kreen)
- Still more 64-bit fixes for `contrib/intagg`
- Prevent incorrect optimization of functions returning RECORD

E.185. Release 7.3.9

Release Date

2005-01-31

This release contains a variety of fixes from 7.3.8, including several security-related issues.

E.185.1. Migration to Version 7.3.9

A dump/restore is not required for those running 7.3.X.

E.185.2. Changes

- Disallow **LOAD** to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), **LOAD** can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Check that creator of an aggregate function has the right to execute the specified transition functions

  This oversight made it possible to bypass denial of EXECUTE permission on a function.
- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles

E.186. Release 7.3.8

Release Date

2004-10-22

This release contains a variety of fixes from 7.3.7.

E.186.1. Migration to Version 7.3.8

A dump/restore is not required for those running 7.3.X.

E.186.2. Changes

- Repair possible failure to update hint bits on disk

  Under rare circumstances this oversight could lead to « could not access transaction status » failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples

  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg_ctl as root

  This is to guard against any possible security issues.
- Avoid using temp files in /tmp in make_oidjoins_check

  This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.

E.187. Release 7.3.7

Release Date

2004-08-16

This release contains one critical fix over 7.3.6, and some minor items.

E.187.1. Migration to Version 7.3.7

A dump/restore is not required for those running 7.3.X.

E.187.2. Changes

- Prevent possible loss of committed transactions during crash

  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL™ 7.1.
- Remove asymmetrical word processing in tsearch (Teodor)
- Properly schema-qualify function names when pg_dump'ing a CAST

E.188. Release 7.3.6

Release Date

2004-03-02

This release contains a variety of fixes from 7.3.5.

E.188.1. Migration to Version 7.3.6

A dump/restore is *not* required for those running 7.3.*.

E.188.2. Changes

- Revert erroneous changes in rule permissions checking

  A patch applied in 7.3.3 to fix a corner case in rule permissions checks turns out to have disabled rule-related permissions checks in many not-so-corner cases. This would for example allow users to insert into views they weren't supposed to have permission to insert into. We have therefore reverted the 7.3.3 patch. The original bug will be fixed in 8.0.
- Repair incorrect order of operations in GetNewTransactionId()

  This bug could result in failure under out-of-disk-space conditions, including inability to restart even after disk space is freed.
- Ensure configure selects -fno-strict-aliasing even when an external value for CFLAGS is supplied

  On some platforms, building with -fstrict-aliasing causes bugs.
- Make pg_restore handle 64-bit off_t correctly

  This bug prevented proper restoration from archive files exceeding 4 GB.
- Make contrib/dblink not assume that local and remote type OIDs match (Joe)
- Quote connectby()'s start_with argument properly (Joe)
- Don't crash when a rowtype argument to a plpgsql function is NULL
- Avoid generating invalid character encoding sequences in corner cases when planning LIKE operations
- Ensure text_position() cannot scan past end of source string in multibyte cases (Korea PostgreSQL Users' Group)
- Fix index optimization and selectivity estimates for LIKE operations on bytea columns (Joe)

E.189. Release 7.3.5

Release Date

2003-12-03

This has a variety of fixes from 7.3.4.

E.189.1. Migration to Version 7.3.5

A dump/restore is *not* required for those running 7.3.*.

E.189.2. Changes

- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of « variable not in subplan target lists »
- Force stats processes to detach from shared memory, ensuring cleaner shutdown
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Added missing SPI_finish() calls to dblink's get_tuple_of_interest() (Joe)
- Fix for possible foreign key violation when rule rewrites INSERT (Jan)
- Support qualified type names in PL/Tcl's spi_prepare command (Jan)
- Make pg_dump handle a procedural language handler located in pg_catalog
- Make pg_dump handle cases where a custom opclass is in another schema
- Make pg_dump dump binary-compatible casts correctly (Jan)
- Fix insertion of expressions containing subqueries into rule bodies
- Fix incorrect argument processing in clusterdb script (Anand Ranganathan)
- Fix problems with dropped columns in plpython triggers
- Repair problems with to_char() reading past end of its input string (Karel)
- Fix GB18030 mapping errors (Tatsuo)
- Fix several problems with SSL error handling and asynchronous SSL I/O
- Remove ability to bind a list of values to a single parameter in JDBC (prevents possible SQL-injection attacks)
- Fix some errors in HAVE_INT64_TIMESTAMP code paths
- Fix corner case for btree search in parallel with first root page split

E.190. Release 7.3.4

Release Date

2003-07-24

This has a variety of fixes from 7.3.3.

E.190.1. Migration to Version 7.3.4

A dump/restore is *not* required for those running 7.3.*.

E.190.2. Changes

- Repair breakage in timestamp-to-date conversion for dates before 2000
- Prevent rare possibility of server startup failure (Tom)
- Fix bugs in interval-to-time conversion (Tom)
- Add constraint names in a few places in pg_dump (Rod)

- Improve performance of functions with many parameters (Tom)
- Fix to_ascii() buffer overruns (Tom)
- Prevent restore of database comments from throwing an error (Tom)
- Work around buggy strxfrm() present in some Solaris releases (Tom)
- Properly escape jdbc setObject() strings to improve security (Barry)

E.191. Release 7.3.3

Release Date

2003-05-22

This release contains a variety of fixes for version 7.3.2.

E.191.1. Migration to Version 7.3.3

A dump/restore is *not* required for those running version 7.3.*.

E.191.2. Changes

- Repair sometimes-incorrect computation of StartUpID after a crash
- Avoid slowness with lots of deferred triggers in one transaction (Stephan)
- Don't lock referenced row when **UPDATE** doesn't change foreign key's value (Jan)
- Use **-fPIC** not **-fpic** on Sparc (Tom Callaway)
- Repair lack of schema-awareness in contrib/reindexdb
- Fix contrib/intarray error for zero-element result array (Teodor)
- Ensure createuser script will exit on control-C (Oliver)
- Fix errors when the type of a dropped column has itself been dropped
- **CHECKPOINT** does not cause database panic on failure in noncritical steps
- Accept 60 in seconds fields of timestamp, time, interval input values
- Issue notice, not error, if TIMESTAMP, TIME, or INTERVAL precision too large
- Fix `abstime-to-time` cast function (fix is not applied unless you initdb)
- Fix pg_proc entry for timestampt_izone (fix is not applied unless you initdb)
- Make `EXTRACT(EPOCH FROM timestamp without time zone)` treat input as local time
- **'now'::timestamptz** gave wrong answer if timezone changed earlier in transaction
- `HAVE_INT64_TIMESTAMP` code for time with timezone overwrote its input
- Accept **GLOBAL TEMP/TEMPORARY** as a synonym for **TEMPORARY**
- Avoid improper schema-privilege-check failure in foreign-key triggers
- Fix bugs in foreign-key triggers for **SET DEFAULT** action
- Fix incorrect time-qual check in row fetch for **UPDATE** and **DELETE** triggers
- Foreign-key clauses were parsed but ignored in **ALTER TABLE ADD COLUMN**
- Fix createlang script breakage for case where handler function already exists
- Fix misbehavior on zero-column tables in pg_dump, COPY, ANALYZE, other places
- Fix misbehavior of `func_error()` on type names containing '%'
- Fix misbehavior of `replace()` on strings containing '%'

- Regular-expression patterns containing certain multibyte characters failed
- Account correctly for **NULL**s in more cases in join size estimation
- Avoid conflict with system definition of `isblank()` function or macro
- Fix failure to convert large code point values in EUC_TW conversions (Tatsuo)
- Fix error recovery for `SSL_read`/`SSL_write` calls
- Don't do early constant-folding of type coercion expressions
- Validate page header fields immediately after reading in any page
- Repair incorrect check for ungrouped variables in unnamed joins
- Fix buffer overrun in `to_ascii` (Guido Notari)
- contrib/ltree fixes (Teodor)
- Fix core dump in deadlock detection on machines where char is unsigned
- Avoid running out of buffers in many-way indexscan (bug introduced in 7.3)
- Fix planner's selectivity estimation functions to handle domains properly
- Fix dbmirror memory-allocation bug (Steven Singer)
- Prevent infinite loop in `ln(numeric)` due to roundoff error
- **GROUP BY** got confused if there were multiple equal GROUP BY items
- Fix bad plan when alled **UPDATE**/**DELETE** references another inherited table
- Prevent clustering on incomplete (partial or non-NULL-storing) indexes
- Service shutdown request at proper time if it arrives while still starting up
- Fix left-links in temporary indexes (could make backwards scans miss entries)
- Fix incorrect handling of client_encoding setting in postgresql.conf (Tatsuo)
- Fix failure to respond to **pg_ctl stop -m fast** after Async_NotifyHandler runs
- Fix SPI for case where rule contains multiple statements of the same type
- Fix problem with checking for wrong type of access privilege in rule query
- Fix problem with **EXCEPT** in **CREATE RULE**
- Prevent problem with dropping temp tables having serial columns
- Fix replace_vars_with_subplan_refs failure in complex views
- Fix regexp slowness in single-byte encodings (Tatsuo)
- Allow qualified type names in **CREATE CAST** and **DROP CAST**
- Accept `SETOF type[]`, which formerly had to be written `SETOF _type`
- Fix pg_dump core dump in some cases with procedural languages
- Force ISO datestyle in pg_dump output, for portability (Oliver)
- pg_dump failed to handle error return from `lo_read` (Oleg Drokin)
- pg_dumpall failed with groups having no members (Nick Eskelinen)
- pg_dumpall failed to recognize --globals-only switch
- pg_restore failed to restore blobs if -X disable-triggers is specified
- Repair intrafunction memory leak in plpgsql
- pltcl's **elog** command dumped core if given wrong parameters (Ian Harding)
- plpython used wrong value of `atttypmod` (Brad McLean)
- Fix improper quoting of boolean values in Python interface (D'Arcy)

- Added `addDataType()` method to PGConnection interface for JDBC
- Fixed various problems with updateable ResultSets for JDBC (Shawn Green)
- Fixed various problems with DatabaseMetaData for JDBC (Kris Jurka, Peter Royal)
- Fixed problem with parsing table ACLs in JDBC
- Better error message for character set conversion problems in JDBC

E.192. Release 7.3.2

Release Date

2003-02-04

This release contains a variety of fixes for version 7.3.1.

E.192.1. Migration to Version 7.3.2

A dump/restore is *not* required for those running version 7.3.*.

E.192.2. Changes

- Restore creation of OID column in CREATE TABLE AS / SELECT INTO
- Fix pg_dump core dump when dumping views having comments
- Dump DEFERRABLE/INITIALLY DEFERRED constraints properly
- Fix UPDATE when child table's column numbering differs from parent
- Increase default value of max_fsm_relations
- Fix problem when fetching backwards in a cursor for a single-row query
- Make backward fetch work properly with cursor on SELECT DISTINCT query
- Fix problems with loading pg_dump files containing contrib/lo usage
- Fix problem with all-numeric user names
- Fix possible memory leak and core dump during disconnect in libpgtcl
- Make plpython's spi_execute command handle nulls properly (Andrew Bosma)
- Adjust plpython error reporting so that its regression test passes again
- Work with bison 1.875
- Handle mixed-case names properly in plpgsql's %type (Neil)
- Fix core dump in pltcl when executing a query rewritten by a rule
- Repair array subscript overruns (per report from Yichen Xie)
- Reduce MAX_TIME_PRECISION from 13 to 10 in floating-point case
- Correctly case-fold variable names in per-database and per-user settings
- Fix coredump in plpgsql's RETURN NEXT when SELECT into record returns no rows
- Fix outdated use of pg_type.typprtlen in python client interface
- Correctly handle fractional seconds in timestamps in JDBC driver
- Improve performance of getImportedKeys() in JDBC
- Make shared-library symlinks work standardly on HPUX (Giles)
- Repair inconsistent rounding behavior for timestamp, time, interval
- SSL negotiation fixes (Nathan Mueller)

- Make libpq's ~/.pgpass feature work when connecting with PQconnectDB
- Update my2pg, ora2pg
- Translation updates
- Add casts between types lo and oid in contrib/lo
- fastpath code now checks for privilege to call function

E.193. Release 7.3.1

Release Date

2002-12-18

This release contains a variety of fixes for version 7.3.

E.193.1. Migration to Version 7.3.1

A dump/restore is *not* required for those running version 7.3. However, it should be noted that the main PostgreSQL™ interface library, libpq, has a new major version number for this release, which might require recompilation of client code in certain cases.

E.193.2. Changes

- Fix a core dump of COPY TO when client/server encodings don't match (Tom)
- Allow pg_dump to work with pre-7.2 servers (Philip)
- contrib/adddepend fixes (Tom)
- Fix problem with deletion of per-user/per-database config settings (Tom)
- contrib/vacuumlo fix (Tom)
- Allow 'password' encryption even when pg_shadow contains MD5 passwords (Bruce)
- contrib/dbmirror fix (Steven Singer)
- Optimizer fixes (Tom)
- contrib/tsearch fixes (Teodor Sigaev, Magnus)
- Allow locale names to be mixed case (Nicolai Tufar)
- Increment libpq library's major version number (Bruce)
- pg_hba.conf error reporting fixes (Bruce, Neil)
- Add SCO Openserver 5.0.4 as a supported platform (Bruce)
- Prevent EXPLAIN from crashing server (Tom)
- SSL fixes (Nathan Mueller)
- Prevent composite column creation via ALTER TABLE (Tom)

E.194. Release 7.3

Release Date

2002-11-27

E.194.1. Overview

Major changes in this release:

Schemas
: Schemas allow users to create objects in separate namespaces, so two people or applications can have tables with the same name. There is also a public schema for shared tables. Table/index creation can be restricted by removing privileges on the public schema.

Drop Column
: PostgreSQL™ now supports the `ALTER TABLE ... DROP COLUMN` functionality.

Table Functions
: Functions returning multiple rows and/or multiple columns are now much easier to use than before. You can call such a « table function » in the `SELECT FROM` clause, treating its output like a table. Also, PL/pgSQL functions can now return sets.

Prepared Queries
: PostgreSQL™ now supports prepared queries, for improved performance.

Dependency Tracking
: PostgreSQL™ now records object dependencies, which allows improvements in many areas. **DROP** statements now take either `CASCADE` or `RESTRICT` to control whether dependent objects are also dropped.

Privileges
: Functions and procedural languages now have privileges, and functions can be defined to run with the privileges of their creator.

Internationalization
: Both multibyte and locale support are now always enabled.

Logging
: A variety of logging options have been enhanced.

Interfaces
: A large number of interfaces have been moved to *http://gborg.postgresql.org* where they can be developed and released independently.

Functions/Identifiers
: By default, functions can now take up to 32 parameters, and identifiers can be up to 63 bytes long. Also, `OPAQUE` is now deprecated: there are specific « pseudo-datatypes » to represent each of the former meanings of `OPAQUE` in function argument and result types.

E.194.2. Migration to Version 7.3

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release. If your application examines the system catalogs, additional changes will be required due to the introduction of schemas in 7.3; for more information, see: *http://developer.postgresql.org/~momjian/upgrade_tips_7.3*.

Observe the following incompatibilities:

- Pre-6.3 clients are no longer supported.
- `pg_hba.conf` now has a column for the user name and additional features. Existing files need to be adjusted.
- Several `postgresql.conf` logging parameters have been renamed.
- `LIMIT #,#` has been disabled; use `LIMIT # OFFSET #`.
- **INSERT** statements with column lists must specify a value for each specified column. For example, `INSERT INTO tab (col1, col2) VALUES ('val1')` is now invalid. It's still allowed to supply fewer columns than expected if the **INSERT** does not have a column list.
- serial columns are no longer automatically `UNIQUE`; thus, an index will not automatically be created.
- A **SET** command inside an aborted transaction is now rolled back.
- **COPY** no longer considers missing trailing columns to be null. All columns need to be specified. (However, one can achieve a similar effect by specifying a column list in the **COPY** command.)
- The data type timestamp is now equivalent to timestamp without time zone, instead of timestamp with time zone.
- Pre-7.3 databases loaded into 7.3 will not have the new object dependencies for serial columns, unique constraints, and foreign keys. See the directory `contrib/adddepend/` for a detailed description and a script that will add such dependencies.

- An empty string (' ') is no longer allowed as the input into an integer field. Formerly, it was silently interpreted as 0.

E.194.3. Changes

E.194.3.1. Server Operation

- Add pg_locks view to show locks (Neil)
- Security fixes for password negotiation memory allocation (Neil)
- Remove support for version 0 FE/BE protocol (PostgreSQL™ 6.2 and earlier) (Tom)
- Reserve the last few backend slots for superusers, add parameter superuser_reserved_connections to control this (Nigel J. Andrews)

E.194.3.2. Performance

- Improve startup by calling localtime() only once (Tom)
- Cache system catalog information in flat files for faster startup (Tom)
- Improve caching of index information (Tom)
- Optimizer improvements (Tom, Fernando Nasser)
- Catalog caches now store failed lookups (Tom)
- Hash function improvements (Neil)
- Improve performance of query tokenization and network handling (Peter)
- Speed improvement for large object restore (Mario Weilguni)
- Mark expired index entries on first lookup, saving later heap fetches (Tom)
- Avoid excessive NULL bitmap padding (Manfred Koizar)
- Add BSD-licensed qsort() for Solaris, for performance (Bruce)
- Reduce per-row overhead by four bytes (Manfred Koizar)
- Fix GEQO optimizer bug (Neil Conway)
- Make WITHOUT OID actually save four bytes per row (Manfred Koizar)
- Add default_statistics_target variable to specify ANALYZE buckets (Neil)
- Use local buffer cache for temporary tables so no WAL overhead (Tom)
- Improve free space map performance on large tables (Stephen Marshall, Tom)
- Improved WAL write concurrency (Tom)

E.194.3.3. Privileges

- Add privileges on functions and procedural languages (Peter)
- Add OWNER to CREATE DATABASE so superusers can create databases on behalf of unprivileged users (Gavin Sherry, Tom)
- Add new object privilege bits EXECUTE and USAGE (Tom)
- Add SET SESSION AUTHORIZATION DEFAULT and RESET SESSION AUTHORIZATION (Tom)
- Allow functions to be executed with the privilege of the function owner (Peter)

E.194.3.4. Server Configuration

- Server log messages now tagged with LOG, not DEBUG (Bruce)

- Add user column to pg_hba.conf (Bruce)
- Have log_connections output two lines in log file (Tom)
- Remove debug_level from postgresql.conf, now server_min_messages (Bruce)
- New ALTER DATABASE/USER ... SET command for per-user/database initialization (Peter)
- New parameters server_min_messages and client_min_messages to control which messages are sent to the server logs or client applications (Bruce)
- Allow pg_hba.conf to specify lists of users/databases separated by commas, group names prepended with +, and file names prepended with @ (Bruce)
- Remove secondary password file capability and pg_password utility (Bruce)
- Add variable db_user_namespace for database-local user names (Bruce)
- SSL improvements (Bear Giles)
- Make encryption of stored passwords the default (Bruce)
- Allow pg_statistics to be reset by calling pg_stat_reset() (Christopher)
- Add log_duration parameter (Bruce)
- Rename debug_print_query to log_statement (Bruce)
- Rename show_query_stats to show_statement_stats (Bruce)
- Add param log_min_error_statement to print commands to logs on error (Gavin)

E.194.3.5. Queries

- Make cursors insensitive, meaning their contents do not change (Tom)
- Disable LIMIT #,# syntax; now only LIMIT # OFFSET # supported (Bruce)
- Increase identifier length to 63 (Neil, Bruce)
- UNION fixes for merging >= 3 columns of different lengths (Tom)
- Add DEFAULT key word to INSERT, e.g., INSERT ... (..., DEFAULT, ...) (Rod)
- Allow views to have default values using ALTER COLUMN ... SET DEFAULT (Neil)
- Fail on INSERTs with column lists that don't supply all column values, e.g., INSERT INTO tab (col1, col2) VALUES ('val1'); (Rod)
- Fix for join aliases (Tom)
- Fix for FULL OUTER JOINs (Tom)
- Improve reporting of invalid identifier and location (Tom, Gavin)
- Fix OPEN cursor(args) (Tom)
- Allow 'ctid' to be used in a view and currtid(viewname) (Hiroshi)
- Fix for CREATE TABLE AS with UNION (Tom)
- SQL99 syntax improvements (Thomas)
- Add statement_timeout variable to cancel queries (Bruce)
- Allow prepared queries with PREPARE/EXECUTE (Neil)
- Allow FOR UPDATE to appear after LIMIT/OFFSET (Bruce)
- Add variable autocommit (Tom, David Van Wie)

E.194.3.6. Object Manipulation

- Make equals signs optional in CREATE DATABASE (Gavin Sherry)

- Make ALTER TABLE OWNER change index ownership too (Neil)
- New ALTER TABLE tabname ALTER COLUMN colname SET STORAGE controls TOAST storage, compression (John Gray)
- Add schema support, CREATE/DROP SCHEMA (Tom)
- Create schema for temporary tables (Tom)
- Add variable search_path for schema search (Tom)
- Add ALTER TABLE SET/DROP NOT NULL (Christopher)
- New CREATE FUNCTION volatility levels (Tom)
- Make rule names unique only per table (Tom)
- Add 'ON tablename' clause to DROP RULE and COMMENT ON RULE (Tom)
- Add ALTER TRIGGER RENAME (Joe)
- New current_schema() and current_schemas() inquiry functions (Tom)
- Allow functions to return multiple rows (table functions) (Joe)
- Make WITH optional in CREATE DATABASE, for consistency (Bruce)
- Add object dependency tracking (Rod, Tom)
- Add RESTRICT/CASCADE to DROP commands (Rod)
- Add ALTER TABLE DROP for non-CHECK CONSTRAINT (Rod)
- Autodestroy sequence on DROP of table with SERIAL (Rod)
- Prevent column dropping if column is used by foreign key (Rod)
- Automatically drop constraints/functions when object is dropped (Rod)
- Add CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- Add ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)
- Prevent alled columns from being removed or renamed (Alvaro Herrera)
- Fix foreign key constraints to not error on intermediate database states (Stephan)
- Propagate column or table renaming to foreign key constraints
- Add CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- Add CREATE OR REPLACE RULE (Gavin, Neil, Tom)
- Have rules execute alphabetically, returning more predictable values (Tom)
- Triggers are now fired in alphabetical order (Tom)
- Add /contrib/adddepend to handle pre-7.3 object dependencies (Rod)
- Allow better casting when inserting/updating values (Tom)

E.194.3.7. Utility Commands

- Have COPY TO output embedded carriage returns and newlines as \r and \n (Tom)
- Allow DELIMITER in COPY FROM to be 8-bit clean (Tatsuo)
- Make pg_dump use ALTER TABLE ADD PRIMARY KEY, for performance (Neil)
- Disable brackets in multistatement rules (Bruce)
- Disable VACUUM from being called inside a function (Bruce)
- Allow dropdb and other scripts to use identifiers with spaces (Bruce)
- Restrict database comment changes to the current database
- Allow comments on operators, independent of the underlying function (Rod)

- Rollback SET commands in aborted transactions (Tom)
- EXPLAIN now outputs as a query (Tom)
- Display condition expressions and sort keys in EXPLAIN (Tom)
- Add 'SET LOCAL var = value' to set configuration variables for a single transaction (Tom)
- Allow ANALYZE to run in a transaction (Bruce)
- Improve COPY syntax using new WITH clauses, keep backward compatibility (Bruce)
- Fix pg_dump to consistently output tags in non-ASCII dumps (Bruce)
- Make foreign key constraints clearer in dump file (Rod)
- Add COMMENT ON CONSTRAINT (Rod)
- Allow COPY TO/FROM to specify column names (Brent Verner)
- Dump UNIQUE and PRIMARY KEY constraints as ALTER TABLE (Rod)
- Have SHOW output a query result (Joe)
- Generate failure on short COPY lines rather than pad NULLs (Neil)
- Fix CLUSTER to preserve all table attributes (Alvaro Herrera)
- New pg_settings table to view/modify GUC settings (Joe)
- Add smart quoting, portability improvements to pg_dump output (Peter)
- Dump serial columns out as SERIAL (Tom)
- Enable large file support, >2G for pg_dump (Peter, Philip Warner, Bruce)
- Disallow TRUNCATE on tables that are involved in referential constraints (Rod)
- Have TRUNCATE also auto-truncate the toast table of the relation (Tom)
- Add clusterdb utility that will auto-cluster an entire database based on previous CLUSTER operations (Alvaro Herrera)
- Overhaul pg_dumpall (Peter)
- Allow REINDEX of TOAST tables (Tom)
- Implemented START TRANSACTION, per SQL99 (Neil)
- Fix rare index corruption when a page split affects bulk delete (Tom)
- Fix ALTER TABLE ... ADD COLUMN for inheritance (Alvaro Herrera)

E.194.3.8. Data Types and Functions

- Fix factorial(0) to return 1 (Bruce)
- Date/time/timezone improvements (Thomas)
- Fix for array slice extraction (Tom)
- Fix extract/date_part to report proper microseconds for timestamp (Tatsuo)
- Allow text_substr() and bytea_substr() to read TOAST values more efficiently (John Gray)
- Add domain support (Rod)
- Make WITHOUT TIME ZONE the default for TIMESTAMP and TIME data types (Thomas)
- Allow alternate storage scheme of 64-bit integers for date/time types using --enable-integer-datetimes in configure (Thomas)
- Make timezone(timestamptz) return timestamp rather than a string (Thomas)
- Allow fractional seconds in date/time types for dates prior to 1BC (Thomas)
- Limit timestamp data types to 6 decimal places of precision (Thomas)
- Change timezone conversion functions from timetz() to timezone() (Thomas)

- Add configuration variables datestyle and timezone (Tom)
- Add OVERLAY(), which allows substitution of a substring in a string (Thomas)
- Add SIMILAR TO (Thomas, Tom)
- Add regular expression SUBSTRING(string FROM pat FOR escape) (Thomas)
- Add LOCALTIME and LOCALTIMESTAMP functions (Thomas)
- Add named composite types using CREATE TYPE typename AS (column) (Joe)
- Allow composite type definition in the table alias clause (Joe)
- Add new API to simplify creation of C language table functions (Joe)
- Remove ODBC-compatible empty parentheses from calls to SQL99 functions for which these parentheses do not match the standard (Thomas)
- Allow macaddr data type to accept 12 hex digits with no separators (Mike Wyer)
- Add CREATE/DROP CAST (Peter)
- Add IS DISTINCT FROM operator (Thomas)
- Add SQL99 TREAT() function, synonym for CAST() (Thomas)
- Add pg_backend_pid() to output backend pid (Bruce)
- Add IS OF / IS NOT OF type predicate (Thomas)
- Allow bit string constants without fully-specified length (Thomas)
- Allow conversion between 8-byte integers and bit strings (Thomas)
- Implement hex literal conversion to bit string literal (Thomas)
- Allow table functions to appear in the FROM clause (Joe)
- Increase maximum number of function parameters to 32 (Bruce)
- No longer automatically create index for SERIAL column (Tom)
- Add current_database() (Rod)
- Fix cash_words() to not overflow buffer (Tom)
- Add functions replace(), split_part(), to_hex() (Joe)
- Fix LIKE for bytea as a right-hand argument (Joe)
- Prevent crashes caused by SELECT cash_out(2) (Tom)
- Fix to_char(1,'FM999.99') to return a period (Karel)
- Fix trigger/type/language functions returning OPAQUE to return proper type (Tom)

E.194.3.9. Internationalization

- Add additional encodings: Korean (JOHAB), Thai (WIN874), Vietnamese (TCVN), Arabic (WIN1256), Simplified Chinese (GBK), Korean (UHC) (Eiji Tokuya)
- Enable locale support by default (Peter)
- Add locale variables (Peter)
- Escape byes >= 0x7f for multibyte in PQescapeBytea/PQunescapeBytea (Tatsuo)
- Add locale awareness to regular expression character classes
- Enable multibyte support by default (Tatsuo)
- Add GB18030 multibyte support (Bill Huang)
- Add CREATE/DROP CONVERSION, allowing loadable encodings (Tatsuo, Kaori)
- Add pg_conversion table (Tatsuo)

- Add SQL99 CONVERT() function (Tatsuo)
- pg_dumpall, pg_controldata, and pg_resetxlog now national-language aware (Peter)
- New and updated translations

E.194.3.10. Server-side Languages

- Allow recursive SQL function (Peter)
- Change PL/Tcl build to use configured compiler and Makefile.shlib (Peter)
- Overhaul the PL/pgSQL FOUND variable to be more Oracle-compatible (Neil, Tom)
- Allow PL/pgSQL to handle quoted identifiers (Tom)
- Allow set-returning PL/pgSQL functions (Neil)
- Make PL/pgSQL schema-aware (Joe)
- Remove some memory leaks (Nigel J. Andrews, Tom)

E.194.3.11. psql

- Don't lowercase psql \connect database name for 7.2.0 compatibility (Tom)
- Add psql \timing to time user queries (Greg Sabino Mullane)
- Have psql \d show index information (Greg Sabino Mullane)
- New psql \dD shows domains (Jonathan Eisler)
- Allow psql to show rules on views (Paul ?)
- Fix for psql variable substitution (Tom)
- Allow psql \d to show temporary table structure (Tom)
- Allow psql \d to show foreign keys (Rod)
- Fix \? to honor \pset pager (Bruce)
- Have psql reports its version number on startup (Tom)
- Allow \copy to specify column names (Tom)

E.194.3.12. libpq

- Add ~/.pgpass to store host/user password combinations (Alvaro Herrera)
- Add PQunescapeBytea() function to libpq (Patrick Welche)
- Fix for sending large queries over non-blocking connections (Bernhard Herzog)
- Fix for libpq using timers on Win9X (David Ford)
- Allow libpq notify to handle servers with different-length identifiers (Tom)
- Add libpq PQescapeString() and PQescapeBytea() to Windows (Bruce)
- Fix for SSL with non-blocking connections (Jack Bates)
- Add libpq connection timeout parameter (Denis A Ustimenko)

E.194.3.13. JDBC

- Allow JDBC to compile with JDK 1.4 (Dave)
- Add JDBC 3 support (Barry)
- Allows JDBC to set loglevel by adding ?loglevel=X to the connection URL (Barry)

- Add Driver.info() message that prints out the version number (Barry)
- Add updateable result sets (Raghu Nidagal, Dave)
- Add support for callable statements (Paul Bethe)
- Add query cancel capability
- Add refresh row (Dave)
- Fix MD5 encryption handling for multibyte servers (Jun Kawai)
- Add support for prepared statements (Barry)

E.194.3.14. Miscellaneous Interfaces

- Fixed ECPG bug concerning octal numbers in single quotes (Michael)
- Move src/interfaces/libpgeasy to http://gborg.postgresql.org (Marc, Bruce)
- Improve Python interface (Elliot Lee, Andrew Johnson, Greg Copeland)
- Add libpgtcl connection close event (Gerhard Hintermayer)
- Move src/interfaces/libpq++ to http://gborg.postgresql.org (Marc, Bruce)
- Move src/interfaces/odbc to http://gborg.postgresql.org (Marc)
- Move src/interfaces/libpgeasy to http://gborg.postgresql.org (Marc, Bruce)
- Move src/interfaces/perl5 to http://gborg.postgresql.org (Marc, Bruce)
- Remove src/bin/pgaccess from main tree, now at http://www.pgaccess.org (Bruce)
- Add pg_on_connection_loss command to libpgtcl (Gerhard Hintermayer, Tom)

E.194.3.15. Source Code

- Fix for parallel make (Peter)
- AIX fixes for linking Tcl (Andreas Zeugswetter)
- Allow PL/Perl to build under Cygwin (Jason Tishler)
- Improve MIPS compiles (Peter, Oliver Elphick)
- Require Autoconf version 2.53 (Peter)
- Require readline and zlib by default in configure (Peter)
- Allow Solaris to use Intimate Shared Memory (ISM), for performance (Scott Brunza, P.J. Josh Rovero)
- Always enable syslog in compile, remove --enable-syslog option (Tatsuo)
- Always enable multibyte in compile, remove --enable-multibyte option (Tatsuo)
- Always enable locale in compile, remove --enable-locale option (Peter)
- Fix for Win9x DLL creation (Magnus Naeslund)
- Fix for link() usage by WAL code on Windows, BeOS (Jason Tishler)
- Add sys/types.h to c.h, remove from main files (Peter, Bruce)
- Fix AIX hang on SMP machines (Tomoyuki Niijima)
- AIX SMP hang fix (Tomoyuki Niijima)
- Fix pre-1970 date handling on newer glibc libraries (Tom)
- Fix PowerPC SMP locking (Tom)
- Prevent gcc -ffast-math from being used (Peter, Tom)
- Bison >= 1.50 now required for developer builds

- Kerberos 5 support now builds with Heimdal (Peter)
- Add appendix in the User's Guide which lists SQL features (Thomas)
- Improve loadable module linking to use RTLD_NOW (Tom)
- New error levels WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- New src/port directory holds replaced libc functions (Peter, Bruce)
- New pg_namespace system catalog for schemas (Tom)
- Add pg_class.relnamespace for schemas (Tom)
- Add pg_type.typnamespace for schemas (Tom)
- Add pg_proc.pronamespace for schemas (Tom)
- Restructure aggregates to have pg_proc entries (Tom)
- System relations now have their own namespace, pg_* test not required (Fernando Nasser)
- Rename TOAST index names to be *_index rather than *_idx (Neil)
- Add namespaces for operators, opclasses (Tom)
- Add additional checks to server control file (Thomas)
- New Polish FAQ (Marcin Mazurek)
- Add Posix semaphore support (Tom)
- Document need for reindex (Bruce)
- Rename some internal identifiers to simplify Windows compile (Jan, Katherine Ward)
- Add documentation on computing disk space (Bruce)
- Remove KSQO from GUC (Bruce)
- Fix memory leak in rtree (Kenneth Been)
- Modify a few error messages for consistency (Bruce)
- Remove unused system table columns (Peter)
- Make system columns NOT NULL where appropriate (Tom)
- Clean up use of sprintf in favor of snprintf() (Neil, Jukka Holappa)
- Remove OPAQUE and create specific subtypes (Tom)
- Cleanups in array internal handling (Joe, Tom)
- Disallow pg_atoi(") (Bruce)
- Remove parameter wal_files because WAL files are now recycled (Bruce)
- Add version numbers to heap pages (Tom)

E.194.3.16. Contrib

- Allow inet arrays in /contrib/array (Neil)
- GiST fixes (Teodor Sigaev, Neil)
- Upgrade /contrib/mysql
- Add /contrib/dbsize which shows table sizes without vacuum (Peter)
- Add /contrib/intagg, integer aggregator routines (mlw)
- Improve /contrib/oid2name (Neil, Bruce)
- Improve /contrib/tsearch (Oleg, Teodor Sigaev)
- Cleanups of /contrib/rserver (Alexey V. Borzov)

- Update /contrib/oracle conversion utility (Gilles Darold)
- Update /contrib/dblink (Joe)
- Improve options supported by /contrib/vacuumlo (Mario Weilguni)
- Improvements to /contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- Add /contrib/reindexdb utility (Shaun Thomas)
- Add indexing to /contrib/isbn_issn (Dan Weston)
- Add /contrib/dbmirror (Steven Singer)
- Improve /contrib/pgbench (Neil)
- Add /contrib/tablefunc table function examples (Joe)
- Add /contrib/ltree data type for tree structures (Teodor Sigaev, Oleg Bartunov)
- Move /contrib/pg_controldata, pg_resetxlog into main tree (Bruce)
- Fixes to /contrib/cube (Bruno Wolff)
- Improve /contrib/fulltextindex (Christopher)

E.195. Release 7.2.8

Release Date

2005-05-09

This release contains a variety of fixes from 7.2.7, including one security-related issue.

E.195.1. Migration to Version 7.2.8

A dump/restore is not required for those running 7.2.X.

E.195.2. Changes

- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix `EXTRACT(EPOCH)` for TIME WITH TIME ZONE values
- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump index names and trigger names containing `%` correctly (Neil)
- Prevent `to_char(interval)` from dumping core for month-related formats
- Fix `contrib/pgcrypto` for newer OpenSSL builds (Marko Kreen)

E.196. Release 7.2.7

Release Date

2005-01-31

This release contains a variety of fixes from 7.2.6, including several security-related issues.

E.196.1. Migration to Version 7.2.7

A dump/restore is not required for those running 7.2.X.

E.196.2. Changes

- Disallow **LOAD** to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), **LOAD** can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Fix display of negative intervals in SQL and GERMAN datestyles

E.197. Release 7.2.6

Release Date

2004-10-22

This release contains a variety of fixes from 7.2.5.

E.197.1. Migration to Version 7.2.6

A dump/restore is not required for those running 7.2.X.

E.197.2. Changes

- Repair possible failure to update hint bits on disk

  Under rare circumstances this oversight could lead to « could not access transaction status » failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples

  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg_ctl as root

  This is to guard against any possible security issues.
- Avoid using temp files in /tmp in make_oidjoins_check

  This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.
- Update to newer versions of Bison

E.198. Release 7.2.5

Release Date

2004-08-16

This release contains a variety of fixes from 7.2.4.

E.198.1. Migration to Version 7.2.5

A dump/restore is not required for those running 7.2.X.

E.198.2. Changes

- Prevent possible loss of committed transactions during crash

  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL™ 7.1.
- Fix corner case for btree search in parallel with first root page split
- Fix buffer overrun in `to_ascii` (Guido Notari)
- Fix core dump in deadlock detection on machines where char is unsigned
- Fix failure to respond to **pg_ctl stop -m fast** after Async_NotifyHandler runs
- Repair memory leaks in pg_dump
- Avoid conflict with system definition of `isblank()` function or macro

E.199. Release 7.2.4

Release Date

2003-01-30

This release contains a variety of fixes for version 7.2.3, including fixes to prevent possible data loss.

E.199.1. Migration to Version 7.2.4

A dump/restore is *not* required for those running version 7.2.*.

E.199.2. Changes

- Fix some additional cases of VACUUM "No one parent tuple was found" error
- Prevent VACUUM from being called inside a function (Bruce)
- Ensure pg_clog updates are sync'd to disk before marking checkpoint complete
- Avoid integer overflow during large hash joins
- Make GROUP commands work when pg_group.grolist is large enough to be toasted
- Fix errors in datetime tables; some timezone names weren't being recognized
- Fix integer overflows in circle_poly(), path_encode(), path_add() (Neil)
- Repair long-standing logic errors in lseg_eq(), lseg_ne(), lseg_center()

E.200. Release 7.2.3

Release Date

2002-10-01

This release contains a variety of fixes for version 7.2.2, including fixes to prevent possible data loss.

E.200.1. Migration to Version 7.2.3

A dump/restore is *not* required for those running version 7.2.*.

E.200.2. Changes

- Prevent possible compressed transaction log loss (Tom)
- Prevent non-superuser from increasing most recent vacuum info (Tom)
- Handle pre-1970 date values in newer versions of glibc (Tom)
- Fix possible hang during server shutdown
- Prevent spinlock hangs on SMP PPC machines (Tomoyuki Niijima)
- Fix pg_dump to properly dump FULL JOIN USING (Tom)

E.201. Release 7.2.2

Release Date

2002-08-23

This release contains a variety of fixes for version 7.2.1.

E.201.1. Migration to Version 7.2.2

A dump/restore is *not* required for those running version 7.2.*.

E.201.2. Changes

- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)
- Fix for compressed transaction log id wraparound (Tom)
- Fix PQescapeBytea/PQunescapeBytea so that they handle bytes > 0x7f (Tatsuo)
- Fix for psql and pg_dump crashing when invoked with non-existent long options (Tatsuo)
- Fix crash when invoking geometric operators (Tom)
- Allow OPEN cursor(args) (Tom)
- Fix for rtree_gist index build (Teodor)
- Fix for dumping user-defined aggregates (Tom)
- contrib/intarray fixes (Oleg)
- Fix for complex UNION/EXCEPT/INTERSECT queries using parens (Tom)
- Fix to pg_convert (Tatsuo)
- Fix for crash with long DATA strings (Thomas, Neil)
- Fix for repeat(), lpad(), rpad() and long strings (Neil)

E.202. Release 7.2.1

Release Date

2002-03-21

This release contains a variety of fixes for version 7.2.

E.202.1. Migration to Version 7.2.1

A dump/restore is *not* required for those running version 7.2.

E.202.2. Changes

- Ensure that sequence counters do not go backwards after a crash (Tom)

- Fix pgaccess kanji-conversion key binding (Tatsuo)
- Optimizer improvements (Tom)
- Cash I/O improvements (Tom)
- New Russian FAQ
- Compile fix for missing AuthBlockSig (Heiko)
- Additional time zones and time zone fixes (Thomas)
- Allow psql \connect to handle mixed case database and user names (Tom)
- Return proper OID on command completion even with ON INSERT rules (Tom)
- Allow COPY FROM to use 8-bit DELIMITERS (Tatsuo)
- Fix bug in extract/date_part for milliseconds/microseconds (Tatsuo)
- Improve handling of multiple UNIONs with different lengths (Tom)
- contrib/btree_gist improvements (Teodor Sigaev)
- contrib/tsearch dictionary improvements, see README.tsearch for an additional installation step (Thomas T. Thai, Teodor Sigaev)
- Fix for array subscripts handling (Tom)
- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)

E.203. Release 7.2

Release Date

2002-02-04

E.203.1. Overview

This release improves PostgreSQL™ for use in high-volume applications.

Major changes in this release:

VACUUM
: Vacuuming no longer locks tables, thus allowing normal user access during the vacuum. A new **VACUUM FULL** command does old-style vacuum by locking the table and shrinking the on-disk copy of the table.

Transactions
: There is no longer a problem with installations that exceed four billion transactions.

OIDs
: OIDs are now optional. Users can now create tables without OIDs for cases where OID usage is excessive.

Optimizer
: The system now computes histogram column statistics during **ANALYZE**, allowing much better optimizer choices.

Security
: A new MD5 encryption option allows more secure storage and transfer of passwords. A new Unix-domain socket authentication option is available on Linux and BSD systems.

Statistics
: Administrators can use the new table access statistics module to get fine-grained information about table and index usage.

Internationalization
: Program and library messages can now be displayed in several languages.

E.203.2. Migration to Version 7.2

A dump/restore using **pg_dump** is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The semantics of the **VACUUM** command have changed in this release. You might wish to update your maintenance procedures accordingly.
- In this release, comparisons using `= NULL` will always return false (or NULL, more precisely). Previous releases automatically transformed this syntax to `IS NULL`. The old behavior can be re-enabled using a `postgresql.conf` parameter.
- The `pg_hba.conf` and `pg_ident.conf` configuration is now only reloaded after receiving a `SIGHUP` signal, not with each connection.
- The function `octet_length()` now returns the uncompressed data length.
- The date/time value `'current'` is no longer available. You will need to rewrite your applications.
- The `timestamp()`, `time()`, and `interval()` functions are no longer available. Instead of `timestamp()`, use `timestamp 'string'` or `CAST`.

The `SELECT ... LIMIT #,#` syntax will be removed in the next release. You should change your queries to use separate LIMIT and OFFSET clauses, e.g. `LIMIT 10 OFFSET 20`.

E.203.3. Changes

E.203.3.1. Server Operation

- Create temporary files in a separate directory (Bruce)
- Delete orphaned temporary files on postmaster startup (Bruce)
- Added unique indexes to some system tables (Tom)
- System table operator reorganization (Oleg Bartunov, Teodor Sigaev, Tom)
- Renamed pg_log to pg_clog (Tom)
- Enable SIGTERM, SIGQUIT to kill backends (Jan)
- Removed compile-time limit on number of backends (Tom)
- Better cleanup for semaphore resource failure (Tatsuo, Tom)
- Allow safe transaction ID wraparound (Tom)
- Removed OIDs from some system tables (Tom)
- Removed "triggered data change violation" error check (Tom)
- SPI portal creation of prepared/saved plans (Jan)
- Allow SPI column functions to work for system columns (Tom)
- Long value compression improvement (Tom)
- Statistics collector for table, index access (Jan)
- Truncate extra-long sequence names to a reasonable value (Tom)
- Measure transaction times in milliseconds (Thomas)
- Fix TID sequential scans (Hiroshi)
- Superuser ID now fixed at 1 (Peter E)
- New pg_ctl "reload" option (Tom)

E.203.3.2. Performance

- Optimizer improvements (Tom)
- New histogram column statistics for optimizer (Tom)
- Reuse write-ahead log files rather than discarding them (Tom)

- Cache improvements (Tom)
- IS NULL, IS NOT NULL optimizer improvement (Tom)
- Improve lock manager to reduce lock contention (Tom)
- Keep relcache entries for index access support functions (Tom)
- Allow better selectivity with NaN and infinities in NUMERIC (Tom)
- R-tree performance improvements (Kenneth Been)
- B-tree splits more efficient (Tom)

E.203.3.3. Privileges

- Change UPDATE, DELETE privileges to be distinct (Peter E)
- New REFERENCES, TRIGGER privileges (Peter E)
- Allow GRANT/REVOKE to/from more than one user at a time (Peter E)
- New has_table_privilege() function (Joe Conway)
- Allow non-superuser to vacuum database (Tom)
- New SET SESSION AUTHORIZATION command (Peter E)
- Fix bug in privilege modifications on newly created tables (Tom)
- Disallow access to pg_statistic for non-superuser, add user-accessible views (Tom)

E.203.3.4. Client Authentication

- Fork postmaster before doing authentication to prevent hangs (Peter E)
- Add ident authentication over Unix domain sockets on Linux, *BSD (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
- Add a password authentication method that uses MD5 encryption (Bruce)
- Allow encryption of stored passwords using MD5 (Bruce)
- PAM authentication (Dominic J. Eidson)
- Load pg_hba.conf and pg_ident.conf only on startup and SIGHUP (Bruce)

E.203.3.5. Server Configuration

- Interpretation of some time zone abbreviations as Australian rather than North American now settable at run time (Bruce)
- New parameter to set default transaction isolation level (Peter E)
- New parameter to enable conversion of "expr = NULL" into "expr IS NULL", off by default (Peter E)
- New parameter to control memory usage by VACUUM (Tom)
- New parameter to set client authentication timeout (Tom)
- New parameter to set maximum number of open files (Tom)

E.203.3.6. Queries

- Statements added by INSERT rules now execute after the INSERT (Jan)
- Prevent unadorned relation names in target list (Bruce)
- NULLs now sort after all normal values in ORDER BY (Tom)
- New IS UNKNOWN, IS NOT UNKNOWN Boolean tests (Tom)
- New SHARE UPDATE EXCLUSIVE lock mode (Tom)

- New EXPLAIN ANALYZE command that shows run times and row counts (Martijn van Oosterhout)
- Fix problem with LIMIT and subqueries (Tom)
- Fix for LIMIT, DISTINCT ON pushed into subqueries (Tom)
- Fix nested EXCEPT/INTERSECT (Tom)

E.203.3.7. Schema Manipulation

- Fix SERIAL in temporary tables (Bruce)
- Allow temporary sequences (Bruce)
- Sequences now use int8 internally (Tom)
- New SERIAL8 creates int8 columns with sequences, default still SERIAL4 (Tom)
- Make OIDs optional using WITHOUT OIDS (Tom)
- Add %TYPE syntax to CREATE TYPE (Ian Lance Taylor)
- Add ALTER TABLE / DROP CONSTRAINT for CHECK constraints (Christopher Kings-Lynne)
- New CREATE OR REPLACE FUNCTION to alter existing function (preserving the function OID) (Gavin Sherry)
- Add ALTER TABLE / ADD [UNIQUE | PRIMARY] (Christopher Kings-Lynne)
- Allow column renaming in views
- Make ALTER TABLE / RENAME COLUMN update column names of indexes (Brent Verner)
- Fix for ALTER TABLE / ADD CONSTRAINT ... CHECK with alled tables (Stephan Szabo)
- ALTER TABLE RENAME update foreign-key trigger arguments correctly (Brent Verner)
- DROP AGGREGATE and COMMENT ON AGGREGATE now accept an aggtype (Tom)
- Add automatic return type data casting for SQL functions (Tom)
- Allow GiST indexes to handle NULLs and multikey indexes (Oleg Bartunov, Teodor Sigaev, Tom)
- Enable partial indexes (Martijn van Oosterhout)

E.203.3.8. Utility Commands

- Add RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP now allow options in any order (Vince)
- Add LOCK A, B, C functionality (Neil Padgett)
- New ENCRYPTED/UNENCRYPTED option to CREATE/ALTER USER (Bruce)
- New light-weight VACUUM does not lock table; old semantics are available as VACUUM FULL (Tom)
- Disable COPY TO/FROM on views (Bruce)
- COPY DELIMITERS string must be exactly one character (Tom)
- VACUUM warning about index tuples fewer than heap now only appears when appropriate (Martijn van Oosterhout)
- Fix privilege checks for CREATE INDEX (Tom)
- Disallow inappropriate use of CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

E.203.3.9. Data Types and Functions

- SUM(), AVG(), COUNT() now uses int8 internally for speed (Tom)
- Add convert(), convert2() (Tatsuo)
- New function bit_length() (Peter E)

- Make the "n" in CHAR(n)/VARCHAR(n) represents letters, not bytes (Tatsuo)
- CHAR(), VARCHAR() now reject strings that are too long (Peter E)
- BIT VARYING now rejects bit strings that are too long (Peter E)
- BIT now rejects bit strings that do not match declared size (Peter E)
- INET, CIDR text conversion functions (Alex Pilosov)
- INET, CIDR operators << and <<= indexable (Alex Pilosov)
- Bytea \### now requires valid three digit octal number
- Bytea comparison improvements, now supports =, <>, >, >=, <, and <=
- Bytea now supports B-tree indexes
- Bytea now supports LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea now supports concatenation
- New bytea functions: position, substring, trim, btrim, and length
- New encode() function mode, "escaped", converts minimally escaped bytea to/from text
- Add pg_database_encoding_max_length() (Tatsuo)
- Add pg_client_encoding() function (Tatsuo)
- now() returns time with millisecond precision (Thomas)
- New TIMESTAMP WITHOUT TIMEZONE data type (Thomas)
- Add ISO date/time specification with "T", yyyy-mm-ddThh:mm:ss (Thomas)
- New xid/int comparison functions (Hiroshi)
- Add precision to TIME, TIMESTAMP, and INTERVAL data types (Thomas)
- Modify type coercion logic to attempt binary-compatible functions first (Tom)
- New encode() function installed by default (Marko Kreen)
- Improved to_*() conversion functions (Karel Zak)
- Optimize LIKE/ILIKE when using single-byte encodings (Tatsuo)
- New functions in contrib/pgcrypto: crypt(), hmac(), encrypt(), gen_salt() (Marko Kreen)
- Correct description of translate() function (Bruce)
- Add INTERVAL argument for SET TIME ZONE (Thomas)
- Add INTERVAL YEAR TO MONTH (etc.) syntax (Thomas)
- Optimize length functions when using single-byte encodings (Tatsuo)
- Fix path_inter, path_distance, path_length, dist_ppath to handle closed paths (Curtis Barrett, Tom)
- octet_length(text) now returns non-compressed length (Tatsuo, Bruce)
- Handle "July" full name in date/time literals (Greg Sabino Mullane)
- Some datatype() function calls now evaluated differently
- Add support for Julian and ISO time specifications (Thomas)

E.203.3.10. Internationalization

- National language support in psql, pg_dump, libpq, and server (Peter E)
- Message translations in Chinese (simplified, traditional), Czech, French, German, Hungarian, Russian, Swedish (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- Make trim, ltrim, rtrim, btrim, lpad, rpad, translate multibyte aware (Tatsuo)
- Add LATIN5,6,7,8,9,10 support (Tatsuo)

- Add ISO 8859-5,6,7,8 support (Tatsuo)
- Correct LATIN5 to mean ISO-8859-9, not ISO-8859-5 (Tatsuo)
- Make mic2ascii() non-ASCII aware (Tatsuo)
- Reject invalid multibyte character sequences (Tatsuo)

E.203.3.11. PL/pgSQL

- Now uses portals for SELECT loops, allowing huge result sets (Jan)
- CURSOR and REFCURSOR support (Jan)
- Can now return open cursors (Jan)
- Add ELSEIF (Klaus Reger)
- Improve PL/pgSQL error reporting, including location of error (Tom)
- Allow IS or FOR key words in cursor declaration, for compatibility (Bruce)
- Fix for SELECT ... FOR UPDATE (Tom)
- Fix for PERFORM returning multiple rows (Tom)
- Make PL/pgSQL use the server's type coercion code (Tom)
- Memory leak fix (Jan, Tom)
- Make trailing semicolon optional (Tom)

E.203.3.12. PL/Perl

- New untrusted PL/Perl (Alex Pilosov)
- PL/Perl is now built on some platforms even if libperl is not shared (Peter E)

E.203.3.13. PL/Tcl

- Now reports errorInfo (Vsevolod Lobko)
- Add spi_lastoid function (bob@redivi.com)

E.203.3.14. PL/Python

- ...is new (Andrew Bosma)

E.203.3.15. psql

- \d displays indexes in unique, primary groupings (Christopher Kings-Lynne)
- Allow trailing semicolons in backslash commands (Greg Sabino Mullane)
- Read password from /dev/tty if possible
- Force new password prompt when changing user and database (Tatsuo, Tom)
- Format the correct number of columns for Unicode (Patrice)

E.203.3.16. libpq

- New function PQescapeString() to escape quotes in command strings (Florian Weimer)
- New function PQescapeBytea() escapes binary strings for use as SQL string literals

E.203.3.17. JDBC

- Return OID of INSERT (Ken K)
- Handle more data types (Ken K)
- Handle single quotes and newlines in strings (Ken K)
- Handle NULL variables (Ken K)
- Fix for time zone handling (Barry Lind)
- Improved Druid support
- Allow eight-bit characters with non-multibyte server (Barry Lind)
- Support BIT, BINARY types (Ned Wolpert)
- Reduce memory usage (Michael Stephens, Dave Cramer)
- Update DatabaseMetaData (Peter E)
- Add DatabaseMetaData.getCatalogs() (Peter E)
- Encoding fixes (Anders Bengtsson)
- Get/setCatalog methods (Jason Davies)
- DatabaseMetaData.getColumns() now returns column defaults (Jason Davies)
- DatabaseMetaData.getColumns() performance improvement (Jeroen van Vianen)
- Some JDBC1 and JDBC2 merging (Anders Bengtsson)
- Transaction performance improvements (Barry Lind)
- Array fixes (Greg Zoller)
- Serialize addition
- Fix batch processing (Rene Pijlman)
- ExecSQL method reorganization (Anders Bengtsson)
- GetColumn() fixes (Jeroen van Vianen)
- Fix isWriteable() function (Rene Pijlman)
- Improved passage of JDBC2 conformance tests (Rene Pijlman)
- Add bytea type capability (Barry Lind)
- Add isNullable() (Rene Pijlman)
- JDBC date/time test suite fixes (Liam Stewart)
- Fix for SELECT 'id' AS xxx FROM table (Dave Cramer)
- Fix DatabaseMetaData to show precision properly (Mark Lillywhite)
- New getImported/getExported keys (Jason Davies)
- MD5 password encryption support (Jeremy Wohl)
- Fix to actually use type cache (Ned Wolpert)

E.203.3.18. ODBC

- Remove query size limit (Hiroshi)
- Remove text field size limit (Hiroshi)
- Fix for SQLPrimaryKeys in multibyte mode (Hiroshi)
- Allow ODBC procedure calls (Hiroshi)
- Improve boolean handing (Aidan Mountford)

- Most configuration options now settable via DSN (Hiroshi)
- Multibyte, performance fixes (Hiroshi)
- Allow driver to be used with iODBC or unixODBC (Peter E)
- MD5 password encryption support (Bruce)
- Add more compatibility functions to odbc.sql (Peter E)

E.203.3.19. ECPG

- EXECUTE ... INTO implemented (Christof Petig)
- Multiple row descriptor support (e.g. CARDINALITY) (Christof Petig)
- Fix for GRANT parameters (Lee Kindness)
- Fix INITIALLY DEFERRED bug
- Various bug fixes (Michael, Christof Petig)
- Auto allocation for indicator variable arrays (int *ind_p=NULL)
- Auto allocation for string arrays (char **foo_pp=NULL)
- ECPGfree_auto_mem fixed
- All function names with external linkage are now prefixed by ECPG
- Fixes for arrays of structures (Michael)

E.203.3.20. Misc. Interfaces

- Python fix fetchone() (Gerhard Haring)
- Use UTF, Unicode in Tcl where appropriate (Vsevolod Lobko, Reinhard Max)
- Add Tcl COPY TO/FROM (ljb)
- Prevent output of default index op class in pg_dump (Tom)
- Fix libpgeasy memory leak (Bruce)

E.203.3.21. Build and Install

- Configure, dynamic loader, and shared library fixes (Peter E)
- Fixes in QNX 4 port (Bernd Tegge)
- Fixes in Cygwin and Windows ports (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- Fix for Windows socket communication failures (Magnus, Mikhail Terekhov)
- Hurd compile fix (Oliver Elphick)
- BeOS fixes (Cyril Velter)
- Remove configure --enable-unicode-conversion, now enabled by multibyte (Tatsuo)
- AIX fixes (Tatsuo, Andreas)
- Fix parallel make (Peter E)
- Install SQL language manual pages into OS-specific directories (Peter E)
- Rename config.h to pg_config.h (Peter E)
- Reorganize installation layout of header files (Peter E)

E.203.3.22. Source Code

- Remove SEP_CHAR (Bruce)
- New GUC hooks (Tom)
- Merge GUC and command line handling (Marko Kreen)
- Remove EXTEND INDEX (Martijn van Oosterhout, Tom)
- New pgjindent utility to indent java code (Bruce)
- Remove define of true/false when compiling under C++ (Leandro Fanzone, Tom)
- pgindent fixes (Bruce, Tom)
- Replace strcasecmp() with strcmp() where appropriate (Peter E)
- Dynahash portability improvements (Tom)
- Add 'volatile' usage in spinlock structures
- Improve signal handling logic (Tom)

E.203.3.23. Contrib

- New contrib/rtree_gist (Oleg Bartunov, Teodor Sigaev)
- New contrib/tsearch full-text indexing (Oleg, Teodor Sigaev)
- Add contrib/dblink for remote database access (Joe Conway)
- contrib/ora2pg Oracle conversion utility (Gilles Darold)
- contrib/xml XML conversion utility (John Gray)
- contrib/fulltextindex fixes (Christopher Kings-Lynne)
- New contrib/fuzzystrmatch with levenshtein and metaphone, soundex merged (Joe Conway)
- Add contrib/intarray boolean queries, binary search, fixes (Oleg Bartunov)
- New pg_upgrade utility (Bruce)
- Add new pg_resetxlog options (Bruce, Tom)

E.204. Release 7.1.3

Release Date

2001-08-15

E.204.1. Migration to Version 7.1.3

A dump/restore is *not* required for those running 7.1.X.

E.204.2. Changes

```
Remove unused WAL segments of large transactions (Tom)
Multiaction rule fix (Tom)
PL/pgSQL memory allocation fix (Jan)
VACUUM buffer fix (Tom)
Regression test fixes (Tom)
pg_dump fixes for GRANT/REVOKE/comments on views, user-defined types (Tom)
Fix subselects with DISTINCT ON or LIMIT (Tom)
BeOS fix
Disable COPY TO/FROM a view (Tom)
Cygwin build (Jason Tishler)
```

E.205. Release 7.1.2

Release Date

2001-05-11

This has one fix from 7.1.1.

E.205.1. Migration to Version 7.1.2

A dump/restore is *not* required for those running 7.1.X.

E.205.2. Changes

```
Fix PL/pgSQL SELECTs when returning no rows
Fix for psql backslash core dump
Referential integrity privilege fix
Optimizer fixes
pg_dump cleanups
```

E.206. Release 7.1.1

Release Date

2001-05-05

This has a variety of fixes from 7.1.

E.206.1. Migration to Version 7.1.1

A dump/restore is *not* required for those running 7.1.

E.206.2. Changes

```
Fix for numeric MODULO operator (Tom)
pg_dump fixes (Philip)
pg_dump can dump 7.0 databases (Philip)
readline 4.2 fixes (Peter E)
JOIN fixes (Tom)
AIX, MSWIN, VAX, N32K fixes (Tom)
Multibytes fixes (Tom)
Unicode fixes (Tatsuo)
Optimizer improvements (Tom)
Fix for whole rows in functions (Tom)
Fix for pg_ctl and option strings with spaces (Peter E)
ODBC fixes (Hiroshi)
EXTRACT can now take string argument (Thomas)
Python fixes (Darcy)
```

E.207. Release 7.1

Release Date

2001-04-13

This release focuses on removing limitations that have existed in the PostgreSQL™ code for many years.

Major changes in this release:

Write-ahead Log (WAL)
: To maintain database consistency in case of an operating system crash, previous releases of PostgreSQL™ have forced all data modifications to disk before each transaction commit. With WAL, only one log file must be flushed to disk, greatly improving performance. If you have been using -F in previous releases to disable disk flushes, you might want to consider discontinuing its use.

TOAST
: TOAST - Previous releases had a compiled-in row length limit, typically 8k - 32k. This limit made storage of long text fields difficult. With TOAST, long rows of any length can be stored with good performance.

Outer Joins
: We now support outer joins. The UNION/NOT IN workaround for outer joins is no longer required. We use the SQL92 outer join syntax.

Function Manager
: The previous C function manager did not handle null values properly, nor did it support 64-bit CPU's (Alpha). The new function manager does. You can continue using your old custom functions, but you might want to rewrite them in the future to use the new function manager call interface.

Complex Queries
: A large number of complex queries that were unsupported in previous releases now work. Many combinations of views, aggregates, UNION, LIMIT, cursors, subqueries, and alled tables now work properly. Inherited tables are now accessed by default. Subqueries in FROM are now supported.

E.207.1. Migration to Version 7.1

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

E.207.2. Changes

```
Bug Fixes
---------
Many multibyte/Unicode/locale fixes (Tatsuo and others)
More reliable ALTER TABLE RENAME (Tom)
Kerberos V fixes (David Wragg)
Fix for INSERT INTO...SELECT where targetlist has subqueries (Tom)
Prompt username/password on standard error (Bruce)
Large objects inv_read/inv_write fixes (Tom)
Fixes for to_char(), to_date(), to_ascii(), and to_timestamp() (Karel,
   Daniel Baldoni)
Prevent query expressions from leaking memory (Tom)
Allow UPDATE of arrays elements (Tom)
Wake up lock waiters during cancel (Hiroshi)
Fix rare cursor crash when using hash join (Tom)
Fix for DROP TABLE/INDEX in rolled-back transaction (Hiroshi)
Fix psql crash from \l+ if MULTIBYTE enabled (Peter E)
Fix truncation of rule names during CREATE VIEW (Ross Reedstrom)
Fix PL/perl (Alex Kapranoff)
Disallow LOCK on views (Mark Hollomon)
Disallow INSERT/UPDATE/DELETE on views (Mark Hollomon)
Disallow DROP RULE, CREATE INDEX, TRUNCATE on views (Mark Hollomon)
Allow PL/pgSQL accept non-ASCII identifiers (Tatsuo)
Allow views to proper handle GROUP BY, aggregates, DISTINCT (Tom)
Fix rare failure with TRUNCATE command (Tom)
Allow UNION/INTERSECT/EXCEPT to be used with ALL, subqueries, views,
   DISTINCT, ORDER BY, SELECT...INTO (Tom)
Fix parser failures during aborted transactions (Tom)
Allow temporary relations to properly clean up indexes (Bruce)
Fix VACUUM problem with moving rows in same page (Tom)
Modify pg_dump to better handle user-defined items in template1 (Philip)
Allow LIMIT in VIEW (Tom)
Require cursor FETCH to honor LIMIT (Tom)
Allow PRIMARY/FOREIGN Key definitions on alled columns (Stephan)
Allow ORDER BY, LIMIT in subqueries (Tom)
Allow UNION in CREATE RULE (Tom)
```

```
Make ALTER/DROP TABLE rollback-able (Vadim, Tom)
Store initdb collation in pg_control so collation cannot be changed (Tom)
Fix INSERT...SELECT with rules (Tom)
Fix FOR UPDATE inside views and subselects (Tom)
Fix OVERLAPS operators conform to SQL92 spec regarding NULLs (Tom)
Fix lpad() and rpad() to handle length less than input string (Tom)
Fix use of NOTIFY in some rules (Tom)
Overhaul btree code (Tom)
Fix NOT NULL use in Pl/pgSQL variables (Tom)
Overhaul GIST code (Oleg)
Fix CLUSTER to preserve constraints and column default (Tom)
Improved deadlock detection handling (Tom)
Allow multiple SERIAL columns in a table (Tom)
Prevent occasional index corruption (Vadim)

Enhancements
------------
Add OUTER JOINs (Tom)
Function manager overhaul (Tom)
Allow ALTER TABLE RENAME on indexes (Tom)
Improve CLUSTER (Tom)
Improve ps status display for more platforms (Peter E, Marc)
Improve CREATE FUNCTION failure message (Ross)
JDBC improvements (Peter, Travis Bauer, Christopher Cain, William Webber,
   Gunnar)
Grand Unified Configuration scheme/GUC.  Many options can now be set in
   data/postgresql.conf, postmaster/postgres flags, or SET commands (Peter E)
Improved handling of file descriptor cache (Tom)
New warning code about auto-created table alias entries (Bruce)
Overhaul initdb process (Tom, Peter E)
Overhaul of alled tables; inherited tables now accessed by default;
  new ONLY key word prevents it (Chris Bitmead, Tom)
ODBC cleanups/improvements (Nick Gorham, Stephan Szabo, Zoltan Kovacs,
   Michael Fork)
Allow renaming of temp tables (Tom)
Overhaul memory manager contexts (Tom)
pg_dumpall uses CREATE USER or CREATE GROUP rather using COPY (Peter E)
Overhaul pg_dump (Philip Warner)
Allow pg_hba.conf secondary password file to specify only username (Peter E)
Allow TEMPORARY or TEMP key word when creating temporary tables (Bruce)
New memory leak checker (Karel)
New SET SESSION CHARACTERISTICS (Thomas)
Allow nested block comments (Thomas)
Add WITHOUT TIME ZONE type qualifier (Thomas)
New ALTER TABLE ADD CONSTRAINT (Stephan)
Use NUMERIC accumulators for INTEGER aggregates (Tom)
Overhaul aggregate code (Tom)
New VARIANCE and STDDEV() aggregates
Improve dependency ordering of pg_dump (Philip)
New pg_restore command (Philip)
New pg_dump tar output option (Philip)
New pg_dump of large objects  (Philip)
New ESCAPE option to LIKE (Thomas)
New case-insensitive LIKE - ILIKE (Thomas)
Allow functional indexes to use binary-compatible type (Tom)
Allow SQL functions to be used in more contexts (Tom)
New pg_config utility (Peter E)
New PL/pgSQL EXECUTE command which allows dynamic SQL and utility statements
   (Jan)
New PL/pgSQL GET DIAGNOSTICS statement for SPI value access (Jan)
New quote_identifiers() and quote_literal() functions (Jan)
New ALTER TABLE table OWNER TO user command (Mark Hollomon)
Allow subselects in FROM, i.e. FROM (SELECT ...) [AS] alias (Tom)
Update PyGreSQL to version 3.1 (D'Arcy)
Store tables as files named by OID (Vadim)
New SQL function setval(seq,val,bool) for use in pg_dump (Philip)
Require DROP VIEW to remove views, no DROP TABLE (Mark)
Allow DROP VIEW view1, view2 (Mark)
Allow multiple objects in DROP INDEX, DROP RULE, and DROP TYPE (Tom)
```

```
Allow automatic conversion to/from Unicode (Tatsuo, Eiji)
New /contrib/pgcrypto hashing functions (Marko Kreen)
New pg_dumpall --globals-only option (Peter E)
New CHECKPOINT command for WAL which creates new WAL log file (Vadim)
New AT TIME ZONE syntax (Thomas)
Allow location of Unix domain socket to be configurable (David J. MacKenzie)
Allow postmaster to listen on a specific IP address (David J. MacKenzie)
Allow socket path name to be specified in hostname by using leading slash
   (David J. MacKenzie)
Allow CREATE DATABASE to specify template database (Tom)
New utility to convert MySQL schema dumps to SQL92 and PostgreSQL (Thomas)
New /contrib/rserv replication toolkit (Vadim)
New file format for COPY BINARY (Tom)
New /contrib/oid2name to map numeric files to table names (B Palmer)
New "idle in transaction" ps status message (Marc)
Update to pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl now defaults to -w (wait) on shutdown, new -l (log) option
Add rudimentary dependency checking to pg_dump (Philip)

Types
-----
Fix INET/CIDR type ordering and add new functions (Tom)
Make OID behave as an unsigned type (Tom)
Allow BIGINT as synonym for INT8 (Peter E)
New int2 and int8 comparison operators (Tom)
New BIT and BIT VARYING types (Adriaan Joubert, Tom, Peter E)
CHAR() no longer faster than VARCHAR() because of TOAST (Tom)
New GIST seg/cube examples (Gene Selkov)
Improved round(numeric) handling (Tom)
Fix CIDR output formatting (Tom)
New CIDR abbrev() function (Tom)

Performance
-----------
Write-Ahead Log (WAL) to provide crash recovery with less performance
   overhead (Vadim)
ANALYZE stage of VACUUM no longer exclusively locks table (Bruce)
Reduced file seeks (Denis Perchine)
Improve BTREE code for duplicate keys (Tom)
Store all large objects in a single table (Denis Perchine, Tom)
Improve memory allocation performance (Karel, Tom)

Source Code
-----------
New function manager call conventions (Tom)
SGI portability fixes (David Kaelbling)
New configure --enable-syslog option (Peter E)
New BSDI README (Bruce)
configure script moved to top level, not /src (Peter E)
Makefile/configuration/compilation overhaul (Peter E)
New configure --with-python option (Peter E)
Solaris cleanups (Peter E)
Overhaul /contrib Makefiles (Karel)
New OpenSSL configuration option (Magnus, Peter E)
AIX fixes (Andreas)
QNX fixes (Maurizio)
New heap_open(), heap_openr() API (Tom)
Remove colon and semi-colon operators (Thomas)
New pg_class.relkind value for views (Mark Hollomon)
Rename ichar() to chr() (Karel)
New documentation for btrim(), ascii(), chr(), repeat() (Karel)
Fixes for NT/Cygwin (Pete Forman)
AIX port fixes (Andreas)
New BeOS port (David Reid, Cyril Velter)
Add proofreader's changes to docs (Addison-Wesley, Bruce)
New Alpha spinlock code (Adriaan Joubert, Compaq)
UnixWare port overhaul (Peter E)
New Darwin/Mac OS X port (Peter Bierman, Bruce Hartzler)
New FreeBSD Alpha port (Alfred)
```

```
Overhaul shared memory segments (Tom)
Add IBM S/390 support (Neale Ferguson)
Moved macmanuf to /contrib (Larry Rosenman)
Syslog improvements (Larry Rosenman)
New template0 database that contains no user additions (Tom)
New /contrib/cube and /contrib/seg GIST sample code (Gene Selkov)
Allow NetBSD's libedit instead of readline (Peter)
Improved assembly language source code format (Bruce)
New contrib/pg_logger
New --template option to createdb
New contrib/pg_control utility (Oliver)
New FreeBSD tools ipc_check, start-scripts/freebsd
```

E.208. Release 7.0.3

Release Date

2000-11-11

This has a variety of fixes from 7.0.2.

E.208.1. Migration to Version 7.0.3

A dump/restore is *not* required for those running 7.0.*.

E.208.2. Changes

```
Jdbc fixes (Peter)
Large object fix (Tom)
Fix lean in COPY WITH OIDS leak (Tom)
Fix backwards-index-scan (Tom)
Fix SELECT ... FOR UPDATE so it checks for duplicate keys (Hiroshi)
Add --enable-syslog to configure (Marc)
Fix abort transaction at backend exit in rare cases (Tom)
Fix for psql \l+ when multibyte enabled (Tatsuo)
Allow PL/pgSQL to accept non ascii identifiers (Tatsuo)
Make vacuum always flush buffers (Tom)
Fix to allow cancel while waiting for a lock (Hiroshi)
Fix for memory allocation problem in user authentication code (Tom)
Remove bogus use of int4out() (Tom)
Fixes for multiple subqueries in COALESCE or BETWEEN (Tom)
Fix for failure of triggers on heap open in certain cases (Jeroen van
   Vianen)
Fix for erroneous selectivity of not-equals (Tom)
Fix for erroneous use of strcmp() (Tom)
Fix for bug where storage manager accesses items beyond end of file
   (Tom)
Fix to include kernel errno message in all smgr elog messages (Tom)
Fix for '.' not in PATH at build time (SL Baur)
Fix for out-of-file-descriptors error (Tom)
Fix to make pg_dump dump 'iscachable' flag for functions (Tom)
Fix for subselect in targetlist of Append node (Tom)
Fix for mergejoin plans (Tom)
Fix TRUNCATE failure on relations with indexes (Tom)
Avoid database-wide restart on write error (Hiroshi)
Fix nodeMaterial to honor chgParam by recomputing its output (Tom)
Fix VACUUM problem with moving chain of update row versions when source
   and destination of a row version lie on the same page (Tom)
Fix user.c CommandCounterIncrement (Tom)
Fix for AM/PM boundary problem in to_char() (Karel Zak)
Fix TIME aggregate handling (Tom)
Fix to_char() to avoid coredump on NULL input (Tom)
Buffer fix (Tom)
Fix for inserting/copying longer multibyte strings into char() data
   types (Tatsuo)
```

```
Fix for crash of backend, on abort (Tom)
```

E.209. Release 7.0.2

Release Date

2000-06-05

This is a repackaging of 7.0.1 with added documentation.

E.209.1. Migration to Version 7.0.2

A dump/restore is *not* required for those running 7.*.

E.209.2. Changes

```
Added documentation to tarball.
```

E.210. Release 7.0.1

Release Date

2000-06-01

This is a cleanup release for 7.0.

E.210.1. Migration to Version 7.0.1

A dump/restore is *not* required for those running 7.0.

E.210.2. Changes

```
Fix many CLUSTER failures (Tom)
Allow ALTER TABLE RENAME works on indexes (Tom)
Fix plpgsql to handle datetime->timestamp and timespan->interval (Bruce)
New configure --with-setproctitle switch to use setproctitle() (Marc, Bruce)
Fix the off by one errors in ResultSet from 6.5.3, and more.
jdbc ResultSet fixes (Joseph Shraibman)
optimizer tunings (Tom)
Fix create user for pgaccess
Fix for UNLISTEN failure
IRIX fixes (David Kaelbling)
QNX fixes (Andreas Kardos)
Reduce COPY IN lock level (Tom)
Change libpqeasy to use PQconnectdb() style parameters (Bruce)
Fix pg_dump to handle OID indexes (Tom)
Fix small memory leak (Tom)
Solaris fix for createdb/dropdb (Tatsuo)
Fix for non-blocking connections (Alfred Perlstein)
Fix improper recovery after RENAME TABLE failures (Tom)
Copy pg_ident.conf.sample into /lib directory in install (Bruce)
Add SJIS UDC (NEC selection IBM kanji) support (Eiji Tokuya)
Fix too long syslog message (Tatsuo)
Fix problem with quoted indexes that are too long (Tom)
JDBC ResultSet.getTimestamp() fix (Gregory Krasnow & Floyd Marinescu)
ecpg changes (Michael)
```

E.211. Release 7.0

Release Date

2000-05-08

This release contains improvements in many areas, demonstrating the continued growth of PostgreSQL™. There are more improvements and fixes in 7.0 than in any previous release. The developers have confidence that this is the best release yet; we do our best to put out only solid releases, and this one is no exception.

Major changes in this release:

Foreign Keys
: Foreign keys are now implemented, with the exception of PARTIAL MATCH foreign keys. Many users have been asking for this feature, and we are pleased to offer it.

Optimizer Overhaul
: Continuing on work started a year ago, the optimizer has been improved, allowing better query plan selection and faster performance with less memory usage.

Updated psql
: psql, our interactive terminal monitor, has been updated with a variety of new features. See the psql manual page for details.

Join Syntax
: SQL92 join syntax is now supported, though only as `INNER JOIN` for this release. `JOIN`, `NATURAL JOIN`, `JOIN/USING`, and `JOIN/ON` are available, as are column correlation names.

E.211.1. Migration to Version 7.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL™. For those upgrading from 6.5.*, you can instead use pg_upgrade to upgrade to this release; however, a full dump/reload installation is always the most robust method for upgrades.

Interface and compatibility issues to consider for the new release include:

- The date/time types datetime and timespan have been superseded by the SQL92-defined types timestamp and interval. Although there has been some effort to ease the transition by allowing PostgreSQL™ to recognize the deprecated type names and translate them to the new type names, this mechanism cannot be completely transparent to your existing application.
- The optimizer has been substantially improved in the area of query cost estimation. In some cases, this will result in decreased query times as the optimizer makes a better choice for the preferred plan. However, in a small number of cases, usually involving pathological distributions of data, your query times might go up. If you are dealing with large amounts of data, you might want to check your queries to verify performance.
- The JDBC and ODBC interfaces have been upgraded and extended.
- The string function `CHAR_LENGTH` is now a native function. Previous versions translated this into a call to `LENGTH`, which could result in ambiguity with other types implementing `LENGTH` such as the geometric types.

E.211.2. Changes

```
Bug Fixes
---------
Prevent function calls exceeding maximum number of arguments (Tom)
Improve CASE construct (Tom)
Fix SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
Fix SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)
Fix GROUP BY scan bug (Tom)
Improvements in SQL grammar processing (Tom)
Fix for views involved in INSERT ... SELECT ... (Tom)
Fix for SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
Fix for subselects in INSERT ... SELECT (Tom)
Prevent INSERT ... SELECT ... ORDER BY (Tom)
Fixes for relations greater than 2GB, including vacuum
Improve propagating system table changes to other backends (Tom)
Improve propagating user table changes to other backends (Tom)
Fix handling of temp tables in complex situations (Bruce, Tom)
```

```
Allow table locking at table open, improving concurrent reliability (Tom)
Properly quote sequence names in pg_dump (Ross J. Reedstrom)
Prevent DROP DATABASE while others accessing
Prevent any rows from being returned by GROUP BY if no rows processed (Tom)
Fix SELECT COUNT(1) FROM table WHERE ...' if no rows matching WHERE (Tom)
Fix pg_upgrade so it works for MVCC (Tom)
Fix for SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
Fix for "f1 datetime DEFAULT 'now'"  (Tom)
Fix problems with CURRENT_DATE used in DEFAULT (Tom)
Allow comment-only lines, and ;;; lines too. (Tom)
Improve recovery after failed disk writes, disk full (Hiroshi)
Fix cases where table is mentioned in FROM but not joined (Tom)
Allow HAVING clause without aggregate functions (Tom)
Fix for "--" comment and no trailing newline, as seen in perl interface
Improve pg_dump failure error reports (Bruce)
Allow sorts and hashes to exceed 2GB file sizes (Tom)
Fix for pg_dump dumping of alled rules (Tom)
Fix for NULL handling comparisons (Tom)
Fix inconsistent state caused by failed CREATE/DROP commands (Hiroshi)
Fix for dbname with dash
Prevent DROP INDEX from interfering with other backends (Tom)
Fix file descriptor leak in verify_password()
Fix for "Unable to identify an operator =$" problem
Fix ODBC so no segfault if CommLog and Debug enabled (Dirk Niggemann)
Fix for recursive exit call (Massimo)
Fix for extra-long timezones (Jeroen van Vianen)
Make pg_dump preserve primary key information (Peter E)
Prevent databases with single quotes (Peter E)
Prevent DROP DATABASE inside  transaction (Peter E)
ecpg memory leak fixes (Stephen Birch)
Fix for SELECT null::text, SELECT int4fac(null) and SELECT 2 + (null) (Tom)
Y2K timestamp fix (Massimo)
Fix for VACUUM 'HEAP_MOVED_IN was not expected' errors (Tom)
Fix for views with tables/columns containing spaces  (Tom)
Prevent privileges on indexes (Peter E)
Fix for spinlock stuck problem when error is generated (Hiroshi)
Fix ipcclean on Linux
Fix handling of NULL constraint conditions (Tom)
Fix memory leak in odbc driver (Nick Gorham)
Fix for privilege check on UNION tables (Tom)
Fix to allow SELECT 'a' LIKE 'a' (Tom)
Fix for SELECT 1 + NULL (Tom)
Fixes to CHAR
Fix log() on numeric type (Tom)
Deprecate ':' and ';' operators
Allow vacuum of temporary tables
Disallow alled columns with the same name as new columns
Recover or force failure when disk space is exhausted (Hiroshi)
Fix INSERT INTO ... SELECT with AS columns matching result columns
Fix INSERT ... SELECT ... GROUP BY groups by target columns not source columns (Tom)
Fix CREATE TABLE test (a char(5) DEFAULT text '', b int4) with INSERT (Tom)
Fix UNION with LIMIT
Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2
Fix CREATE TABLE test(col char(2) DEFAULT user)
Fix mismatched types in CREATE TABLE ... DEFAULT
Fix SELECT * FROM pg_class where oid in (0,-1)
Fix SELECT COUNT('asdf') FROM pg_class WHERE oid=12
Prevent user who can create databases can modifying pg_database table (Peter E)
Fix btree to give a useful elog when key > 1/2 (page - overhead) (Tom)
Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

Enhancements
------------
New CLI interface include file sqlcli.h, based on SQL3/SQL98
Remove all limits on query length, row length limit still exists (Tom)
Update jdbc protocol to 2.0 (Jens Glaser <jens@jens.de>)
Add TRUNCATE command to quickly truncate relation (Mike Mascari)
Fix to give super user and createdb user proper update catalog rights (Peter E)
Allow ecpg bool variables to have NULL values (Christof)
```

```
Issue ecpg error if NULL value for variable with no NULL indicator (Christof)
Allow ^C to cancel COPY command (Massimo)
Add SET FSYNC and SHOW PG_OPTIONS commands(Massimo)
Function name overloading for dynamically-loaded C functions (Frankpitt)
Add CmdTuples() to libpq++(Vince)
New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands(Jan)
Allow CREATE FUNCTION/WITH clause to be used for all language types
configure --enable-debug adds -g (Peter E)
configure --disable-debug removes -g (Peter E)
Allow more complex default expressions (Tom)
First real FOREIGN KEY constraint trigger functionality (Jan)
Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
Add FOREIGN KEY ... MATCH <unspecified> referential actions (Don Baccus)
Allow WHERE restriction on ctid (physical heap location) (Hiroshi)
Move pginterface from contrib to interface directory, rename to pgeasy (Bruce)
Change pgeasy connectdb() parameter ordering (Bruce)
Require SELECT DISTINCT target list to have all ORDER BY columns (Tom)
Add Oracle's COMMENT ON command (Mike Mascari <mascarim@yahoo.com>)
libpq's PQsetNoticeProcessor function now returns previous hook(Peter E)
Prevent PQsetNoticeProcessor from being set to NULL (Peter E)
Make USING in COPY optional (Bruce)
Allow subselects in the target list (Tom)
Allow subselects on the left side of comparison operators (Tom)
New parallel regression test (Jan)
Change backend-side COPY to write files with permissions 644 not 666 (Tom)
Force permissions on PGDATA directory to be secure, even if it exists (Tom)
Added psql LASTOID variable to return last inserted oid (Peter E)
Allow concurrent vacuum and remove pg_vlock vacuum lock file (Tom)
Add privilege check for vacuum (Peter E)
New libpq functions to allow asynchronous connections: PQconnectStart(),
  PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),
  PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)
New libpq PQsetenv() function (Ewan Mellor)
create/alter user extension (Peter E)
New postmaster.pid and postmaster.opts under $PGDATA (Tatsuo)
New scripts for create/drop user/db (Peter E)
Major psql overhaul (Peter E)
Add const to libpq interface (Peter E)
New libpq function PQoidValue (Peter E)
Show specific non-aggregate causing problem with GROUP BY (Tom)
Make changes to pg_shadow recreate pg_pwd file (Peter E)
Add aggregate(DISTINCT ...) (Tom)
Allow flag to control COPY input/output of NULLs (Peter E)
Make postgres user have a password by default (Peter E)
Add CREATE/ALTER/DROP GROUP (Peter E)
All administration scripts now support --long options (Peter E, Karel)
Vacuumdb script now supports --all option (Peter E)
ecpg new portable FETCH syntax
Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF
       and EXEC SQL ENDIF directives
Add pg_ctl script to control backend start-up (Tatsuo)
Add postmaster.opts.default file to store start-up flags (Tatsuo)
Allow --with-mb=SQL_ASCII
Increase maximum number of index keys to 16 (Bruce)
Increase maximum number of function arguments to 16 (Bruce)
Allow configuration of maximum number of index keys and arguments (Bruce)
Allow unprivileged users to change their passwords (Peter E)
Password authentication enabled; required for new users (Peter E)
Disallow dropping a user who owns a database (Peter E)
Change initdb option --with-mb to --enable-multibyte
Add option for initdb to prompts for superuser password (Peter E)
Allow complex type casts like col::numeric(9,2) and col::int2::float8 (Tom)
Updated user interfaces on initdb, initlocation, pg_dump, ipcclean (Peter E)
New pg_char_to_encoding() and pg_encoding_to_char() functions (Tatsuo)
libpq non-blocking mode (Alfred Perlstein)
Improve conversion of types in casts that don't specify a length
New plperl internal programming language (Mark Hollomon)
Allow COPY IN to read file that do not end with a newline (Tom)
Indicate when long identifiers are truncated (Tom)
Allow aggregates to use type equivalency (Peter E)
```

```
Add Oracle's to_char(), to_date(), to_datetime(), to_timestamp(), to_number()
       conversion functions (Karel Zak <zakkr@zf.jcu.cz>)
Add SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
Check to be sure ORDER BY is compatible with the DISTINCT operation (Tom)
Add NUMERIC and int8 types to ODBC
Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)
Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)
Enable backward sequential scan even after reaching EOF (Hiroshi)
Add btree indexing of boolean values, >= and <= (Don Baccus)
Print current line number when COPY FROM fails (Massimo)
Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)
Add DEC as synonym for DECIMAL (Thomas)
Add SESSION_USER as SQL92 key word, same as CURRENT_USER (Thomas)
Implement SQL92 column aliases (aka correlation names) (Thomas)
Implement SQL92 join syntax (Thomas)
Make INTERVAL reserved word allowed as a column identifier (Thomas)
Implement REINDEX command (Hiroshi)
Accept ALL in aggregate function SUM(ALL col) (Tom)
Prevent GROUP BY from using column aliases (Tom)
New psql \encoding option (Tatsuo)
Allow PQrequestCancel() to terminate when in waiting-for-lock state (Hiroshi)
Allow negation of a negative number in all cases
Add ecpg descriptors (Christof, Michael)
Allow CREATE VIEW v AS SELECT f1::char(8) FROM tbl
Allow casts with length, like foo::char(8)
New libpq functions PQsetClientEncoding(), PQclientEncoding() (Tatsuo)
Add support for SJIS user defined characters (Tatsuo)
Larger views/rules supported
Make libpq's PQconndefaults() thread-safe (Tom)
Disable // as comment to be ANSI conforming, should use -- (Tom)
Allow column aliases on views CREATE VIEW name (collist)
Fixes for views with subqueries (Tom)
Allow UPDATE table SET fld = (SELECT ...) (Tom)
SET command options no longer require quotes
Update pgaccess to 0.98.6
New SET SEED command
New pg_options.sample file
New SET FSYNC command (Massimo)
Allow pg_descriptions when creating tables
Allow pg_descriptions when creating types, columns, and functions
Allow psql \copy to allow delimiters (Peter E)
Allow psql to print nulls as distinct from "" [null] (Peter E)

Types
-----
Many array fixes (Tom)
Allow bare column names to be subscripted as arrays (Tom)
Improve type casting of int and float constants (Tom)
Cleanups for int8 inputs, range checking, and type conversion (Tom)
Fix for SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg Sharoiko)
Add btree index on NUMERIC (Jan)
Perl fix for large objects containing NUL characters (Douglas Thomson)
ODBC fix for large objects (free)
Fix indexing of cidr data type
Fix for Ethernet MAC addresses (macaddr type) comparisons
Fix for date/time types when overflows happened in computations (Tom)
Allow array on int8 (Peter E)
Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)
Allow NUMERIC arrays
Fix bugs in NUMERIC ceil() and floor() functions (Tom)
Make char_length()/octet_length including trailing blanks (Tom)
Made abstime/reltime use int4 instead of time_t (Peter E)
New lztext data type for compressed text fields
Revise code to handle coercion of int and float constants (Tom)
Start at new code to implement a BIT and BIT VARYING type (Adriaan Joubert)
NUMERIC now accepts scientific notation (Tom)
NUMERIC to int4 rounds (Tom)
Convert float4/8 to NUMERIC properly (Tom)
```

```
Allow type conversion with NUMERIC (Thomas)
Make ISO date style (2000-02-16 09:33) the default (Thomas)
Add NATIONAL CHAR [ VARYING ] (Thomas)
Allow NUMERIC round and trunc to accept negative scales (Tom)
New TIME WITH TIME ZONE type (Thomas)
Add MAX()/MIN() on time type (Thomas)
Add abs(), mod(), fac() for int8 (Thomas)
Rename functions to round(), sqrt(), cbrt(), pow() for float8 (Thomas)
Add transcendental math functions (e.g. sin(), acos()) for float8 (Thomas)
Add exp() and ln() for NUMERIC type
Rename NUMERIC power() to pow() (Thomas)
Improved TRANSLATE() function (Edwin Ramirez, Tom)
Allow X=-Y operators  (Tom)
Allow SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
Allow LOCALE to use indexes in regular expression searches (Tom)
Allow creation of functional indexes to use default types

Performance
-----------
Prevent exponential space consumption with many AND's and OR's (Tom)
Collect attribute selectivity values for system columns (Tom)
Reduce memory usage of aggregates (Tom)
Fix for LIKE optimization to use indexes with multibyte encodings (Tom)
Fix r-tree index optimizer selectivity (Thomas)
Improve optimizer selectivity computations and functions (Tom)
Optimize btree searching for cases where many equal keys exist (Tom)
Enable fast LIKE index processing only if index present (Tom)
Re-use free space on index pages with duplicates (Tom)
Improve hash join processing (Tom)
Prevent descending sort if result is already sorted(Hiroshi)
Allow commuting of index scan query qualifications (Tom)
Prefer index scans in cases where ORDER BY/GROUP BY is required (Tom)
Allocate large memory requests in fix-sized chunks for performance (Tom)
Fix vacuum's performance by reducing memory allocation requests (Tom)
Implement constant-expression simplification (Bernard Frankpitt, Tom)
Use secondary columns to be used to determine start of index scan (Hiroshi)
Prevent quadruple use of disk space when doing internal sorting (Tom)
Faster sorting by calling fewer functions (Tom)
Create system indexes to match all system caches (Bruce, Hiroshi)
Make system caches use system indexes (Bruce)
Make all system indexes unique (Bruce)
Improve pg_statistics management for VACUUM speed improvement (Tom)
Flush backend cache less frequently (Tom, Hiroshi)
COPY now reuses previous memory allocation, improving performance (Tom)
Improve optimization cost estimation (Tom)
Improve optimizer estimate of range queries x > lowbound AND x < highbound (Tom)
Use DNF instead of CNF where appropriate (Tom, Taral)
Further cleanup for OR-of-AND WHERE-clauses (Tom)
Make use of index in OR clauses (x = 1 AND y = 2) OR (x = 2 AND y = 4) (Tom)
Smarter optimizer computations for random index page access (Tom)
New SET variable to control optimizer costs (Tom)
Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications (Tom)
Reduce optimizer internal housekeeping of join paths for speedup (Tom)
Major subquery speedup (Tom)
Fewer fsync writes when fsync is not disabled (Tom)
Improved LIKE optimizer estimates (Tom)
Prevent fsync in SELECT-only queries (Vadim)
Make index creation use psort code, because it is now faster (Tom)
Allow creation of sort temp tables > 1 Gig

Source Tree Changes
-------------------
Fix for linux PPC compile
New generic expression-tree-walker subroutine (Tom)
Change form() to varargform() to prevent portability problems
Improved range checking for large integers on Alphas
Clean up #include in /include directory (Bruce)
Add scripts for checking includes (Bruce)
Remove un-needed #include's from *.c files (Bruce)
Change #include's to use <> and "" as appropriate (Bruce)
```

```
Enable Windows compilation of libpq
Alpha spinlock fix from Uncle George <gatgul@voicenet.com>
Overhaul of optimizer data structures (Tom)
Fix to cygipc library (Yutaka Tanida)
Allow pgsql to work on newer Cygwin snapshots (Dan)
New catalog version number (Tom)
Add Linux ARM
Rename heap_replace to heap_update
Update for QNX (Dr. Andreas Kardos)
New platform-specific regression handling (Tom)
Rename oid8 -> oidvector and int28 -> int2vector (Bruce)
Included all yacc and lex files into the distribution (Peter E.)
Remove lextest, no longer needed (Peter E)
Fix for libpq and psql on Windows (Magnus)
Internally change datetime and timespan into timestamp and interval (Thomas)
Fix for plpgsql on BSD/OS
Add SQL_ASCII test case to the regression test (Tatsuo)
configure --with-mb now deprecated (Tatsuo)
NT fixes
NetBSD fixes (Johnny C. Lam <lamj@stat.cmu.edu>)
Fixes for Alpha compiles
New multibyte encodings
```

E.212. Release 6.5.3

Release Date

1999-10-13

This is basically a cleanup release for 6.5.2. We have added a new PgAccess that was missing in 6.5.2, and installed an NT-specific fix.

E.212.1. Migration to Version 6.5.3

A dump/restore is *not* required for those running 6.5.*.

E.212.2. Changes

```
Updated version of pgaccess 0.98
NT-specific patch
Fix dumping rules on alled tables
```

E.213. Release 6.5.2

Release Date

1999-09-15

This is basically a cleanup release for 6.5.1. We have fixed a variety of problems reported by 6.5.1 users.

E.213.1. Migration to Version 6.5.2

A dump/restore is *not* required for those running 6.5.*.

E.213.2. Changes

```
subselect+CASE fixes(Tom)
Add SHLIB_LINK setting for solaris_i386 and solaris_sparc ports(Daren Sefcik)
Fixes for CASE in WHERE join clauses(Tom)
Fix BTScan abort(Tom)
Repair the check for redundant UNIQUE and PRIMARY KEY indexes(Thomas)
```

```
Improve it so that it checks for multicolumn constraints(Thomas)
Fix for Windows making problem with MB enabled(Hiroki Kataoka)
Allow BSD yacc and bison to compile pl code(Bruce)
Fix SET NAMES working
int8 fixes(Thomas)
Fix vacuum's memory consumption(Hiroshi,Tatsuo)
Reduce the total memory consumption of vacuum(Tom)
Fix for timestamp(datetime)
Rule deparsing bugfixes(Tom)
Fix quoting problems in mkMakefile.tcldefs.sh.in and mkMakefile.tkdefs.sh.in(Tom)
This is to re-use space on index pages freed by vacuum(Vadim)
document -x for pg_dump(Bruce)
Fix for unary operators in rule deparser(Tom)
Comment out FileUnlink of excess segments during mdtruncate()(Tom)
IRIX linking fix from Yu Cao >yucao@falcon.kla-tencor.com<
Repair logic error in LIKE: should not return LIKE_ABORT
  when reach end of pattern before end of text(Tom)
Repair incorrect cleanup of heap memory allocation during transaction abort(Tom)
Updated version of pgaccess 0.98
```

E.214. Release 6.5.1

Release Date

1999-07-15

This is basically a cleanup release for 6.5. We have fixed a variety of problems reported by 6.5 users.

E.214.1. Migration to Version 6.5.1

A dump/restore is *not* required for those running 6.5.

E.214.2. Changes

```
Add NT README file
Portability fixes for linux_ppc, IRIX, linux_alpha, OpenBSD, alpha
Remove QUERY_LIMIT, use SELECT...LIMIT
Fix for EXPLAIN on allance(Tom)
Patch to allow vacuum on multisegment tables(Hiroshi)
R-Tree optimizer selectivity fix(Tom)
ACL file descriptor leak fix(Atsushi Ogawa)
New expression subtree code(Tom)
Avoid disk writes for read-only transactions(Vadim)
Fix for removal of temp tables if last transaction was aborted(Bruce)
Fix to prevent too large row from being created(Bruce)
plpgsql fixes
Allow port numbers 32k - 64k(Bruce)
Add ^ precedence(Bruce)
Rename sort files called pg_temp to pg_sorttemp(Bruce)
Fix for microseconds in time values(Tom)
Tutorial source cleanup
New linux_m68k port
Fix for sorting of NULL's in some cases(Tom)
Shared library dependencies fixed (Tom)
Fixed glitches affecting GROUP BY in subselects(Tom)
Fix some compiler warnings (Tomoaki Nishiyama)
Add Win1250 (Czech) support (Pavel Behal)
```

E.215. Release 6.5

Release Date

1999-06-09

This release marks a major step in the development team's mastery of the source code we alled from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team.

Here is a brief summary of the more notable changes:

Multiversion concurrency control(MVCC)
: This removes our old table-level locking, and replaces it with a locking system that is superior to most commercial database systems. In a traditional system, each row that is modified is locked until committed, preventing reads by other users. MVCC uses the natural multiversion nature of PostgreSQL™ to allow readers to continue reading consistent data during writer activity. Writers continue to use the compact pg_log transaction system. This is all performed without having to allocate a lock for every row like traditional database systems. So, basically, we no longer are restricted by simple table-level locking; we have something better than row-level locking.

Hot backups from pg_dump
: pg_dump takes advantage of the new MVCC features to give a consistent database dump/backup while the database stays online and available for queries.

Numeric data type
: We now have a true numeric data type, with user-specified precision.

Temporary tables
: Temporary tables are guaranteed to have unique names within a database session, and are destroyed on session exit.

New SQL features
: We now have CASE, INTERSECT, and EXCEPT statement support. We have new LIMIT/OFFSET, SET TRANSACTION ISOLATION LEVEL, SELECT ... FOR UPDATE, and an improved LOCK TABLE command.

Speedups
: We continue to speed up PostgreSQL™, thanks to the variety of talents within our team. We have sped up memory allocation, optimization, table joins, and row transfer routines.

Ports
: We continue to expand our port list, this time including `Windows NT/ix86` and `NetBSD/arm32`.

Interfaces
: Most interfaces have new versions, and existing functionality has been improved.

Documentation
: New and updated material is present throughout the documentation. New FAQs have been contributed for `SGI` and `AIX` platforms. The *Tutorial* has introductory information on SQL from Stefan Simkovics. For the *User's Guide*, there are reference pages covering the postmaster and more utility programs, and a new appendix contains details on date/time behavior. The *Administrator's Guide* has a new chapter on troubleshooting from Tom Lane. And the *Programmer's Guide* has a description of query processing, also from Stefan, and details on obtaining the PostgreSQL™ source tree via anonymous CVS™ and CVSup™.

E.215.1. Migration to Version 6.5

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL™. pg_upgrade can *not* be used to upgrade to this release because the on-disk structure of the tables has changed compared to previous releases.

The new Multiversion Concurrency Control (MVCC) features can give somewhat different behaviors in multiuser environments. *Read and understand the following section to ensure that your existing applications will give you the behavior you need.*

E.215.1.1. Multiversion Concurrency Control

Because readers in 6.5 don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In other words, if a row is returned by **SELECT** it doesn't mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from being deleted or updated by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existence of a row and protect it against concurrent updates one must use **SELECT FOR UPDATE** or an appropriate **LOCK TABLE** statement. This should be taken into account when porting applications from previous releases of PostgreSQL™ and other environments.

Keep the above in mind if you are using `contrib/refint.*` triggers for referential integrity. Additional techniques are required now. One way is to use **LOCK parent_table IN SHARE ROW EXCLUSIVE MODE** command if a transaction is going to update/delete a primary key and use **LOCK parent_table IN SHARE MODE** command if a transaction is going to update/insert

a foreign key.

> **Note**
>
> Note that if you run a transaction in SERIALIZABLE mode then you must execute the **LOCK** commands above before execution of any DML statement (**SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO**) in the transaction.

These inconveniences will disappear in the future when the ability to read dirty (uncommitted) data (regardless of isolation level) and true referential integrity will be implemented.

E.215.2. Changes

```
Bug Fixes
---------
Fix text<->float8 and text<->float4 conversion functions(Thomas)
Fix for creating tables with mixed-case constraints(Billy)
Change exp()/pow() behavior to generate error on underflow/overflow(Jan)
Fix bug in pg_dump -z
Memory overrun cleanups(Tatsuo)
Fix for lo_import crash(Tatsuo)
Adjust handling of data type names to suppress double quotes(Thomas)
Use type coercion for matching columns and DEFAULT(Thomas)
Fix deadlock so it only checks once after one second of sleep(Bruce)
Fixes for aggregates and PL/pgsql(Hiroshi)
Fix for subquery crash(Vadim)
Fix for libpq function PQfnumber and case-insensitive names(Bahman Rafatjoo)
Fix for large object write-in-middle, no extra block, memory consumption(Tatsuo)
Fix for pg_dump -d or -D and  quote special characters in INSERT
Repair serious problems with dynahash(Tom)
Fix INET/CIDR portability problems
Fix problem with selectivity error in ALTER TABLE ADD COLUMN(Bruce)
Fix executor so mergejoin of different column types works(Tom)
Fix for Alpha OR selectivity bug
Fix OR index selectivity problem(Bruce)
Fix so \d shows proper length for char()/varchar()(Ryan)
Fix tutorial code(Clark)
Improve destroyuser checking(Oliver)
Fix for Kerberos(Rodney McDuff)
Fix for dropping database while dirty buffers(Bruce)
Fix so sequence nextval() can be case-sensitive(Bruce)
Fix !!= operator
Drop buffers before destroying database files(Bruce)
Fix case where executor evaluates functions twice(Tatsuo)
Allow sequence nextval actions to be case-sensitive(Bruce)
Fix optimizer indexing not working for negative numbers(Bruce)
Fix for memory leak in executor with fjIsNull
Fix for aggregate memory leaks(Erik Riedel)
Allow user name containing a dash to grant privileges
Cleanup of NULL in inet types
Clean up system table bugs(Tom)
Fix problems of PAGER and \? command(Masaaki Sakaida)
Reduce default multisegment file size limit to 1GB(Peter)
Fix for dumping of CREATE OPERATOR(Tom)
Fix for backward scanning of cursors(Hiroshi Inoue)
Fix for COPY FROM STDIN when using \i(Tom)
Fix for subselect is compared inside an expression(Jan)
Fix handling of error reporting while returning rows(Tom)
Fix problems with reference to array types(Tom,Jan)
Prevent UPDATE SET oid(Jan)
Fix pg_dump so -t option can handle case-sensitive tablenames
Fixes for GROUP BY in special cases(Tom, Jan)
Fix for memory leak in failed queries(Tom)
DEFAULT now supports mixed-case identifiers(Tom)
Fix for multisegment uses of DROP/RENAME table, indexes(Ole Gjerde)
Disable use of pg_dump with both -o and -d options(Bruce)
Allow pg_dump to properly dump group privileges(Bruce)
```

```
Fix GROUP BY in INSERT INTO table SELECT * FROM table2(Jan)
Fix for computations in views(Jan)
Fix for aggregates on array indexes(Tom)
Fix for DEFAULT handles single quotes in value requiring too many quotes
Fix security problem with non-super users importing/exporting large objects(Tom)
Rollback of transaction that creates table cleaned up properly(Tom)
Fix to allow long table and column names to generate proper serial names(Tom)

Enhancements
------------
Add "vacuumdb" utility
Speed up libpq by allocating memory better(Tom)
EXPLAIN all indexes used(Tom)
Implement CASE, COALESCE, NULLIF  expression(Thomas)
New pg_dump table output format(Constantin)
Add string min()/max() functions(Thomas)
Extend new type coercion techniques to aggregates(Thomas)
New moddatetime contrib(Terry)
Update to pgaccess 0.96(Constantin)
Add routines for single-byte "char" type(Thomas)
Improved substr() function(Thomas)
Improved multibyte handling(Tatsuo)
Multiversion concurrency control/MVCC(Vadim)
New Serialized mode(Vadim)
Fix for tables over 2gigs(Peter)
New SET TRANSACTION ISOLATION LEVEL(Vadim)
New LOCK TABLE IN ... MODE(Vadim)
Update ODBC driver(Byron)
New NUMERIC data type(Jan)
New SELECT FOR UPDATE(Vadim)
Handle "NaN" and "Infinity" for input values(Jan)
Improved date/year handling(Thomas)
Improved handling of backend connections(Magnus)
New options ELOG_TIMESTAMPS and USE_SYSLOG options for log files(Massimo)
New TCL_ARRAYS option(Massimo)
New INTERSECT and EXCEPT(Stefan)
New pg_index.indisprimary for primary key tracking(D'Arcy)
New pg_dump option to allow dropping of tables before creation(Brook)
Speedup of row output routines(Tom)
New READ COMMITTED isolation level(Vadim)
New TEMP tables/indexes(Bruce)
Prevent sorting if result is already sorted(Jan)
New memory allocation optimization(Jan)
Allow psql to do \p\g(Bruce)
Allow multiple rule actions(Jan)
Added LIMIT/OFFSET functionality(Jan)
Improve optimizer when joining a large number of tables(Bruce)
New intro to SQL from S. Simkovics' Master's Thesis (Stefan, Thomas)
New intro to backend processing from S. Simkovics' Master's Thesis (Stefan)
Improved int8 support(Ryan Bradetich, Thomas, Tom)
New routines to convert between int8 and text/varchar types(Thomas)
New bushy plans, where meta-tables are joined(Bruce)
Enable right-hand queries by default(Bruce)
Allow reliable maximum number of backends to be set at configure time
     (--with-maxbackends and postmaster switch (-N backends))(Tom)
GEQO default now 10 tables because of optimizer speedups(Tom)
Allow NULL=Var for MS-SQL portability(Michael, Bruce)
Modify contrib check_primary_key() so either "automatic" or "dependent"(Anand)
Allow psql \d on a view show query(Ryan)
Speedup for LIKE(Bruce)
Ecpg fixes/features, see src/interfaces/ecpg/ChangeLog file(Michael)
JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)
Make % operator have precedence like /(Bruce)
Add new postgres -O option to allow system table structure changes(Bruce)
Update contrib/pginterface/findoidjoins script(Tom)
Major speedup in vacuum of deleted rows with indexes(Vadim)
Allow non-SQL functions to run different versions based on arguments(Tom)
Add -E option that shows actual queries sent by \dt and friends(Masaaki Sakaida)
Add version number in start-up banners for psql(Masaaki Sakaida)
```

```
New contrib/vacuumlo removes large objects not referenced(Peter)
New initialization for table sizes so non-vacuumed tables perform better(Tom)
Improve error messages when a connection is rejected(Tom)
Support for arrays of char() and varchar() fields(Massimo)
Overhaul of hash code to increase reliability and performance(Tom)
Update to PyGreSQL 2.4(D'Arcy)
Changed debug options so -d4 and -d5 produce different node displays(Jan)
New pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
Better optimization statistics for system table access(Tom)
Better handling of non-default block sizes(Massimo)
Improve GEQO optimizer memory consumption(Tom)
UNION now supports ORDER BY of columns not in target list(Jan)
Major libpq++ improvements(Vince Vielhaber)
pg_dump now uses -z(ACL's) as default(Bruce)
backend cache, memory speedups(Tom)
have pg_dump do everything in one snapshot transaction(Vadim)
fix for large object memory leakage, fix for pg_dumping(Tom)
INET type now respects netmask for comparisons
Make VACUUM ANALYZE only use a readlock(Vadim)
Allow VIEWs on UNIONS(Jan)
pg_dump now can generate consistent snapshots on active databases(Vadim)

Source Tree Changes
-------------------
Improve port matching(Tom)
Portability fixes for SunOS
Add Windows NT backend port and enable dynamic loading(Magnus and Daniel Horak)
New port to Cobalt Qube(Mips) running Linux(Tatsuo)
Port to NetBSD/m68k(Mr. Mutsuki Nakajima)
Port to NetBSD/sun3(Mr. Mutsuki Nakajima)
Port to NetBSD/macppc(Toshimi Aoki)
Fix for tcl/tk configuration(Vince)
Removed CURRENT key word for rule queries(Jan)
NT dynamic loading now works(Daniel Horak)
Add ARM32 support(Andrew McMurry)
Better support for HP-UX 11 and UnixWare
Improve file handling to be more uniform, prevent file descriptor leak(Tom)
New install commands for plpgsql(Jan)
```

E.216. Release 6.4.2

Release Date

1998-12-20

The 6.4.1 release was improperly packaged. This also has one additional bug fix.

E.216.1. Migration to Version 6.4.2

A dump/restore is *not* required for those running 6.4.*.

E.216.2. Changes

```
Fix for datetime constant problem on some platforms(Thomas)
```

E.217. Release 6.4.1

Release Date

1998-12-18

This is basically a cleanup release for 6.4. We have fixed a variety of problems reported by 6.4 users.

E.217.1. Migration to Version 6.4.1

A dump/restore is *not* required for those running 6.4.

E.217.2. Changes

```
Add pg_dump -N flag to force double quotes around identifiers.  This is
       the default(Thomas)
Fix for NOT in where clause causing crash(Bruce)
EXPLAIN VERBOSE coredump fix(Vadim)
Fix shared-library problems on Linux
Fix test for table existence to allow mixed-case and whitespace in
       the table name(Thomas)
Fix a couple of pg_dump bugs
Configure matches template/.similar entries better(Tom)
Change builtin function names from SPI_* to spi_*
OR WHERE clause fix(Vadim)
Fixes for mixed-case table names(Billy)
contrib/linux/postgres.init.csh/sh fix(Thomas)
libpq memory overrun fix
SunOS fixes(Tom)
Change exp() behavior to generate error on underflow(Thomas)
pg_dump fixes for memory leak, allance constraints, layout change
update pgaccess to 0.93
Fix prototype for 64-bit platforms
Multibyte fixes(Tatsuo)
New ecpg man page
Fix memory overruns(Tatsuo)
Fix for lo_import() crash(Bruce)
Better search for install program(Tom)
Timezone fixes(Tom)
HP-UX fixes(Tom)
Use implicit type coercion for matching DEFAULT values(Thomas)
Add routines to help with single-byte (internal) character type(Thomas)
Compilation of libpq for Windows fixes(Magnus)
Upgrade to PyGreSQL 2.2(D'Arcy)
```

E.218. Release 6.4

Release Date

1998-10-30

There are *many* new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the *Programmer's Guide*.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple-byte character set support from Tatsuo Ishii to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coercion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of PostgreSQL™. There is a new chapter in the *User's Guide* which covers this topic.
- Three new data types have been added. Two types, inet and cidr, support various forms of IP network, subnet, and machine addressing. There is now an 8-byte integer type available on some platforms. See the chapter on data types in the *User's Guide* for details. A fourth type, serial, is now supported by the parser as an amalgam of the int4 type, a sequence, and a unique index.

- Several more SQL92-compatible syntax features have been added, including **INSERT DEFAULT VALUES**
- The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

E.218.1. Migration to Version 6.4

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL™.

E.218.2. Changes

```
Bug Fixes
---------
Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)
Remove char2-16 data types, use char/varchar(Darren)
Pqfn not handles a NOTICE message(Anders)
Reduced busywaiting overhead for spinlocks with many backends (dg)
Stuck spinlock detection (dg)
Fix up "ISO-style" timespan decoding and encoding(Thomas)
Fix problem with table drop after rollback of transaction(Vadim)
Change error message and remove non-functional update message(Vadim)
Fix for COPY array checking
Fix for SELECT 1 UNION SELECT NULL
Fix for buffer leaks in large object calls(Pascal)
Change owner from oid to int4 type(Bruce)
Fix a bug in the oracle compatibility functions btrim() ltrim() and rtrim()
Fix for shared invalidation cache overflow(Massimo)
Prevent file descriptor leaks in failed COPY's(Bruce)
Fix memory leak in libpgtcl's pg_select(Constantin)
Fix problems with username/passwords over 8 characters(Tom)
Fix problems with handling of asynchronous NOTIFY in backend(Tom)
Fix of many bad system table entries(Tom)

Enhancements
------------
Upgrade ecpg and ecpglib,see src/interfaces/ecpc/ChangeLog(Michael)
Show the index used in an EXPLAIN(Zeugswetter)
EXPLAIN  invokes  rule system and shows plan(s) for rewritten queries(Jan)
Multibyte awareness of many data types and functions, via configure(Tatsuo)
New configure --with-mb option(Tatsuo)
New initdb --pgencoding option(Tatsuo)
New createdb -E multibyte option(Tatsuo)
Select version(); now returns PostgreSQL version(Jeroen)
libpq now allows asynchronous clients(Tom)
Allow cancel from client of backend query(Tom)
psql now cancels query with Control-C(Tom)
libpq users need not issue dummy queries to get NOTIFY messages(Tom)
NOTIFY now sends sender's PID, so you can tell whether it was your own(Tom)
PGresult struct now includes associated error message, if any(Tom)
Define "tz_hour" and "tz_minute" arguments to date_part()(Thomas)
Add routines to convert between varchar and bpchar(Thomas)
Add routines to allow sizing of varchar and bpchar into target columns(Thomas)
Add bit flags to support timezonehour and minute in data retrieval(Thomas)
Allow more variations on valid floating point numbers (e.g. ".1", "1e6")(Thomas)
Fixes for unary minus parsing with leading spaces(Thomas)
Implement TIMEZONE_HOUR, TIMEZONE_MINUTE per SQL92 specs(Thomas)
Check for and properly ignore FOREIGN KEY column constraints(Thomas)
Define USER as synonym for CURRENT_USER per SQL92 specs(Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as bpchar)(Thomas)
Save string type if specified for DEFAULT clause handling(Thomas)
Coerce operations involving different data types(Thomas)
Allow some index use for columns of different types(Thomas)
Add capabilities for automatic type conversion(Thomas)
Cleanups for large objects, so file is truncated on open(Peter)
Readline cleanups(Tom)
```

```
Allow psql  \f \ to make spaces as delimiter(Bruce)
Pass pg_attribute.atttypmod to the frontend for column field lengths(Tom,Bruce)
Msql compatibility library in /contrib(Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be
included in the target list(David)
Convert columns to match columns in UNION clauses(Thomas)
Remove fork()/exec() and only do fork()(Bruce)
Jdbc cleanups(Peter)
Show backend status on ps command line(only works on some platforms)(Bruce)
Pg_hba.conf now has a sameuser option in the database field
Make lo_unlink take oid param, not int4
New DISABLE_COMPLEX_MACRO for compilers that cannot handle our macros(Bruce)
Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy queries(Tom)
libpgtcl cleanups(Tom)
Add -error option to libpgtcl's pg_result command(Tom)
New locale patch, see docs/README/locale(Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct(ccb)
New contrib/lo code for large object orphan removal(Peter)
New psql command "SET CLIENT_ENCODING TO 'encoding'" for multibytes
feature, see /doc/README.mb(Tatsuo)
contrib/noupdate code to revoke update permission on a column
libpq can now be compiled on Windows(Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS support(Thomas)
Better support for quoted table/column names(Thomas)
Surround table and column names with double-quotes in pg_dump(Thomas)
PQreset() now works with passwords(Tom)
Handle case of GROUP BY target list column number out of range(David)
Allow UNION in subselects
Add auto-size to screen to \d? commands(Bruce)
Use UNION to show all \d? results in one query(Bruce)
Add \d? field search feature(Bruce)
Pg_dump issues fewer \connect requests(Tom)
Make pg_dump -z flag work better, document it in manual page(Tom)
Add HAVING clause with full support for subselects and unions(Stephan)
Full text indexing routines in contrib/fulltextindex(Maarten)
Transaction ids now stored in shared memory(Vadim)
New PGCLIENTENCODING when issuing COPY command(Tatsuo)
Support for SQL92 syntax "SET NAMES"(Tatsuo)
Support for LATIN2-5(Tatsuo)
Add UNICODE regression test case(Tatsuo)
Lock manager cleanup, new locking modes for LLL(Vadim)
Allow index use with OR clauses(Bruce)
Allows "SELECT NULL ORDER BY 1;"
Explain VERBOSE prints the plan, and now pretty-prints the plan to
the postmaster log file(Bruce)
Add indexes display to \d command(Bruce)
Allow GROUP BY on functions(David)
New pg_class.relkind for large objects(Bruce)
New way to send libpq NOTICE messages to a different location(Tom)
New \w write command to psql(Bruce)
New /contrib/findoidjoins scans oid columns to find join relationships(Bruce)
Allow binary-compatible indexes to be considered when checking for valid
Indexes for restriction clauses containing a constant(Thomas)
New ISBN/ISSN code in /contrib/isbn_issn
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN constraint(Thomas)
New rewrite system fixes many problems with rules and views(Jan)
       * Rules on relations work
       * Event qualifications on insert/update/delete work
       * New OLD variable to reference CURRENT, CURRENT will be remove in future
       * Update rules can reference NEW and OLD in rule qualifications/actions
       * Insert/update/delete rules on views work
       * Multiple rule actions are now supported, surrounded by parentheses
       * Regular users can create views/rules on tables they have RULE permits
       * Rules and views all the privileges of the creator
       * No rules at the column level
       * No UPDATE NEW/OLD rules
       * New pg_tables, pg_indexes, pg_rules and pg_views system views
       * Only a single action on SELECT rules
```

```
       * Total rewrite overhaul, perhaps for 6.5
       * handle subselects
       * handle aggregates on views
       * handle insert into select from view works
System indexes are now multikey(Bruce)
Oidint2, oidint4, and oidname types are removed(Bruce)
Use system cache for more system table lookups(Bruce)
New backend programming language PL/pgSQL in backend/pl(Jan)
New SERIAL data type, auto-creates sequence/index(Thomas)
Enable assert checking without a recompile(Massimo)
User lock enhancements(Massimo)
New setval() command to set sequence value(Massimo)
Auto-remove unix socket file on start-up if no postmaster running(Massimo)
Conditional trace package(Massimo)
New UNLISTEN command(Massimo)
psql and libpq now compile under Windows using win32.mak(Magnus)
Lo_read no longer stores trailing NULL(Bruce)
Identifiers are now truncated to 31 characters internally(Bruce)
Createuser options now available on the command line
Code for 64-bit integer supported added, configure tested, int8 type(Thomas)
Prevent file descriptor leaf from failed COPY(Bruce)
New pg_upgrade command(Bruce)
Updated /contrib directories(Massimo)
New CREATE TABLE DEFAULT VALUES statement available(Thomas)
New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)
New DECLARE and FETCH feature(Thomas)
libpq's internal structures now not exported(Tom)
Allow up to 8 key indexes(Bruce)
Remove ARCHIVE key word, that is no longer used(Thomas)
pg_dump -n flag to suppress quotes around indentifiers
disable system columns for views(Jan)
new INET and CIDR types for network addresses(TomH, Paul)
no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)

Source Tree Changes
-------------------
/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
HP-UX cleanups(Tom)
Multibyte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRESDIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsdi shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ_CVS
Update backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getrusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
SCO 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documentation(Thomas)
Nextstep support(Jacek)
Aix support(David)
pginterface manual page(Bruce)
shared libraries all have version numbers
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)
```

E.219. Release 6.3.2

Release Date

1998-04-07

This is a bug-fix release for 6.3.x. Refer to the release notes for version 6.3 for a more complete summary of new features.

Summary:

- Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in version 6.3.1.
- Correctly handles function calls on the left side of BETWEEN and LIKE clauses.

A dump/restore is NOT required for those running 6.3 or 6.3.1. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL™ libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

E.219.1. Changes

```
Configure detection improvements for tcl/tk(Brook Milligan, Alvin)
Manual page improvements(Bruce)
BETWEEN and LIKE fix(Thomas)
fix for psql \connect used by pg_dump(Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups(Jeroen)
fix for buffer over-runs detected(Maurice Gittens)
fix for buffer overrun in libpgtcl(Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY(Bruce)
gettimeofday configure check(Doug Winterburn)
Fix "indexes not used" bug(Vadim)
docs additions(Thomas)
Fix for backend memory leak(Bruce)
libreadline cleanup(Erwan MAS)
Remove DISTDIR(Bruce)
Makefile dependency cleanup(Jeroen van Vianen)
ASSERT fixes(Bruce)
```

E.220. Release 6.3.1

Release Date

1998-03-23

Summary:

- Additional support for multibyte character sets.
- Repair byte ordering for mixed-endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.

A dump/restore is NOT required for those running 6.3. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL™ libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

E.220.1. Changes

```
ecpg cleanup/fixes, now version 1.1(Michael Meskes)
pg_user cleanup(Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions(Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multibyte character set support, see doc/README.mb(Tatsuo)
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications(Kataoka)
SUBSTR() and substring() fix(Jan)
several jdbc fixes(Peter)
libpgtcl improvements, see libptcl/README(Randy Kunkee)
Fix for "Datasize = 0" error(Vadim)
Prevent \do from wrapping(Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE key word in LOCK and SELECT INTO(Thomas)
CREATE SEQUENCE options to allow a negative integer(Thomas)
Add "PASSWORD" as an allowed column identifier(Thomas)
Add checks for UNION target fields(Bruce)
Fix Alpha port(Dwayne Bailey)
Fix for text arrays containing quotes(Doug Gibson)
Solaris compile fix(Albert Chin-A-Young)
Better identify tcl and tk libs and includes(Bruce)
```

E.221. Release 6.3

Release Date

1998-03-01

There are *many* new features and improvements in this release. Here is a brief, incomplete summary:

- Many new SQL features, including full SQL92 subselect capability (everything is here but target-list subselects).
- Support for client-side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you might need to start postmaster with the `-i` flag.
- Better password authorization mechanisms. Default table privileges have changed.
- Old-style *time travel* has been removed. Performance has been improved.

Note

Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that cannot be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you might think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you cannot do with subselects is to use them in the target list.

Second, 6.3 uses Unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster -i option, and of course edit `pg_hba.conf`. Also, for this reason, the format of `pg_hba.conf` has changed.

Third, char() fields will now allow faster access than varchar() or text. Specifically, the text and varchar() have a penalty for ac-

cess to any columns after the first column of this type. char() used to also have this access penalty, but it no longer does. This might suggest that you redesign some of your tables, especially if you have short character columns that you have defined as varchar() or text. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable independent of any Unix file. There are new SQL USER commands. See the *Administrator's Guide* for more information. There is a new table, pg_shadow, which is used to store user information and user passwords, and it by default only SELECT-able by the `postgres` super-user. pg_user is now a view of pg_shadow, and is SELECT-able by PUBLIC. You should keep using pg_user in your application without changes.

User-created tables now no longer have SELECT privilege to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any privileges you want after the table is created. System tables continue to be SELECT-able by PUBLIC.

We also have real deadlock detection code. No more sixty-second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documentation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the doc/ directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see `pgsql/contrib/spi/README`). Please check out the new \d command for types, operators, etc. Also, views have their own privileges now, not based on the underlying tables, so privileges on them have to be set separately. Check `/pgsql/interfaces` for some new ways to talk to PostgreSQL™.

This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work-arounds people were using are no longer needed.

E.221.1. Migration to Version 6.3

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL™.

E.221.2. Changes

```
Bug Fixes
---------
Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pqtrace(Bruce)
Fix psql \e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on alled tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)(Bruce)
Fix notty output to show status result.  -q option still turns it off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
       lock before older waiters, and having readlock people not share
       locks if a writer is waiting for a lock, and waiting writers not
       getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external files(James)
Fix problem with multiple order by columns, with the first one having
       NULL values(Jeroen)
Use correct hash table support functions for float8 and int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true','false','yes','no','1','0'
       and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
```

```
Fix for GROUP BY showing duplicates(Vadim)
Fix for index scans in MergeJoin(Vadim)

Enhancements
------------
Subselects with EXISTS, IN, ALL, ANY key words (Vadim, Bruce, Thomas)
New User Manual(Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts(Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
       CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately before)(Thomas)
Allow PostgreSQL-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")(Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
       (e.g. "DATETIME 'now'")(Thomas)
Add conversions for int2, int4, and OID types to and from text(Thomas)
Use shared lock when building indexes(Vadim)
Free memory allocated for an user query inside transaction block after
       this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New PostgreSQL™ Procedural Language (PL) backend interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indexes for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for \d and \z, and fix \i(Bruce)
Add Unix domain socket support to backend and to frontend library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation utility(Thomas)
Allow more SQL92 and/or PostgreSQL™ reserved words as column identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO
       frontend library initialization environment variables(Thomas)
Regression tests time zone automatically set with "setenv PGTZ PST8PDT"(Thomas)
Add pg_description table for info on tables, columns, operators, types, and
       aggregates(Bruce)
Increase 16 char limit on system table/index names to 32 characters(Bruce)
Rename system indexes(Bruce)
Add 'GERMAN' option to SET DATESTYLE(Thomas)
Define an "ISO-style" timespan output format with "hh:mm:ss" fields(Thomas)
Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
Validate numeric input more carefully for delta times(Thomas)
Implement day of year as possible input to date_part()(Thomas)
Define timespan_finite() and text_timespan() functions(Thomas)
Remove archive stuff(Bruce)
Allow for a pg_password authentication database that is separate from
       the system password file(Todd)
Dump ACLs, GRANT, REVOKE privileges(Matt)
Define text, varchar, and bpchar string length functions(Thomas)
Fix Query handling for allance, and cost computations(Bruce)
Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)
Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
Implement UNIONs for SELECT(Bruce)
Add UNION, GROUP, DISTINCT to INSERT(Bruce)
varchar() stores only necessary bytes on disk(Bruce)
Fix for BLOBs(Peter)
Mega-Patch for JDBC...see README_6.3 for list of changes(Peter)
```

```
Remove unused "option" from PQconnectdb()
New LOCK command and lock manual page describing deadlocks(Bruce)
Add new psql \da, \dd, \df, \do, \dS, and \dT commands(Bruce)
Enhance psql \z to show sequences(Bruce)
Show NOT NULL and DEFAULT in psql \d table(Bruce)
New psql .psqlrc file start-up(Andrew)
Modify sample start-up script in contrib/linux to show syslog(Thomas)
New types for IP and MAC addresses in contrib/ip_and_mac(TomH)
Unix system time conversions with date/time types in contrib/unixdate(Thomas)
Update of contrib stuff(Massimo)
Add Unix socket support to DBD::Pg(Goran)
New python interface (PyGreSQL 2.0)(D'Arcy)
New frontend/backend protocol has a version number, network byte order(Phil)
Security features in pg_hba.conf enhanced and documented, many cleanups(Phil)
CHAR() now faster access than VARCHAR() or TEXT
ecpg embedded SQL preprocessor
Reduce system column overhead(Vadmin)
Remove pg_time table(Vadim)
Add pg_type attribute to identify types that need length (bpchar, varchar)
Add report of offending line when COPY command fails
Allow VIEW privileges to be set separately from the underlying tables.
       For security, use GRANT/REVOKE on views as appropriate(Jan)
Tables now have no default GRANT SELECT TO PUBLIC.  You must
       explicitly grant such privileges.
Clean up tutorial examples(Darren)

Source Tree Changes
-------------------
Add new html development tools, and flow chart in /tools/backend
Fix for SCO compiles
Stratus computer port Robert Gillies
Added support for shlib for BSD44_derived & i386_solaris
Make configure more automated(Brook)
Add script to check regression test results
Break parser functions into smaller files, group together(Bruce)
Rename heap_create to heap_create_and_catalog, rename heap_creatr
       to heap_create()(Bruce)
Sparc/Linux patch for locking(TomS)
Remove PORTNAME and reorganize port-specific stuff(Marc)
Add optimizer README file(Bruce)
Remove some recursion in optimizer and clean up some code there(Bruce)
Fix for NetBSD locking(Henry)
Fix for libptcl make(Tatsuo)
AIX patch(Darren)
Change IS TRUE, IS FALSE, ... to expressions using "=" rather than
       function calls to istrue() or isfalse() to allow optimization(Thomas)
Various fixes NetBSD/Sparc related(TomH)
Alpha linux locking(Travis,Ryan)
Change elog(WARN) to elog(ERROR)(Bruce)
FAQ for FreeBSD(Marc)
Bring in the PostODBC source tree as part of our standard distribution(Marc)
A minor patch for HP/UX 10 vs 9(Stan)
New pg_attribute.atttypmod for type-specific info like varchar length(Bruce)
UnixWare patches(Billy)
New i386 'lock' for spinlock asm(Billy)
Support for multiplexed backends is removed
Start an OpenBSD port
Start an AUX port
Start a Cygnus port
Add string functions to regression suite(Thomas)
Expand a few function names formerly truncated to 16 characters(Thomas)
Remove un-needed malloc() calls and replace with palloc()(Bruce)
```

E.222. Release 6.2.1

Release Date

1997-10-17

6.2.1 is a bug-fix and usability release on 6.2.

Summary:

- Allow strings to span lines, per SQL92.
- Include example trigger function for inserting user names on table updates.

This is a minor bug-fix release on 6.2. For upgrades from pre-6.2 systems, a full dump/reload is required. Refer to the 6.2 release notes for instructions.

E.222.1. Migration from version 6.2 to version 6.2.1

This is a minor bug-fix release. A dump/reload is not required from version 6.2, but is required from any release prior to 6.2.

In upgrading from version 6.2, if you choose to dump/reload you will find that avg(money) is now calculated correctly. All other bug fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from **psql** to update the existing system table:

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including template1.

E.222.2. Changes

```
Allow TIME and TYPE column names(Thomas)
Allow larger range of true/false as boolean values(Thomas)
Support output of "now" and "current"(Thomas)
Handle DEFAULT with INSERT of NULL properly(Vadim)
Fix for relation reference counts problem in buffer manager(Vadim)
Allow strings to span lines, like ANSI(Thomas)
Fix for backward cursor with ORDER BY(Vadim)
Fix avg(cash) computation(Thomas)
Fix for specifying a column twice in ORDER/GROUP BY(Vadim)
Documented new libpq function to return affected rows, PQcmdTuples(Bruce)
Trigger function for inserting user names for INSERT/UPDATE(Brook Milligan)
```

E.223. Release 6.2

Release Date

1997-10-02

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL™.

E.223.1. Migration from version 6.1 to version 6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.

Note that the **pg_dump** and **pg_dumpall** utility from 6.2 should be used to dump the 6.1 database.

E.223.2. Migration from version 1.*x* to version 6.2

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.223.3. Changes

```
Bug Fixes
---------
```

```
Fix problems with pg_dump for allance, sequences, archive tables(Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad prototypes
        from Solaris(Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection calculations)(Thomas)
Check for geometric intersections at endpoints to avoid rounding ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bug which made rows changed/inserted by a command
       visible to the command itself (so we had multiple update of
       updated rows, etc.)(Vadim)
Fix for SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isnull didn't return rows with x NULL) (Vadim)
Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
       WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
       in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

Enhancements
------------
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of rows inserted/affected by INSERT/UPDATE/DELETE etc.(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows execution of queries inside
       C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and unions(Thomas)
Implement extended comments ("/* ... */") using exclusive states(Thomas)
Add "//" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim & Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
       and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s) (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)(Thomas)
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
Allow hh:mm:ss time entry for timespan/reltime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
       using an axis-crossing algorithm(Thomas)
Add routine to convert circle-box(Thomas)
Merge conflicting operators for different geometric data types(Thomas)
Replace distance operator "<===>" with "<->"(Thomas)
Replace "above" operator "!^" with ">^" and "below" operator "!|" with "<^"(Thomas)
Add routines for text trimming on both ends, substring, and string position(Thomas)
Added conversion routines circle(box) and poly(circle)(Thomas)
Allow internal sorts to be stored in memory rather than in files(Bruce & Vadim)
Allow functions and operators on internally-identical types to succeed(Bruce)
Speed up backend start-up after profiling analysis(Bruce)
Inline frequently called functions for performance(Bruce)
Reduce open() calls(Bruce)
psql:  Add PAGER for \h and \?,\C fix
Fix for psql pager when no tty(Bruce)
New entab utility(Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
```

```
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)
MOVE implementation (Vadim)

Source Tree Changes
-------------------
HP-UX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
MkLinux patches (Tatsuo Ishii)
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy example code(Thomas)
Add -o to psql manual page(Bruce)
Prevent relname unallocated string length from being copied into database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce & Thomas)
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg_attribute.attnvals to attdispersion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop_table"(Bruce)
Add "create_trigger", "drop_trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D'Arcy)
```

E.224. Release 6.1.1

Release Date

1997-07-22

E.224.1. Migration from version 6.1 to version 6.1.1

This is a minor bug-fix release. A dump/reload is not required from version 6.1, but is required from any release prior to 6.1. Refer to the release notes for 6.1 for more details.

E.224.2. Changes

```
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/objects(Bruce)
new psql \connect option allows changing usernames without changing databases
fix for initdb --debug option(Yoshihiko Ichikawa))
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
fix timezone daylight handling for some ports(Thomas, Bruce, Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's \d now case-insensitive(Bruce)
psql's backslash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using \g(Bruce)
```

```
major fix for endian handling of communication to server(Thomas, Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

E.225. Release 6.1

Release Date

1997-06-08

The regression tests have been adapted and extensively modified for the 6.1 release of PostgreSQL™.

Three new data types (datetime, timespan, and circle) have been added to the native set of PostgreSQL™ types. Points, boxes, paths, and polygons have had their output formats made consistent across the data types. The polygon output in misc.out has only been spot-checked for correctness relative to the original regression output.

PostgreSQL™ 6.1 introduces a new, alternate optimizer which uses *genetic* algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g. points and time intervals) and tests involving those types are explicitly bracketed with **set geqo to 'off'** and **reset geqo**.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current `./expected/*.out` files reflect this new interpretation, which might not be correct!

The float8 regression test fails on at least some platforms. This is due to differences in implementations of `pow()` and `exp()` and the signaling mechanisms used for overflow and underflow conditions.

The « random » results in the random test should cause the « random » test to be « failed », since the regression tests are evaluated using a simple diff. However, « random » does not seem to produce random results on my test machine (Linux/gcc/i686).

E.225.1. Migration to Version 6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1.

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.225.2. Changes

```
Bug Fixes
---------
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multitable join fix(Vadim)
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D'Arcy)
select SUM(x) can return NULL on no rows(Thomas)
internal optimizer, executor bug fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indexes (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
```

```
fix file manager memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted rows reincarnation caused by vacuum (Vadim)
fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify (Igor)

Enhancements
------------
attribute optimization statistics(Bruce)
much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debug code(Massimo)
massive changes to libpg++(Leo)
new GEQO optimizer speeds table multitable optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce,Thomas,Dan)
debug backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend(Martin)
multicolumn btree indexes(Vadim)
new SET var TO value command(Martin)
update transaction status on reads(Dan)
new locale settings for character types(Oleg)
new SEQUENCE serial number generator(Vadim)
GROUP BY function now possible(Vadim)
re-organize regression test(Thomas,Marc)
new optimizer operation weights(Vadim)
new psql \z grant/permit option(Marc)
new MONEY data type(D'Arcy,Thomas)
tcp socket communication speed improved(Vadim)
new VACUUM option for attribute statistics, and for certain columns (Vadim)
many geometric type improvements(Thomas,Keith)
additional regression tests(Thomas)
new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes
-------------------
new configuration script(Marc)
readline configuration option added(Marc)
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor Hoffleit)
removed Windows-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
```

```
warn about buggy flex(Bruce)
DG/UX, Ultrix, IRIX, AIX portability fixes
```

E.226. Release 6.0

Release Date

1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL™.

E.226.1. Migration from version 1.09 to version 6.0

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

E.226.2. Migration from pre-1.09 to version 6.0

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.226.3. Changes

```
Bug Fixes
---------
ALTER TABLE bug - running postgres process needs to re-read table definition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bug(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export()(Bruce)
Fix uninitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bug(Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl privileges
Memory leaks (hunt and destroy with tools like Purify(Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and increase speed
Pg_dump -d now handles NULL's properly(Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
psql \g filename now works(Bruce)
psql fixed problem with multiple statements on one line with multiple outputs
Removed duplicate system OIDs
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table exists(Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors(Bruce)
Submitting an empty query now returns empty status, not just " " query(Bruce)

Enhancements
------------
Add EXPLAIN manual page(Bruce)
Add UNIQUE index capability(Dan)
Add hostname/user level access control rather than just hostname and user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias table.column(Bruce)
Allow COPY from the frontend(Bryan)
Allow GROUP BY to use alias column name(Bruce)
```

```
Allow actual compression, not just reuse on the same page(Vadim)
Allow installation-configuration option to auto-add all local users(Bryan)
Allow libpq to distinguish between text value '' and null(Bruce)
Allow non-postgres users with createdb privs to destroydb's
Allow restriction on who can create C functions(Bryan)
Allow restriction on who can do backend COPY(Bryan)
Can shrink tables, pg_time and pg_log(Vadim & Erich)
Change debug level 2 to print queries only, changed debug heading layout(Bruce)
Change default decimal constant representation from float4 to float8(Bruce)
European date format now set when postmaster is started
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x+y) from z'
Gist now included in the distribution(Marc)
Idend authentication of local users(Bryan)
Implement BETWEEN qualifier(Bruce)
Implement IN qualifier(Bruce)
libpq has PQgetisnull()(Bruce)
libpq++ improvements
New options to initdb(Bryan)
Pg_dump allow dump of OIDs(Bruce)
Pg_dump create indexes after tables are loaded for speed(Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values(Bruce)
Prevent postmaster from being run as root
psql \h and \? is now readable(Bruce)
psql allow backslashed, semicolons anywhere on the line(Bruce)
psql changed command prompt for lines in query or in quotes(Bruce)
psql char(3) now displays as (bp)char in \d output(Bruce)
psql return code now more accurate(Bryan?)
psql updated help syntax(Bruce)
Re-visit and fix vacuum(Vadim)
Reduce size of regression diffs, remove timezone name difference(Bruce)
Remove compile-time parameters to enable binary distributions(Bryan)
Reverse meaning of HBA masks(Bryan)
Secure Authentication of local users(Bryan)
Speed up vacuum(Vadim)
Vacuum now had VERBOSE option(Bruce)

Source tree changes
-------------------
All functions now have prototypes that are compared against the calls
Allow asserts to be disabled easily from Makefile.global(Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines(Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable constructs
Major include file reorganization/reduction(Marc)
Make now stops on compile failure(Bryan)
Makefile restructuring(Bryan, Marc)
Merge bsdi_2_1 to bsdi(Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file(Marc, Bryan)
PG_VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
Reduced the number of #define's, centralized #define's
Remove duplicate OIDS in system tables(Dan)
Remove duplicate system catalog info or report mismatches(Dan)
Removed many os-specific #define's
Restructured object file generation/location(Bryan, Marc)
Restructured port-specific file locations(Bryan, Marc)
Unused/uninitialized variables corrected
```

E.227. Release 1.09

Release Date

1996-11-04

Sorry, we didn't keep track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

E.228. Release 1.02

> **Release Date**
>
> 1996-08-01

E.228.1. Migration from version 1.02 to version 1.02.1

Here is a new migration file for 1.02.1. It includes the 'copy' change and a script to convert old ASCII files.

> **Note**
>
> The following notes are for the benefit of users who want to migrate databases from Postgres95™ 1.01 and 1.02 to Postgres95™ 1.02.1.
>
> If you are starting afresh with Postgres95™ 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older Postgres95™ version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster
2. Add the new built-in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries attached at the end of the file. This can be done easily through **psql**. If your 1.01 or 1.02 database is named `testdb` and you have cut the commands from the end of this file and saved them in `addfunc.sql`:

   ```
   % psql testdb -f addfunc.sql
   ```

   Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

E.228.2. Dump/Reload Procedure

If you are trying to reload a pg_dump or text-mode, `copy tablename to stdout` generated with a previous version, you will need to run the attached **sed** script on the ASCII file before loading it into the database. The old format used '.' as end-of-data, while '\.' is now the end-of-data marker. Also, empty strings are now loaded in as '' rather than NULL. See the copy manual page for full details.

```
sed 's/^\.$/\\./g' <in_file >out_file
```

If you are loading an older binary copy or non-`stdout` copy, there is no end-of-data character, and hence no conversion necessary.

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

E.228.3. Changes

```
Source code maintenance and development
 * worldwide team of volunteers
```

```
 * the source tree now in CVS at ftp.ki.net

Enhancements
 * psql (and underlying libpq library) now has many more options for
   formatting output, including HTML
 * pg_dump now output the schema and/or the data, with many fixes to
   enhance completeness.
 * psql used in place of monitor in administration shell scripts.
   monitor to be deprecated in next release.
 * date/time functions enhanced
 * NULL insert/update/comparison fixed/enhanced
 * TCL/TK lib and shell fixed to work with both tck7.4/tk4.0 and tcl7.5/tk4.1

Bug Fixes (almost too numerous to mention)
 * indexes
 * storage management
 * check for NULL pointer before dereferencing
 * Makefile fixes

New Ports
 * added SolarisX86 port
 * added BSD/OS 2.1 port
 * added DG/UX port
```

E.229. Release 1.01

Release Date

1996-02-23

E.229.1. Migration from version 1.0 to version 1.01

The following notes are for the benefit of users who want to migrate databases from Postgres95™ 1.0 to Postgres95™ 1.01.

If you are starting afresh with Postgres95™ 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to Postgres95™ version 1.01 with databases created with Postgres95™ version 1.0, the following steps are required:

1. Set the definition of NAMEDATALEN in `src/Makefile.global` to 16 and OIDNAMELEN to 20.
2. Decide whether you want to use Host based authentication.
   a. If you do, you must create a file name `pg_hba` in your top-level data directory (typically the value of your `$PGDATA`). `src/libpq/pg_hba` shows an example syntax.
   b. If you do not want host-based authentication, you can comment out the line:

      ```
      HBA = 1
      ```

      in `src/Makefile.global`

      Note that host-based authentication is turned on by default, and if you do not take steps A or B above, the out-of-the-box 1.01 will not allow you to connect to 1.0 databases.
3. Compile and install 1.01, but DO NOT do the **initdb** step.
4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing `$PGDATA` directory.
5. Set your `PGDATA` environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.
6. Modify the file `$PGDATA/PG_VERSION` from 5.0 to 5.1
7. Start up a new 1.01 postmaster
8. Add the new built-in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file 1.0_to_1.01.sql. This can be done easily

through **psql**. If your 1.0 database is name `testdb`:

```
% psql testdb -f 1.0_to_1.01.sql
```

and then execute the following commands (cut and paste from here):

```
-- add builtin functions that are new to 1.01

create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure =
char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure =
char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);
```

E.229.2. Changes

```
Incompatibilities:
 * 1.01 is backwards compatible with 1.0 database provided the user
   follow the steps outlined in the MIGRATION_from_1.0_to_1.01 file.
   If those steps are not taken, 1.01 is not compatible with 1.0 database.

Enhancements:
 * added PQdisplayTuples() to libpq and changed monitor and psql to use it
 * added NeXT port (requires SysVIPC implementation)
 * added CAST .. AS ... syntax
 * added ASC and DESC key words
 * added 'internal' as a possible language for CREATE FUNCTION
```

```
   internal functions are C functions which have been statically linked
   into the postgres backend.
 * a new type "name" has been added for system identifiers (table names,
   attribute names, etc.)  This replaces the old char16 type.   The
   of name is set by the NAMEDATALEN #define in src/Makefile.global
 * a readable reference manual that describes the query language.
 * added host-based access control.  A configuration file ($PGDATA/pg_hba)
   is used to hold the configuration data.  If host-based access control
   is not desired, comment out HBA=1 in src/Makefile.global.
 * changed regex handling to be uniform use of Henry Spencer's regex code
   regardless of platform.  The regex code is included in the distribution
 * added functions and operators for case-insensitive regular expressions.
   The operators are ~* and !~*.
 * pg_dump uses COPY instead of SELECT loop for better performance

Bug fixes:
 * fixed an optimizer bug that was causing core dumps when
   functions calls were used in comparisons in the WHERE clause
 * changed all uses of getuid to geteuid so that effective uids are used
 * psql now returns non-zero status on errors when using -c
 * applied public patches 1-14
```

E.230. Release 1.0

Release Date

1995-09-05

E.230.1. Changes

```
Copyright change:
 * The copyright of Postgres™ 1.0 has been loosened to be freely modifiable
   and modifiable for any purpose.  Please read the COPYRIGHT file.
   Thanks to Professor Michael Stonebraker for making this possible.

Incompatibilities:
 *  date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're using
   EUROPEAN STYLE).  This follows SQL-92 specs.
 *  "delimiters" is now a key word

Enhancements:
 *  sql LIKE syntax has been added
 *  copy command now takes an optional USING DELIMITER specification.
   delimiters can be any single-character string.
 *  IRIX 5.3 port has been added.
   Thanks to Paul Walmsley and others.
 *  updated pg_dump to work with new libpq
 *  \d has been added psql
   Thanks to Keith Parks
 *  regexp performance for architectures that use POSIX regex has been
   improved due to caching of precompiled patterns.
   Thanks to Alistair Crooks
 *  a new version of libpq++
   Thanks to William Wanders

Bug fixes:
 *  arbitrary userids can be specified in the createuser script
 *  \c to connect to other databases in psql now works.
 *  bad pg_proc entry for float4inc() is fixed
 *  users with usecreatedb field set can now create databases without
   having to be usesuper
 *  remove access control entries when the entry no longer has any
   privileges
 *  fixed non-portable datetimes implementation
 *  added kerberos flags to the src/backend/Makefile
```

```
 *  libpq now works with kerberos
 *  typographic errors in the user manual have been corrected.
 *  btrees with multiple index never worked, now we tell you they don't
   work when you try to use them
```

E.231. Postgres95™ Release 0.03

Release Date

1995-07-21

E.231.1. Changes

```
Incompatible changes:
 * BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS VERSIONS
   (due to system catalog changes and indexing structure changes).
 * double-quote (") is deprecated as a quoting character for string literals;
   you need to convert them to single quotes (').
 * name of aggregates (eg. int4sum) are renamed in accordance with the
   SQL standard (eg. sum).
 * CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
 * float literals (eg. 3.14) are now of type float4 (instead of float8 in
   previous releases); you might have to do typecasting if you depend on it
   being of type float8.  If you neglect to do the typecasting and you assign
   a float literal to a field of type float8, you might get incorrect values
   stored!
 * LIBPQ has been totally revamped so that frontend applications
   can connect to multiple backends
 * the usesysid field in pg_user has been changed from int2 to int4 to
   allow wider range of Unix user ids.
 * the netbsd/freebsd/bsd o/s ports have been consolidated into a
   single BSD44_derived port.  (thanks to Alistair Crooks)

SQL standard-compliance (the following details changes that makes postgres95
more compliant to the SQL-92 standard):
 * the following SQL types are now built-in: smallint, int(eger), float, real,
   char(N), varchar(N), date and time.

   The following are aliases to existing postgres types:
                 smallint -> int2
                 integer, int -> int4
                 float, real  -> float4
   char(N) and varchar(N) are implemented as truncated text types. In
   addition, char(N) does blank-padding.
 * single-quote (') is used for quoting string literals; '' (in addition to
   \') is supported as means of inserting a single quote in a string
 * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used
   (Also, aggregates can now be overloaded, i.e. you can define your
   own MAX aggregate to take in a user-defined type.)
 * CHANGE ACL removed. GRANT/REVOKE syntax added.
   - Privileges can be given to a group using the "GROUP" key word.
        For example:
                 GRANT SELECT ON foobar TO GROUP my_group;
        The key word 'PUBLIC' is also supported to mean all users.

        Privileges can only be granted or revoked to one user or group
        at a time.

        "WITH GRANT OPTION" is not supported.  Only class owners can change
        access control
   - The default access control is to grant users readonly access.
     You must explicitly grant insert/update access to users.  To change
     this, modify the line in
                 src/backend/utils/acl.h
     that defines ACL_WORLD_DEFAULT
```

```
Bug fixes:
 * the bug where aggregates of empty tables were not run has been fixed. Now,
   aggregates run on empty tables will return the initial conditions of the
   aggregates. Thus, COUNT of an empty  table will now properly return 0.
   MAX/MIN of an empty table will return a row of value NULL.
 * allow the use of \; inside the monitor
 * the LISTEN/NOTIFY asynchronous notification mechanism now work
 * NOTIFY in rule action bodies now work
 * hash indexes work, and access methods in general should perform better.
   creation of large btree indexes should be much faster.  (thanks to Paul
   Aoki)

Other changes and enhancements:
 * addition of an EXPLAIN statement used for explaining the query execution
   plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the execution plan for
   the query).
 * WARN and NOTICE messages no longer have timestamps on them. To turn on
   timestamps of error messages, uncomment the line in
   src/backend/utils/elog.h:
        /* define ELOG_TIMESTAMPS */
 * On an access control violation, the message
        "Either no such class or insufficient privilege"
   will be given.  This is the same message that is returned when
   a class is not found.  This dissuades non-privileged users from
   guessing the existence of privileged classes.
 * some additional system catalog changes have been made that are not
   visible to the user.

libpgtcl changes:
 * The -oid option has been added to the "pg_result" tcl command.
   pg_result -oid returns oid of the last row inserted.   If the
   last command was not an INSERT, then pg_result -oid returns "".
 * the large object interface is available as pg_lo* tcl commands:
   pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:
 * flex/lex problems have been cleared up.  Now, you should be able to use
   flex instead of lex on any platforms.  We no longer make assumptions of
   what lexer you use based on the platform you use.
 * The Linux-ELF port is now supported.  Various configuration have been
   tested:  The following configuration is known to work:
        kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   with everything in ELF format,

New utilities:
 * ipcclean added to the distribution
   ipcclean usually does not need to be run, but if your backend crashes
   and leaves shared memory segments hanging around, ipcclean will
   clean them up for you.

New documentation:
 * the user manual has been revised and libpq documentation added.
```

E.232. Postgres95™ Release 0.02

Release Date

1995-05-25

E.232.1. Changes

```
Incompatible changes:
 * The SQL statement for creating a database is 'CREATE DATABASE' instead
   of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE' instead
```

```
   of 'DESTROYDB'. However, the names of the executables 'createdb' and
   'destroydb' remain the same.

New tools:
 * pgperl - a Perl (4.036) interface to Postgres95
 * pg_dump - a utility for dumping out a postgres database into a
        script file containing query commands. The script files are in a ASCII
        format and can be used to reconstruct the database, even on other
        machines and other architectures. (Also good for converting
        a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-beta-0.02:
 * the NetBSD port by Alistair Crooks
 * the AIX port by Mike Tung
 * the Windows NT port by Jon Forrest (more stuff but not done yet)
 * the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:
 * new lines not escaped in COPY OUT and problem with COPY OUT when first
   attribute is a '.'
 * cannot type return to use the default user id in createuser
 * SELECT DISTINCT on big tables crashes
 * Linux installation problems
 * monitor doesn't allow use of 'localhost' as PGHOST
 * psql core dumps when doing \c or \l
 * the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
 * libpgtcl has a hard-wired default port number
 * SELECT DISTINCT INTO TABLE hangs
 * CREATE TYPE doesn't accept 'variable' as the internallength
 * wrong result using more than 1 aggregate in a SELECT
```

E.233. Postgres95™ Release 0.01

Release Date

1995-05-01

Initial release.

E.234. Timing Results

These timing results are from running the regression test with the commands

```
% cd src/test/regress
% make all
% time make runtest
```

Timing under Linux 2.0.27 seems to have a roughly 5% variation from run to run, presumably due to the scheduling vagaries of multitasking systems.

E.234.1. Version 6.5

As has been the case for previous releases, timing between releases is not directly comparable since new regression tests have been added. In general, 6.5 is faster than previous releases.

Timing with `fsync()` disabled:

```
Time   System
02:00  Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc 2.7.2.3 -O2 -m486
04:38  Sparc Ultra 1 143MHz, 64MB, Solaris 2.6
```

Timing with `fsync()` enabled:

```
Time   System
```

```
04:21  Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc 2.7.2.3 -O2 -m486
```

For the `Linux` system above, using UW-SCSI disks rather than (older) IDE disks leads to a 50% improvement in speed on the regression test.

E.234.2. Version 6.4beta

The times for this release are not directly comparable to those for previous releases since some additional regression tests have been included. In general, however, 6.4 should be slightly faster than the previous release (thanks, Bruce!).

```
Time   System
02:26  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc 2.7.2.1 -O2 -m486
```

E.234.3. Version 6.3

The times for this release are not directly comparable to those for previous releases since some additional regression tests have been included and some obsolete tests involving time travel have been removed. In general, however, 6.3 is substantially faster than previous releases (thanks, Bruce!).

```
Time   System
02:30  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc 2.7.2.1 -O2 -m486
04:12  Dual Pentium Pro 180, 96MB, EIDE, Linux 2.0.30, gcc 2.7.2.1 -O2 -m486
```

E.234.4. Version 6.1

```
Time   System
06:12  Pentium Pro 180, 32MB, EIDE, Linux 2.0.30, gcc 2.7.2 -O2 -m486
12:06  P-100, 48MB, Linux 2.0.29, gcc
39:58  Sparc IPC 32MB, Solaris 2.5, gcc 2.7.2.1 -O -g
```

Annexe F. Modules supplémentaires fournis

Cette annexe contient des informations concernant les modules disponibles dans le répertoire `contrib` de la distribution PostgreSQL™. Ce sont des outils de portage, des outils d'analyse, des fonctionnalités supplémentaires qui ne font pas partie du système PostgreSQL de base, principalement parce qu'ils s'adressent à une audience limitée ou sont trop expérimentaux pour faire partie de la distribution de base. Cela ne concerne en rien leur utilité.

Lors de la construction à partir des sources de la distribution, ces modules ne sont pas construits automatiquement, sauf si vous utilisez la cible « world » (voir Étape 2). Ils peuvent être construits et installés en exécutant :

```
gmake
gmake install
```

dans le répertoire `contrib` d'un répertoire des sources configuré ; ou pour ne construire et installer qu'un seul module sélectionné, on exécute ces commandes dans le sous-répertoire du module. Beaucoup de ces modules ont des tests de régression qui peuvent être exécutés en lançant la commande :

```
gmake installcheck
```

une fois que le serveur PostgreSQL™ est démarré. (`gmake check` n'est pas supporté ; un serveur de bases de données opérationnel est nécessaire pour réaliser ces tests, et le module doit avoir été construit et installé pour être testé.)

Lorsqu'une version packagée de PostgreSQL™ est utilisée, ces modules sont typiquement disponibles dans un package séparé, comme par exemple `postgresql-contrib`.

Beaucoup de ces modules fournissent de nouvelles fonctions, de nouveaux opérateurs ou types utilisateurs. Pour pouvoir utiliser un de ces modules, après avoir installé le code, il faut enregistrer les nouveaux objets dans la base de données en exécutant les commandes SQL contenus dans le fichier `.sql` fourni par le module. Par exemple :

```
psql -d nom_base -f SHAREDIR/contrib/module.sql
```

Ici, *`SHAREDIR`* est le répertoire « share » de l'installation (`pg_config --sharedir` indique de quel répertoire il s'agit). Dans la plupart de cas, le script doit être exécuté par un super-utilisateur de la base de données.

Le fichier `.sql` doit être exécuté dans chaque base de données où le module doit être disponible. Il peut également être exécuté dans la base `template1` pour que le module soit automatiquement copié dans toute nouvelle base de données créée.

La première commande du fichier `.sql` peut être modifiée pour déterminer le schéma de la base où sont créés les objets. Par défaut, ils sont placés dans `public`.

Après une mise à jour majeure de PostgreSQL™, le script d'installation doit être réexécuté, même si les objets du module sont éventuellement créés par une sauvegarde de l'ancienne installation. Cela assure que toute nouvelle fonction est disponible et tout correction nécessaire appliquée.

F.1. adminpack

L'`adminpack` fournit un certain nombre de fonctions de support que pgAdmin ou d'autres outils de gestion et d'administration peuvent utiliser pour fournir des fonctionnalités supplémentaires, comme la gestion à distance de journaux applicatifs.

F.1.1. Fonctions codées

Les fonctions ajoutées par `adminpack` ne peuvent être exécutées que par un super-utilisateur. La liste des fonctions est la suivante :

```
int8 pg_catalog.pg_file_write(fname text, data text, append bool)
bool pg_catalog.pg_file_rename(oldname text, newname text, archivename text)
bool pg_catalog.pg_file_rename(oldname text, newname text)
bool pg_catalog.pg_file_unlink(fname text)
setof record pg_catalog.pg_logdir_ls()

/* Renommage des fonctions internes existantes pour une compatibilité
avec pgAdmin */
int8 pg_catalog.pg_file_read(fname text, data text, append bool)
bigint pg_catalog.pg_file_length(text)
int4 pg_catalog.pg_logfile_rotate()
```

F.2. auto_explain

Le module `auto_explain` fournit un moyen de tracer les plans d'exécution des requêtes lentes automatiquement, sans qu'il soit nécessaire de lancer EXPLAIN(7) manuellement. C'est particulièrement utile pour repérer les requêtes non optimisées sur de grosses applications.

Le module ne fournit pas de fonctions accessibles par SQL. Pour l'utiliser, il suffit de le charger sur le serveur. Il peut être chargé dans une session individuelle :

```
LOAD 'auto_explain';
```

(Seul le super-utilisateur peut le faire.) Un usage plus caractéristique est de le précharger dans toutes les sessions en incluant `auto_explain` dans shared_preload_libraries dans le fichier `postgresql.conf`. Il est alors possible de récupérer les requêtes lentes non prévues, quel que soit le moment où elles se produisent. Évidemment, il y a un prix à payer pour cela.

F.2.1. Paramètres de configuration

Plusieurs paramètres de configuration contrôlent le comportement d'`auto_explain`. Le comportement par défaut est de ne rien faire. Il est donc nécessaire de préciser au minimum `auto_explain.log_min_duration` pour obtenir un résultat.

`auto_explain.log_min_duration` (integer)
: `auto_explain.log_min_duration` est la durée minimale d'exécution de requête à partir de laquelle le plan d'exécution sera tracé. Son unité est la milliseconde. La positionner à zéro trace tous les plans. -1 (la valeur par défaut) désactive l'écriture des plans. Positionnée à `250ms`, tous les ordres qui durent 250 ms ou plus seront tracés. Seuls les super-utilisateurs peuvent modifier ce paramétrage.

`auto_explain.log_analyze` (boolean)
: `auto_explain.log_analyze` entraîne l'écriture du résultat de **EXPLAIN ANALYZE**, à la place du résultat de **EXPLAIN**, lorsqu'un plan d'exécution est tracé. Ce paramètre est désactivé par défaut. Seuls les super-utilisateurs peuvent modifier ce paramètre.

> **Note**
>
> Lorsque ce paramètre est activé, un chronométrage par nœud du plan est calculé pour tous les ordres exécutés, qu'ils durent suffisamment longtemps pour être réellement tracés, ou non. Ceci peut avoir des conséquences très négatives sur les performances.

`auto_explain.log_verbose` (boolean)
: `auto_explain.log_verbose` entraîne l'écriture du résultat de **EXPLAIN VERBOSE** au lieu du résultat de la commande simple **EXPLAIN**, lorsqu'un plan d'exécution est tracé. Ce paramètre est désactivé par défaut. Seuls les super-utilisateurs peuvent modifier ce paramètre.

`auto_explain.log_buffers` (boolean)
: `auto_explain.log_buffers` permet d'obtenir la même sortie qu'un **EXPLAIN (ANALYZE, BUFFERS)**, plutôt que la sortie **EXPLAIN** habituelle quand un plan d'exécution est tracé. Ce paramètre est désactivé par défaut. Seuls les superutilisateurs peuvent modifier cette configuration. Ce paramètre n'a pas d'effet sauf si le paramètre `auto_explain.log_analyze` est activé.

`auto_explain.log_format` (enum)
: `auto_explain.log_format` sélectionne le format de sortie du **EXPLAIN** à utiliser. Les valeurs autorisées sont `text`, `xml`, `json` et `yaml`. La valeur par défaut est text. Seuls les super-utilisateurs peuvent modifier ce paramètre.

`auto_explain.log_nested_statements` (boolean)
: `auto_explain.log_nested_statements` entraîne la prise en compte des ordres imbriqués (les requêtes exécutées dans une fonction) dans la trace. Quand il est désactivé, seuls les plans d'exécution de plus haut niveau sont tracés. Ce paramètre est désactivé par défaut. Seuls les super-utilisateurs peuvent modifier ce paramètre.

Pour positionner ces paramètres dans le fichier `postgresql.conf`, il convient d'ajouter `auto_explain` à custom_variable_classes. Un usage classique serait :

```
        # postgresql.conf
```

```
        shared_preload_libraries = 'auto_explain'

        custom_variable_classes = 'auto_explain'
        auto_explain.log_min_duration = '3s'
```

F.2.2. Exemple

```
                postgres=# LOAD 'auto_explain';
                postgres=# SET auto_explain.log_min_duration = 0;
                postgres=# SELECT count(*)
                FROM pg_class, pg_index
                WHERE oid = indrelid AND indisunique;
```

Ceci devrait produire un résultat de ce style dans les journaux applicatifs :

```
  LOG:  duration: 3.651 ms  plan:
    Query Text: SELECT count(*)
                FROM pg_class, pg_index
               WHERE oid = indrelid AND indisunique;
    Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1
loops=1)
      ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594
rows=92 loops=1)
            Hash Cond: (pg_class.oid = pg_index.indrelid)
            ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual
time=0.016..0.140 rows=255 loops=1)
            ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238
rows=92 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 4kB
                  ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual
time=0.008..3.187 rows=92 loops=1)
                        Filter: indisunique
```

F.2.3. Auteur

Takahiro Itagaki `<itagaki.takahiro@oss.ntt.co.jp>`

F.3. btree_gin

`btree_gin` fournit des échantillons de classes d'opérateurs GIN qui codent un comportement équivalent à un B-tree pour les types int2, int4, int8, float4, float8, timestamp with time zone, timestamp without time zone, time with time zone, time without time zone, date, interval, oid, money, "char", varchar, text, bytea, bit, varbit, macaddr, inet, et cidr.

En général, ces classes d'opérateurs ne sont pas plus rapides que les méthodes standard d'indexation B-tree équivalentes, et il leur manque une fonctionnalité majeure du code B-tree standard : la capacité à forcer l'unicité. Toutefois, elles sont utiles pour tester GIN et comme base pour développer d'autres classes d'opérateurs GIN. Par ailleurs, pour des requêtes qui testent à la fois une colonnne indexable via GIN et une colonne indexable par B-tree, il peut être plus efficace de créer un index GIN multicolonne qui utilise une de ces classes d'opérateurs que de créer deux index séparés qui devront être combinés via une opération de bitmap ET.

F.3.1. Exemple d'utilisation

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING gin (a);
-- requête
SELECT * FROM test WHERE a < 10;
```

F.3.2. Auteurs

Teodor Sigaev (<teodor@stack.net>) et Oleg Bartunov (<oleg@sai.msu.su>). Voir *http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin* pour plus d'informations.

F.4. btree_gist

`btree_gist` fournit des exemples de classes d'opérateur GiST qui codent un comportement équivalent à celui du B-tree pour les types de données int2, int4, int8, float4, float8, numeric, timestamp with time zone, timestamp without time zone, time with time zone, time without time zone, date, interval, oid, money, char, varchar, text, bytea, bit, varbit, macaddr, inet et cidr.

En règle général, ces classes d'opérateur ne dépassent pas en performance les méthodes d'indexage standard équivalentes du B-tree. Il leur manque une fonctionnalité majeure : la possibilité d'assurer l'unicité. Néanmoins, elles sont utiles pour des tests de GiST et comme base pour le développement d'autres classes d'opérateur GiST.

F.4.1. Exemple d'utilisation

```
CREATE TABLE test (a int4);
-- création de l'index
CREATE INDEX testidx ON test USING gist (a);
-- requête
SELECT * FROM test WHERE a < 10;
```

F.4.2. Auteurs

Teodor Sigaev (<teodor@stack.net>), Oleg Bartunov (<oleg@sai.msu.su>) et Janko Richter (<jankorichter@yahoo.de>). Voir le *site sur GiST* pour plus d'information.

F.5. chkpass

Ce module implante le type de données chkpass, conçu pour stocker des mots de passe chiffrés. Chaque mot de passe est automatiquement converti dans sa forme chiffrée et est stocké ainsi. Pour comparer un mot de passe, il suffit de comparer le champ avec un mot de passe en clair. La fonction de comparaison le chiffre automatiquement avant la comparaison.

Il existe des parties dans le code pour reporter une erreur si le mot de passe est facilement trouvable. Néanmoins, ce code ne fait rien actuellement.

Une chaîne précédée d'un symbole deux-points est supposée déjà chiffrée et se retrouve stockée sans autre traitement. Ceci permet la saisie des mots de passe précédemment chiffrés.

En sortie, le symbole deux-points est ajouté en préfixe. Cela rend possible la sauvegarde et la restauration des mots de passe sans les chiffrer de nouveau. Pour obtenir le mot de passe (chiffré) sans le symbole deux-points, on utilise la fonction `raw()`. Ceci permet d'utiliser le type avec d'autres choses comme, par exemple, le module `Auth_PostgreSQL` d'Apache.

Le chiffrement utilise la fonction `crypt()` du standard Unix. Il souffre donc des limitations habituelles de cette fonction, notamment le fait que seuls les huit premiers caractères d'un mot de passe sont pris en compte.

Le type de données chkpass n'est pas indexable.

Exemple d'utilisation :

```
test=# create table test (p chkpass);
CREATE TABLE
test=# insert into test values ('hello');
INSERT 0 1
test=# select * from test;
       p
----------------
 :dVGkpXdOrE3ko
(1 row)

test=# select raw(p) from test;
      raw
---------------
 dVGkpXdOrE3ko
(1 row)
```

```
test=# select p = 'hello' from test;
 ?column?
----------
 t
(1 row)

test=# select p = 'goodbye' from test;
 ?column?
----------
 f
(1 row)
```

F.5.1. Auteur

D'Arcy J.M. Cain `<darcy@druid.net>`

F.6. citext

Le module `citext` fournit un type chaîne de caratères insensible à la casse, citext. En réalité, il appelle en interne la fonction `lower` lorsqu'il compare des valeurs. Dans les autres cas, il se comporte presque exactement comme le type text.

F.6.1. Intérêt

L'approche standard pour effectuer des rapprochements insensibles à la casse avec PostgreSQL™ était d'utiliser la fonction `lower` pour comparer des valeurs. Par exemple :

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

Ceci fonctionne plutôt bien, mais présente quelques inconvénients :

- Cela rend les ordres SQL bavards, et vous devez sans arrêt vous souvenir d'utiliser la fonction `lower` à la fois sur la colonne et la valeur de la requête.
- Cela n'utilise pas les index, à moins que vous ne créiez un index fonctionnel avec la fonction `lower`.
- Si vous déclarez une colonne `UNIQUE` ou `PRIMARY KEY`, l'index généré implicitement est sensible à la casse. Il est donc inutile pour des recherches insensibles à la casse, et il ne va pas garantir l'unicité de manière insensible à la casse.

Le type de données citext vous permet d'éviter les appels à `lower` dans les requêtes SQL, et peut rendre une clé primaire insensible à la casse. citext tient compte de la locale, comme text, ce qui signifie que la comparaison entre caractères majuscules et minuscules dépend des règles de la locale paramétrée par `LC_CTYPE` . Ici également, le comportement est identique à l'utilisation de la fonction `lower` dans les requêtes. Mais comme cela est fait de manière transparente par le type de données, vous n'avez pas à vous souvenir de faire quelque chose de particulier dans vos requêtes.

F.6.2. Comment l'utiliser

Voici un exemple simple d'utilisation :

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  md5(random()::text) );
INSERT INTO users VALUES ( 'Tom',    md5(random()::text) );
INSERT INTO users VALUES ( 'Damian', md5(random()::text) );
INSERT INTO users VALUES ( 'NEAL',   md5(random()::text) );
INSERT INTO users VALUES ( 'Bjørn',  md5(random()::text) );

SELECT * FROM users WHERE nick = 'Larry';
```

L'ordre **SELECT** va renvoyer un enregistrement, bien que la colonne *`nick`* ait été positionnée à `larry` et que la requête soit

pour `Larry`.

F.6.3. Comportement des comparaisons de chaînes

Afin d'émuler un tri insensible à la casse de la manière la plus fidèle possible, il existe des versions spécifiques à citext de plusieurs opérateurs et fonctions de comparaison. Ainsi, par exemple, les opérateurs pour les expressions rationnelles `~` and `~*` ont le même comportement quand ils sont appliqués au type citext : ils comparent tous les deux de manière insensible à la casse. Cela est aussi vraie pour `!~` et `!~*`, et également pour les opérateurs `LIKE`, `~~`, `~~*`, et `!~~` et `!~~*`. Si vous voulez faire une comparaison sensible à la casse, vous pouvez toujours convertir dans un text avant de comparer.

De la même façon, toutes les fonctions ci-dessous font une comparaison insensible à la casse si leurs arguments sont de type citext :

- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

Pour les fonctions regexp, si vous voulez effectuer des comparaisons sensibles à la casse, vous pouvez positionner l'indicateur « c » pour forcer une comparaison sensible à la casse. Sinon, si vous souhaitez un comportement sensible à la casse, vous devez convertir dans un type text avant d'utiliser une de ces fonctions.

F.6.4. Limitations

- Le comportement de citext dépend du paramètre `LC_CTYPE` de votre base de données. Par conséquent, la manière dont il compare les valeurs est fixée lorsque initdb est exécuté pour créer le cluster. Il n'est pas réellement insensible à la casse dans les termes définis par le standard Unicode. En pratique, ce que cela signifie est que, tant que vous êtes satisfait de votre tri, vous devriez être satisfait des comparaisons de citext. Mais si vous avez des données stockées dans différentes langues dans votre base, des utilisateurs de certains langages pourraient trouver que les résultats de leurs requêtes sont inattendus si le tri est déterminé pour un autre langage.
- citext n'est pas aussi performant que text parce que les fonctions opérateurs et les fonctions de comparaison B-tree doivent faire des copies des données et les convertir en minuscules pour les comparaisons. C'est cependant légèrement plus efficace qu'utiliser `lower` pour obtenir des comparaisons insensibles à la casse.
- citext n'aide pas réellement dans un certain conctexte. Vos données doivent être comparées de manière sensible à la casse dans certains contextes, et de manière sensible à la casse dans d'autres contextes. La réponse habituelle à cette question est d'utiliser le type text et d'utiliser manuellement la fonction `lower` lorsque vous avez besoin d'une comparaison insensible à la casse ; ceci fonctionne très bien si vous avez besoin peu fréquemment de comparaisons insensibles à la casse. Si vous avez besoin de comparaisons insensibles à la casse la plupart du temps, pensez à stocker les données en citext et à convertir explicitement les colonnes en text quand vous voulez une comparaison sensible à la casse. Dans les deux situations, vous aurez besoin de deux index si vous voulez que les deux types de recherche soient rapides.
- Le schéma contenant les opérateurs citext doit être dans le `search_path` (généralement `public`) ; dans le cas contraire, une comparaison standard sur text, c'est-à-dire sensible à la casse, sera réalisée.

F.6.5. Auteur

David E. Wheeler `<david@kineticode.com>`

Inspiré par le module original citext par Donald Fraser.

F.7. cube

Ce module code le type de données cube pour représenter des cubes à plusieurs dimensions.

F.7.1. Syntaxe

Tableau F.1, « Représentations externes d'un cube » affiche les représentations externes valides pour le type cube. *x*, *y*, etc. dénotent des nombres flottants.

Tableau F.1. Représentations externes d'un cube

x	point uni-dimensionnel (ou interval unidimensionnel de longueur nulle)
(*x*)	Identique à ci-dessus
x1,*x2*,...,*xn*	Un point dans un espace à n dimensions, représenté en interne comme un cube de volume nul
(*x1*,*x2*,...,*xn*)	Identique à ci-dessus
(*x*),(*y*)	Interval uni-dimensionnel débutant à *x* et finissant à *y* ou vice-versa ; l'ordre n'importe pas
[(*x*),(*y*)]	Identique à ci-dessus
(*x1*,...,*xn*),(*y1*,...,*yn*)	Cube à n dimensions représenté par paires de coins diagonalement opposés
[(*x1*,...,*xn*),(*y1*,...,*yn*)]	Identique à ci-dessus

L'ordre de saisie des coins opposés d'un cube n'a aucune importance. Les fonctions cube s'occupent de la bascule nécessaire à l'obtention d'une représentation uniforme « bas gauche, haut droit ».

Les espaces sont ignorées, [(*x*),(*y*)] est donc identique à [(*x*), (*y*)].

F.7.2. Précision

Les valeurs sont enregistrées en interne sous la forme de nombres en virgule flottante. Cela signifie que les nombres avec plus de 16 chiffres significatifs sont tronqués.

F.7.3. Utilisation

Le module `cube` inclut une classe d'opérateur pour index GiST pour les valeurs de type cube. Les opérateurs supportés par la classe d'opérateur GiST sont montrés dans Tableau F.2, « Opérateurs GiST du type cube ».

Tableau F.2. Opérateurs GiST du type cube

Opérateur	Description
a = b	Les cubes a et b sont identiques.
a && b	Les cubes a et b se chevauchent.
a @> b	Le cube a contient le cube b.
a <@ b	Le cube a est contenu dans le cube b.

(Avant PostgreSQL 8.2, les opérateurs de contenance @> et <@ étaient appelés respectivement @ et ~. Ces noms sont toujours disponibles mais sont déclarés obsolètes et seront supprimés un jour. Les anciens noms sont inversés par rapport à la convention suivie par les types de données géométriques !)

Les opérateurs du standard B-tree sont aussi fournis, par exemple :

Opérateur	Description
[a, b] < [c, d]	Plus petit que
[a, b] > [c, d]	Plus grand que

Ces opérateurs n'ont vraiment de sens que pour le tri. Ces opérateurs comparent en premier (a) à (c) et, s'ils sont égaux, comparent (b) à (d). Cela fait un bon tri dans la plupart des cas, ce qui permet d'utiliser ORDER BY avec ce type.

Tableau F.3, « Fonctions cube » indique les fonctions disponibles.

Tableau F.3. Fonctions cube

`cube(float8) returns cube`	Crée un cube uni-dimensionnel de coordonnées identiques. `cube(1) == '(1)'`
`cube(float8, float8) returns cube`	Crée un cube uni-dimensionnel. `cube(1,2) == '(1),(2)'`
`cube(float8[]) returns cube`	Crée un cube de volume nul en utilisant les coordonnées définies par le tableau. `cube(ARRAY[1,2]) == '(1,2)'`
`cube(float8[], float8[]) returns cube`	Crée un cube avec les coordonnées haut droit et bas gauche définies par deux tableaux de flottants, obligatoirement de même taille. `cube('{1,2}'::float[], '{3,4}'::float[]) == '(1,2),(3,4)'`
`cube(cube, float8) returns cube`	Construit un nouveau cube en ajoutant une dimension à un cube existant avec les mêmes valeurs pour les deux parties de la nouvelle coordonnée. Ceci est utile pour construire des cubes pièce par pièce à partir de valeurs calculées. `cube('(1)',2) == '(1,2),(1,2)'`
`cube(cube, float8, float8) returns cube`	Crée un nouveau cube en ajoutant une dimension à un cube existant. Ceci est utile pour construire des cubes pièce par pièce à partir de valeurs calculées. `cube('(1,2)',3,4) == '(1,3),(2,4)'`
`cube_dim(cube) returns int`	Renvoie le nombre de dimensions du cube.
`cube_ll_coord(cube, int) returns double`	Renvoie la n-ième coordonnée pour le coin en bas à gauche d'un cube.
`cube_ur_coord(cube, int) returns double`	Renvoie la n-ième coordonnée pour le coin en haut à droite d'un cube.
`cube_is_point(cube) returns bool`	Renvoie true si un cube est aussi un point, c'est-à-dire si les deux coins de définition sont identiques.
`cube_distance(cube, cube) returns double`	Renvoie la distance entre deux cubes. Si les deux cubes sont des points, il s'agit de la fonction de distance habituelle.
`cube_subset(cube, int[]) returns cube`	Crée un nouveau cube à partir d'un cube existant en utilisant une liste d'index de dimension d'un tableau. Peut être utilisé pour trouver les coordonnées bas gauche et haut droit d'une dimension, par exemple : `cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2]) = '(3),(7)'`. Peut aussi être utilisé pour supprimer des dimensions, ou pour les réordonner, par exemple : `cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1]) = '(5, 3, 1, 1),(8, 7, 6, 6)'`.
`cube_union(cube, cube) returns cube`	Réalise l'union de deux cubes.
`cube_inter(cube, cube) returns cube`	Réalise l'intersection de deux cubes.
`cube_enlarge(cube c, double r, int n) returns cube`	Augmente la taille d'un cube suivant un rayon précisé dans au moins n dimensions. Si le rayon est négatif, la boîte est diminuée. C'est utile pour créer des boîtes limitantes autour d'un point dans le but de rechercher les points voisins. Toutes les dimensions définies sont modifiées de la valeur du rayon r. Les coordonnées bas gauche sont décrémentées de r et les coordonnées haut droit sont incrémentées de r. Si une coordonnée bas gauche est incrémentée au-delà de la valeur correspondante haut droit (ce qui ne peut arriver que lorsque r < 0), les deux coordonnées sont positionnées à leur moyenne. Si n est plus grand que le nombre de dimensions définies et que le cube est augmenté (r >= 0), alors 0 est utilisé comme base des coordonnées supplémentaires.

F.7.4. Par défaut

Le développeur pense que l'union :

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

n'est pas en contradiction avec le bon sens. Pas plus que l'intersection

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

Dans toutes les opérations binaires sur des boîtes de tailles différentes, l'auteur suppose que la plus petite est une projection cartésienne, c'est-à-dire qu'il y a des zéros à la place des coordonnées omises dans la représentation sous forme de chaîne. Les exemples ci-dessus sont équivalents à :

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

Le prédicat de contenance suivant utilise la syntaxe en points alors qu'en fait, le second argument est représenté en interne par une boîte. Cette syntaxe rend inutile la définition du type point et des fonctions pour les prédicats (boîte,point).

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.7.5. Notes

Pour des exemples d'utilisation, voir les tests de régression `sql/cube.sql`.

Pour éviter toute mauvaise utilisation, le nombre de dimensions des cubes est limité à 100. Cela se configure dans `cubedata.h`.

F.7.6. Crédits

Auteur d'origine : Gene Selkov, Jr. `<selkovjr@mcs.anl.gov>`, Mathematics and Computer Science Division, Argonne National Laboratory.

F.7.7. Note de l'auteur

Mes remerciements vont tout particulièrement au professeur Joe Hellerstein (*http://db.cs.berkeley.edu/jmh/*) qui a su extraire l'idée centrale de GiST (*http://gist.cs.berkeley.edu/*), et à son étudiant précédant, Andy Dong (*http://best.berkeley.edu/~adong/rtree/index.html*), pour son exemple rédigé dans Illustra. Mes remerciements vont également aux développeurs de PostgreSQL qui m'ont permis de créer mon propre monde et de pouvoir y vivre sans être dérangé. Toute ma gratitude aussi à Argonne Lab et au département américain de l'énergie pour les années de support dans mes recherches sur les bases de données.

Des modifications mineures ont été effectuées sur ce module par Bruno Wolff III `<bruno@wolff.to>` en août/septembre 2002. Elles incluent la modification de la précision (de simple à double) et l'ajout de quelques nouvelles fonctions.

Des mises à jour supplémentaires ont été réalisées par Joshua Reich `<josh@root.net>` en juillet 2006. Elles concernent l'ajout de `cube(float8[], float8[])` et le nettoyage du code pour utiliser le protocole d'appel V1 à la place de la forme V0 maintenant obsolète.

F.8. dblink

`dblink` est un module qui permet de se connecter à d'autres bases de données PostgreSQL™ depuis une session de base de données.

Nom

dblink_connect — ouvre une connexion persistante vers une base de données distante.

Synopsis

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

Description

`dblink_connect()` établit une connexion à une base de données PostgreSQL™ distante. Le serveur et la base de données à contacter sont identifiées par une chaine de connexion standard de la libpq. Il est possible d'affecter un nom à la connexion. Plusieurs connexions nommées peuvent être ouvertes en une seule fois, mais il ne peut y avoir qu'une seule connexion anonyme à la fois. Toute connexion est maintenue jusqu'à ce qu'elle soit close ou que la session de base de données soit terminée.

La chaîne de connexion peut aussi être le nom d'un serveur distant existant. Il est recommandé d'utiliser `postgresql_fdw_validator` lors de la définition d'un wrapper de données distantes correspondant. Voir l'exemple ci-dessous, ainsi que : CREATE FOREIGN DATA WRAPPER(7), CREATE SERVER(7), CREATE USER MAPPING(7)

Arguments

`conname`
: Le nom à utiliser pour la connexion ; en cas d'omission, une connexion sans nom est ouverte, qui remplace toute autre connexion sans nom.

`connstr`
: Chaîne de connexion au format standard de la libpq, par exemple `hostaddr=127.0.0.1 port=5432 dbname=mabase user=postgres password=monmotdepasse`. Pour les détails, voir `PQconnectdb` dans Section 31.1, « Fonctions de contrôle de connexion à la base de données ».

Valeur de retour

Renvoie le statut qui est toujours `OK` (puisque toute erreur amène la fonction à lever une erreur, sans retour).

Notes

Seuls les super-utilisateurs peuvent utiliser `dblink_connect` pour créer des connexions authentifiées sans mot de passe. Si des utilisateurs standard ont ce besoin, il faut utiliser la fonction `dblink_connect_u` à sa place.

Il est déconseillé de choisir des noms de connexion contenant des signes d'égalité car ils peuvent introduire des risques de confusion avec les chaînes de connexion dans les autres fonctions `dblink`.

Exemple

```
SELECT dblink_connect('dbname=postgres');
  dblink_connect
 ----------------
  OK
 (1 row)

 SELECT dblink_connect('myconn', 'dbname=postgres');
  dblink_connect
 ----------------
  OK
 (1 row)

 -- Fonctionnalité FOREIGN DATA WRAPPER
 -- Note : la connexion locale nécessite l'authentification password pour que
 --        cela fonctionne correctement
 --        Sinon, vous recevrez le message d'erreur suivant provenant de
 --        dblink_connect() :
 --       ----------------------------------------------------------------------
```

```
 --       ERROR:  password is required
 --       DETAIL:  Non-superuser cannot connect if the server does not request a
password.
 --       HINT:  Target server's authentication method must be changed.
 CREATE USER dblink_regression_test WITH PASSWORD 'secret';
 CREATE FOREIGN DATA WRAPPER postgresql VALIDATOR postgresql_fdw_validator;
 CREATE SERVER fdtest FOREIGN DATA WRAPPER postgresql OPTIONS (hostaddr '127.0.0.1',
dbname 'contrib_regression');

 CREATE USER MAPPING FOR dblink_regression_test SERVER fdtest OPTIONS (user
'dblink_regression_test', password 'secret');
 GRANT USAGE ON FOREIGN SERVER fdtest TO dblink_regression_test;
 GRANT SELECT ON TABLE foo TO dblink_regression_test;

 \set ORIGINAL_USER :USER
 \c - dblink_regression_test
 SELECT dblink_connect('myconn', 'fdtest');
  dblink_connect
 ----------------
  OK
 (1 row)

 SELECT * FROM dblink('myconn','SELECT * FROM foo') AS t(a int, b text, c text[]);
  a  | b |       c
 ----+---+---------------
   0 | a | {a0,b0,c0}
   1 | b | {a1,b1,c1}
   2 | c | {a2,b2,c2}
   3 | d | {a3,b3,c3}
   4 | e | {a4,b4,c4}
   5 | f | {a5,b5,c5}
   6 | g | {a6,b6,c6}
   7 | h | {a7,b7,c7}
   8 | i | {a8,b8,c8}
   9 | j | {a9,b9,c9}
  10 | k | {a10,b10,c10}
 (11 rows)

 \c - :ORIGINAL_USER
 REVOKE USAGE ON FOREIGN SERVER fdtest FROM dblink_regression_test;
 REVOKE SELECT ON TABLE foo FROM  dblink_regression_test;
 DROP USER MAPPING FOR dblink_regression_test SERVER fdtest;
 DROP USER dblink_regression_test;
 DROP SERVER fdtest;
 DROP FOREIGN DATA WRAPPER postgresql;
```

Nom

dblink_connect_u — ouvre une connexion distante à une base de données de façon non sécurisée.

Synopsis

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

Description

`dblink_connect_u()` est identique à `dblink_connect()`, à ceci près qu'elle permet à des utilisateurs non-privilégiés de se connecter par toute méthode d'authentification.

Si le serveur distant sélectionne une méthode d'authentification qui n'implique pas de mot de passe, une impersonnalisation et une escalade de droits peut survenir car la session semble émaner de l'utilisateur qui exécute le serveur PostgreSQL™ local. De plus, même si le serveur distant réclame un mot de passe, il est possible de fournir le mot de passe à partir de l'environnement du serveur, par exemple en utilisant un fichier `~/.pgpass` appartenant à l'utilisateur du serveur. Cela apporte un risque supplémentaire d'impersonnification, sans parler de la possibilité d'exposer un mot de passe sur un serveur distant qui ne mérite pas votre confiance. C'est pourquoi, `dblink_connect_u()` est installé initialement sans aucun droit pour `PUBLIC`, ce qui restreint son utilisation aux seuls super-utilisateurs. Dans certaines cas, le droit `EXECUTE` sur `dblink_connect_u()` peut être accordé à quelque utilisateur spécifique digne de confiance, mais cela doit se faire avec une extrême prudence. Il est aussi recommandé que tout fichier `~/.pgpass` appartenant à l'utilisateur du serveur ne contienne *pas* de joker dans le nom de l'hôte.

Pour plus de détails, voir `dblink_connect()`.

Nom

dblink_disconnect — ferme une connexion persistante vers une base de données distante.

Synopsis

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

Description

`dblink_disconnect()` ferme une connexion ouverte par `dblink_connect()`. La forme sans argument ferme une connexion non nommée.

Arguments

`conname`
Le nom de la connexion à fermer

Valeur de retour

Renvoie le statut qui est toujours `OK` (puisque toute erreur amène la fonction à lever une erreur, sans retour).

Exemple

```
SELECT dblink_disconnect();
  dblink_disconnect
 -------------------
  OK
 (1 row)

 SELECT dblink_disconnect('myconn');
  dblink_disconnect
 -------------------
  OK
 (1 row)
```

Nom

dblink — exécute une requête sur une base de données distante

Synopsis

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

Description

`dblink` exécute une requête (habituellement un **SELECT**, mais toute instruction SQL qui renvoie des lignes est valable) sur une base de données distante.

Si deux arguments text sont présents, le premier est d'abord considéré comme nom de connexion persistante ; si cette connexion est trouvée, la commande est exécutée sur cette connexion. Dans le cas contraire, le premier argument est considéré être une chaîne de connexion comme dans le cas de `dblink_connect`, et la connexion indiquée n'est conservée que pour la durée d'exécution de cette commande.

Arguments

`conname`
: Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`connstr`
: Une chaîne de connexion similaire à celle décrite précédemment pour `dblink_connect`.

`sql`
: L'instruction SQL à exécuter sur l'hôte distant, par exemple `select * from foo`.

`fail_on_error`
: Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la fonction ne retourne aucune ligne.

Valeur de retour

La fonction renvoie les lignes produites par la requête. Comme `dblink` peut être utilisée avec toute requête, elle est déclarée comme renvoyant le type record, plutôt que de préciser un ensemble particulier de colonnes. Cela signifie que l'ensemble des colonnes attendues doit être précisé dans la requête appelante -- sinon PostgreSQL™ ne sait pas quoi attendre. Voici un exemple :

```
SELECT *
    FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

La partie « alias » de la clause `FROM` doit spécifier les noms et types des colonnes retournés par la fonction. (La précision des noms des colonnes dans un alias est une syntaxe du standard SQL mais la précision des types des colonnes est une extension PostgreSQL™.) Cela permet au système de savoir comment étendre `*`, et à quoi correspond proname dans la clause `WHERE` avant de tenter l'exécution de la fonction. À l'exécution, une erreur est renvoyée si le nombre de colonnes du résultat effectif de la requête sur la base de données distante diffère de celui indiqué dans la clause `FROM`. Les noms de colonnes n'ont pas besoin de correspondre et `dblink` n'impose pas une correspondance exacte des types. L'opération réussit si les chaînes de données renvoyées sont valides pour le type déclaré dans la clause `FROM`.

Notes

`dblink` récupère l'intégralité des résultats de la requête avant de les renvoyer au système local. Si la requête doit renvoyer un grand nombre de lignes, il est préférable d'ouvrir un curseur avec `dblink_open` puis de récupérer un nombre gérable de lignes.

Il est souvent plus pratique de créer une vue pour utiliser `dblink` avec des requêtes prédéterminées. Cela permet de laisser la vue gérer le type de la colonne plutôt que d'avoir à le saisir pour chaque requête. Par exemple :

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

Exemple

```
SELECT * FROM dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
```

```
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

Nom

dblink_exec — exécute une commande sur une base de données distante

Synopsis

```
    dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
    dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
    dblink_exec(text sql [, bool fail_on_error]) returns text
```

Description

`dblink_exec` exécute une commande (c'est-à-dire toute instruction SQL qui ne renvoie pas de lignes) dans une base de données distante.

Quand deux arguments de type text sont fournis, le premier est d'abord considéré comme nom d'une connexion persistante ; si cette connexion est trouvée, la commande est exécutée sur cette connexion. Dans le cas contraire, le premier argument est traitée comme une chaîne de connexion pour `dblink_connect`, et la connexion indiquée n'est maintenue que pour la durée d'exécution de cette commande.

Arguments

`conname`
: Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`connstr`
: Une chaîne de connexion similaire à celle décrite précédemment pour `dblink_connect`.

`sql`
: La commande SQL à exécuter sur la base de données distante ; par exemple `INSERT INTO foo VALUES(0,'a','{"a0","b0","c0"}')`.

`fail_on_error`
: Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la valeur de retour de la fonction est positionné à `ERROR`.

Valeur de retour

Renvoie le statut de la commande ou `ERROR` en cas d'échec.

Exemple

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
   dblink_exec
```

```
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

Nom

dblink_open — ouvre un curseur sur une base de données distante

Synopsis

```
    dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
    dblink_open(text connname, text cursorname, text sql [, bool fail_on_error])
returns text
```

Description

`dblink_open()` ouvre un curseur sur une base de données distante. Le curseur peut ensuite être manipulé avec `dblink_fetch()` et `dblink_close()`.

Arguments

`conname`
: Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`cursorname`
: Nom à affecter au curseur.

`sql`
: L'instruction **SELECT** à exécuter sur l'hôte distant, par exemple `SELECT * FROM pg_class`.

`fail_on_error`
: Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la valeur de retour de la fonction est positionné à `ERROR`.

Valeur de retour

Renvoie le statut, soit `OK` soit `ERROR`.

Notes

Puisqu'un curseur ne peut persister qu'au sein d'une transaction, `dblink_open` lance un bloc de transaction explicite (**BEGIN**) côté distant, si le côté distant n'est pas déjà à l'intérieur d'une transaction. Cette transaction est refermée à l'exécution de l'instruction `dblink_close`. Si `dblink_exec` est utilisée pour modifier les données entre `dblink_open` et `dblink_close`, et qu'une erreur survient ou que `dblink_disconnect` est utilisé avant `dblink_close`, les modifications *sont perdues* car la transaction est annulée.

Exemple

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

Nom

dblink_fetch — renvoie des lignes à partir d'un curseur ouvert sur une base de données distante

Synopsis

```
    dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof
record
    dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error])
returns setof record
```

Description

`dblink_fetch` récupère des lignes à partir d'un curseur déjà ouvert par `dblink_open`.

Arguments

`conname`
: Nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`cursorname`
: Le nom du curseur à partir duquel récupérer les lignes.

`howmany`
: Nombre maximum de lignes à récupérer. Les *`howmany`* lignes suivantes sont récupérées, en commençant à la position actuelle du curseur, vers l'avant. Une fois le curseur arrivé à la fin, aucune ligne supplémentaire n'est renvoyée.

`fail_on_error`
: Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la fonction ne retourne aucune ligne.

Valeur de retour

La fonction renvoie les lignes récupérées à partir du curseur. Pour utiliser cette fonction, l'ensemble des colonnes attendues doit être spécifié, comme décrit précédemment pour `dblink`.

Notes

Si le nombre de colonnes de retour spécifiées dans la clause `FROM`, et le nombre réel de colonnes renvoyées par le curseur distant diffèrent, une erreur est remontée. Dans ce cas, le curseur distant est tout de même avancé du nombre de lignes indiqué, comme si l'erreur n'avait pas eu lieu. Il en est de même pour toute autre erreur survenant dans la requête locale après l'exécution du **FETCH** distant.

Exemple

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like
''bytea%''');
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
```

```
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

Nom

dblink_close — ferme un curseur sur une base de données distante

Synopsis

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

Description

`dblink_close` ferme un curseur précédemment ouvert avec `dblink_open`.

Arguments

`conname`
: Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`cursorname`
: Nom du curseur à fermer.

`fail_on_error`
: Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la valeur de retour est positionnée à `ERROR`.

Valeur de retour

Renvoie le statut, soit `OK` soit `ERROR`.

Notes

Si `dblink_open` a ouvert un bloc de transaction explicite, et que c'est le dernier curseur ouvert restant dans cette connexion, `dblink_close` exécute le **COMMIT** correspondant.

Exemple

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

Nom

dblink_get_connections — renvoie les noms de toutes les connexions nommées ouvertes

Synopsis

```
dblink_get_connections() returns text[]
```

Description

`dblink_get_connections` renvoie un tableau contenant le nom de toutes les connexions nommées ouvertes de `dblink`.

Valeur de retour

Renvoie un tableau texte des noms des connexions, ou NULL s'il n'y en a pas.

Exemple

```
SELECT dblink_get_connections();
```

Nom

dblink_error_message — récupère le dernier message d'erreur sur la connexion nommée

Synopsis

```
dblink_error_message(text connname) returns text
```

Description

`dblink_error_message` récupère le dernier message d'erreur sur une connexion donnée.

Arguments

`conname`
Nom de la connexion à utiliser.

Return Value

Renvoie le dernier message, ou une chaîne vide s'il n'y a pas eu d'erreur sur cette connexion.

Exemple

```
SELECT dblink_error_message('dtest1');
```

Nom

dblink_send_query — envoie une requête asynchrone à une base de données distante

Synopsis

```
dblink_send_query(text connname, text sql) returns int
```

Description

`dblink_send_query` envoie une requête à exécuter de façon asynchrone, c'est-à-dire sans attendre immédiatement le résultat. Il ne doit pas déjà exister de requête asynchrone en exécution sur la connexion.

Après l'envoi réussi d'une requête asynchrone, le statut de fin d'exécution de la requête se vérifie avec `dblink_is_busy`, et les résultats sont finalement récupérés avec `dblink_get_result`. Il est aussi possible de tenter l'annulation d'une rquête asynchrone active en utilisant `dblink_cancel_query`.

Arguments

`conname`
: Le nom de la connexion à utiliser.

`sql`
: L'instruction SQL à exécuter dans la base de données distante, par exemple `select * from pg_class`.

Valeur de retour

Renvoie 1 si la requête a été envoyée avec succès, 0 sinon.

Exemple

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

Nom

dblink_is_busy — vérifie si la connexion est occupée par le traitement d'une requête asynchrone

Synopsis

```
dblink_is_busy(text connname) returns int
```

Description

`dblink_is_busy` teste si une requête asynchrone est en cours d'exécution.

Arguments

`conname`
Le nom de la connexion à vérifier.

Valeur de retour

Renvoie 1 si la connexion est occupée, 0 dans le cas contraire. Si cette fonction renvoie 0, il est garanti que l'appel à `dblink_get_result` ne bloque pas.

Exemple

```
SELECT dblink_is_busy('dtest1');
```

Nom

dblink_get_notify — récupère les notifications asynchrones sur une connection

Synopsis

```
    dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
    dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra
text)
```

Description

`dblink_get_notify` récupère les notifications soit sur une connexion anonyme (sans nom), soit sur une connexion nommée si le nom est précisé. Pour recevoir des notifications via dblink, `LISTEN` doit d'abord être lancé en utilisant `dblink_exec`. Pour les détails, voir LISTEN(7) et NOTIFY(7).

Arguments

`conname`
Le nom d'une connexion nommée qui veut récupérer les notifications.

Valeur de retour

Renvoit setof (notify_name text, be_pid int, extra text) ou un ensemble vide.

Exemple

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

Nom

dblink_get_result — récupère le résultat d'une requête asynchrone

Synopsis

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

Description

`dblink_get_result` récupère le résultat d'une requête asynchrone précédemment envoyée avec `dblink_send_query`. Si la requête n'est pas terminée, `dblink_get_result` en attend la fin.

Arguments

`conname`
: Le nom de la connexion à utiliser.

`fail_on_error`
: Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la fonction ne retourne aucune ligne.

Valeur de retour

Pour une requête asynchrone (c'est-à-dire une instruction SQL renvoyant des lignes), la fonction renvoie les lignes produites par la requête. Pour utiliser cette fonction, il faut spécifier l'ensemble des colonnes attendues, comme indiqué pour `dblink`.

Pour une commande asynchrone (c'est-à-dire une instruction SQL ne renvoyant aucune ligne), la fonction renvoie une seule ligne avec une colonne texte contenant la chaîne de statut de la commande. Il est impératif d'indiquer dans la clause `FROM` appelante que le résultat est constitué d'une unique colonne texte .

Notes

Cette fonction *doit* être appelée si `dblink_send_query` a renvoyé 1. Elle doit l'être une fois pour chaque requête envoyée, et une fois de plus pour obtenir un ensemble vide, avant de pouvoir utiliser à nouveau la connexion.

Exemple

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS
t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
```

```
f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3;
select * from foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)
```

Nom

dblink_cancel_query — annule toute requête en cours d'exécution sur la connexion nommée

Synopsis

```
dblink_cancel_query(text connname) returns text
```

Description

`dblink_cancel_query` tente d'annuler toute requête en cours d'exécution sur la connexion nommée. La réussite de la fonction n'est pas assurée (la requête distante pourrait, par exemple, être déjà terminée). Une demande d'annulation augmente simplement la probabilité que la requête échoue rapidement. Le protocoel de requête normal doit toujours être terminé, par exemple en appelant `dblink_get_result`.

Arguments

`conname`
: Le nom de la connexion à utiliser.

Valeur de retour

Renvoie `OK` si la demande d'annulation a été envoyée, ou le texte d'un message d'erreur en cas d'échec.

Exemple

```
SELECT dblink_cancel_query('dtest1');
```

Nom

dblink_get_pkey — renvoie la position et le nom des champs de clé primaire d'une relation

Synopsis

```
    dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

Description

`dblink_get_pkey` fournit des informations sur la clé primaire d'une relation de la base de données locale. Il est parfois utile de produire des requêtes à transmettre à des bases distantes.

Arguments

`relname`
: Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom a une casse mixte, ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

Valeur de retour

Renvoie une ligne pour chaque champ de clé primaire, ou aucune ligne si la relation n'a pas de clé primaire. Le type de ligne résultante est défini ainsi :

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

La colonne `position` comme à 1 et va jusqu'à *`N`* ; elle correspond au numéro du champ dans la clé primaire, pas au numéro de colonne dans la liste des colonnes de la table.

Exemple

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

Nom

dblink_build_sql_insert — construit une instruction d'insertion en utilisant un tuple local, remplaçant les valeurs des champs de la clé primaire avec les valeurs fournies

Synopsis

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Description

`dblink_build_sql_insert` peut être utile pour réaliser une réplication sélective d'une table locale vers une base distante. Elle sélectionne une ligne de la table locale sur la base de la clé primaire et construit une commande SQL **INSERT** qui duplique cette ligne, mais avec pour valeurs de clé primaire celles du dernier argument. (Pour réaliser une copie exacte de la ligne, il suffit d'indiquer les mêmes valeurs pour les deux derniers arguments.)

Arguments

`relname`
: Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom est en casse mixte ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

`primary_key_attnums`
: Les numéros des attributs (commençant à 1) des champs de la clé primaire, par exemple `1 2`.

`num_primary_key_atts`
: Le nombre de champs de la clé primaire.

`src_pk_att_vals_array`
: Les valeurs des champs de la clé primaire à utiliser pour identifier le tuple local. Chaque champ est représenté dans sa forme textuelle. Une erreur est renvoyée s'il n'y a pas de lignes locales avec ces valeurs de clé primaire.

`tgt_pk_att_vals_array`
: Les valeurs des champs de la clé primaire à placer dans la commande **INSERT** résultante. Chaque champ est représenté dans sa forme textuelle.

Valeur de retour

Renvoie l'instruction SQL demandée en tant que texte.

Notes

À partir de PostgreSQL™ 9.0, les numéros des attributs dans *`primary_key_attnums`* sont interprétés comme des numéros logiques de colonnes correspondant à la position de la colonne dans `SELECT * FROM relation`. Les versions précédentes interprétaient les numéros comme des positions physiques de colonnes. Une différence existe si une des colonnes à gauche de la colonne indiquée a été supprimé de la table.

Exemple

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
             dblink_build_sql_insert
----------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

Nom

dblink_build_sql_delete — construit une instruction de suppression en utilisant les valeurs fournies pour les champs de la clé primaire

Synopsis

```
    dblink_build_sql_delete(text relname,
                            int2vector primary_key_attnums,
                            integer num_primary_key_atts,
                            text[] tgt_pk_att_vals_array) returns text
```

Description

`dblink_build_sql_delete` peut être utile pour réaliser une réplication sélective d'une table locale vers une base distante. Elle construit une commande SQL **DELETE** qui supprime la ligne avec les valeurs indiquées de clé primaire.

Arguments

`relname`
: Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom est en casse mixte ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

`primary_key_attnums`
: Les numéros des attributs (commençant à 1) des champs de la clé primaire, par exemple `1 2`.

`num_primary_key_atts`
: Le nombre de champs de la clé primaire.

`tgt_pk_att_vals_array`
: Les valeurs de champs de la clé primaire, à utiliser dans la commande **DELETE** résultante. Chaque champ est représenté dans sa forme textuelle.

Valeur de retour

Renvoie l'instruction SQL demandée en tant que texte.

Notes

À partir de PostgreSQL™ 9.0, les numéros des attributs dans *`primary_key_attnums`* sont interprétés comme des numéros logiques de colonnes correspondant à la position de la colonne dans `SELECT * FROM relation`. Les versions précédentes interprétaient les numéros comme des positions physiques de colonnes. Une différence existe si une des colonnes à gauche de la colonne indiquée a été supprimé de la table.

Exemple

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
--------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

Nom

dblink_build_sql_update — construit une instruction de mise à jour à partir d'un tuple local, en remplaçant les valeurs des champs de la clé primaire par celles fournies

Synopsis

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Description

`dblink_build_sql_update` peut être utile pour réaliser une réplication sélective d'une table locale vers une base de donnée distante. Elle sélectionne une ligne à partir de la table locale en se basant sur la clé primaire, puis construit une commande SQL **UPDATE** qui duplique cette ligne, mais avec pour valeurs de clé primaire celles du dernier argument. (Pour faire une copie exacte de la ligne, on indique les mêmes valeurs pour les deux derniers arguments.) La commande **UPDATE** affecte toujours tous les champs de la ligne -- la différence principale entre cette instruction et `dblink_build_sql_insert` est l'hypothèse de l'existence de la ligne cible dans la table distante.

Arguments

`relname`
: Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom est en casse mixte ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

`primary_key_attnums`
: Les numéros des attributs (commençant à 1) des champs de la clé primaire, par exemple `1 2`.

`num_primary_key_atts`
: Le nombre de champs de la clé primaire.

`src_pk_att_vals_array`
: Les valeurs des champs de la clé primaire à utiliser pour identifier le tuple local. Chaque champ est représenté dans sa forme textuelle. Une erreur est renvoyée s'il n'y a pas de lignes locales avec ces valeurs de clé primaire.

`tgt_pk_att_vals_array`
: Les valeurs des champs de la clé primaire à placer dans la commande **UPDATE** résultante. Chaque champ est représenté dans sa forme textuelle.

Valeur de retour

Renvoie l'instruction SQL demandée en tant que texte.

Notes

À partir de PostgreSQL™ 9.0, les numéros des attributs dans *`primary_key_attnums`* sont interprétés comme des numéros logiques de colonnes correspondant à la position de la colonne dans `SELECT * FROM relation`. Les versions précédentes interprétaient les numéros comme des positions physiques de colonnes. Une différence existe si une des colonnes à gauche de la colonne indiquée a été supprimé de la table.

Exemple

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-----------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.9. dict_int

`dict_int` est un exemple de modèle de dictionnaire pour la recherche plein texte. La création de ce dictionnaire à été motivée par la volonté de pouvoir contrôler l'indexage d'entiers (signés et non signés), pour permettre à de tels nombres d'être indexés sans grossissement excessif du nombre de mots uniques, ce qui affecte grandement la performance de la recherche.

F.9.1. Configuration

Le dictionnaire accepte deux options :

- le paramètre `maxlen` indique le nombre maximum de chiffres autorisés dans un mot de type entier. La valeur par défaut est 6 ;
- Le paramètre `rejectlong` précise si un entier trop long doit être tronqué ou ignoré. Si `rejectlong` vaut `false` (valeur par défaut), le dictionnaire renvoie les `maxlen` premiers chiffres de l'entier. Si `rejectlong` vaut `true`, le dictionnaire traite l'entier comme un terme courant, l'entier n'est donc pas indexé. Cela signifie aussi qu'un tel nombre ne peut pas être recherché.

F.9.2. Utilisation

L'exécution du script d'installation crée un modèle de recherche plein texte `intdict_template` et un dictionnaire `intdict` basé sur ce dernier avec les paramètres par défaut. Les paramètres peuvent être modifiés, par exemple :

```
mabase ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

ou de nouveaux dictionnaires basés sur le modèle créés.

Pour tester le dictionnaire :

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

mais une utilisation réelle nécessite de l'inclure dans une configuration de recherche plein texte comme celle décrite dans Chapitre 12, Recherche plein texte. Cela peut ressembler à ceci :

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.10. dict_xsyn

Le module `dict_xsyn` (*Extended Synonym Dictionary*, dictionnaire étendu de synonymes) est un exemple de modèle de dictionnaire pour la recherche plein texte. Ce type de dictionnaire remplace des mots avec un ensemble de synonymes, ce qui rend possible la recherche d'un mot en utilisant un de ses synonymes.

F.10.1. Configuration

Un dictionnaire `dict_xsyn` accepte les options suivantes :

- `matchorig` contrôle si le mot original est accepté par le dictionnaire. Par défaut à `true`.
- `matchsynonyms` contrôle si les synonymes sont acceptés par le dictionnaire. Par défaut à `false`.
- `keeporig` contrôle si le mot original est inclus dans la sortie du dictionnaire. Par défaut à `true`.
- `keepsynonyms` contrôle si les synonymes sont inclus dans la sortie du dictionnaire. Par défaut à `true`.
- `rules` est le nom du fichier contenant la liste des synonymes. Ce fichier doit être stocké dans `$SHAREDIR/tsearch_data/` (où `$SHAREDIR` est le répertoire des données partagées de la distribution PostgreSQL™). Son nom doit se terminer par `.rules` (cette extension n'est pas à inclure dans le paramètre `rules`).

Le fichier rules a le format suivant :

- chaque ligne représente un groupe de synonymes pour un mot simple, donné en premier sur la ligne. Les synonymes sont séparés par une espace :

  ```
  mot syn1 syn2 syn3
  ```

- le signe dièse (#) est un délimiteur de commentaires. Il peut apparaître dans la ligne. Le reste de la ligne est ignoré.

Un exemple est donné dans `xsyn_sample.rules` qui est installé dans `$SHAREDIR/tsearch_data/`.

F.10.2. Utilisation

L'exécution du script d'installation crée un modèle `xsyn_template` de recherche plein texte et un dictionnaire `xsyn` basé sur le modèle, avec des paramètres par défaut. Il est possible de modifier les paramètres, par exemple :

```
ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

ou de créer de nouveaux dictionnaires basés sur le modèle.

Pour tester le dictionnaire :

```
ma_base=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

ma_base=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false,
MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

ma_base=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true,
MATCHORIG=false, KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

ma_base=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

Une utilisation réelle implique son ajout dans une configuration de recherche plein texte comme décrit dans Chapitre 12, Recherche plein texte. Cela pourrait ressembler à ceci :

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.11. earthdistance

Le module `earthdistance` fournit deux approches différentes pour calculer de grandes distances circulaires à la surface de la Terre. La première dépend du package `cube` (qui *doit* être installé pour que le module `earthdistance` puisse l'être aussi). La seconde est basée sur le type de données interne point et utilise longitude et latitude pour les coordonnées.

Dans ce module, la Terre est supposée parfaitement sphérique (si cette hypothèse n'est pas acceptable, le projet *PostGIS* doit être considéré.)

F.11.1. Distances sur Terre à partir de cubes

Les données sont stockées dans des cubes qui sont des points (les coins sont identiques), les trois coordonnées représentant la distance x, y et z au centre de la Terre. Un domaine earth sur cube est fourni. Il inclut des contraintes de vérification pour que la valeur respecte ces restrictions et reste raisonnablement proche de la surface réelle de la Terre.

Le rayon de la Terre, obtenu à partir de la fonction `earth()`, est donné en mètres. Il est toutefois possible de modifier le module pour changer l'unité, ou pour utiliser une autre valeur de rayon.

Ce paquet peut être appliqué aux bases de données d'astronomie. Les astronomes peuvent modifier `earth()` pour que le rayon renvoyé soit `180/pi()`, de sorte que les distances soient en degrés.

Les fonctions acceptent latitude et longitude en entrée et en sortie (en degrés), calculent la distance circulaire entre deux points et permettent de préciser facilement une boîte utilisable par les recherches par index.

Les fonctions suivantes sont fournies :

Tableau F.4. Fonctions earthdistance par cubes

Fonction	Retour	Description
`earth()`	float8	Renvoie le rayon estimé de la Terre.
`sec_to_gc(float8)`	float8	Convertit la distance en ligne droite (sécant) entre deux points à la surface de la Terre en distane circulaire.
`gc_to_sec(float8)`	float8	Convertit la distance circulaire entre deux points à la surface de la Terre en une distance en ligne droite (sécant).
`ll_to_earth(float8, float8)`	earth	Renvoie l'emplacement d'un point à la surface de la Terre étant données sa latitude (argument 1) et sa longitude (argument 2) en degrés.
`latitude(earth)`	float8	Renvoie la latitude en degrés d'un point à la surface de la Terre.
`longitude(earth)`	float8	Renvoie la longitude en degrés d'un point à la surface de la Terre.
`earth_distance(earth, earth)`	float8	Renvoie la distance circulaire entre deux points à la surface de la Terre.
`earth_box(earth, float8)`	cube	Renvoie une boîte autorisant une recherche par index avec l'opérateur `@>` du type cube pour les points situés au maximum à une distance circulaire donnée d'un emplacement. Certains points de cette boîte sont plus éloignés que la distance circulaire indiquée. Une deuxième vérification utilisant `earth_distance` doit, donc, être incluse dans la requête.

F.11.2. Distances sur Terre à partir de points

La seconde partie du module se fonde sur la représentation des emplacements sur Terre comme valeurs de type point, pour lesquelles le premier composant représente la longitude en degrés, et le second la latitude en degrés. Les points ont la forme (longitude, latitude) et non l'inverse, car intuitivement, la longitude se compare à l'axe X, la latitude à l'axe Y.

Un opérateur unique est fourni :

Tableau F.5. Opérateurs earthdistance par points

Opérateur	Retour	Description
point <@> point	float8	Donne la distance en miles entre deux points à la surface de la Terre.

Contrairement à la partie fondée sur cube, les unités ne sont pas modifiables : une modification de la fonction `earth()` n'affecte pas les résultats de l'opérateur.

La représentation longitude/latitude a pour inconvénient d'obliger à tenir compte des conditions particulières près des pôles et près des longitudes de +/- 180 degrés. La représentation par cube évite ces discontinuités.

F.12. fuzzystrmatch

Le module `fuzzystrmatch` fournit diverses fonctions qui permettent de déterminer les similarités et la distance entre des chaînes.

> **Attention**
>
> Actuellement, `fuzzystrmatch` ne fonctionne pas bien avec les encodages multi-octets (comme l'UTF-8).

F.12.1. Soundex

Le système Soundex est une méthode qui permet d'associer des noms (ou des mots) dont la prononciation est proche en les convertissant dans le même code. Elle a été utilisée à l'origine par le « United States Census » en 1880, 1900 et 1910. Soundex n'est pas très utile pour les noms qui ne sont pas anglais.

Le module `fuzzystrmatch` fournit deux fonctions pour travailler avec des codes Soundex :

```
   soundex(text) returns text
   difference(text, text) returns int
```

La fonction `soundex` convertit une chaîne en son code Soundex. La fonction `difference` convertit deux chaînes en leur codes Soundex, puis rapporte le nombre de positions de code correspondant. Comme les codes Soundex ont quatre caractères, le résultat va de zéro à quatre. Zéro correspond à aucune correspondance, quatre à une correspondance exacte. (Du coup, la fonction est mal nommée -- `similarity` aurait été un meilleur nom.)

Voici quelques exemples d'utilisation :

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.12.2. Levenshtein

Cette fonction calcule la distance de Levenshtein entre deux chaînes :

```
   levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost)
returns int
   levenshtein(text source, text target) returns int
```

La `source` et la cible (`target`) sont des chaînes quelconques non NULL de 255 bytes. Les paramètres de coût indiquent respectivement le coût d'une insertion, suppression ou substitution d'un paramètre. Vous pouvez omettre les paramètres de coût, comme dans la deuxième version de la version. Dans ce cas, elles ont 1 comme valeur par défaut.

Exemples :

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)
```

F.12.3. Metaphone

Metaphone, comme Soundex, construit un code représentatif de la chaîne en entrée. Deux chaînes sont considérées similaires si elles ont le même code.

Cette fonction calcule le code metaphone d'une chaîne en entrée :

```
   metaphone(text source, int max_output_length) returns text
```

`source` doit être une chaîne non NULL de 255 caractères au maximum. `max_output_length` fixe la longueur maximale du code metaphone résultant ; s'il est plus long, la sortie est tronquée à cette taille.

Exemple

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.12.4. Double Metaphone

Le système « Double Metaphone » calcule deux chaînes « qui se ressemblent » pour une chaîne en entrée -- une « primaire » et une « alternative ». Dans la plupart des cas, elles sont identiques mais, tout spécialement pour les noms autres qu'anglais, elles peuvent être légèrement différentes, selon la prononciation. Ces fonctions calculent le code primaire et le code alternatif :

```
   dmetaphone(text source) returns text
   dmetaphone_alt(text source) returns text
```

Il n'y a pas de limite de longueur sur les chaînes en entrée.

Exemple :

```
test=# select dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.13. hstore

Ce module code le type de données hstore pour stocker des ensembles de paires clé/valeur à l'intérieur d'une simple valeur PostgreSQL™. Cela peut s'avérer utile dans divers cas, comme les lignes à attributs multiples rarement examinées ou les données semi-structurées. Les clés et les valeurs sont de simples chaînes de texte.

F.13.1. Représentation externe de hstore

La représentation textuelle d'une valeur hstore, utilisée en entrée et en sortie, inclut zéro ou plusieurs paires *clé* => *valeur* séparées par des virgules. Par exemple :

```
    k => v
    foo => bar, baz => whatever
    "1-a" => "anything at all"
```

L'ordre des paires n'est pas significatif (et pourrait ne pas être reproduit en sortie). Les espaces blancs entre les paires ou autour des signes => sont ignorés. Les clés et valeurs entre guillemets peuvent inclure des espaces blancs, virgules, = ou >. Pour inclure un guillemet double ou un antislash dans une clé ou une valeur, échappez-le avec un antislash.

Chaque clé dans un hstore est unique. Si vous déclarez un hstore avec des clés dupliquées, seule une sera stockée dans hstore et il n'y a pas de garantie sur celle qui sera conservée :

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

Une valeur, mais pas une clé, peut être un NULL SQL. Par exemple :

```
    key => NULL
```

Le mot-clé NULL est insensible à la casse. La chaîne NULL entre des guillemets doubles fait que le chaîne est traitées comme tout autre chaîne.

> **Note**
>
> Gardez en tête que le format texte hstore, lorsqu'il est utilisé en entrée, s'applique *avant* tout guillemet ou échappement nécessaire. Si vous passez une valeur litérale de type hstore via un paramètre, aucun traitement supplémentaire n'est nécessaire. par contre, si vous la passez comme constante litérale entre guillemets, alors les guillemets simples et, suivant la configuration du paramètre standard_conforming_strings, les caractères antislash doivent être échappés correctement. Voir Section 4.1.2.1, « Constantes de chaînes » pour plus d'informations sur la gestion des chaînes constantes.

En sortie, guillemets doubles autour des clés et valeurs, en permanence, même quand cela n'est pas strictement nécessaire.

F.13.2. Opérateurs et fonctions hstore

Tableau F.6. Opérateurshstore

Opérateur	Description	Exemple	Résultat
hstore -> text	obtenir la valeur de la clé (NULL si inexistante)	'a=>x, b=>y'::hstore -> 'a'	x
hstore -> text[]	obtenir les valeurs pour les clés (NULL si inexistant)	'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']	{"z","x"}
text => text	créer un hstore à un seul élément	'a' => 'b'	"a"=>"b"
hstore || hstore	concaténation de hstore	'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore	"a"=>"b", "c"=>"x", "d"=>"q"

Opérateur	Description	Exemple	Résultat
		`d=>q'::hstore`	
hstore ? text	hstore contient-il une clé donnée ?	`'a=>1'::hstore ? 'a'`	`t`
hstore ?& text[]	hstore contient-il toutes les clés indiquées ?	`'a=>1,b=>2'::hstore ?& ARRAY['a','b']`	`t`
hstore ?| text[]	hstore contient-il une des clés spécifiées ?	`'a=>1,b=>2'::hstore ?| ARRAY['b','c']`	`t`
hstore @> hstore	l'opérande gauche contient-il l'opérande droit ?	`'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'`	`t`
hstore <@ hstore	l'opérande gauche est-il contenu dans l'opérande droit ?	`'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'`	`f`
hstore - text	supprimer la clé à partir de l'opérande gauche	`'a=>1, b=>2, c=>3'::hstore - 'b'::text`	`"a"=>"1", "c"=>"3"`
hstore - text[]	supprimer les clés à partir de l'opérande gauche	`'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']`	`"c"=>"3"`
hstore - hstore	supprimer les paires correspondantes à partir de l'opérande gauche	`'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore`	`"a"=>"1", "c"=>"3"`
record #= hstore	remplacer les chanmps dans record avec des valeurs correspondantes à hstore	see Examples section	
%% hstore	convertir hstore en un tableau de clés et valeurs alternatives	`%% 'a=>foo, b=>bar'::hstore`	`{a,foo,b,bar}`
%# hstore	convertir hstore en un tableau clé/valeur à deux dimensions	`%# 'a=>foo, b=>bar'::hstore`	`{{a,foo},{b,bar}}`

Avant PostgreSQL 8.2, les opérateurs de contenance @> et <@ étaient appelés respectivement @ et ~. Ces noms sont toujours disponibles mais sont devenus obsolètes et pourraient éventuellement être supprimés. Les anciens noms sont inversés par rapport à la convention suivie par les types de données géométriques.

Note

L'opérateur `=>` est obsolète et pourrait être supprimé dans une prochaine version. À la place, utilisez la fonction `hstore(text, text)`.

Tableau F.7. Fonctions hstore

Fonction	Type en retour	Description	Exemple	Résultat
`hstore(record)`	hstore	construire un hstore à partir d'un RECORD ou d'un ROW	`hstore(ROW(1,2))`	`f1=>1,f2=>2`
`hstore(text[])`	hstore	construire un hstore à partir d'un tableau, qui peut être soit un tableau clé/valeur soit un tableau à deux dimensions	`hstore(ARRAY['a','1','b','2']) || hstore(ARRAY[['c','3'],['d','4']])`	`a=>1, b=>2, c=>3, d=>4`
`hstore(text[], text[])`	hstore	construire un hstore à partir des tableaux séparés pour les clés et valeurs	`hstore(ARRAY['a','b'], ARRAY['1','2'])`	`"a"=>"1","b"=>"2"`

Fonction	Type en retour	Description	Exemple	Résultat
`hstore(text, text)`	hstore	construire un hstore à un seul élément	`hstore('a', 'b')`	`"a"=>"b"`
`akeys(hstore)`	text[]	récupérer les clés du hstore dans un tableau	`akeys('a=>1,b=>2')`	`{a,b}`
`skeys(hstore)`	setof text	récupérer les clés du hstore dans un ensemble	`skeys('a=>1,b=>2')`	`a` `b`
`avals(hstore)`	text[]	récupérer les valeurs du hstore dans un tableau	`avals('a=>1,b=>2')`	`{1,2}`
`svals(hstore)`	setof text	récupérer les valeurs du hstore dans un ensemble	`svals('a=>1,b=>2')`	`1` `2`
`hstore_to_array(hstore)`	text[]	récupérer les clés et les valeurs du hstore sous la forme d'un tableau de clés et valeurs alternées	`hstore_to_array('a=>1,b=>2')`	`{a,1,b,2}`
`hstore_to_matrix(hstore)`	text[]	récupérer les clés et valeurs hstore sous la forme d'un tableau à deux dimensions	`hstore_to_matrix('a=>1,b=>2')`	`{{a,1},{b,2}}`
`slice(hstore, text[])`	hstore	extraire un sous-ensemble d'un hstore	`slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x'])`	`"b"=>"2", "c"=>"3"`

`each(hstore)`	setof (key text, value text)	récupérer les clés et valeurs du hstore dans un ensemble	`select * from each('a=>1,b=>2')`	`key | value`
				`-----+-------`
				`a | 1`
				`b | 2`

`exist(hstore,text)`	boolean	le hstore contient-il une clé donnée ?	`exist('a=>1','a')`	`t`
`defined(hstore,text)`	boolean	le hstore contient-il une valeur non `NULL` pour la clé ?	`defined('a=>NULL','a')`	`f`
`delete(hstore,text)`	hstore	supprimer toute paire correspondant à une clé donnée	`delete('a=>1,b=>2','b')`	`"a"=>"1"`
`delete(hstore,text[])`	hstore	supprimer toute paire de clés correspondante	`delete('a=>1,b=>2,c=>3',ARRAY['a','b'])`	`"c"=>"3"`
`delete(hstore,hstore)`	hstore	supprimer les paires correspondant à celle du second argument	`delete('a=>1,b=>2','a=>4,b=>2'::hstore)`	`"a"=>"1"`
`populate_record(record,hstore)`	record	remplacer les champs dans record avec les valeurs correspondant au hstore	voir la section Exemples	

Note

> La fonction `populate_record` est en fait déclarée avec anyelement, et non pas record, en tant que premier argument mais elle rejettera les types qui ne sont pas des RECORD avec une erreur d'exécution.

F.13.3. Index

hstore dispose du support pour les index GiST et GIN pour les opérateurs `@>`, `?`, `?&` et `?|`. Par exemple :

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

hstore supporte aussi les index btree ou hash pour l'opérateur `=`. Cela permet aux colonnes hstore d'être déclarées `UNIQUE` et d'être utilisées dans des expressions `GROUP BY`, `ORDER BY` et `DISTINCT`. L'ordre de tri pour les valeurs hstore n'est pas particulièrement utile mais ces index pourraient l'être pour des recherches d'équivalence. Créer des index de comparaisons `=` de la façon suivante :

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.13.4. Exemples

Ajouter une clé, ou mettre à jour une clé existante avec une nouvelle valeur :

```
UPDATE tab SET h = h || ('c' => '3');
```

Supprimer une clé :

```
UPDATE tab SET h = delete(h, 'k1');
```

Convertiir un type record en un hstore :

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                   hstore
--------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

Convertir un type hstore en un type record prédéfini :

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

Modifier une enregistrement existant en utilisant les valeurs provenant d'un hstore :

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.13.5. Statistiques

Le type hstore, du fait de sa libéralité intrinsèque, peut contenir beaucoup de clés différentes. C'est à l'application de vérifier la validité des clés. Les exemples ci-dessous présentent plusieurs techniques pour vérifier les clés et obtenir des statistiques.

Exemple simple :

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

En utilisant une table :

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

Statistiques en ligne :

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
...................
```

F.13.6. Compatibilité

Lors d'une mise à jour à partir d'une ancienne version, chargez toujours la nouvelle version de ce module dans la base de données avant de restaurer la sauvegarde. Dans le cas contraire, certaines nouvelles fonctionnalités ne seront pas disponibles.

À partir de PostgreSQL 9.0, hstore utilise une représentation interne différente des anciennes versions. Cela ne présente aucun obstacle pour les mises à jour par sauvegarde/restauration car la représentation textuelle utilisée dans la sauvegarde n'est pas changée.

Dans le cas d'une mise à jour binaire, la compatibilité ascendante est maintenue en faisant en sorte que le nouveau code reconnaisse les données dans l'ancien format. Ceci aura pour conséquence une légère pénalité au niveau des performances lors du traitement de données qui n'aura pas été modifiée par le nouveau code. Il est possible de forcer une mise à jour de toutes les valeurs d'une colonne de la table en réalisant la requête `UPDATE` suivante :

```
UPDATE nom_table SET col_hstore = col_hstore || '';
```

Une autre façon de le faire :

```
ALTER TABLE nom_table ALTER col_hstore TYPE col_hstore USING hstorecol || '';
```

La méthode **ALTER TABLE** requiert un verrou exclusif sur la table mais n'a pas pour résultat une fragmentation de la table avec d'anciennes versions des lignes.

F.13.7. Auteurs

Oleg Bartunov `<oleg@sai.msu.su>`, Moscou, Université de Moscou, Russie

Teodor Sigaev `<teodor@sigaev.ru>`, Moscou, Delta-Soft Ltd., Russie

Additional enhancements by Andrew Gierth `<andrew@tao11.riddles.org.uk>`, United Kingdom

F.14. intagg

Le module `intagg` fournit un agrégateur d'entiers et un énumérateur. `intagg` est maintenant obsolète car il existe des fonctions intégrées qui fournissent les mêmes fonctionnalités. Néanmoins, le module est toujours disponible pour fournir des fonctions de compatibilité.

F.14.1. Fonctions

L'agrégateur est une fonction d'agrégat `int_array_aggregate(integer)` qui produit un tableau d'entiers contenant exactement les entiers fournis en argument. Cette fonction appelle `array_agg` pour des raisons de compatibilité.

L'énumérateur est une fonction `int_array_enum(integer[])` qui renvoie setof integer. C'est essentiellement une opération reverse de l'agrégateur : elle étend un tableau d'entiers en un ensemble de lignes. Cette fonction appelle `unnest`, pour des raisons de compatibilité.

F.14.2. Exemples d'utilisation

Un grand nombre de bases de données utilisent la notion de table « une vers plusieurs » (*one to many*). Ce type de table se trouve habituellement entre deux tables indexés, par exemple :

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

Il est typiquement utilisé de cette façon :

```
  SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
    WHERE one_to_many.left = item;
```

Cela renvoie tous les éléments de la table de droite pour un enregistrement de la table de gauche donné. Il s'agit d'une construction assez commune en SQL.

Cette méthode devient complexe lorsqu'il existe de nombreuses entrées dans la table one_to_many. Souvent, une jointure de ce type résulte en un parcours d'index et une récupération de chaque enregistrement de la table de droite pour une entrée particulière de la table de gauche. Sur un système dynamique, il n'y a pas grand chose à faire. Au contraire, lorsqu'une partie des données est statique, une table de résumé peut être créée par agrégation.

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

Ceci crée une table avec une ligne par élément gauche et un tableau d'éléments droits. En l'absence de méthode d'utilisation de tableau, c'est réellement inutilisable, d'où l'énumérateur.

Exemple :

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

La requête ci-dessus, qui utilise `int_array_enum`, produit les mêmes résultats que celle-ci :

```
SELECT left, right FROM one_to_many WHERE left = item;
```

La différence tient dans le fait que la requête qui utilise la table de résumé ne récupère qu'une ligne de la table alors que la requête directe à one_to_many doit faire un parcours d'index et récupérer une ligne par enregistrement.

Sur un système particulier, un **EXPLAIN** a montré qu'une requête avec un coût de 8488 a été réduite à une requête d'un coût de 329. La requête originale était une jointure impliquant la table one_to_many, remplacée par :

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.15. intarray

Le module `intarray` fournit un certain nombre de fonctions et d'opérateurs utiles pour manipuler des tableaux d'entiers sur une dimension. Il y a aussi un support pour les recherches par index en utilisant certains des opérateurs.

F.15.1. Fonctions et opérateurs d'`intarray`

Tableau F.8. Fonctions `intarray`

Fonction	Type en retour	Description	Exemple	Résultat
`icount(int[])`	int	nombre d'éléments dans un tableau	`icount('{1,2,3}'::int[])`	`3`
`sort(int[], text dir)`	int[]	tri du tableau -- *dir* doit valoir `asc` ou `desc`	`sort('{1,2,3}'::int[], 'desc')`	`{3,2,1}`
`sort(int[])`	int[]	tri en ordre ascendant	`sort(array[11,77,44])`	`{11,44,77}`
`sort_asc(int[])`	int[]	tri en ordre descendant		
`sort_desc(int[])`	int[]	tri en ordre descendant		
`uniq(int[])`	int[]	supprime les duplicats adjacents	`uniq(sort('{1,2,3,2,1}'::int[]))`	`{1,2,3}`
`idx(int[], int item)`	int	index du premier élément correspondant à *item* (0 si aucune correspondance)	`idx(array[11,22,33,22,11], 22)`	`2`
`subarray(int[], int start, int len)`	int[]	portion du tableau commençant à la position *start*, de longueur *len*	`subarray('{1,2,3,2,1}'::int[], 2, 3)`	`{2,3,2}`
`subarray(int[], int start)`	int[]	portion du tableau commençant à la position *start*	`subarray('{1,2,3,2,1}'::int[], 2)`	`{2,3,2,1}`
`intset(int)`	int[]	crée un tableau à un élément	`intset(42)`	`{42}`

Tableau F.9. Opérateurs d'`intarray`

Opérateur	Renvoie	Description
`int[] && int[]`	boolean	surcharge -- `true` si les tableaux ont au moins un élément en commun
`int[] @> int[]`	boolean	contient -- `true` si le tableau gauche contient le tableau droit
`int[] <@ int[]`	boolean	est contenu -- `true` si le tableau gauche est contenu dans le tableau droit
`# int[]`	int	nombre d'éléments dans le tableau
`int[] # int`	int	index (identique à la fonction `idx`)
`int[] + int`	int[]	pousse l'élément dans le tableau (l'ajoute à la fin du tableau)
`int[] + int[]`	int[]	concaténation de tableau (le tableau à droite est ajouté à la fin du tableau à gauche)
`int[] - int`	int[]	supprime les entrée correspondant à l'argument droit du tableau
`int[] - int[]`	int[]	supprime les éléments du tableau droit à partir de la gauche
`int[] | int`	int[]	union des arguments
`int[] | int[]`	int[]	union des tableaux
`int[] & int[]`	int[]	intersection des tableaux
`int[] @@ query_int`	boolean	`true` si le tableau satisfait la requête (voir ci-dessous)
`query_int ~~ int[]`	boolean	`true` si le tableau satisfait la requête (commutateur de @@)

(Avant PostgreSQL 8.2, les opérateurs de contenance @> et <@ étaient respectivement appelés @ et ~. Ces noms sont toujours disponibles mais sont considérés comme obsolètes et seront un jour supprimés. Notez que les anciens noms sont inversés par rapport à la convention suivie par les types de données géométriques !)

Les opérateurs de contenance @ et ~ sont équivalents fonctionnellement aux opérateurs internes de PostgreSQL™, respectivement @> et <@, en dehors du fait que @ et ~ fonctionnent seulement avec des tableaux d'entiers alors que les opérateurs internes fonctionnent avec tout type de tableau. Une différence importante est que les opérateurs de `intarray` ne considèrent pas un tableau vide comme appartenant à autre chose. Ceci est cohérent avec le comportement des requêtes utilisant des index GIN, mais pas avec la définition mathématique habituelle de contenance.

Les opérateurs @@ et ~~ testent si un tableau satisfait une *requête*, qui est exprimée comme une valeur d'un type de données spécialisé query_int. Une *requête* consiste en des valeurs de type integer qui sont vérifiées avec les éléments du tableau, parfois combinées en utilisant les opérateurs & (AND), | (OR) et ! (NOT). Les parenthèses peuvent être utilisées si nécessaire. Par exemple, la requête `1&(2|3)` établit une correspondance avec les tableaux qui contiennent 1 et aussi soit 2 soit 3.

F.15.2. Support des index

`intarray` fournit un support d'index pour les opérateurs &&, @>, <@ et @@, ainsi que pour l'égalité de tableaux.

Deux classes d'opérateur pour index GiST sont fournies : `gist__int_ops` (utilisé par défaut) convient pour des tableaux d'ensembles de données de petites et moyennes tailles alors que `gist__intbig_ops` utilise une signature plus importante et est donc plus intéressant pour indexer des gros ensembles de données. (c'est-à-dire les colonnes contenant un grand nombre de valeurs de tableaux distinctes). L'implantation utilise une structure de données RD-tree avec une compression interne à perte.

Il y a aussi une classe d'opérateur GIN,`gin__int_ops` supporting the same operators, qui n'est pas disponible par défaut.

Le choix d'un indexage GiST ou IN dépend des caractéristiques relatives de performance qui sont discutées ailleurs. Comme règle de base, un index GIN est plus rapide pour la recherche qu'un index GiST mais plus lent pour la construction et la mise à jour ; donc GIN est préférable pour des données statiques et GiST pour des données souvent mises à jour.

F.15.3. Exemple

```
-- un message peut être dans un ou plusieurs « sections »
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- crée un index spécialisé
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- sélectionne les messages dans la section 1 ou 2 - opérateur OVERLAP
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- sélectionne les messages dans sections 1 et 2 - opérateur CONTAINS
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- idem, en utilisant l'opérateur QUERY
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.15.4. Tests de performance

Le répertoire des sources (`contrib/intarray/bench`) contient une suite de tests de performance. Pour l'exécuter :

```
cd .../bench
createdb TEST
psql TEST < ../_int.sql
./create_test.pl | psql TEST
./bench.pl
```

Le script `bench.pl` contient un grand nombre d'options. Elles sont affichées quand il est exécuté sans arguments.

F.15.5. Auteurs

Ce travail a été réalisé par Teodor Sigaev (`<teodor@sigaev.ru>`) et Oleg Bartunov (`<oleg@sai.msu.su>`). Voir le *site de GiST* pour des informations supplémentaires. Andrey Oktyabrski a fait un gros travail en ajoutant des nouvelles fonctions et opérateurs.

F.16. isn

Le module `isn` fournit des types de données pour les standards internationaux de numérotation suivants : EAN13, UPC, ISBN (livres), ISMN (musique) et ISSN (numéro de série). Les nombres sont validés en saisie, et correctement affichés (avec les tirets) en sortie.

F.16.1. Types de données

Tableau F.10, « Types de données `isn` » shows the data types provided by the `isn` module.

Tableau F.10. Types de données `isn`

Type de données	Description
EAN13	Numéro d'article européen (*European Article Numbers*), toujours affiché dans le format de l'EAN13
ISBN13	Numéro standard international pour les livres (*International Standard Book Numbers*) à afficher dans le nouveau format EAN13
ISMN13	Numéro standard international pour la musique (*International Standard Music Numbers*) à afficher dans le nouveau format EAN13
ISSN13	Numéro de série au standard international (*International Standard Serial Numbers*) à afficher dans le nouveau format EAN13
ISBN	Numéro standard international pour les livres (*International Standard Book Numbers*) à afficher dans l'ancien format court
ISMN	Numéro standard international pour la musique (*International*

Type de données	Description
	Standard Music Numbers) à afficher dans l'ancien format court
ISSN	Numéro de série au standard international (*International Standard Serial Numbers*) à afficher dans l'ancien format court
UPC	Code produit universel (*Universal Product Codes*)

Quelques notes :

1. Les nombres ISBN13, ISMN13, ISSN13 sont tous des nombres EAN13.
2. Les nombres EAN13 ne sont pas toujours des ISBN13, ISMN13 ou ISSN13 (mais certains le sont).
3. Certains nombres ISBN13 peuvent être affichés comme des ISBN.
4. Certains nombres ISMN13 peuvent être affichés comme des ISMN.
5. Certains nombres ISSN13 peuvent être affichés comme des ISSN.
6. Les nombres UPC sont un sous-ensemble des nombres EAN13 (ce sont basiquement des EAN13 sans le premier `0`).
7. Tous les nombres UPC, ISBN, ISMN et ISSN numbers peuvent être représentés sous la forme EAN13.

En interne, tous ces types utilisent la même représentation (un entier sur 64 bits), et tous sont interchangeables. Plusieurs types sont fournis pour contrôler le formatage de l'affichage et pour permettre une vérification très fine de la validité des entrées qui est supposée dénoter un type particulier de nombre.

Les types ISBN, ISMN et ISSN afficheront la version courte du nombre (ISxN 10) quand c'est possible, et afficheront la version au format ISxN 13 pour les nombres qui ne tiennent pas dans la version courte. Les types EAN13, ISBN13, ISMN13 et ISSN13 afficheront toujours la version longue de l'ISxN (EAN13).

F.16.2. Conversions

Le module `isn` fournit les paires suivantes pour les conversions de types :

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

Lors d'une conversion d'EAN13 vers un autre type, il y a une vérification à l'exécution que la valeur est dans le domaine de l'autre type et une erreur est renvoyée dans le cas contraire. Les autres conversions sont simplement un renommage qui succèdera à chaque fois.

F.16.3. Fonctions et opérateurs

Le module `isn` fournit des opérateurs de comparaison standard, plus un support des index B-Tree et hachés pour tous les types de données. De plus, il existe plusieurs fonctions spécialisées, listées dans Tableau F.11, « Fonctions de `isn` ». Dans cette table, isn signifie un des types de données de ce module :

Tableau F.11. Fonctions de `isn`

Fonction	Retour	Description
`isn_weak(boolean)`	boolean	Configure le mode de saisie faible

Fonction	Retour	Description
		(renvoie le nouveau paramétrage)
`isn_weak()`	boolean	Récupère le statut actuel du mode faible
`make_valid(isn)`	isn	Valide un nombre invalide (efface le drapeau d'invalidité)
`is_valid(isn)`	boolean	Vérifie la présence du drapeau d'invalidité

Le mode *faible* est utilisé pour insérer des données invalides dans une table. Invalide signifie que le chiffre de vérification est mauvais, pas qu'il manque des numéros.

Pourquoi voudriez-vous utiliser le mode faible ? Tout simplement parce que vous pouvez avoir une grosse collection de nombres ISBN, et que beaucoup d'entre eux, quelque soit la raison, ont un mauvais chiffre de vérification (peut-être que les nombres ont été scannés à partir d'une liste imprimée et que l'OCR s'est trompé sur les numéros, peut-être que les numéros ont été saisis manuellement... qui sait). Bref, le fait est que vous pouvez vouloir corriger ça, mais que vous voulez être capable d'avoir tous les nombres dans votre base de données pour que vous puissiez vérifier l'information et peut-être utiliser un outil externe pour localiser les nombres invalides dans la base de données, puis les vérifier et valider plus facilement ; donc par exemple, vous voudrez sélectionner tous les nombres invalides dans la table.

Quand vous insérez des nombres invalides dans une table en utilisant le mode faible, le nombre sera inséré avec le chiffre de vérification corrigé, mais il sera affiché avec un point d'exclamation (`!`) à la fin, par exemple `0-11-000322-5!`. Ce marqueur d'invalidité peut être vérifié avec la fonction `is_valid` et effacé avec la fonction `make_valid`.

Vous pouvez aussi forcer l'insertion de nombres invalides, même quand vous n'êtes pas dans le mode faible, en ajoutant le caractère `!` à la fin du nombre.

Une autre fonctionnalité spéciale est que, durant la saisie, vous pouvez écrire `?` à la place du chiffre de vérification. Ce dernier sera calculé et inséré automatiquement.

F.16.4. Exemples

```
--Using the types directly:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--Casting types:
-- note that you can only cast from ean13 to another type when the
-- number would be valid in the realm of the target type;
-- thus, the following will NOT work: select isbn(ean13('0220356483481'));
-- but these will:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--Create a table with a single column to hold ISBN numbers:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--Automatically calculate check digits (observe the '?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--Using the weak mode:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;
```

```
SELECT isbn13(id) FROM test;
```

F.16.5. Bibliographie

Les informations qui ont permis l'implémentation de ce module ont été récupérées sur plusieurs sites, dont :

- *http://www.isbn-international.org/*
- *http://www.issn.org/*
- *http://www.ismn-international.org/*
- *http://www.wikipedia.org/*

Les préfixes utilisées pour le formatage ont été récupérés à partir de :

- *http://www.gs1.org/productssolutions/idkeys/support/prefix_list.html*
- *http://www.isbn-international.org/en/identifiers.html*
- *http://www.ismn-international.org/ranges.html*

Nous avons porté une grande attention lors de la création des algorithmes et ils ont été vérifiés méticuleusement par rapport aux algorithmes suggérés dans les manuels utilisateurs officiels ISBN, ISMN et ISSN.

F.16.6. Auteur

Germen Mendez Bravo (Kronuz), 2004 - 2006

Ce module est inspiré du code `isbn_issn` de Garrett A. Wollman.

F.17. lo

Le module `lo` ajoute un support des « Large Objects » (aussi appelé LO ou BLOB). Il inclut le type de données lo et un trigger `lo_manage`.

F.17.1. Aperçu

Un des problèmes avec le pilote JDBC (mais cela affecte aussi le pilote ODBC) est que la spécification suppose que les références aux BLOB (Binary Large OBject) sont stockées dans une table et que, si une entrée est modifiée, le BLOB associé est supprimé de cette base.

Au niveau de PostgreSQL™, ceci n'arrive pas. Les « Large Objects » sont traités comme des objets propres ; une entrée de table peut référencer un « Large Object » par son OID, mais plusieurs tables peuvent référencer le même OID. Donc, le système ne peut pas supprimer un « Large Object » simplement parce que vous avez modifié ou supprimé une entrée contenant son OID.

Ceci n'est pas un problème pour les nouvelles applications spécifiques à PostgreSQL™ mais celles qui existent déjà, qui utilisent JDBC ou ODBC, ne suppriment pas les objets, ceci aboutissant à des « Large Objects » orphelins - des objets qui ne sont référencés par personne et occupant donc un espace disque précieux sans raison.

Le module `lo` permet de corriger ceci en attachant un trigger aux tables contenant des colonnes de référence des LO. Le trigger fait essentiellement un `lo_unlink` quand vous supprimez ou modifiez une valeur référence un « Large Object ». Quand vous utilisez ce trigger, vous supposez que, dans toute la base de données, il n'existe qu'une référence d'un « Large Object » référencé dans une colonne contrôlée par un trigger.

Le module fournit aussi un type de données lo, qui est tout simplement un domaine sur le type oid. Il est utile pour différencier les colonnes de la base qui contiennent des références d'objet de ceux qui contiennent des OID sur d'autres choses. Vous n'avez pas besoin d'utiliser le type lo pour utiliser le trigger mais cela facilite le travail pour garder trace des colonnes de votre base de données qui représentent des « Large Objects » que vous gérez avec le trigger. Une rumeur indique aussi que le pilote ODBC a du mal si vous n'utilisez pas le type lo pour les colonnes BLOB.

F.17.2. Comment l'utiliser

Voici un exemple d'utilisation :

```
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
  FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

Pour chaque colonne qui contiendra des références uniques aux « Large Objects », créez un trigger `BEFORE UPDATE OR DELETE` trigger, et donnez le nom de la colonne comme argument du trigger. Si vous avez plusieurs colonnes lo dans la même table, créez un trigger séparé pour chacune en vous souvenant de donner un nom différent à chaque trigger sur la même table.

F.17.3. Limites

- Supprimer une table résultera quand même en des objets orphelins pour tous les objets qu'elle contient, car le trigger n'est pas exécuté. Vous pouvez éviter ceci en faisant précéder le **DROP TABLE** avec **DELETE FROM** ***`table`***.

  TRUNCATE a le même comportement.

  Si vous avez déjà, ou suspectez avoir, des « Large Objects » orphelins, voir le module `contrib/vacuumlo` (Section F.41, « vacuumlo ») pour vous aider à les nettoyer. Une bonne idée est d'exécuter vacuumlo occasionnellement pour s'assurer du ménage réalisé par le trigger `lo_manage`.
- Quelques interfaces peuvent créer leur propres tables et n'ajouteront pas les triggers associés. De plus, les utilisateurs peuvent ne pas se rappeler (ou savoir) pour créer les triggers.

F.17.4. Auteur

Peter Mount `<peter@retep.org.uk>`

F.18. ltree

Ce module implémente le type de données ltree pour représenter des labels de données stockés dans une structure hiérarchique de type arbre. Des fonctionnalités étendues de recherche sont fournies.

F.18.1. Définitions

Un *label* est une séquence de caractères alphanumériques et de tirets bas (par exemple, dans la locale C, les caractères `A-Za-z0-9_` sont autorisés). La longueur d'un label est limité par 256 octets.

Exemples : `42`, `Personal_Services`

Le *chemin de label* est une séquence de zéro ou plusieurs labels séparés par des points, par exemple `L1.L2.L3`, ce qui représente le chemin de la racine jusqu'à un nœud particulier. La longueur d'un chemin est limité à 65 Ko, mais une longueur inférieure ou égale à 2 Ko est préférable. Nous considérons qu'il ne s'agit pas d'une limitation stricte. Par exemple, lataille maximum d'un chemin de label dans le *catalogue DMOZ* fait environ 240 octets !

Exemple : `Top.Countries.Europe.Russia`

Le module `ltree` fournit plusieurs types de données :

- ltree stocke un chemin de label.
- lquery représente un type d'expression rationnelle du chemin pour la correspondance de valeurs de type ltree. Un mot simple établit une correspondance avec ce label dans un chemin. Le caractère joker (`*`) est utilisé pour spécifier tout nombre de labels (niveaux). Par exemple :

  ```
  foo         Correspond au chemin exact foo
  *.foo.*     Correspond à tout chemin contenant le label foo
  *.foo       Correspond à tout chemin dont le dernier label est
  foo
  ```

  Les caractères joker peuvent être quantifiés pour restreindre le nombre de labels de la correspondance :

  ```
  *{n}        Correspond à exactement n labels
  ```

```
*{n,}       Correspond à au moins n labels
*{n,m}      Correspond à au moins n labels mais à pas plus de m
*{,m}       Correspond à au plus m labels  --  identique à *{0,m}
```

Il existe plusieurs modificateurs qui peuvent être placés à la fin d'un label sans joker dans un lquery pour que la correspondance se fasse sur plus que la correspondance exacte :

```
@           Correspondance sans vérification de casse, par exemple a@ établit une
correspondance avec A
*           Correspondance d'un préfixe pour un label, par exemple foo* établit une
correspondance avec foobar
%           Correspondance avec les mots séparés par des tirets bas
```

Le comportement de `%` est un peu complexe. Il tente d'établir une correspondance avec des mots plutôt qu'avec un label complet. Par exemple, `foo_bar%` établit une correspondance avec `foo_bar_baz` mais pas avec `foo_barbaz`. S'il est combiné avec `*`, la correspondance du préfixe s'applique à chaque mot séparément. Par exemple, `foo_bar%*` établit une correspondance avec `foo1_bar2_baz`, mais pas avec `foo1_br2_baz`.

De plus, vous pouvez écrire plusieurs labels séparés avec des `|` (OR) pour établir une correspondance avec un des labels, et vous pouvez placer un `!` (NOT) au début pour établir une correspondance avec tout sauf une des différentes alternatives.

Voici un exemple annoté d'une lquery :

```
     Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
     a.  b.     c.      d.               e.
```

Cette requête établira une correspondance avec tout chemin qui :

a. commence avec le label `Top`

b. et suit avec zéro ou deux labels jusqu'à

c. un label commençant avec le préfixe `sport` quelque soit la casse

d. ensuite un label ne correspondant ni à `football` ni à `tennis`

e. et se termine enfin avec un label commençant par `Russ` ou correspond strictement à `Spain`.

- ltxtquery represente en quelque sorte une recherche plein texte pour la correspondance de valeurs ltree. Une valeur ltxtquery contient des mots, quelque fois avec les modificateurs `@`, `*`, `%` à la fin ; les modifications ont la même signification que dans un lquery. Les mots peuvent être combinés avec `&` (AND), `|` (OR), `!` (NOT) et des parenthèses. La différence clé d'une lquery est que ltxtquery établit une correspondance avec des mots sans relation avec leur position dans le chemin de labels.

  Voici un exemple de ltxtquery :

  ```
     Europe & Russia*@ & !Transportation
  ```

  Ceci établira une correspondance avec les chemins contenant le label `Europe` et tout label commençant par `Russia` (quelque soit la casse), mais pas les chemins contenant le label `Transportation`. L'emplacement de ces mots dans le chemin n'est pas important. De plus, quand `%` est utilisé, le mot peut établir une correspondance avec tout mot séparé par un tiret bas dans un label, quelque soit sa position.

Note : ltxtquery autorise un espace blanc entre des symboles mais ltree et lquery ne le permettent pas.

F.18.2. Opérateurs et fonctions

Le type ltree dispose des opérateurs de comparaison habituels `=`, `<>`, `<`, `>`, `<=`, `>=`. Les comparaisons trient dans l'ordre du parcours d'un arbre, avec les enfants d'un nœud trié par le texte du label. De plus, il existe les opérateurs spécialisés suivant :

Tableau F.12. Opérateurs ltree

Opérateur	Retour	Description
ltree `@>` ltree	boolean	l'argument gauche est-il un ancêtre de l'argument droit (ou identique) ?

Opérateur	Retour	Description
ltree <@ ltree	boolean	l'argument gauche est-il un descendant de l'argument droit (ou identique) ?
ltree ~ lquery	boolean	est-ce que ltree établie une correspondance avec lquery ?
lquery ~ ltree	boolean	est-ce que ltree établie une correspondance avec lquery ?
ltree ? lquery[]	boolean	est-ce que ltree établie une correspondance avec tout any lquery dans ce tableau ?
lquery[] ? ltree	boolean	est-ce que ltree établie une correspondance avec tout lquery dans ce tableau ?
ltree @ ltxtquery	boolean	est-ce que ltree établie une correspondance avec ltxtquery ?
ltxtquery @ ltree	boolean	est-ce que ltree établie une correspondance avec ltxtquery ?
ltree || ltree	ltree	concatène des chemins ltree
ltree || text	ltree	convertit du texte en ltree et concatène
text || ltree	ltree	convertit du texte en ltree et concatène
ltree[] @> ltree	boolean	est-ce que le tableau contient un ancêtre de ltree ?
ltree <@ ltree[]	boolean	est-ce que le tableau contient un ancêtre de ltree ?
ltree[] <@ ltree	boolean	est-ce que le tableau contient un descendant de ltree ?
ltree @> ltree[]	boolean	est-ce que le tableau contient un descendant de ltree ?
ltree[] ~ lquery	boolean	est-ce que le tableau contient tout chemin correspondant à lquery ?
lquery ~ ltree[]	boolean	est-ce que le tableau contient tout chemin correspondant à lquery ?
ltree[] ? lquery[]	boolean	est-ce que le tableau ltree contient tout chemin correspondant à un lquery ?
lquery[] ? ltree[]	boolean	est-ce que le tableau ltree contient tout chemin correspondant à un lquery ?
ltree[] @ ltxtquery	boolean	est-ce que le tableau contient tout chemin correspondant à ltxtquery ?
ltxtquery @ ltree[]	boolean	est-ce que le tableau contient tout chemin correspondant à ltxtquery ?
ltree[] ?@> ltree	ltree	première entrée du tableau ancêtre de ltree ; NULL si aucun
ltree[] ?<@ ltree	ltree	première entrée du tableau descendant de ltree ; NULL si aucun
ltree[] ?~ lquery	ltree	première entrée du tableau établissant une correspondance avec lquery ; NULL si aucune
ltree[] ?@ ltxtquery	ltree	première entrée du tableau établissant une correspondance avec ltxtquery ; NULL si aucune

Lesopérateurs operators <@, @>, @ et ~ ont des versions analogues ^<@, ^@>, ^@, ^~, qui sont identiques sauf qu'elles n'utilisent pas les index. Elles sont utiles pour tester.

Les fonctions suivantes sont disponibles :

Tableau F.13. Fonctions ltree

Fonction	Type en retour	Description	Exemple	Résultat
`subltree(ltree, int start, int end)`	ltree	sous-chemin de of ltree de la position *start* à la position *end*-1 (counting from 0)	`subltree('Top.Child1.Child2',1,2)`	`Child1`
`subpath(ltree, int offset, int len)`	ltree	sous-chemin de ltree commençant à la position *offset*, de longueur *len*. Si *offset* est négatif, le sous-chemin commence de ce nombre à partir de la fin du chemin. Si *len* est négatif, laisse ce nombre de labels depuis la fin du chemin.	`subpath('Top.Child1.Child2',0,2)`	`Top.Child1`
`subpath(ltree, int offset)`	ltree	sous-chemin de ltree commençant à la position *offset*, s'étendant à la fin du chemin. Si *offset* est négatif, le sous-chemin commence de ce point jusqu'à la fin du chemin.	`subpath('Top.Child1.Child2',1)`	`Child1.Child2`
`nlevel(ltree)`	integer	nombre de labels dans le chemin	`nlevel('Top.Child1.Child2')`	`3`
`index(ltree a, ltree b)`	integer	position de la première occurrence de *b* dans *a* ; -1 si introuvable	`index('0.1.2.3.5.4.5.6.8.5.6.8','5.6')`	`6`
`index(ltree a, ltree b, int offset)`	integer	position de la première occurrence de *b* dans *a*, la recherche commence à *offset* ; un *offset* négatif signifie un commencement à *-offset* labels de la fin du chemin	`index('0.1.2.3.5.4.5.6.8.5.6.8','5.6',-4)`	`9`
`text2ltree(text)`	ltree	convertit du text en ltree		
`ltree2text(ltree)`	text	convertit du ltree en text		
`lca(ltree, ltree, ...)`	ltree	plus petit ancêtre commun, c'est-à-dire préfixe commun le plus long des chemins (jusqu'à huit arguments supportés)	`lca('1.2.2.3','1.2.3.4.5.6')`	`1.2`
`lca(ltree[])`	ltree	plus petit ancêtre commun, c'est-à-dire préfixe commun le plus long des chemins	`lca(array['1.2.2.3'::ltree,'1.2.3'])`	`1.2`

F.18.3. Index

`ltree` accepte différents types d'index pouvant améliorer les performances des oopérateurs indiqués :

- Index B-tree sur ltree : <, <=, =, >=, >
- Index GiST sur ltree : <, <=, =, >=, >, @>, <@, @, ~, ?

  Exemple de la création d'un tel index :

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path);
  ```

- Index GiST sur ltree[] : `ltree[] <@ ltree`, `ltree @> ltree[]`, @, ~, ?

  Exemple de la création d'un tel index :

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

  Note : ce type d'index est à perte.

F.18.4. Exemple

Cet exemple utilise les données suivantes (disponibles dans le fichier `contrib/ltree/ltreetest.sql` des sources) :

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING gist(path);
CREATE INDEX path_idx ON test USING btree(path);
```

Maintenant, nous avons une table test peuplée avec des données décrivant la hiérarchie ci-dessous :

```
                            Top
                         /   |  \
                 Science Hobbies Collections
                      /       |              \
            Astronomy   Amateurs_Astronomy Pictures
               /  \                            |
    Astrophysics  Cosmology                Astronomy
                                            /  |    \
                                     Galaxies Stars Astronauts
```

Nous pouvons faire de l'héritage :

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
------------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

Voici quelques exemples de correspondance de chemins :

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                      path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.*.Astronomy.*';
                path
-------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Voici quelques exemples de recherche plein texte :

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
-------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
-------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Construction d'un chemin en utilisant les fonctions :

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@
'Top.Science.Astronomy';
                 ?column?
-----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

Nous pouvons simplifier ceci en créant une fonction SQL qui insère un label à une position spécifié dans un chemin :

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@
'Top.Science.Astronomy';
                ins_label
-----------------------------------------
 Top.Science.Space.Astronomy
```

```
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.18.5. Auteurs

Tout le travail a été réalisé par Teodor Sigaev (`<teodor@stack.net>`) et Oleg Bartunov (`<oleg@sai.msu.su>`). Voir *http://www.sai.msu.su/~megera/postgres/gist* pour des informations supplémentaires. Les auteurs voudraient remercier Eugeny Rodichev pour son aide. Commentaires et rapports de bogue sont les bienvenus.

F.19. oid2name

oid2name est un outil qui aide les administrateurs à examiner la structure des fichiers utilisée par PostgreSQL. Pour l'utiliser, vous devez être connaître la structure de fichiers utilisée de la base de données. Elle est décrite dans Chapitre 54, Stockage physique de la base de données.

> **Note**
>
> Le nom « oid2name » est historique, et est maintenant plutôt contradictoire car la plupart du temps, quand vous l'utiliserez, vous aurez besoin de connaître les numéroes filenode des tables (qui sont le nom des fichiers visibles dans les répertoires des bases de données). Assurez-vous de bien comprendre la différence entre les OID des tables et leur filenode !

F.19.1. Aperçu

oid2name se connecte à une base de données cible et extrait OID, filenode, et/ou nom de table. Vous pouvez aussi afficher les OID des bases et des tablespaces.

F.19.2. oid2name options

oid2name accepte les arguments suivants en ligne de commande :

`-o` *`oid`*
: affiche des informations pour la table dont l'OID est *`oid`*

`-f` *`filenode`*
: affiche des informations pour la table dont le filenode est *`filenode`*

`-t` *`motif_nomtable`*
: affiche des informations pour les tables dont le nom respecte *`motif_nomtable`*

`-s`
: affiche les OID des tablespaces

`-S`
: inclut les objets systèmes (ceux compris dans les schémas `information_schema`, `pg_toast` et `pg_catalog`)

`-i`
: inclut les index et séquences dans la liste

`-x`
: affiche plus d'informations sur chaque objet affiché : nom du tablespace, nom du schéma et OID

`-q`
: omet les entêtes (facilite le scriptage)

`-d` *`nom_base`*
: base de données où se connecter

`-H` *`hôte`*
: hôte du serveur de base de données

`-p` *`port`*
: port du serveur de base de données

`-U` *`nom_utilisateur`*

nom d'utilisateur pour la connexion

`-P` *`mot_de_passe`*
mot de passe (obsolète -- placer cette information sur la ligne de commande introduit un risque de sécurité)

Pour afficher des tables spécifiques, sélectionnez les tables à afficher en utilisant `-o`, `-f` et/ou `-t`. `-o` prend un OID, `-f` prend un filenode, et `-t` prend un nom de table (en fait, c'est un modèle de type `LIKE`, donc vous pouvez utiliser `foo%` par exemple). Vous pouvez utiliser autant d'options que vous le souhaitez, et la liste incluera tous les objets en se basant sur chaque options. Mais notez que ces options peuvent seulement afficher des objets appartenant à la base de données indiquée par l'option `-d`.

Si vous n'utilisez pas `-o`, `-f` et `-t`, mais que vous passez l'option `-d`, cela listera toutes les tables dans la base nommée par l'option `-d`. Dans ce mode, les options `-S` et `-i` contrôlent ce qui est listé.

Si vous ne passez pas non plus `-d`, cela affichera une liste des OID de bases de données. Autrement, vous pouvez passer l'option `-s` pour obtenir une liste des tablespaces.

F.19.3. Exemples

```
$ # quelles sont les bases disponibles ?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # OK, jetons un œil à la base alvherre
$ cd $PGDATA/base/17228

$ # récupérons les 10 premiers objets de la base dans le tablespace par défaut
$ # et triés par taille
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ # à quoi correspond le fichier 155173 ?
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # vous pouvez demander plus d'un objet à la fois
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
------------------------
    155173       accounts
   1155291  accounts_pkey
```

```
$ # vous pouvez mélanger les options et obtenir plus de détails avec -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
-----------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # affiche l'espace disque pour chaque objet d'une base de données
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # pareil, mais trié par taille
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561            1155291  accounts_pkey
1177              16717  pg_proc_proname_args_nsp_index
...

$ # Si vous voulez voir ce qu'il y a dans un tablespace, utilisez le répertoire
$ # pg_tblspc
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # quelle base de données a des objets dans le tablespace "fastdisk" ?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # Oh, quelle était la base de données 17228 ?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # Voyons si quels objets de cette base sont dans ce tablespace.
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # OK, c'est une table très petite, mais laquelle est-ce ?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

F.19.4. Limites

oid2name requiert que la serveur soit en cours d'exécution avec des catalogues systèmes non corrompus. Son utilisation est donc très limitée en ce qui concerne la récupération à partir de situations catastrophiques de récupération.

F.19.5. Auteur

B. Palmer `<bpalmer@crimelabs.net>`

F.20. pageinspect

Le module `pageinspect` fournit des fonctions qui vous permettent d'inspecter le contenu des pages de la base de données à un bas niveau, ce qui utile pour le débogage. Toutes ces fonctions ne sont utilisables que par les super-utilisateurs.

F.20.1. Fonctions

`get_raw_page(relname text, fork text, blkno int) returns bytea`

`get_raw_page` lit le bloc spécifié de la table nommée et renvoie une copie en tant que valeur de type bytea. Ceci permet la récupération de la copie cohérente à un instant t d'un bloc spécifique. *`fork`* devrait être `'main'` pour les données, et `'fsm'` pour la carte des espaces libres, ou `'vm'` pour la carte de visibilité.

`get_raw_page(relname text, blkno int) returns bytea`

Une version raccourcie de `get_raw_page`, pour le lire que la partie des données. Équivalent à `get_raw_page(relname, 'main', blkno)`.

`page_header(page bytea) returns record`

`page_header` affiche les champs communs à toutes les pages des tables et index PostgreSQL™.

L'image d'une page obtenu avec `get_raw_page` doit être passé en argument. Par exemple :

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | tli | flags | lower | upper | special | pagesize | version | prune_xid
-----------+-----+-------+-------+-------+---------+----------+---------+-----------
 0/24A1B50 |   1 |     1 |   232 |   368 |    8192 |     8192 |       4 |         0
```

Les colonnes renvoyées correspondent aux champs de la structure PageHeaderData. Voir `src/include/storage/bufpage.h` pour les détails.

`heap_page_items(page bytea) returns setof record`

`heap_page_items` affiche tous les pointeurs de ligne dans une page de table. Pour les pointeurs de ligne en utilisation, les en-têtes de ligne sont aussi affichées. Toutes les lignes sont affichées, qu'elles soient ou non visibles dans l'image MVCC au moment où la page brute a été copiée.

Une image d'une page de table obtenue avec `get_raw_page` doit être fournie en argument. Par exemple :

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

Voir `src/include/storage/itemid.h` et `src/include/access/htup.h` pour des explications sur les champs renvoyés.

`bt_metap(relname text) returns record`

`bt_metap` renvoie des informations sur une méta-page d'un index B-tree. Par exemple :

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 1
level     | 0
fastroot  | 1
fastlevel | 0
```

`bt_page_stats(relname text, blkno int) returns record`

`bt_page_stats` renvoie un résumé des informations sur les pages enfants des index B-tree. Par exemple :

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 256
dead_items    | 0
avg_item_size | 12
page_size     | 8192
free_size     | 4056
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

`bt_page_items(relname text, blkno int) returns setof record`

`bt_page_items` renvoie des informations détaillées sur tous les éléments d'une page d'index btree. Par exemple :

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

`fsm_page_contents(page bytea) returns text`

`fsm_page_contents` affiche la structure interne d'une page FSM. La sortie est une chaîne multi-lignes, chaque ligne décrivant un nœud de l'arbre binaire d'une page. Seuls les nœuds différents de zéro sont affichés. Le pointeur appelé « next », qui pointe vers le prochain slot à renvoyer pour cette page, est aussi affiché.

Voir `src/backend/storage/freespace/README` pour plus d'informations sur la structure d'une page FSM.

F.21. passwordcheck

Le module `passwordcheck` vérifie les mots de passe des utilisateurs quand ils sont configurés avec CREATE ROLE(7) ou ALTER ROLE(7). Si un mot de passe est considéré trop faible, il sera rejeté et la commande terminera avec une erreur.

Pour activer ce module, ajoutez `'$libdir/passwordcheck'` dans le paramètre shared_preload_libraries du fichier `postgresql.conf`, puis redémarrez le serveur.

Vous pouvez adapter ce module à vos besoins en changeant son code source. Par exemple, vous pouvez utiliser *CrackLib* pour vérifier les mots de passe -- ceci requiert seulement la suppression des commentaires sur deux lignes du `Makefile` et la reconstruction du module. (Nous ne pouvons pas inclure CrackLib™ par défaut pour des raisons de licence.) Sans CrackLib™, le module force quelques règles simples sur la force du mot de passe, que vous pouvez modifier ou étendre au besoin.

Attention

Pour empêcher l'envoi en clair de mot de passe sur le réseau, leur écriture dans les journaux applicatifs ou leur récupération par un administrateur de bases de données, PostgreSQL™ permet à l'utilisateur de fournir des mots de passe déjà chiffrés. Beaucoup de programmes clients utilisent cette fonctionnalité et chiffrent le mot de passe avant de l'envoyer au serveur.

Ceci limite l'utilité du module `passwordcheck` car, dans ce cas, il peut seulement tenter de deviner le mot de passe. Pour cette raison, `passwordcheck` n'est pas recommandé si vos besoins en sécurité sont importants. Il est plus intéressant d'utiliser une méthode d'authentification externe comme Kerberos (voir Chapitre 19, Authentification du client) plutôt que de se fier aux mots de passe internes.

> Autrement, vous pouvez modifier `passwordcheck` pour rejeter les mots de passe pré-chiffrés, mais forcer les utilisateurs à initialiser les mots de passe en clair apporte son lot de problèmes de sécurité.

F.22. pg_archivecleanup

pg_archivecleanup est conçu pour être utilisé via le paramètre `archive_cleanup_command` pour nettoyer les archives des journaux de transactions lorsque le serveur est exécuté en tant que serveur en standby (voir Section 25.2, « Serveurs de Standby par transfert de journaux »). pg_archivecleanup peut aussi être utilisé comme un programme autonome pour nettoyer les archives de journaux de transactions.

Les fonctionnalités de pg_archivecleanup incluent :

- Écrit en C, donc très portable et facile à installer
- Code source facile à modifier, avec des sections spécialement conçues pour être modifier selon vos besoins

F.22.1. Utilisation

Pour configurer un serveur en attente de façon à ce qu'il utilise pg_archivecleanup, placez ce qui suit dans le fichier de configuration `recovery.conf` :

```
archive_cleanup_command = 'pg_archivecleanup RépArchive %r'
```

où *`RépArchive`* est le répertoire à partir duquel les journaux de transactions doivent être supprimés.

S'il est utilisé via archive_cleanup_command, tous les fichiers des journaux de transactions qui précèdent la valeur de `%r` de façon logique seront supprimés de *`RépArchive`*. Ceci minimise le nombre de fichiers à conserver tout en préservant la capcité à redémarrer suite à un arrêt brutal. L'utilisation de ce paramètre est approprié si *`RépArchive`* est une aire de passage temporaire pour ce serveur en attente particulier, mais *pas* quand *`RépArchive`* a pour but d'être une archive à long terme des journaux de transactions ou quand plusieurs serveurs en standby utilisent le même emplacement pour les archives.

La syntaxe complète de la ligne de commande de pg_archivecleanup est la suivante :

```
pg_archivecleanup [ option ... ] RépArchive JournalRedémarrage
```

S'il est utilisé en tant que programme autonome, tous les fichiers de journaux de transactions avant le `JournalRedémarrage` seront supprimés de *`RépArchive`*. Dans ce mode, si vous indiquez un nom de fichier avec une extension `.backup`, alors seul le préfixe du fichier sera utilisé comme `JournalRedémarrage`. Ceci vous permet de supprimer tous les fichiers des journaux de transactions archivés avant une sauvegarde de base spécifique sans erreur. Par exemple, l'exemple suivant supprime tous les fichiers plus anciens que le fichier `000000010000003700000010` :

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanup suppose que *`RépArchive`* est un répertoire lisible et modifiable par l'utilisateur qui a lancé le serveur.

F.22.2. Options de pg_archivecleanup

pg_archivecleanup accepte les arguments suivants en ligne de commande :

`-d`
: Affiche des informations de débogage sur `stderr`.

F.22.3. Exemples

Sur des systèmes Linux ou Unix, vous pouvez utiliser :

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

Dans ce cas, le répertoire des archives est situé physiquement sur le serveur en attente. `archive_command` y accède via un montage NFS mais les fichiers sont accessibles directement sur le serveur en attente. La commande va :

- produire une sortie de débogage dans `cleanup.log`
- supprimer les fichiers qui ne sont plus nécessaires à partir du répertoire d'archivage

F.22.4. Versions serveurs supportés

pg_archivecleanup est conçu pour fonctionner dès la version 8.0 de PostgreSQL™ ainsi que sur toutes les versions ultérieures quand il est utilisé de façon autonome. Il n'est utilisable comme commande de nettoyage des archives qu'à partir de la version 9.0 de PostgreSQL™.

F.22.5. Auteur

Simon Riggs `<simon@2ndquadrant.com>`

F.23. pgbench

pgbench est un programme simple qui exécute un test de performances (*benchmark* en anglais) sur PostgreSQL™. Il exécute la même séquence de commandes SQL de nombreuses fois, si possible dans plusieurs sessions en parallèle, puis calcule le taux de transaction moyen (transactions par seconde). Par défaut, pgbench teste un scénario vaguement basé sur TPC-B, impliquant cinq **SELECT**, **UPDATE** et **INSERT** par transaction. Néanmoins il est facile de tester d'autres cas en écrivant ses propres fichiers de transaction.

Voici un exemple d'affichage de pgbench :

```
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

Les six premières lignes indiquent simplement certains des paramètres les plus importants. La ligne suivant rapporte le nombre de transactions terminées et souhaitées (le dernier étant juste le produit du nombre de clients avec le nombre de transactions par client) ; ils doivent être égaux sauf si un échec est arrivé avant la fin. Les deux dernières lignes précisent le taux de transactions, avec et sans le temps de lancement de la session.

F.23.1. Aperçu

Le test de transaction par défaut, ressemblant à TPC-B, nécessite des tables particulières qu'il faut créer. pgbench doit être appelé avec l'option `-i` (initialisation) pour les créer et les peupler. (Cette étape n'est pas nécessaire dans le cas de script personnalisé, mais la base doit être configurée en conséquence.) L'initialisation ressemble à :

```
pgbench -i [ autres-options ] nom_base
```

où *`nom_base`* est le nom d'une base de données déjà créée et à utiliser pour les tests. (Les options `-h`, `-p`, et/ou `-U` peuvent être utilisées pour indiquer la façon de se connecter au serveur de bases de données.)

> **Attention**
>
> `pgbench -i` crée quatre tables : pgbench_accounts, pgbench_branches, pgbench_history et pgbench_tellers, détruisant toute table existante de ce nom. Il convient d'y être attentif s'il existe des tables de même nom !

Avec un « facteur d'échelle » de 1, les tables contiennent initialement ce nombre de lignes :

```
table                   # de lignes
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

Le nombre de lignes peut être augmenté en utilisant l'option `-s` (facteur d'échelle). L'option `-F` (facteur de remplissage) peut aussi être utilisée à ce moment.

Une fois la configuration exécutée, le test de performance peut être lancé avec une commande qui n'inclut pas l'option `-i`, c'est-à-dire :

```
pgbench [ options ] nom_base
```

Dans la plupart des cas, vous aurez besoin de quelques options pour que ce test soit réellement intéressant. Les options les plus importantes sont `-c` (nombre de clients), `-t` (nombre de transactions), `-T` (limite de temps) et `-f` (pour spécifier un script personnalisé). Voir ci-dessous pour une liste complète.

Section F.23.2, « Options de démarrage de pgbench » affiche les options qui sont utiles lors de l'initialisation de la base de données, alors que Section F.23.3, « Options de benchmarking pgbench » affiche celles qui sont utiles lors de l'exécution des tests de performance et Section F.23.4, « Options habituels de pgbench » affiches les options utiles dans les deux cas.

F.23.2. Options de démarrage de pgbench

pgbench comprend les options de la ligne de commande suivants :

`-F` *`facteur de remplissage`*
: Crée les tables pgbench_accounts, pgbench_tellers et pgbench_branches avec le facteur de remplissage donné. La valeur par défaut est de 100.

`-i`
: Réquis lorsqu'on veut utiliser le mode d'initialisation.

`-s` *`facteur d'echelle`*
: Multiplie le nombre de rangées par le facteur d'echelle. Par exemple, `-s 100` créera 10,000,000 rangées dans la table pgbench_accounts. La valeur par défaut est de 1.

F.23.3. Options de benchmarking pgbench

pgbench comprend les options de la ligne de commande suivants :

`-c` *`clients`*
: Nombre de clients simulés et donc le nombre de sessions concurrentes. La valeur par défaut est de 1.

`-C`
: Établit une nouvelle connexion pour chaque transaction, plutôt que le faire une fois par session. Ceci est utile pour mésurer les ressources consommées par la connexion.

`-d`
: Afficher la sortie de déboguage.

`-D` *`varname=value`*
: Définit une variable pouvant être utilisé par un script personalisé (voir plus bas). Plusieurs options `-D` peuvent être utilisées.

`-f` *`fichier`*
: Lire le script de transaction à partir de *`fichier`*. Voir plus bas pour les détails. `-N`, `-S` et `-f` ne peuvent pas être utilisés en même temps.

`-j` *`threads`*
: Nombre de threads au sein de pgbench. L'utilisation de plus d'un thread peut être utile sur des machines multi-processeurs. Le nombre de client doit être un multiple du nombre de threads puisqu'on donne à chaque thread le même nombre de sessions

clients à gérer. La valeur par défaut est de 1.

`-l`
Écrire le temps pris par chaque transaction dans le fichier de log. Voir plus bas pour les détails.

`-M` *`mode de requête`*
Protocole à utiliser pour soumettre les requêtes au serveur :

- `simple` : utiliser le protocole de requêtes simples.
- `extended` : utiliser le protocole de requêtes étendues.
- `prepared`: utiliser le protocole de requêtes étendues avec instructions préparées.

Le protocole par defaut est celui de requêtes simples. (Voir Chapitre 46, Protocole client/serveur pour plus d'information.)

`-n`
Ne pas faire de vacuuming avant de lancer le test. Cette option est *obligatoire* si vous lancez un scenario de test personalisé qui n'inclut pas les tables standards pgbench_accounts, pgbench_branches, pgbench_history et pgbench_tellers.

`-N`
Ne pas mettre à jour pgbench_tellers et pgbench_branches. Ceci évitera les problemes de mises à jour sur ces tables mais rendra les tests moins semblables à TPC-B.

`-s` *`scale_factor`*
Rapporter le facteur d'echelle spécifié dans la sortie de pgbench. Avec les tests intégrés par défaut, ceci n'est pas necessaie ; le facteur sera déduit en comptant les nombres de rangées dans la table pgbench_branches. Par contre, lors que vous testerez des benchmarks personalisés (avec l'option `-f`), le facteur rapporté sera de 1 sauf si cette option est utilisée.

`-S`
Faire des tests à base de select uniquement plutôt que TPC-B.

`-t` *`transactions`*
Le nombre de transactions que chaque client doit jouer. Le défaut est de 10.

`-T` *`secondes`*
Lancer ce test autant de secondes, plitôt qu'un nombre fixe de transactions par clients. `-t` et `-T` ne peuvent pas être utilisés en même temps.

`-v`
Faire un vacuum sur les quatre tables standard avant de lancer le test. Sans `-n` ou `-v`, pgbench fera un vacuum sur les tables pgbench_tellers et pgbench_branches et tronquera pgbench_history.

F.23.4. Options habituels de pgbench

pgbench reconnait les options habituels suivantes de ligne de commande :

`-h` *`nom de machine`*
Le nom de machine du serveur base de données

`-p` *`port`*
Le numéro de port du serveur base de données

`-U` *`login`*
Le nom avec lequel se connecter

F.23.5. Quelle « transaction » est réalisée dans pgbench ?

Le script de transaction par défaut exécute sept commandes :

1. `BEGIN;`
2. `UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;`
3. `SELECT abalance FROM pgbench_accounts WHERE aid = :aid;`
4. `UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;`
5. `UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;`

6. `INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);`
7. `END;`

Si vous indiquez `-N`, les étapes 4 et 5 ne sont pas inclus dans la transaction. Si vous indiquez `-S`, seul le **SELECT** est exécuté.

F.23.6. Scripts personnalisés

pgbench sait exécuter des scénarios personnalisés en remplaçant le script de transaction par défaut (décrit ci-dessus) avec un script de transaction lu à partir d'un fichier (option `-f`). Dans ce cas, une « transaction » compte en tant qu'une exécution du fichier. Vous pouvez même indiquer plusieurs scripts (avec plusieurs options `-f`), auquel cas un script est choisi au hasard à chaque fois qu'une session client exécute une nouvelle transaction.

Le fichier doit contenir une commande par ligne ; les commandes SQL multi-lignes ne sont pas acceptées. Les lignes vides et les lignes commençant par `--` sont ignorées. Les lignes du fichier peuvent aussi contenir des « meta commandes », qui sont interprétées par pgbench lui-même, comme décrit ci-dessous.

Il existe une fonctionnalité de substitution de variables pour les fichiers. Les variables sont configurables par l'option `-D` en ligne de commande, comme expliqué ci-dessus, ou par les méta-commandes expliquées ci-dessous. En plus des variables pré-initialisées par les options `-D`, la variable `scale` est pré-initialisée avec le facteur d'échelle actuel. Une fois configurée, la valeur d'une variable peut être insérée dans une commande SQL en écrivant `:`*`nom_variable`*. Lors de l'exécution de plusieurs sessions clients, chaque session a son propre ensemble de variables.

Les méta-commandes du script commencent par un antislash (`\`). Les arguments d'une méta-commande sont séparés par des espaces blancs. Voici la liste des méta-commandes acceptées :

`\set` *`nom_variable operande1`* `[` *`operateur operande2`* `]`
: Initialise la variable *`varname`* avec une valeur entière calculée. Chaque *`operande`* est soit une constante entière soit une référence `:`*`nom_variable`* à une variable entière. L'*`operateur`* peut être `+`, `-`, `*` ou `/`.

  Exemple :

  ```
  \set ntellers 10 * :scale
  ```

`\setrandom` *`nom_variable min max`*
: Initialise la variable *`nom_variable`* à une valeur entière prise au hasard entre les limites *`min`* et *`max`*, limites inclues. Chaque limite peut être soit une constante entière soit une référence `:`*`nom_variable`* à une valeur entière.

  Exemple :

  ```
  \setrandom aid 1 :naccounts
  ```

`\sleep` *`nombre`* `[ us | ms | s ]`
: Provoque un endormissement de l'exécution du script pour la durée indiquée en microsecondes (`us`), millisecondes (`ms`) ou secondes (`s`). Si l'unité est omise, alors ce seront par défaut des secondes. *`nombre`* peut être soit un entier soit une référence `:`*`nom_variable`* à un entier.

  Exemple :

  ```
  \sleep 10 ms
  ```

`\setshell` *`nom_variable commande`* `[` *`argument ...`* `]`
: Configure la variable *`nom_variable`* avec le résultat de la commande shell *`commande`*. La commande doit renvoyer une valeur entière via sa sortie standard.

  `argument` peut être soit une constante de type texte soit une référence `:`*`nom_variable`* à une variable de n'importe quel type. Si vous voulez utiliser *`argument`* en commençant avec un signe deux-points, vous devez ajouer un signe deux-points supplémentaires au début de *`argument`*.

  Exemple :

```
\setshell variable_to_be_assigned command literal_argument :variable
::literal_starting_with_colon
```

`\shell` *`commande`* `[` *`argument`* `... ]`
Idem à `\setshell`, mais le résultat est ignoré.

Exemple :

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

En exemple, voici la définition complète d'une transaction style TPC-B :

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid,
:delta, CURRENT_TIMESTAMP);
END;
```

Ce script permet à chaque itération de la transaction de référencer des lignes différentes, prises au hasard. (Cet exemple montre aussi pourquoi il est imporant que chaque session client ait ses propres variables -- sinon elles ne traiteraient pas des lignes différentes.)

F.23.7. Journalisation par transaction

Avec l'option `-l`, pgbench écrit le temps pris par chaque transaction dans un journal applicatif. Le journal sera nommé `pgbench_log.`*`nnn`*, où *`nnn`* est le PID du processus pgbench. Si l'option `-j` vaut 2 ou plus, ce qui va créer plusieurs threads de travail, chacun d'entre eux aura son propre journal applicatif. Le premier utilise le même nom de fichier que s'il avait été seul. Les autres fichiers des autres travailleurs seront nommés `pgbench_log.`*`nnn.mmm`*, où *`mmm`* est un numéro séquentiel pour chaque travailleur et commençant à 1.

Le format de ce journal est :

```
client_id transaction_no time file_no time_epoch time_us
```

où *`time`* est le durée de la transaction en microsecondes, *`file_no`* identifie le script qui a été utilisé (utile quand plusieurs scripts sont indiqués avec `-f`) et *`time_epoch`*/*`time_us`* sont une date/heure au format epoch Unix et un décalage en microsecondes (convenable pour la création d'un horodatage ISO 8601 avec des secondes en fration) indiquant la date et heure de la fin de la transaction.

Voici un exemple de journal :

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

F.23.8. Bonnes practiques

Il est très facile d'utiliser pgbench pour produire des nombres sans signification. Voici quelques lignes de conduite pour vous aider à obtenir des résultats intéressants.

En premier lieu, ne *jamais* croire tout test qui ne s'exécute que pendant quelques secondes. Utilisez l'option `-t` ou `-T` pour que le test dure plusieurs minutes pour rendre le bruit insignifiant. Dans certains cas, nous avez besoin de quelques heures pour obtenir des chiffres reproductibles. Exécuter le test plusieurs fois est une bonne idée pour savoir si vos résultats sont reproductibles.

Pour le scénario par défaut, style TPC-B, le facteur d'échelle à l'initialisation (`-s`) doit être au moins aussi important que le plus grand nombre de clients que vous souhaitez supporter (`-c`) ; sinon vous ne ferez que mesurer la contention des mises à jour. Il n'y a que `-s` lignes dans la table pgbench_branches, et chaque transaction veut en mettre une à jour, donc les valeurs `-c` supérieures à `-s` résulteront sans doute en beaucoup de transactions bloquées, en attente d'autres transactions.

Le scénario test par défaut est aussi assez sensible du moment où les tables ont été initialisées : une accumulation de lignes morts et d'espace vide dans les tables modifient les résultats. Pour comprendre les résultats, vous devez garder trace de nombre total de mises à jour et des moments où un VACUUM est exécuté. Si l'autovacuum est activé, cela peut causer des modifications non prévisibles dans les performances mesurées.

Une limite de pgbench est qu'il peut devenir lui-même le goulot d'étranglement lors de tentative de tests d'un grand nombre de sessions clients. Ceci peut se voir allégé en exécutant pgbench sur une autre machine que le serveur de bases de données, bien que la latence du réseau est essentielle. Il pourrait même être utile d'exécuter plusieurs instances pgbench en parallèle sur plusieurs machines client, pour le même serveur de bases de données.

F.24. pg_buffercache

Le module `pg_buffercache` fournit un moyen pour examiner ce qui se passe dans le cache partagé en temps réel.

Ce module propose une fonction C, `pg_buffercache_pages`, qui renvoie un ensemble d'enregistrements, ainsi qu'une vue, pg_buffercache, qui englobe la fonction pour une utilisation facilitée.

Par défaut, l'accès publique est refusé pour ces deux objets au cas où une faille de sécurité serait présente.

F.24.1. La vue pg_buffercache

Voici la définition des colonnes exposées par la vue affichée dans Tableau F.14, « Colonnes de pg_buffercache » :

Tableau F.14. Colonnes de pg_buffercache

Nom	Type	Références	Description
bufferid	integer		ID, qui va de 1 à `shared_buffers`
relfilenode	oid	`pg_class.relfilenode`	Numéro filenode de la relation
reltablespace	oid	`pg_tablespace.oid`	OID du tablespace de la relation
reldatabase	oid	`pg_database.oid`	OID de la base de données de la relation
relblocknumber	bigint		Numéro de page dans la relation
relforknumber	smallint		Numéro du fork dans la relation
isdirty	boolean		Page modifiée ?
usagecount	smallint		Compteur LRU de la page

Il y a une ligne pour chaque tampon dans le cache partagé. Les tampons inutilisés sont affichés avec des champs NULL sauf pour *bufferid*. Les catalogues systèmes partagés sont affichés comme appartenant à la base de données zéro.

Comme le cache est partagé par toutes les bases de données, il y aura des pages de relations n'appartenant pas à la base de données courante. Cela signifie qu'il pourrait y avoir des lignes sans correspondance dans pg_class, ou qu'il pourrait y avoir des jointures incorrectes. Si vous essayez une jointure avec pg_class, une bonne idée est de restreindre la jointure aux lignes ayant un *reldatabase* égal à l'OID de la base de données actuelle ou à zéro.

Quand la vue pg_buffercache est utilisée, les verrous du gestionnaire de tampons sont posés suffisamment longtemps pour copier toutes les données d'état du tampon que la vue affichera. Ceci assure à la vue un ensemble cohérent des résultats tout en évitant un blocage trop important de l'activité des tampons. Néanmoins, cela pourrait avoir un impact sur la performance de la base de données si cette vue est lue fréquemment.

F.24.2. Affichage en sortie

```
regression=# SELECT c.relname, count(*) AS buffers
             FROM pg_buffercache b INNER JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             GROUP BY c.relname
             ORDER BY 2 DESC
             LIMIT 10;

             relname             | buffers
---------------------------------+---------
 tenk2                           |     345
 tenk1                           |     141
 pg_proc                         |      46
 pg_class                        |      45
 pg_attribute                    |      43
 pg_class_relname_nsp_index      |      30
 pg_proc_proname_args_nsp_index  |      28
 pg_attribute_relid_attnam_index |      26
 pg_depend                       |      22
 pg_depend_reference_index       |      20
(10 rows)
```

F.24.3. Auteurs

Mark Kirkwood `<markir@paradise.net.nz>`

Suggestions de conception : Neil Conway `<neilc@samurai.com>`

Conseils pour le débogage : Tom Lane `<tgl@sss.pgh.pa.us>`

F.25. pgcrypto

Le module `pgcrypto` propose des fonctions de cryptographie pour PostgreSQL™.

F.25.1. Fonctions de hachage généralistes

F.25.1.1. `digest()`

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Calcule un hachage binaire de *`data`*. *`type`* est l'algorithme utilisé. Les algorithmes standards sont `md5` et `sha1`. Si `pgcrypto` a été construit avec OpenSSL, d'autres algorithmes sont disponibles comme le détaille Tableau F.18, « Résumé de fonctionnalités avec et sans OpenSSL ».

Si vous voulez en résultat une chaîne hexadécimale, utilisez `encode()` sur le résultat. Par exemple :

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
  SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.25.1.2. `hmac()`

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key text, type text) returns bytea
```

Calcule un MAC haché sur *`data`* avec la clé *`key`*. *`type`* est identique à `digest()`.

C'est similaire à `digest()` mais le hachage peut être recalculé en connaissant seulement la clé. Ceci évite le scénario où quelqu'un modifie les données et le hachage en même temps.

Si la clé est plus grosse que le bloc haché, il sera tout d'abord haché puis le résultat sera utilisé comme clé.

F.25.2. Fonctions de hachage de mot de passe

Les fonctions `crypt()` et `gen_salt()` sont spécialement conçues pour hacher les mots de passe. `crypt()` s'occupe du hachage et `gen_salt()` prépare les paramètres de l'algorithme pour ça.

Les algorithmes de `crypt()` diffèrent des algorithmes de hachage habituels comme MD5 ou SHA1 :

1. Ils sont lents. Comme la quantité de données est petite, c'est le seul moyen de rendre difficile la découverte par la force des mots de passe.
2. Ils incluent une valeur aléatoire appelée sel (*salt* en anglais) avec le résultat, pour que les utilisateurs qui ont le même mot de passer puissent avoir des mots de passe chiffrés différents. C'est aussi une défense supplémentaire comme l'inversion de l'algorithme.
3. Ils incluent le type de l'algorithme dans le résultat pour que les mots de passe hachés avec différents algorithmes puissent coexister.
4. Certains s'adaptent. Cela signifie que, une fois que les ordinateurs iront plus vite, vous pourrez configurer l'algorithme pour qu'il soit plus lent, ceci sans introduire d'incompatibilité avec les mots de passe existant.

Tableau F.15, « Algorithmes supportés par `crypt()` » liste les algorithmes supportés par la fonction `crypt()`.

Tableau F.15. Algorithmes supportés par `crypt()`

Algorithme	Longueur maximum du mot de passe	Adaptif ?	Bits sel	Description
`bf`	72	oui	128	Basé sur Blowfish, variante 2a
`md5`	unlimited	non	48	crypt() basé sur MD5
`xdes`	8	oui	24	DES étendu
`des`	8	non	12	crypt original UNIX

F.25.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

Calcule un hachage de mot de passe (*`password`*) d'après crypt(3) UN*X. Lors du stockage d'un nouveau mot de passe, vous devez utiliser la fonction `gen_salt()` pour générer un nouveau sel (*`salt`*). Lors de la vérification de mot de passe, passez la valeur hachée stockée *`salt`*, et testez si le résultat correspond à la valeur stockée.

Exemple d'ajout d'un nouveau mot de passe :

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

Exemple d'authentification :

```
SELECT pswhash = crypt('entered password', pswhash) FROM ... ;
```

Ceci renvoie `true` si le mot de passe saisi est correct.

F.25.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

Génère une nouvelle valeur aléatoire sel pour son utilisation avec `crypt()`. La chaîne sel indique aussi à `crypt()` l'algorithme à utiliser.

Le paramètre *`type`* précise l'algorithme de hachage. Les types acceptées sont : `des`, `xdes`, `md5` et `bf`.

Le paramètre *`iter_count`* laisse l'utilisateur indiquer le nombre d'itération, pour les algorithmes qui en ont. Plus le nombre est important, plus le hachage du mot de passe prendra du temps, et du coup plus le craquage du mot de passe prendre du temps. Cela étant dit, un nombre trop important rend pratiquement impossible le calcul du hachage. Si le paramètre *`iter_count`* est omis, le nombre d'itération par défaut est utilisé. Les valeurs autorisées pour *`iter_count`* dépendent de l'algorithme et sont affichées dans Tableau F.16, « Nombre d'itération pour `crypt()` ».

Tableau F.16. Nombre d'itération pour `crypt()`

Algorithme	Par défaut	Min	Max
`xdes`	725	1	16777215
`bf`	6	4	31

Pour `xdes`, il existe une limite supplémentaire qui fait que ce nombre doit être un nombre impair.

Pour utiliser un nombre d'itération approprié, pensez que la fonction crypt DES original a été conçu pour avoir la vitesse de quatre hachages par seconde sur le matériel de l'époque. Plus lent que quatre hachages par secondes casserait probablement la facilité d'utilisation. Plus rapide que cent hachages à la seconde est probablement trop rapide.

Tableau F.17, « Vitesse de l'algorithm de hachage » donne un aperçu de la lenteur relative de différents algorithmes de hachage. La table montre le temps que prendrait le calcul de toutes les combinaisons réalisables pour un mot de passe sur huit caractères, en supposant que le mot de passe contient soit que des lettres minuscules, soit des lettres minuscules et majuscules et des chiffres. Dans les entrées `crypt-bf`, le nombre après un slash est le paramètre *`iter_count`* de `gen_salt`.

Tableau F.17. Vitesse de l'algorithm de hachage

Algorithme	Hachages/sec	Pour `[a-z]`	Pour `[A-Za-z0-9]`
`crypt-bf/8`	28	246 years	251322 years
`crypt-bf/7`	57	121 années	123457 années
`crypt-bf/6`	112	62 années	62831 années
`crypt-bf/5`	211	33 années	33351 années
`crypt-md5`	2681	2.6 années	2625 années
`crypt-des`	362837	7 jours	19 années
`sha1`	590223	4 jours	12 années
`md5`	2345086	1 jour	3 années

Notes :

- La machine utilisée est un Pentium 4 à 1,5 GHz.
- Les numéros des algorithmes `crypt-des` et `crypt-md5` sont pris de la sortie du -test de John the Ripper v1.6.38.
- Les nombres`md5` font partie de mdcrack 1.2.
- Les nombres `sha1` font partie de lcrack-20031130-beta.
- Les nombres `crypt-bf` sont pris en utilisant le programme simple qui boucle sur 1000 mots de passe de huit caractères. De cette façon, je peux afficher la vitesse avec les différents nombres de tours. Pour référence : `john -test` affiche 213 tours/sec pour `crypt-bf/5`. (La petite différence dans les résultats est dû au fait que l'implémentation de `crypt-bf` dans `pgcrypto` est la même que celle utilisée dans John the Ripper.)

Notez que « tenter toutes les combinaisons » n'est pas un exercice réaliste. Habituellement, craquer les mots de passe se fait avec l'aide de dictionnaires contenant les mots standards et différentes variantes. Donc, même des mots de passe qui ressemblent vaguement à des mots peuvent être craqués plus rapidement que les nombres ci-dessus le suggèrent alors qu'un mot de passe sur six ca-

ractères qui ne ressemble pas à un mot pourrait ne pas être craqué.

F.25.3. Fonctions de chiffrement PGP

Les fonctions implémentent la partie chiffrement du standard OpenPGP (RFC 2440). Les chiffrements à clés symétriques et publiques sont supportés.

Un message PGP chiffré consiste en deux parties ou *paquets* :

- Un paquet contenant la clé de session -- soit une clé symétrique soit une clé publique chiffrée.
- Paquet contenant les données chiffrées avec la clé de session containing data encrypted with the session key.

Lors du chiffrement avec une clé symétrique (par exemple, un mot de passe) :

1. Le mot de passe est haché en utilisant l'algorithme String2Key (S2K). C'est assez similaire à l'algorithme `crypt()` -- lenteur voulue et nombre aléatoire pour le sel -- mais il produit une clé binaire de taille complète.
2. Si une clé de session séparée est demandée, une nouvelle clé sera générée au hasard. Sinon une clé S2K sera utilisée directement en tant que clé de session.
3. Si une clé S2K est à utiliser directement, alors seuls les paramètres S2K sont placés dans le paquet de session. Sinon la clé de session sera chiffrée avec la clé S2K et placée dans le paquet de session.

Lors du chiffrement avec une clé publique :

1. Une nouvelle clé de session est générée au hasard.
2. Elle est chiffrée en utilisant la clé public et placée dans le paquet de session.

Dans les deux cas, les données à chiffrer sont traitées ainsi :

1. Manipulation optionnelle des données : compression, conversion vers UTF-8, conversion de retours à la ligne.
2. Les données sont préfixées avec un bloc d'octets pris au hasard. C'est identique à l'utilisation de *random IV*.
3. Un hachage SHA1 d'un préfixe et de données au hasard est ajouté.
4. Tout ceci est chiffré avec la clé de la session et placé dans la paquet de données.

F.25.3.1. `pgp_sym_encrypt()`

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Chiffre *`data`* avec une clé PGP symétrique *`psw`*. Le paramètre *`options`* peut contenir des options décrites ci-dessous.

F.25.3.2. `pgp_sym_decrypt()`

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

Déchiffre un message PGP chiffré avec une clé symétrique.

Déchiffrer des données bytea avec `pgp_sym_decrypt` est interdit. Ceci a pour but d'éviter la sortie de données de type caractère invalides. Déchiffrer des données textuelles avec `pgp_sym_decrypt_bytea` ne pose pas de problème.

Le paramètre *`options`* peut contenir les paramètres décrits ci-dessous.

F.25.3.3. `pgp_pub_encrypt()`

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Chiffre *data* avec la clé PGP publique *key*. Avec cette fonction, une clé privée renverra une erreur.

Le paramètre *options* peut contenir des options décrites ci-dessous.

F.25.3.4. pgp_pub_decrypt()

```
    pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
    pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns
bytea
```

Déchiffre un message chiffré avec une clé publique. *key* doit être la clé secrète correspondant à la clé publique utilisée pour chiffrer. Si la clé secrète est protégée par un mot de passe, vous devez saisir le mot de passe dans *psw*. S'il n'y a pas de mot de passe mais que vous devez indiquer des options, vous devez saisir un mot de passe vide.

Déchiffrer des données bytea avec `pgp_pub_decrypt` est interdit. Ceci a pour but d'éviter la sortie de données de type caractère invalides. Déchiffrer des données textuelles avec `pgp_pub_decrypt_bytea` ne pose pas de problème.

Le paramètre *options* peut contenir des options décrites ci-dessous.

F.25.3.5. pgp_key_id()

```
    pgp_key_id(bytea) returns text
```

`pgp_key_id` extrait l'identifiant de la clé pour une clé PGP publique ou secrète. Ou il donne l'identifiant de la clé utilisé pour chiffrer les données si un message chiffré est fourni.

Elle peut renvoyer deux identifiants de clés spéciaux :

- `SYMKEY`

  Le message est chiffré avec une clé symétrique.

- `ANYKEY`

  La donnée est chiffrée avec une clé publique mais l'identifiant de la clé est effacé. Cela signifie que vous avez besoin d'essayer toutes les clés secrètes pour voir laquelle la déchiffre. `pgcrypto` ne réalise pas lui-même de tels messages.

Notez que des clés différentes peuvent avoir le même identifiant. C'est rare mais normal. L'application client doit alors essayer de déchiffer avec chacune d'elle pour voir laquelle correspond -- ce qui revient à la gestion de `ANYKEY`.

F.25.3.6. armor(), dearmor()

```
    armor(data bytea) returns text
    dearmor(data text) returns bytea
```

Ces fonctions enveloppent les données dans une armure ASCII PGP qui est basiquement en Base64 avec CRC et un formatage supplémentaire.

F.25.3.7. Options pour les fonctions PGP

Les options sont nommées de façon similaires à GnuPG. Les valeurs sont fournies après un signe d'égalité ; les options sont séparées par des virgules. Par exemple :

```
    pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

Toutes les options en dehors de `convert-crlf` s'appliquent seulement aux fonctions de chiffrement. Les fonctions de déchiffrement obtiennent des paramètres des données PGP.

Les options les plus intéressantes sont problablement `compression-algo` et `unicode-mode`. Le reste doit avoir des valeurs

par défaut raisonnables.

F.25.3.7.1. cipher-algo

Quel algorithme de chiffrement à utiliser.

```
Valeurs : bf, aes128, aes192, aes256 (OpenSSL seulement :
3des, cast5)
Par défaut : aes128
Applique à : pgp_sym_encrypt, pgp_pub_encrypt
```

F.25.3.7.2. compress-algo

Algorithme de compression à utiliser. Seulement disponible si PostgreSQL™ a été construit avec zlib.

```
Valeurs :
  0 - sans compression
  1 - compression ZIP
  2 -  compression ZLIB [=ZIP plus meta-data and block-CRC's]
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt
```

F.25.3.7.3. compress-level

Niveau de compression. Les grands niveaux compressent mieux mais sont plus lents. 0 désactive la compression.

```
Valeurs : 0, 1-9
Par défaut : 6
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt
```

F.25.3.7.4. convert-crlf

Précise si `\n` doit être converti en `\r\n` lors du chiffrement et `\r\n` en `\n` lors du déchiffrement. La RFC 4880 spécifie que les données texte doivent être stockées en utilisant les retours chariot `\r\n`. Utilisez cette option pour obtenir un comportement respectant la RFC.

```
Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt,
pgp_pub_decrypt
```

F.25.3.7.5. disable-mdc

Ne protège pas les données avec SHA-1. La seule bonne raison pour utiliser cette option est d'avoir une compatibilité avec les anciens produits PGP précédant l'ajout de paquets protégés SHA-1 dans la RFC 4880. Les versions récentes des logiciels de gnupg.org et pgp.com le supportent.

```
Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt
```

F.25.3.7.6. enable-session-key

Utilise la clé de session séparée. Le chiffrement par clé publique utilise toujours une clé de session séparée, c'est pour le chiffrement de clé symétrique, qui utilise directement par défaut S2K.

```
Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt
```

F.25.3.7.7. s2k-mode

Algorithme S2K à utiliser.

```
Valeurs :
  0 - Sans sel. Dangereux !
  1 - Avec sel mais avec un décompte fixe des itérations.
  3 - Décompte variables des itérations.
Par défaut : 3
S'applique à : pgp_sym_encrypt
```

F.25.3.7.8. s2k-digest-algo

Algorithme digest à utiliser dans le calcul S2K.

```
Valeurs : md5, sha1
Par défaut : sha1
S'applique à : pgp_sym_encrypt
```

F.25.3.7.9. s2k-cipher-algo

Chiffrement à utiliser pour le chiffrage de la clé de session séparée.

```
Valeurs : bf, aes, aes128, aes192, aes256
Par défaut : use cipher-algo
S'applique à : pgp_sym_encrypt
```

F.25.3.7.10. unicode-mode

Sélection de la conversion des données texte à partir de l'encodage interne de la base vers l'UTF-8 et inversement. Si votre base de données est déjà en UTF-8, aucune conversion ne sera réalisée, seules les données seront marquées comme étant en UTF-8. Sans cette option, cela ne se fera pas.

```
Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt
```

F.25.3.8. Générer des clés PGP avec GnuPG

Pour générer une nouvelle clé :

```
gpg --gen-key
```

Le type de clé préféré est « DSA and Elgamal ».

Pour le chiffrement RSA, vous devez créer soit une clé de signature seulement DSA ou RSA en tant que maître, puis ajouter la sous-clé de chiffrement RSA avec `gpg --edit-key`.

Pour lister les clés :

```
gpg --list-secret-keys
```

Pour exporter une clé publique dans un format armure ASCII :

```
gpg -a --export KEYID > public.key
```

Pour exporter une clé secrète dans un format armure ASCII :

```
gpg -a --export-secret-keys KEYID > secret.key
```

Vous avez besoin d'utiliser la fonction `dearmor()` sur ces clés avant de les passer aux fonctions PGP. Ou si vous gérez des données binaires, vous pouvez supprimer l'option `-a` pour la commande.

Pour plus de détails, voir la `page de référence de gpg`, le *livre « GNU Privacy Handbook »* et d'autres documents sur le *site gnupg.org*.

F.25.3.9. Limites du code PGP

- Pas de support des signatures. Cela signifie aussi qu'on ne peut pas vérifier si la sous-clé de chiffrage appartient bien à la clé maître.
- Pas de support de la clé de chiffrement en tant que clé maître. Cela ne devrait pas être un problème étant donné que cette pratique n'est pas encouragée.
- Pas de support pour plusieurs sous-clés. Ceci peut être un problème car c'est une pratique courante. D'un autre côté, vous ne devez pas utiliser vos clés GPG/PGP habituelles avec `pgcrypto`, mais en créer de nouvelles car l'utilisation est assez différente.

F.25.4. Fonctions de chiffrement brut (Raw)

Ces fonctions exécutent directement un calcul des données ; ils n'ont pas de fonctionnalités avancées de chiffrement PGP. Du coup, ils ont les problèmes majeurs suivant :

1. Elles utilisent directement la clé de l'utilisateur comme clé de calcul.
2. Elles ne fournissent pas une vérification de l'intégrité pour savoir si les données chiffrées ont été modifiées.
3. Elles s'attendent à ce que les utilisateurs gèrent eux-même tous les paramètres du chiffrement, même IV.
4. Elles ne gèrent pas le texte.

Donc, avec l'introduction du chiffrement PGP, l'utilisation des fonctions de chiffrement brut n'est pas encouragée.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

Chiffrer/déchiffrer les données en utilisant la méthode de calcul spécifiée par *`type`*. La syntaxe de la chaîne *`type`* est :

```
algorithm [ - mode ] [ /pad: padding ]
```

où *`algorithm`* fait partie de :

- `bf` -- Blowfish
- `aes` -- AES (Rijndael-128)

et *mode* fait partie de :

- `cbc` -- le bloc suivant dépend du précédent. (par défaut)
- `ecb` -- chaque bloc est chiffré séparément. (seulement pour les tests)

et *padding* fait partie de :

- `pkcs` -- les données peuvent avoir n'importe quelle longueur (par défault)
- `none` -- les données doivent être des multiples de la taille du bloc de calcul.

Donc, pour exemple, ces derniers sont équivalents :

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

Dans `encrypt_iv` et `decrypt_iv`, le paramètre *iv* est la valeur initiale pour le mode CBC ; elle est ignorée pour ECB. Elle est remplie de zéro pour l'alignement si la taille de données ne correspond à un multiple de la taille du bloc. It defaults to all zeroes in the functions without this parameter.

F.25.5. Fonctions d'octets au hasard

```
gen_random_bytes(count integer) returns bytea
```

Renvoie *count*) octets pour un chiffrement fort. Il peut y avoir au maximum 1024 octets extrait à un instant t, ceci pour éviter de vider le contenu du générateur de nombres aléatoires.

F.25.6. Notes

F.25.6.1. Configuration

`pgcrypto` se configure lui-même suivant les découvertes du scrip `configure` principal de PostgreSQL. Les options qui l'affectent sont `--with-zlib` et `--with-openssl`.

Quand il est compilé avec zlib, les fonctions de chiffrement PGP peuvent compresser les données avant chiffrement.

Quand il est compilé avec OpenSSL, plus d'algorithmes seront disponibles. De plus, les fonctions de chiffrement à clé publique seront plus rapides car OpenSSL a des fonctions BIGNUM plus optimisées.

Tableau F.18. Résumé de fonctionnalités avec et sans OpenSSL

Fonctionnalité	Interne	Avec OpenSSL
MD5	oui	oui
SHA1	oui	oui
SHA224/256/384/512	oui	oui (Note 1)
D'autres algorithmes digest	non	oui (Note 2)
Blowfish	oui	oui
AES	oui	oui (Note 3)
DES/3DES/CAST5	non	oui
Raw encryption	oui	oui
PGP Symmetric encryption	oui	oui
PGP Public-Key encryption	oui	oui

Notes :

1. Les algorithmes SHA2 ont été ajoutés à OpenSSL version 0.9.8. Pour les anciennes versions, `pgcrypto` utilisera du code interne.
2. Tout algorithme digest qu'OpenSSL supporte est automatiquement choisi. Ce n'est pas possible avec les chiffreurs qui doivent être supportés explicitement.
3. AES est inclus dans OpenSSL depuis la version 0.9.7. Pour les anciennes versions, `pgcrypto` utilisera du code interne.

F.25.6.2. Gestion des NULL

Comme le standard SQL le demande, toutes les fonctions renvoient NULL si un des arguments est NULL. Cela peut permettre une faille de sécurité si c'est utilisé sans précaution.

F.25.6.3. Limites de la sécurité

Toutes les fonctions de `pgcrypto` sont exécutées au sein du serveur de bases de données. Cela signifie que toutes les données et les mots de passe sont passés entre `pgcrypto` et l'application client en texte clair. Donc, vous devez :

1. Vous connecter localement ou utiliser des connexions SSL ;
2. Faire confiance à votre administrateur système et de base de données.

Si vous ne le pouvez pas, alors il est préférable de chiffrer directement au sein de l'application client.

F.25.6.4. Lectures intéressantes

- *http://www.gnupg.org/gph/en/manual.html*

  The GNU Privacy Handbook.
- *http://www.openwall.com/crypt/*

  Décrit l'algorithme crypt-blowfish.
- *http://www.stack.nl/~galactus/remailers/passphrase-faq.html*

  Comment choisir un bon mot de passe.
- *http://world.std.com/~reinhold/diceware.html*

  Idée intéressante pour choisir des mots de passe.
- *http://www.interhack.net/people/cmcurtin/snake-oil-faq.html*

  Décrit la bonne et la mauvaise cryptographie.

F.25.6.5. Références tecyhniques

- *http://www.ietf.org/rfc/rfc4880.txt*

  Format du message OpenPGP.
- *http://www.ietf.org/rfc/rfc1321.txt*

  Algorithme MD5.
- *http://www.ietf.org/rfc/rfc2104.txt*

  HMAC: Keyed-Hashing for Message Authentication.
- *http://www.usenix.org/events/usenix99/provos.html*

  Comparaison des algorithmes crypt-des, crypt-md5 et bcrypt.
- *http://csrc.nist.gov/cryptval/des.htm*

  Standards pour DES, 3DES et AES.
- *http://en.wikipedia.org/wiki/Fortuna_(PRNG)*

  Description de Fortuna CSPRNG.

- *http://jlcooke.ca/random/*

  Jean-Luc Cooke Fortuna-based `/dev/random` driver for Linux.
- *http://research.cyber.ee/~lipmaa/crypto/*

  Collection de pointeurs sur le chiffrement.

F.25.7. Auteur

Marko Kreen `<markokr@gmail.com>`

`pgcrypto` utilise du code provenant des sources suivantes :

Algorithme	Auteur	Origine du source
DES crypt	David Burren and others	FreeBSD libcrypt
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com
Blowfish cipher	Simon Tatham	PuTTY
Rijndael cipher	Brian Gladman	OpenBSD sys/crypto
MD5 and SHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.26. pg_freespacemap

Le module `pg_freespacemap` fournit un moyen pour examiner la carte des espaces libres (*free space map*, FSM). Il fournit une fonction appelée `pg_freespace`, ou plutôt deux fonctions qui se surchargent. Les fonctions indiquent la valeur enregistrée dans la carte des espaces libres pour une page donnée ou pour toutes les pages de la relation.

Par défaut, l'accès publique est refusé pour ces fonctions, au cas il y aurait des problèmes de sécurité.

F.26.1. Fonctions

`pg_freespace(rel regclass IN, blkno bigint IN) returns int2`
: Renvoit la quantité d'espace libre dans la page de la relation, spécifiée par `blkno`, suivant la FSM.

`pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)`
: Affiche la quantité d'espace libre sur chaque page de la relation suivant la FSM. Un ensemble de lignes du type `(blkno bigint, avail int2)` est renvoyé, une ligne pour chaque page de la relation.

Les valeurs stockées dans la carte des espaces libres ne sont pas exactes. Elles sont arrondies à une précision de 1/256th du BLCKSZ (32 octets pour un BLCKSZ par défaut), et elles ne sont pas mises à jour complètement car des lignes sont insérées et mises à jour.

Pour les index, ne sont tracées que les pages entièrement inutilisées, et non pas les pages ayant un peu d'espace vide. Du coup, les valeurs ne sont pas significatives. Elles indiquent simplement si la page est remplie ou vide.

Note : l'interface a été modifiée en 8.4 pour refléter la nouvelle implémentation de la FSM introduite dans cette version.

F.26.2. Exemple de sortie

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
```

```
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.26.3. Auteur

Version originale par Mark Kirkwood <markir@paradise.net.nz>. Ré-écrit en version 8.4 pour s'adapter à la nouvelle implémentation de la FSM par Heikki Linnakangas <heikki@enterprisedb.com>

F.27. pgrowlocks

Le module `pgrowlocks` fournit une fonction pour afficher les informations de verrouillage de lignes pour une table spécifiée.

F.27.1. Aperçu

```
pgrowlocks(text) returns setof record
```

Le paramètre est le nom d'une table. Le résultat est un ensemble d'enregistrements, avec une ligne pour chaque ligne verrouillée dans la table. Les colonnes en sortie sont affichées dans Tableau F.19, « Colonnes de `pgrowlocks` ».

Tableau F.19. Colonnes de `pgrowlocks`

Nom	Type	Description
locked_row	tid	ID de ligne (TID) d'une ligne verrouillée
lock_type	text	`Shared` pour un verrou partagé, ou `Exclusive` pour un verrou exclusif
locker	xid	ID de transaction de la pose du verrou, ou ID multixact dans le case d'une multi-transaction
multi	boolean	True si le verrou a été posé dans une multi-transaction
xids	xid[]	ID de transaction des verrouilleurs (plus d'une dans le cas de multi-transaction)
pids	integer[]	ID de processus des serveurs qui ont posé des verrous (plus d'une dans le cas des multi-transactions)

`pgrowlocks` prend un verrou `AccessShareLock` pour la table cible et lit chaque ligne une par une pour récupérer les informations de verrouillage de lignes. Ce n'est pas très rapide pour une grosse table. Notez que :

1. Si la table entière est verrouillée exclusivement par quelqu'un d'autre, `pgrowlocks` sera bloqué.
2. `pgrowlocks` isne garantie pas de produire une image cohérente. Il est possible qu'un nouveau verrou de ligne est pris ou qu'un ancien verrou est libéré lors de son exécution.

`pgrowlocks` ne montre pas le contenu des lignes verrouillées. Si vous voulez jeter un œil au contenu de la ligne en même temps, vous pouvez le faire ainso :

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

Faites attention au fait que (depuis PostgreSQL™ 8.3) une telle requête sera particulièrement inefficace.

F.27.2. Exemple d'affichage

```
test=# SELECT * FROM pgrowlocks('t1');
 locked_row | lock_type | locker | multi |   xids    |     pids
------------+-----------+--------+-------+-----------+---------------
      (0,1) | Shared    |     19 | t     | {804,805} | {29066,29068}
      (0,2) | Shared    |     19 | t     | {804,805} | {29066,29068}
      (0,3) | Exclusive |    804 | f     | {804}     | {29066}
      (0,4) | Exclusive |    804 | f     | {804}     | {29066}
(4 rows)
```

F.27.3. Auteur

Tatsuo Ishii

F.28. pg_standby

pg_standby facilite la création d'un serveur en attente (« warm standby server »). Il est conçu pour être immédiatement utilisable, mais peut aussi être facilement personnalisé si vous en avez le besoin.

pg_standby s'utilise au niveau du paramètre `restore_command`. IL est utile pour transformer une récupération d'archives ordinaire en restauration en attente. Une autre configuration est nécessaire, elle est décrite dans le manuel du serveur (voir Section 25.2, « Serveurs de Standby par transfert de journaux »).

Les fonctionnalités de pg_standby incluent :

- Écrit en C, donc très portable et facile à installer
- Code source facile à modifier, avec des sections spécialement conçues pour modifier selon vos besoins
- Déjà testé sur Linux et Windows

F.28.1. Utilisation

Pour configurer un serveur en attente à utiliser pg_standby, placez ceci dans le fichier de configuration `recovery.conf` :

```
restore_command = 'pg_standby archiveDir %f %p %r'
```

où *`archiveDir`* est le répertoire à partir duquel les journaux de transaction seront restaurés.

La syntaxe complète de la ligne de commande de pg_standby est :

```
pg_standby [ option ... ] archivelocation nextwalfile xlogfilepath [ restartwalfile ]
```

Lorsqu'il est utilisé avec `restore_command`, les macros `%f` et `%p` doivent être spécifiées pour, respectivement, *`nextwalfile`* et *`xlogfilepath`*, ce qui fournit ainsi le fichier réel et le chemin requis pour la restauration.

Si *restartwalfile* est spécifié, normalement en utilisant la macro `%r`, alors tous les journaux de transactions précédant logiquement ce fichier seront supprimés de *archivelocation*. Ceci minimise le nombre de fichiers à conserver tout en préservant la possibilité de redémarrer après un crash. L'utilisation de ce paramètre est appropriée si *archivelocation* est une aire pour ce serveur en attente particulier mais ne convient *pas* quand *archivelocation* est prévu pour un archivage à long terme des journaux de transaction.

pg_standby suppose que *archivelocation* est un répertoire lisible par l'utilisateur qui exécute le serveur. Si *restartwalfile* (ou l'option `-k`) est spécifié, le répertoire *archivelocation* doit être accessible aussi en écriture.

Il existe deux façons de basculer un serveur « en attente » quand le maître échoue :

Bascule intelligente
: Dans une bascule intelligente, le serveur est disponible après avoir appliqué tous les fichiers des journaux de transactions dans l'archive. Cela résulte en une perte nulle, même si le serveur en attente n'était pas complètement à jour. Du coup, s'il restait beaucoup de journaux à ré-exécuter, cela peut prendre un long moment avant que le serveur en attente devienne disponible. Pour déclencher une bascule intelligente, créez un fichier trigger contenant le mot `smart`, ou créez-le en le laissant vide.

Bascule rapide
: Lors d'une bascule rapide, le serveur est disponible immédiatement. Tout journal de transaction non rejoué sera ignoré. Du coup, toutes les transactions contenues dans ces fichiers seront perdues. Pour déclencher une bascule rapide, créez un fichier trigger contenant le mot `fast`. pg_standby peut aussi être configuré pour basculer automatiquement si aucun nouveau journal de transactions n'apparaît dans un certain laps de temps.

F.28.2. pg_standby Options

pg_standby accepte les arguments suivantes en ligne de commande :

`-c`
: Utilise la commande `cp` ou `copy` pour restaurer les journaux de transaction à partir de l'archive. C'est le seul comportement supporté donc cette option est inutile.

`-d`
: Affiche de nombreux messages de débogage sur `stderr`.

`-k`
: Supprime les fichiers de *archivelocation* pour qu'il n'existe pas plus de ce nombre de journaux de transactions avant le journal actuel dans l'archive. Zéro (la valeur par défaut) signifie qu'il ne supprime aucun fichier de *archivelocation*. Ce paramètre sera ignoré si *restartwalfile* est spécifié car cette méthode de spécification est plus fiable dans la détermination du point correct de séparation des archives. L'utilisation de ce paramètre est *obsolète* dès PostgreSQL™ 8.3 ; il est préférable et plus efficace d'utiliser le paramètre *restartwalfile*. Une configuration trop basse pourrait résulter en des suppressions de journaux qui sont toujours nécessaire pour un relancement du serveur en attente alors qu'un paramétrage trop important aurait pour conséquence un gachis en espace disque.

`-r` *maxretries*
: Set the maximum number of times to retry the copy command if it fails (default 3). After each failure, we wait for *sleeptime* * *num_retries* so that the wait time increases progressively. So by default, we will wait 5 secs, 10 secs, then 15 secs before reporting the failure back to the standby server. This will be interpreted as end of recovery and the standby will come up fully as a result. Configure le nombre maximum de tentatives pour la commande de copie si celle-ci échoue. Après chaque échec, l'attente est de *sleeptime* * *num_retries* pour que le temps d'attente augmente progressivement. Donc, par défaut, l'outil attend 5 secondes, puis 10, puis 15 avant de rapporter l'échec au serveur en attente. Cela sera interprété comme une fin de récupération par le serveur en attente, ce qui aura pour conséquence que le serveur en attente deviendra disponible.

`-s` *sleeptime*
: Initialise le nombre de seconde (jusqu'à 60, par défaut 5) d'endormissement entre les tests pour voir si le journal de transactions à restaurer est disponible à partir de l'archive. La configuration par défaut n'est pas forcément recommandée ; consultez Section 25.2, « Serveurs de Standby par transfert de journaux » pour plus d'informations.

`-t` *triggerfile*
: Specify a trigger file whose presence should cause failover. It is recommended that you use a structured filename to avoid confusion as to which server is being triggered when multiple servers exist on the same system; for example `/tmp/pgsql.trigger.5432`. Spécifie un fichier trigger dont la présence cause une bascule du serveur maître. Il est recommandé que vous utilisiez un nom de fichier structuré pour éviter la confusion sur le serveur à déclencher au cas où plusieurs serveurs existent sur le même système ; par exemple `/tmp/pgsql.trigger.5442`.

`-w` *maxwaittime*

Configure le nombre maximum de secondes pour attendre le prochain journal de transactions, délai après lequel une bascule du maître est automatique exécuté. Une configuration à zéro (la valeur par défaut) fait qu'il attend indéfiniment. La valeur par défaut n'est pas forcément recommandée ; voir Section 25.2, « Serveurs de Standby par transfert de journaux » pour plus d'informations.

F.28.3. Exemples

Sur des systèmes Linux ou Unix, vous pouvez utiliser (le premier paramètre concerne le maître, le second concerne l'esclave) :

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r
2>>standby.log'

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

alors que le répertoire d'archive est situé physiquement sur le serveur en attente, de façon à ce que `archive_command` y accède via un montage NFS, mais les fichiers sont en local pour le serveur en attente (ce qui permet l'utilisation de `ln`).

- produit une sortie de débogage dans `standby.log`
- s'endort pour deux secondes entre les vérifications de disponibilité du prochain journal de transaction
- arrête l'attente seulement quand un fichier trigger nommé `/tmp/pgsql.trigger.5442` apparaît, et exécute la bascule suivant son contenu
- supprime le fichier trigger quand la restauration se termine
- supprime les fichiers inutiles du répertoire des archives

Sur Windows, vous pouvez utiliser :

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r
2>>standby.log'

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

Notez que les antislashs doivent être doublés dans `archive_command`, mais *pas* dans `restore_command` ou `recovery_end_command`. Cela va :

- utiliser la commande `copy` pour restaurer les journaux de transaction à partir de l'archive
- produire une sortie de débogage dans `standby.log`
- l'endormir pendant cinq secondes entre les vérifications de disponibilité du prochain journal de transaction
- arrêter l'attente seulement quand un fichier trigger nommé `C:\pgsql.trigger.5442` apparaît, et exécuter la bascule suivant son contenu
- supprimer le fichier trigger quand la restauration se termine
- supprimer les fichiers inutiles du répertoire des archives

La commande `copy` sur Windows configure la taille du fichier final avant que le fichier ne soit entièrement copié, ce qui pourrait gêner pg_standby. Du coup, pg_standby attend `sleeptime` secondes une fois qu'il a remarqué que le fichier faisait la bonne taille. `cp` de GNUWin32 configure la taille du fichier seulement lorsque la copie du fichier est terminée.

Comme l'exemple Windows utilise `copy` aux deux bouts, soit l'un soit les deux serveurs pourront accéder au répertoire d'archive via le réseau.

F.28.4. Versions serveur supportées

pg_standby est conçu pour fonctionner avec PostgreSQL™ 8.2 et ultérieurs.

PostgreSQL™ 8.3 fournit la macro `%r`, qui est conçue pour indiquer à pg_standby le dernier fichier qu'il a besoin de conserver. Avec PostgreSQL™ 8.2, l'option `-k` doit être utilisée si le nettoyage de l'archive est démandé. Cette option reste disponible en 8.3, mais est devenue obsolète.

PostgreSQL™ 8.4 fournit le paramètre `recovery_end_command`. Sans lui, il est possible de laisser un fichier trigger, ce qui comporte un risque.

F.28.5. Auteur

Simon Riggs `<simon@2ndquadrant.com>`

F.29. pg_stat_statements

Le module `pg_stat_statements` fournit un moyen de surveiller les statistiques d'exécution de tous les ordres SQL exécutés par un serveur.

Le module doit être chargé par l'ajout de `pg_stat_statements` à shared_preload_libraries dans `postgresql.conf` parce qu'il a besoin de mémoire partagée supplémentaire. Ceci signifie qu'il faut redémarrer le serveur pour ajouter ou supprimer le module.

F.29.1. La vue pg_stat_statements

Les statistiques collectées par le module sont rendues disponibles par une vue système nommée pg_stat_statements. Cette vue contient une ligne pour chaque texte de requête, identifiant de base de données et identifiant utilisateur distincts (jusqu'au nombre maximum d'ordres distincts que le module peut surveiller). Les colonnes de la vue sont affichées dans Tableau F.20, « Colonnes de pg_stat_statements ».

Tableau F.20. Colonnes de pg_stat_statements

Nom	Type	Référence	Description
userid	oid	`pg_authid.oid`	OID de l'utilisateur qui a exécuté l'ordre SQL
dbid	oid	`pg_database.oid`	OID de la base de données dans laquelle l'ordre SQL a été exécuté
query	text		Texte de l'ordre SQL (jusqu'à track_activity_query_size octets)
calls	bigint		Nombre d'exécutions
total_time	double precision		Durée d'exécution de l'instruction SQL, en secondes
rows	bigint		Nombre total de lignes renvoyées ou affectées par l'ordre SQL
shared_blks_hit	bigint		Nombre total de blocs partagés lus dans le cache par l'ordre SQL
shared_blks_read	bigint		Nombre total de blocs partagés lus sur disque par l'ordre SQL
shared_blks_written	bigint		Nombre total de blocs partagés écrits sur disque par l'ordre SQL
local_blks_hit	bigint		Nombre total de blocs locaux lus dans le cache par l'ordre SQL
local_blks_read	bigint		Nombre total de blocs locaux lus sur disque par l'ordre SQL
local_blks_written	bigint		Nombre total de blocs locaux écrits sur disque par l'ordre

Nom	Type	Référence	Description
			SQL
temp_blks_read	bigint		Nombre total de blocs temporaires lus par l'ordre SQL
temp_blks_written	bigint		Nombre total de blocs temporaires écrits par l'ordre SQL

Cette vue, et la fonction `pg_stat_statements_reset`, sont disponibles seulement dans les bases de données dans lesquelles elles ont été installées spécifiquement via l'exécution du script d'installation `pg_stat_statements.sql`. Cependant, si le module `pg_stat_statements` est chargé sur le serveur, les statistiques sont pistées à travers toutes les bases de données du serveur, sans tenir compte de la présence de la vue.

Pour des raisons de sécurité, les utilisateurs qui ne sont pas super-utilisateurs ne sont pas autorisés à voir le texte des requêtes exécutées par les autres utilisateurs. Ils peuvent cependant voir les statistiques si la vue a été installée sur leur base de données.

Notez que les ordres SQL sont considérés identiques s'ils ont le même texte, quelles que soient les valeurs des variables de substitution utilisées dans les ordres SQL. L'utilisation des variables liées va aider à regrouper les ordres SQL et rendre les statistiques plus utiles.

F.29.2. Fonctions

`pg_stat_statements_reset() returns void`
: `pg_stat_statements_reset` ignore toutes les statistiques collectées jusque-là par `pg_stat_statements`. Par défaut, cette fonction peut uniquement être exécutée par les super-utilisateurs.

F.29.3. Paramètres de configuration

`pg_stat_statements.max` (integer)
: `pg_stat_statements.max` est le nombre maximum d'ordres tracés par le module (c'est-à-dire le nombre maximum de lignes dans la vue pg_stat_statements). Si un nombre supérieur d'ordres SQL distincts a été observé, c'est l'information sur les ordres les moins exécutés qui est ignorée. La valeur par défaut est 1000. Ce paramètre peut uniquement être positionné au démarrage du serveur.

`pg_stat_statements.track` (enum)
: `pg_stat_statements.track` contrôle quels sont les ordres comptabilisés par le module. Spécifiez `top` pour suivre les ordres de plus haut niveau (ceux qui sont soumis directement par les clients), `all` pour suivre également les ordres imbriqués (tels que les ordres invoqués dans les fonctions) ou `none` pour désactiver. La valeur par défaut est `top`. Seuls les super-utilisateurs peuvent changer ce paramétrage.

`pg_stat_statements.track_utility` (boolean)
: `pg_stat_statements.track_utility` contrôle si les commandes utilitaires sont tracées par le module. Les commandes utilitaires sont toutes les commandes SQL sauf **SELECT**, **INSERT**, **UPDATE** et **DELETE**. La valeur par défaut est `on`. Seuls les superutilisateurs peuvent modifier cette configuration.

`pg_stat_statements.save` (boolean)
: `pg_stat_statements.save` précise s'il faut sauvegarder les statistiques lors des arrêts du serveur. S'il est `off`, alors les statistiques ne sont pas sauvegardées lors de l'arrêt ni rechargées au démarrage du serveur. La valeur par défaut est `on`. Ce paramètre peut uniquement être positionné dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Le module a besoin de mémoire partagée supplémentaire d'environ `pg_stat_statements.max` * track_activity_query_size octets. Notez que cette mémoire est consommée quand le module est chargé, même si `pg_stat_statements.track` est positionné à `none`.

Afin de positionner ces paramètres dans votre fichier `postgresql.conf`, vous devez ajouter `pg_stat_statements` à custom_variable_classes. Un usage courant pourrait être :

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

custom_variable_classes = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.29.4. Exemple de sortie

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 -M prepared bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]----------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2;
calls       | 3000
total_time  | 9.60900100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]----------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2;
calls       | 3000
total_time  | 8.015156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]----------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
total_time  | 0.310624
rows        | 100000
hit_percent | 0.30395136778115501520
-[ RECORD 4 ]----------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2;
calls       | 3000
total_time  | 0.271741999999997
rows        | 3000
hit_percent | 93.7968855088209426
-[ RECORD 5 ]----------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
calls       | 1
total_time  | 0.08142
rows        | 0
hit_percent | 34.4947735191637631
```

F.29.5. Auteur

Takahiro Itagaki `<itagaki.takahiro@oss.ntt.co.jp>`

F.30. pgstattuple

Le module `pgstattuple` fournit plusieurs fonctions pour obtenir des statistiques au niveau ligne.

F.30.1. Fonctions

`pgstattuple(text) returns record`

`pgstattuple` renvoie la longueur physique d'une relation, le pourcentage des lignes « mortes », et d'autres informations. Ceci peut aider les utilisateurs à déterminer si une opération de VACUUM est nécessaire. L'argument est le nom de la relation cible (qui peut être qualifié par le nom du schéma). Par exemple :

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
```

```
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

Les colonnes en sortie sont décrites dans Tableau F.21, « Colonnes de `pgstattuple` ».

Tableau F.21. Colonnes de `pgstattuple`

Colonne	Type	Description
table_len	bigint	Longueur physique de la relation en octets
tuple_count	bigint	Nombre de lignes vivantes
tuple_len	bigint	Longueur totale des lignes vivantes en octets
tuple_percent	float8	Pourcentage des lignes vivantes
dead_tuple_count	bigint	Nombre de lignes mortes
dead_tuple_len	bigint	Longueur totale des lignes mortes en octets
dead_tuple_percent	float8	Pourcentage des lignes mortes
free_space	bigint	Espace libre total en octets
free_percent	float8	Pourcentage de l'espace libre

`pgstattuple` acquiert seulement un verrou en lecture sur la relation. Donc les relations ne reflètent pas une image instantanée des mises à jour en parallèle peuvent les affecter.

`pgstattuple` juge qu'une ligne est « morte » si `HeapTupleSatisfiesNow` renvoie false.

`pgstattuple(oid) returns record`

Identique à `pgstattuple(text)`, sauf que la relation cible est désignée par son OID.

`pgstatindex(text) returns record`

`pgstatindex` renvoie un enregistrement affichant des informations sur un index B-Tree. Par exemple :

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 8192
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 50.27
leaf_fragmentation | 0
```

En voici les colonnes:

Colonne	Type	Description
version	integer	Numéro de version du B-tree
tree_level	integer	Niveau de l'arbre pour la page racine
index_size	bigint	Nombre total de page dans l'index
root_block_no	bigint	Enmplacement du bloc racine
internal_pages	bigint	Nombre de pages « internes » (niveau supérieur)

Colonne	Type	Description
leaf_pages	bigint	Nombre de pages feuilles
empty_pages	bigint	Nombre de pages vides
deleted_pages	bigint	Nombre de pages supprimées
avg_leaf_density	float8	Densité moyenne des pages feuilles
leaf_fragmentation	float8	Fragmentation des pages feuilles

Comme pour `pgstattuple`, les résultats sont accumulés page par page, et ne représentent pas forcément une image instantannée de l'index complet.

`pg_relpages(text) returns bigint`
: `pg_relpages` renvoie le nombre de pages dans la relation.

F.30.2. Auteurs

Tatsuo Ishii et Satoshi Nagayasu

F.31. pg_trgm

Le module `pg_trgm` fournit des fonctions et opérateurs qui permettent de déterminer des similarités de textes alphanumériques en ASCII en fonction de correspondances de trigrammes. Il fournit également des classes d'opérateur accélérant les recherches de chaînes similaires.

F.31.1. Concepts du trigramme (ou trigraphe)

Un trigramme est un groupe de trois caractères consécutifs pris dans une chaîne. Nous pouvons mesurer la similarité de deux chaînes en comptant le nombre de trigrammes qu'elles partagent. Cette idée simple est très efficace pour mesurer la similarité des mots dans la plupart des langues.

> **Note**
>
> Une chaîne est considérée avoir deux espaces en préfixe et une espace en suffixe lors de la détermination de l'ensemble de trigrammes contenu dans la chaîne. Par exemple, l'ensemble des trigrammes dans la chaîne « `cat` » est « `  c` » (' c'), « `  ca` » (' ca'), « `cat` » et « `at  ` » ('at '). (Les espaces de début et de fin de chaînes sont importantes.)

F.31.2. Fonctions et opérateurs

Tableau F.22. Fonctions de `pg_trgm`

Fonction	Retour	Description
`similarity(text, text)`	real	Renvoie un nombre indiquant la similarité des deux arguments. L'échelle du résultat va de zéro (indiquant que les deux chaînes sont complètement différentes) à un (indiquant que les deux chaînes sont identiques).
`show_trgm(text)`	text[]	Renvoie un tableau de tous les trigrammes d'une chaîne donnée. (En pratique, ceci est peu utile, sauf pour le débogage.)
`show_limit()`	real	Renvoie la limite de similarité utilisée par l'opérateur %. Ceci configure la similarité minimale entre deux mots pour qu'ils soient considérés suffisamment proches.
`set_limit(real)`	real	Configure la limite de similarité actuelle utilisée par l'opérateur %. Le limite se positionne entre 0 et 1, elle vaut par défaut

Fonction	Retour	Description
		0,3. Renvoie la valeur passée.

Tableau F.23. Opérateurs de `pg_trgm`

Opérateur	Retour	Description
text % text	boolean	Renvoie `true` si les arguments ont une similarité supérieure à la limite configurée par `set_limit`.

F.31.3. Support des index

Le module `pg_trgm` fournit des classes d'opérateurs pour les index GiST et GIN qui vous permettent de créer un index sur une colonne de type text dans le but d'accélérer les recherches de similarité. Ces types d'index supportent l'opérateur de similarité % (et aucun autre opérateur, donc vous pouvez aussi créer un index B-tree standard).

Exemple :

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING gist (t gist_trgm_ops);
```

ou

```
CREATE INDEX trgm_idx ON test_trgm USING gin (t gin_trgm_ops);
```

À ce point, vous aurez un index sur la colonne *t* que vous pouvez utiliser pour une recherche de similarité. Une requête typique est :

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

Ceci renverra toutes les valeurs dans la colonne texte qui sont suffisamment similaire à *word*, triées de la meilleure correspondance à la pire. L'index sera utilisé pour accélérer l'opération même sur un grand ensemble de données.

Le choix d'un indexage GiST ou GIN dépend des caractéristiques relatives de performance qui sont discutées ailleurs. Comme règle de base, un index GIN est plus rapide pour la recherche qu'un index GiST mais plus lent pour la construction et la mise à jour ; donc GIN est préférable pour des données statiques et GiST pour des données souvent mises à jour.

F.31.4. Intégration à la recherche plein texte

La correspondance de trigramme est un outil très utile lorsqu'il est utilisé en conjonction avec un index plein texte. En particulier, il peut aider à la reconnaissance des mots mal orthographiés (ou tout simplement mal saisis), mots pour lesquels le mécanisme de recherche plein texte ne pourra pas faire une reconnaissance.

La première étape est la génération d'une table auxiliaire contenant tous les mots uniques dans les documents :

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

où documents est une table qui a un champ texte *bodytext* où nous voulons faire nos recherches. La raison de l'utilisation de la configuration `simple` avec la fonction `to_tsvector`, au lieu d'une configuration spécifique à la langue, est que nous voulons une liste des mots originaux.

Ensuite, nous créons un index trigramme sur la colonne word :

```
CREATE INDEX words_idx ON words USING gin(word gin_trgm_ops);
```

Maintenant, une requête **SELECT** similaire à l'exemple précédent peut être utilisée pour suggérer des mots dans les termes de la recherche de l'utilisateur. Un test utile supplémentaire vient à demander que les mots sélectionnés soient aussi d'une longueur similaire au mot mal orthographié.

> **Note**
>
> Comme la table words a été générée comme une table statique, séparée, il sera nécessaire de la regénérer périodiquement pour qu'elle reste raisonnablement à jour avec la collection des documents. Qu'elle soit exactement identique en permanence n'est habituellement pas nécessaire.

F.31.5. Références

Site de développement de GiST

Site de développement de TSearch2

F.31.6. Auteurs

Oleg Bartunov `<oleg@sai.msu.su>`, Moscou, Université de Moscou, Russie

Teodor Sigaev `<teodor@sigaev.ru>`, Moscou, Delta-Soft Ltd., Russie

Documentation : Christopher Kings-Lynne

Ce module est sponsorisé par Delta-Soft Ltd., Moscou, Russie.

F.32. pg_upgrade

pg_upgrade (auparavant appelé pg_migrator) permet la migration des fichiers de données d'une version majeure de PostgreSQL™ vers une autre version majeure sans nécessiter la partie sauvegarde/restauration typiquement requise pour les mises à jour majeures, par exemple de la version 8.4.7 à la version majeure courante de PostgreSQL™. Cet outil n'est pas utile pour les mises à jour mineures, par exemple pour une migration de la 9.0.1 à la 9.0.4.

pg_upgrade fonctionne bien car, bien que de nouvelles fonctionnalités soient ajoutées à chaque version majeure de PostgreSQL™, le format de stockage interne des données change rarement. pg_upgrade fait de son mieux pour s'assurer que les clusters, ancien et nouveau, soient compatibles binairement, c'est-à-dire en vérifiant les paramètres de temps modifiables à la compilation, en incluant les binaires 32/64 bits. Il est important que tout module externe soit aussi compatible binairement, bien que cela ne soit pas vérifié par pg_upgrade.

F.32.1. Versions supportées

pg_upgrade supporte la mise à jour de version 8.3.X et ultérieures vers la dernière version majeure de PostgreSQL™, ceci incluant les versions intermédiaires (snapshots) et les versions alpha.

F.32.2. Options de pg_upgrade

pg_upgrade accepte les arguments suivant en ligne de commande :

`-b` *`ancien_dir_exec`*, `--old-bindir` *`ANCIENDIREXEC`*
: indique le répertoire des exécutables de l'ancienne instance

`-B` *`nouveau_dir_exec`*, `--new-bindir` *`NOUVEAUDIREXEC`*
: indique le répertoire des exécutables de la nouvelle instance

`-c`, `--check`
: vérifie seulement les instances, ne modifie pas les données

`-d` *`ancien_repdonnées`*, `--old-datadir` *`ANCIENREPDONNÉES`*
: indique le répertoire des données de l'ancienne instance

`-D` *`nouveau_repdonnées`*, `--new-datadir` *`NOUVEAUREPDONNÉES`*
: indique le répertoire des données de la nouvelle instance

`-g`, `--debug`

active les traces de débogage

`-G` *`fichier_debug`*, `--debugfile` *`FICHIERDEBUG`*
enregistre les traces de débogage dans un fichier

`-k`, `--link`
utilise des liens plutôt que de copier les fichiers vers la nouvelle instance

`-l` *`fichier_trace`*, `--logfile` *`FICHIERTRACE`*
trace l'activité de la session dans un fichier

`-p` *`ancien_port`*, `--old-port` *`port`*
indique le numéro de port de l'ancienne instance

`-P` *`nouveau_port`*, `--new-port` *`port`*
indique le numéro de port de la nouvelle instance

`-u` *`nom_utilisateur`*, `--user` *`nom_utilisateur`*
superutilisateur de l'instance

`-v`, `--verbose`
active la sortie verbeuse

`-V`, `--version`
affiche la version puis quitte

`-?`, `-h`, `--help`
affiche l'aide puis quitte

F.32.3. Étapes de mise à jour

1. **Si nécessaire, déplacez l'ancienne instance**

   Si vous utilisez un répertoire d'installation spécifique à la version, par exemple `/opt/PostgreSQL/8.4`, vous n'avez pas besoin de déplacer l'ancien répertoire de l'instance. Les installeurs one-click utilisent tous des répertoires d'installation spécifiques à la version.

   Si votre répertoire d'installation n'est pas spécifique à la version, par exemple `/usr/local/pgsql`, il est nécessaire de déplacer le répertoire d'installation actuelle de PostgreSQL pour qu'il n'interfère pas avec la nouvelle installation de PostgreSQL™. Une fois que le serveur actuel PostgreSQL™ est arrêté, il est bon de renommer le répertoire d'installation de PostgreSQL ; en supposant que l'ancien répertoire est `/usr/local/pgsql`, vous pouvez saisir ceci :

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   pour renommer le répertoire.

2. **Pour une installation par les sources, construisez la nouvelle version**

   Construisez la nouvelle version de PostgreSQL avec des options de compilation compatibles avec l'ancienne instance. pg_upgrade utilisera **pg_controldata** pour vérifier que tous les paramètres sont compatibles avant de lancer la mise à jour.

3. **Installez les nouveaux exécutables de PostgreSQL**

   Installez les nouveaux exécutables du serveur ainsi que les fichiers de support. Vous pouvez utiliser les même numéros de port, habituellement le 5432, car l'ancienne et la nouvelle instance ne seront pas exécutées en même temps.

   Pour les installations à partir des sources, si vous préférez installer le nouveau serveur dans un emplacement particulier, utilisez le mot clé `prefix` :

   ```
   gmake prefix=/usr/local/pgsql.new install
   ```

4. **Installez pg_upgrade et pg_upgrade_support**

   Installez pg_upgrade et pg_upgrade_support dans la nouvelle instance PostgreSQL

5. **Initialisez la nouvelle instance PostgreSQL**

   Initialisez la nouvelle instance avec **initdb**. Encore une fois, utilisez des options d'**initdb** qui correspondent à celles de l'ancienne instance. Beaucoup d'installeurs font cette étape automatiquement. Il n'est pas nécessaire de démarrer la nouvelle

instance.

6. **Installez les fichiers objets partagés personnalisés (ou DLL)**

   Installez tous fichiers objets partagés personnalisés (ou DLL) utilisés par l'ancien cluster dans le nouveau cluster, par exemple `pgcrypto.so`, qu'ils proviennent des modules `contrib` ou de toute autre source. N'installez pas les définitions du schéma, par exemple `pgcrypto.sql`, car elles seront aussi migrées à partir de l'ancien cluster.

7. **Ajustez l'authentification**

   pg_upgrade se connectera à l'ancien et au nouveau serveurs plusieurs fois. Vous devez donc configurer l'authentification `ident` pour les connexions locales via une socket de domaine Unix ou utiliser un fichier `~/.pgpass` (voir Section 31.14, « Fichier de mots de passe »).

8. **Arrêtez les deux serveurs**

   Assurez-vous que les deux serveurs de bases de données sont stoppés. Pour cela, vous pouvez utiliser la commande suivante sous Unix :

```
pg_ctl -D /opt/PostgreSQL/8.4 stop
pg_ctl -D /opt/PostgreSQL/9.0 stop
```

   et la commande suivante sous Windows (en utilisant les bons noms de service) :

```
NET STOP postgresql-8.4
NET STOP postgresql-9.0
```

   ou encore :

```
NET STOP pgsql-8.3  (PostgreSQL™ 8.3 et antérieurs
utilisent un autre nom de service)
```

9. **Exécutez pg_upgrade**

   Exécutez toujours le binaire pg_upgrade dans le nouveau serveur, pas l'ancien. pg_upgrade nécessite de spécifier les repértoires de l'instance (`PGDATA`) et des exécutables pour l'ancienne et la nouvelle instance (`bin`). Vous pouvez aussi indiquer des valeurs pour l'utilisateur et le port.

   Si vous utilisez les liens, la mise à jour sera bien plus rapide (pas de copie de données), mais vous ne serez plus capable d'accéder à votre ancienne instance une fois que vous aurez démarré la nouvelle instance après la mise à jour. Voir `pg_upgrade --help` pour une liste complète des options.

   Pour les utilisateurs Windows, vous devez être connecté avec un compte administrateur, puis lancé un shell en tant qu'utilisateur `postgres` et configuré la variable PATH correctement :

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\9.0\bin;
```

   Enfin, vous lancez pg_upgrade avec les noms des répertoires entre guillemets doubles, par exemple :

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/8.4/data"
        --new-datadir "C:/Program Files/PostgreSQL/9.0/data"
        --old-bindir "C:/Program Files/PostgreSQL/8.4/bin"
        --new-bindir "C:/Program Files/PostgreSQL/9.0/bin"
```

   Une fois lancé, **pg_upgrade** vérifiera que les deux instances sont compatibles puis lancera la migration. vous pouvez utiliser **pg_upgrade --check** pour ne faire que les vérifications, même si l'ancienne installation est en cours d'exécution. **pg_upgrade --check** indiquera aussi tout ajustement manuel que vous devrez faire après la migration. **pg_upgrade** réclame des droits en écriture dans le répertoire courant.

   Évidemment, personne ne doit accéder aux instances pendant la migration. En utilisant un port autre que celui par défaut, par exemple 50432, pour les instances (ancienne et nouvelle), il est possible de diminuer le risque de connexions non voulues pendant la mise à jour.

   Si une erreur survient lors de la restauration du schéma de la base de données, **pg_upgrade** quittera et vous devrez retourner sur l'ancienne instance comme indiqué dans Étape 14. Pour essayer de nouveau pg_upgrade, vous aurez besoin de modifier

l'ancienne instance pour que la restauration du schéma par **pg_upgrade** réussisse. Si le problème est dû à un module contrib, vous pourriez avoir besoin de désinstaller le module contrib à partir de l'ancienne instance et de l'installer sur la nouvelle après la migration en espérant que le module n'est pas utilisé pour stocker des données de l'utilisateur.

10. **Restaurez `pg_hba.conf`**

    Si vous avez modifié `pg_hba.conf`, restaurez-le à sa configuration d'origine.

11. **Traitement post-migration**

    Si un traitement post-mise à jour est requis, pg_upgrade lancera des avertissements à la fin de son travail. Il générera aussi des fichiers de script à exécuter par l'administrateur. Les fichiers de script se connecteront à chaque base de données pour réaliser un traitement post-migration. Chaque script doit être exécuté avec la commande suivante :

    ```
    psql --username postgres --file script.sql postgres
    ```

    Ces scripts peuvent être exécutés dans n'importe quel ordre et peuvent être supprimés une fois qu'ils ont été exécutés.

    > **Attention**
    >
    > En général, il n'est pas prudent d'accéder aux tables référencées dans les scripts de reconstruction tant que ses scripts n'ont pas été exécutés entièrement ; le faire malgré tout pourrait amener des résultats incorrects ou des contre-performances. Les tables non référencées dans les scripts de reconstruction peuvent être utilisées immédiatement.

12. **Statistiques**

    > **Attention**
    >
    > Comme les statistiques de l'optimiseur ne sont pas transférées par pg_upgrade, il vous sera demandé d'exécuter une commande pour regénérer les informations statistiques à la fin de la migration.

13. **Supprimez l'ancienne instance**

    Une fois que vous être satisfait par la mise à jour, vous pouvez supprimer les répertoires de données de l'ancienne instance en exécutant le script mentionné à la fin de **pg_upgrade**. Vous devrez supprimer manuellement les anciens répertoires d'installation, par exemple `bin`, `share`.

14. **Retourner sur l'ancienne instance**

    Si, après avoir exécuté **pg_upgrade**, vous souhaitez retourner à l'ancienne instance, il existe plusieurs options.

    - Si vous avez exécuté **pg_upgrade** avec l'option `--check`, aucune modification n'a eu lieu sur l'ancienne instance, donc vous pouvez la réutiliser immédiatement.
    - Si vous avez exécuté **pg_upgrade** avec l'option `--link`, les fichiers de données sont partagés entre l'ancienne et la nouvelle instances. Si vous avez démarré la nouvelle instance, le nouveau serveur a écrit dans les fichiers partagés et il est du coup dangereux d'utiliser l'ancienne instance.
    - Si vous avez exécuté **pg_upgrade** *sans* l'option `--link` ou si vous n'avez pas lancé le nouveau serveur, l'ancienne instance n'a pas été modifiée mais un suffixe `.old` a été ajouté au fichier `$PGDATA/global/pg_control` et peut-être aux répertoires des tablespaces. Pour réutiliser l'ancienne instance, supprimez le suffixe `.old` du fichier `$PGDATA/global/pg_control`. De plus, si vous migrez d'une version 8.4 ou antérieure, supprimez les répertoires des tablespaces créés par l'outil de migration et supprimez le suffixe `.old` du nom des répertoires des tablespaces ; après cela, vous pouvez redémarré l'ancienne instance.

F.32.4. Limitations pour les migrations *à partir de* PostgreSQL 8.3

La migration à partir de PostgreSQL 8.3 comporte quelques restrictions supplémentaires. Par exemple, pg_upgrade ne fonctionnera pas pour une migration à partir d'une 8.3 si une colonne utilisateur est définie comme :

- le type de données tsquery
- le type de données name et qu'il ne s'agit pas de la première colonne

pg_upgrade ne fonctionnera pas si le module contrib `ltree` est installé dans une base de données.

Vous devez supprimer ce type de colonnes et les migrer manuellement.

pg_upgrade nécessitera une reconstruction de la table si :

- une colonne utilisateur est de type tsvector

pg_upgrade nécessitera un réindexage si :

- un index est de type hash ou GIN
- un index utilise `bpchar_pattern_ops`

De plus, le format de stockage des dates et heures par défaut a changé vers l'entier après l'arrivée de PostgreSQL™ 8.3. pg_upgrade vérifiera que le format de stockage utilisé sur l'ancienne instance correspond à celui utilisé sur la nouvelle instance. Assurez-vous que votre nouvelle instance est construite avec le drapeau de configuration `--disable-integer-datetimes`.

Pour les utilisateurs Windows, notez que, dû à une configuration différente sur ce paramètre par l'installeur one-click et l'installeur MSI, il est seulement possible de migrer une version 8.3 provenant de l'installeur one-click vers la version 8.4 ou ultérieure de l'installeur one-click. Il n'est pas possible de migrer un répertoire des données créé par l'installeur MSI à un répertoire de données créé par l'installeur one-click.

F.32.5. Notes

pg_upgrade n'offre pas le support de la migration de bases de données contenant ces types de données référençant les OID systèmes : regproc, regprocedure, regoper, regoperator, regconfig et regdictionary. (regtype peut être migré.)

Tout échec, reconstruction, réindexage sera indiqué par pg_upgrade si ils affectent votre installation ; les scripts post-mise à jour pour reconstruire les tables et index seront automatiquement générés.

Pour un test en déploiement, créez une copie du schéma de l'ancienne instance, insérez des données au hasard et faites un test de mise à jour là-dessus.

Si vous voulez utiliser le mode des liens et que vous ne voulez pas que votre ancienne instance soit modifiée quand vous lancez la nouvelle instance, faites une copie de l'ancienne instance et migrez-la avec le mode des liens. Pour obtenir une copie valide de l'ancienne instance, utilisez **rsync** pour créer une copie sale de l'ancienne instance et exécutez **rsync** de nouveau après arrêt de l'ancienne instance pour mettre à jour la copie avec toutes les modifications ultérieures.

F.33. seg

Ce module code le type de données seg pour représenter des segments de ligne ou des intervalles de nombres à virgule flottante. seg peut représenter l'incertitude des points extrèmes d'un intervalle, ce qui le rend particulièrement utile pour représenter des mesures de laboratoires.

F.33.1. Explications

La géométrie des mesures est habituellement plus complexe qu'un point dans un continuum numérique. Une mesure est habituellement un segment de ce continuum avec des limites non définissables. Les mesures apparaissent comme des intervalles à cause de ce côté incertain et du hasard, ainsi qu'à cause du fait que la valeur mesurée peut naturellement être un intervalle indiquant certaines conditions comme une échelle de température pour la stabilité d'une protéine.

En utilisant le bon sens, il apparaît plus agréable de stocker de telles données sous la forme d'intervalle, plutôt que sous la forme d'une paire de nombres. En pratique, c'est même plus efficace dans la plupart des applications.

En allant plus loin, le côté souple des limites suggère que l'utilisation des types de données numériques traditionnels amène en fait une certaine perte d'informations. Pensez à ceci : votre instrument lit 6.50, et vous saisissez cette valeur dans la base de données. Qu'obtenez-vous en la récupérant ? Regardez :

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

Dans le monde des mesures, 6.50 n'est pas identique à 6.5. La différence pourrait même être critique. Les personnes ayant réalisé

l'expérience écrivent habituellement (et publient) les chiffres qu'ils connaissent. 6.50 est en fait un intervalle incertain compris dans un intervalle plus grand et encore plus incertain, 6.5, le point central étant (probablement) la seule fonctionnalité commune qu'ils partagent. Nous ne voulons pas que de telles différences de données apparaissent de façon identique.

La conclusion ? il est agréable d'avoir un type de données spécial qui peut enregistrer les limites d'un intervalle avec une précision variable arbitraire. Variable dans le sens où chaque élément de données enregistre sa propre précision.

Vérifiez ceci :

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.33.2. Syntaxe

La représentation externe d'un intervalle se forme en utilisant un ou deux nombres à virgule flottante joint par l'opérateur d'échelle (`..` ou `...`). Sinon, il peut être spécifié comme un point central plus ou moins une déviation. Des indicateurs optionels (`<`, `>` et `~`) peuvent aussi être stockés. (Néanmoins, ces indicateurs sont ignorés par la logique interne.) Tableau F.24, « Représentations externes de seg » donne un aperçu des représentations autorisées ; Tableau F.25, « Exemples d'entrées valides de type seg » montre quelques exemples.

Dans Tableau F.24, « Représentations externes de seg », *x*, *y* et *delta* dénotent des nombres à virgule flottante. *x* et *y*, mais pas *delta*, peuvent être précédés par un indicateur de certitude :

Tableau F.24. Représentations externes de seg

x	Valeur seule (intervalle de longueur zéro)
x `..` *y*	Intervalle de *x* à *y*
x `(+-)` *delta*	Intervalle de *x* - *delta* à *x* + *delta*
x `..`	Intervalle ouvert avec une limite inférieure *x*
`..` *x*	Intervalle ouvert avec une limite supérieure *x*

Tableau F.25. Exemples d'entrées valides de type seg

`5.0`	Crée un segment de longueur zéro (un point si vous préférez)
`~5.0`	Crée un segment de taille nulle et enregistre `~` dans les données. `~` est ignoré par les opérations seg mais conservé en commentaire.
`<5.0`	Crée un point à 5.0. `<` est ignoré mais conservé en commentaire.
`>5.0`	Crée un point à 5.0. `>` est ignoré mais conservé en commentaire.
`5(+-)0.3`	Crée un intervalle `4.7 .. 5.3`. Notez que la notation `(+-)` n'est pas conservée.
`50 ..`	Tout ce qui supérieur ou égal à 50
`.. 0`	Tout ce qui est inférieur ou égal à 0
`1.5e-2 .. 2E-2`	Crée un intervalle `0.015 .. 0.02`
`1 ... 2`	Identifique à `1...2`, ou `1 .. 2`, ou `1..2` (les espaces autour de l'opérateur d'échelle sont ignorés)

Comme `...` est largement utilisé dans les sources de données, il est autorisé comme autre orthographe possible de `..`. Malheureusement, ceci crée une ambiguité pour l'analyseur : la limite supérieure dans `0...23` est `23` ou `0.23`. Ceci se résout en réclamant au moins un chiffre avant le point décimal dans tous les nombres de type seg.

Comme vérification, seg rejette les intervalles dont la limite inférieure est supérieure à la limite supérieure, par exemple `5 .. 2`.

F.33.3. Précision

Les valeurs seg sont stockés en interne sous la forme de paires de nombres en virgule flottante de 32 bits. Cela signifie que les nombres avec plus de sept chiffres significatifs sont tronqués.

Les nombres avec moins ou avec exactement sept chiffres significatifs conservent leur précision originale. C'est-à-dire que, si votre requête renvoie 0.00, vous serez sûr que les zéros qui suivent ne sont pas des conséquences du formatage : elles reflètent la précision de la donnée originale. Le nombre de zéro au début n'affectent pas la précision : deux chiffres significatifs sont considérés pour la valeur 0.0067.

F.33.4. Utilisation

Le module `seg` inclut une classe d'opérateur pour les index GiST dans le cas des valeurs seg. Les opérateurs supportés par la classe d'opérateur GiST are shown in Tableau F.26, « Opérateurs GiST du type Seg ».

Tableau F.26. Opérateurs GiST du type Seg

Opérateur	Description
`[a, b] << [c, d]`	[a, b] est entièrement à gauche de [c, d]. Autrement dit, [a, b] << [c, d] est vérifié si b < c
`[a, b] >> [c, d]`	[a, b] est entièrement à droite de [c, d]. Autrement dit, [a, b] >> [c, d] est vérifié si a > d
`[a, b] &< [c, d]`	Couvre une partie ou est à gauche de -- Cela se lit mieux de cette façon « ne s'étend pas à droite de ». C'est vrai quand b <= d.
`[a, b] &> [c, d]`	Couvre une partie ou est à droite de -- Cela se lit mieux de cette façon « ne s'étend pas à gauche de ». C'est vrai quand a >= c.
`[a, b] = [c, d]`	Identique à -- Les segments [a, b] et [c, d] sont identiques, autrement dit a == b et c == d.
`[a, b] && [c, d]`	Les segments [a, b] et [c, d] se chevauchent en partie.
`[a, b] @> [c, d]`	Le segment [a, b] contient le segment [c, d], autrement dit a <= c et b >= d
`[a, b] <@ [c, d]`	Le segment [a, b] est contenu dans [c, d], autrement dit a >= c et b <= d.

(Avant PostgreSQL 8.2, les opérateurs de contenance `@>` et `<@` étaient appelés respectivement `@` et `~`. Ces noms sont toujours disponibles mais sont déclarés obsolètes et seront supprimés un jour. Notez que les anciens noms sont inversés par rapport à la convention suivie par les types de données géométriques !)

Les opérateurs B-tree standard sont aussi fournis, par exemple :

Opérateur	Description
`[a, b] < [c, d]`	Plus petit que
`[a, b] > [c, d]`	Plus grand que

Ces opérateurs n'ont pas vraiment de sens sauf en ce qui concerne le tri. Ces opérateurs comparent en premier (a) à (c) et, s'ils sont égaux, comparent (b) à (d). Cela fait un bon tri dans la plupart des cas, ce qui est utile si vous voulez utiliser ORDER BY avec ce type.

F.33.5. Notes

Pour des exemples d'utilisation, voir les tests de régression `sql/seg.sql`.

Le mécanisme qui convertit `(+-)` en échelles standards n'est pas entièrement précis pour déterminer le nombre de chiffres significatifs pour les limites. Par exemple, si vous ajoutez un chiffre supplémentaire à la limite basse si l'intervalle résultat inclut une puissance de dix :

```
postgres=> select '10(+-)1'::seg as seg;
      seg
```

```
---------
9.0 .. 11             -- should be: 9 .. 11
```

La performance d'un index R-tree peut dépendre largement de l'ordre des valeurs en entrée. Il pourrait être très utile de trier la table en entrée sur la colonne seg ; voir le script `sort-segments.pl` pour un exemple.

F.33.6. Crédits

Auteur original : Gene Selkov, Jr. `<selkovjr@mcs.anl.gov>`, Mathematics and Computer Science Division, Argonne National Laboratory.

Mes remerciements vont principalement au professeur Joe Hellerstein (*http://db.cs.berkeley.edu/jmh/*) pour avoir élucidé l'idée centrale de GiST (*http://gist.cs.berkeley.edu/*). Mes remerciements aussi aux développeurs de PostgreSQL pour m'avoir permis de créer mon propre monde et de pouvoir y vivre sans pertubation. Argonne Lab et le département américain de l'énergie ont aussi toute ma gratitude pour les années de support dans ma recherche sur les bases de données.

F.34. spi

Le module `contrib/spi` fournit plusieurs exemples fonctionnels d'utilisation de SPI et des déclencheurs. Bien que ces fonctions aient un intérêt certain, elles sont encore plus utiles en tant qu'exemples à modifier pour atteindre ses propres buts. Les fonctions sont suffisamment généralistes pour être utilisées avec une table quelconque, mais la création d'un déclencheur impose que les noms des tables et des champs soient précisés (comme cela est décrit ci-dessous).

F.34.1. refint.c -- fonctions de codage de l'intégrité référentielle

`check_primary_key()` et `check_foreign_key()` sont utilisées pour vérifier les contraintes de clé étrangère. (Cette fonctionnalité est dépassée depuis longtemps par le mécanisme interne, mais le module conserve un rôle d'exemple.)

`check_primary_key()` vérifie la table de référence. Pour l'utiliser, on crée un déclencheur `BEFORE INSERT OR UPDATE` qui utilise cette fonction sur une table référençant une autre table. En arguments du déclencheur, on trouve : le nom de la colonne de la table référençant qui forme la clé étrangère, le nom de la table référencée et le nom de la colonne de la table référencée qui forme la clé primaire/unique. Il peut y avoir plusieurs colonnes. Pour gérer plusieurs clés étrangères, on crée un déclencheur pour chaque référence.

`check_foreign_key()` vérifie la table référencée. Pour l'utiliser, on crée un déclencheur `BEFORE DELETE OR UPDATE` qui utilise cette fonction sur une table référencée par d'autres tables. En arguments du déclencheur, on trouve : le nombre de tables référençant pour lesquelles la fonction réalise la vérification, l'action à exécuter si une clé de référence est trouvée (`cascade` -- pour supprimer une ligne qui référence, `restrict` -- pour annuler la transaction si des clés de référence existent, `setnull` -- pour initialiser les champs des clés référençant à NULL), les noms des colonnes de la table surveillées par le déclencheur, colonnes qui forment la clé primaire/unique, puis le nom de la table référençant et les noms des colonnes (répétés pour autant de tables référençant que cela est précisé par le premier argument). Les colonnes de clé primaire/unique doivent être marquées NOT NULL et posséder un index d'unicité.

Il y a des exemples dans `refint.example`.

F.34.2. timetravel.c -- fonctions de codage du voyage dans le temps

Dans le passé, PostgreSQL™ disposait d'une fonctionnalité de voyage dans le temps, permettant de conserver l'heure d'insertion et de suppression de chaque ligne. Ce comportement peut être émulé en utilisant ces fonctions. Pour les utiliser, il faut ajouter deux champs de type abstime à la table pour stocker le moment où une ligne a été insérée (start_date) et le moment où elle a été modifiée/supprimée (stop_date) :

```
CREATE TABLE mytab (
        ...             ...
        start_date      abstime,
        stop_date       abstime
        ...             ...
);
```

Le nom des colonnes n'a aucune importance, mais dans ce chapitre, elles sont nommées start_date et stop_date.

À l'insertion d'une nouvelle ligne, start_date doit normalement être initialisée à l'heure courante et stop_date à `infinity`. Le déclencheur substitue automatiquement ces valeurs si les données insérées sont NULL pour ces colonnes. L'insertion de données explicitement non-NULL dans ces colonnes n'intervient qu'au rechargement de données sauvegardées.

Les lignes pour lesquelles stop_date vaut `infinity` sont des lignes « actuellement valides », et peuvent être modifiées. Les lignes dont stop_date est fini ne peuvent plus être modifiées -- le déclencheur les protège. (Pour les modifier, il est nécessaire de désactiver le voyage dans le temps comme indiqué ci-dessous.)

Pour une ligne modifiable en mise à jour, seul stop_date est modifié (positionné à l'heure courante) et une nouvelle ligne avec la donnée modifiée est insérée. Pour cette nouvelle ligne, start_date est positionné à l'heure courante et stop_date à `infinity`.

Une suppression ne supprime pas réellement la ligne mais positionne stop_date à l'heure courante.

Pour trouver les lignes « actuellement valides », on ajoute la clause `stop_date = 'infinity'` dans la condition WHERE de la requête. (Cela peut se faire au travers d'une vue.) De façon similaire, une requête peut être exécutée sur les lignes valides à un moment du passé si des conditions adéquates sont posées sur start_date et stop_date.

`timetravel()` est la fonction déclencheur générique associée à ce fonctionnement. On crée un déclencheur `BEFORE INSERT OR UPDATE OR DELETE` qui utilise cette fonction pour chaque table sur laquelle la fonctionnalité de voyage dans le temps est activée. Le déclencheur accepte deux arguments : les noms réels des colonnes start_date et stop_date. La fonction accepte jusqu'à trois arguments optionnels qui doivent faire référence à des colonnes de type text. Le déclencheur stocke le nom de l'utilisateur courant dans la première de ces colonnes lors d'un INSERT, dans la seconde lors d'un UPDATE et dans la troisième lors un DELETE.

`set_timetravel()` permet d'activer et de désactiver la fonctionnalité de voyage dans le temps pour une table. `set_timetravel('ma_table', 1)` l'active pour la table `ma_table`. `set_timetravel('ma_table', 0)` la désactive pour la table `ma_table`. Dans les deux cas, l'ancien statut est rapporté. Quand elle est désactivée, les colonnes start_date et stop_date peuvent être librement modifiées. Le statut actif/inactif est local à la session courante -- toute session commence avec cette fonctionnalité activée sur toutes les tables.

`get_timetravel()` renvoie l'état de la fonctionnalité du voyage dans le temps pour une table sans le modifier.

Il y a un exemple dans `timetravel.example`.

F.34.3. autoinc.c -- fonctions pour l'incrémentation automatique d'un champ

`autoinc()` est un déclencheur qui stocke la prochaine valeur d'une séquence dans un champ de type integer. Cela recouvre quelque peu la fonctionnalité interne de la colonne « serial », mais ce n'est pas strictement identique : `autoinc()` surcharge les tentatives de substitution d'une valeur différente pour ce champ lors des insertions et, optionnellement, peut aussi être utilisé pour incrémenter le champ lors des mises à jour.

Pour l'utiliser, on crée un déclencheur `BEFORE INSERT` (ou en option `BEFORE INSERT OR UPDATE`) qui utilise cette fonction. Le déclencheur accepte deux arguments : le nom de la colonne de type integer à modifier et le nom de la séquence qui fournit les valeurs. (En fait, plusieurs paires de noms peuvent être indiquées pour actualiser plusieurs colonnes.)

Un exemple est fourni dans `autoinc.example`.

F.34.4. insert_username.c -- fonctions pour tracer les utilisateurs qui ont modifié une table

`insert_username()` est un déclencheur qui stocke le nom de l'utilisateur courant dans un champ texte. C'est utile pour savoir quel est le dernier utilisateur à avoir modifié une ligne particulière d'une table.

Pour l'utiliser, on crée un déclencheur `BEFORE INSERT` et/ou `UPDATE` qui utilise cette fonction. Le déclencheur prend pour seul argument le nom de la colonne texte à modifier.

Un exemple est fourni dans `insert_username.example`.

F.34.5. moddatetime.c -- fonctions pour tracer la date et l'heure de la dernière modification

`moddatetime()` est un déclencheur qui stocke la date et l'heure de la dernière modification dans un champ de type timestamp. C'est utile pour savoir quand a eu lieu la dernière modification sur une ligne particulière d'une table.

Pour l'utiliser, on crée un déclencheur `BEFORE UPDATE` qui utilise cette fonction. Le déclencheur prend pour seul argument le nom de la colonne de type timestamp à modifier.

Un exemple est fourni dans `moddatetime.example`.

F.35. sslinfo

Le module `sslinfo` fournit des informations sur le certificat SSL que le client actuel a fourni lors de sa connexion à

™. Le module est inutile (la plupart des fonctions renvoient NULL) si la connexion actuelle n'utilise pas SSL.

Cette extension ne se construira pas du tout sauf si l'installation était configurée avec `--with-openssl`.

F.35.1. Fonctions

`ssl_is_used() returns boolean`
: Renvoie TRUE si la connexion actuelle au serveur utilise SSL.

`ssl_client_cert_present() returns boolean`
: Renvoie TRUE si le client actuel a présenté un certificat client SSL au serveur. (Le serveur pourrait être configuré pour réclamer un certificat client.)

`ssl_client_serial() returns numeric`
: Renvoie un numéro de série du certificat actuel du client. La combinaison du numéro de série de certificat et du créateur du certificat garantit une identification unique du certificat (mais pas son propriétaire -- le propriétaire doit régulièrement changer ses clés et obtenir de nouveaux certifications à partir du créateur).

  Donc, si vous utilisez votre propre CA et autorisez seulement les certificats de ce CA par le serveur, le numéro de série est le moyen le plus fiable (bien que difficile à retenir) pour identifier un utilisateur.

`ssl_client_dn() returns text`
: Renvoie le sujet complet du certificat actuel du client, convertissant des données dans l'encodage actuel de la base de données. Nous supposons que si vous utilisez des caractères non ASCII dans le noms des certificats, votre base de données est capable de représenter ces caractères aussi. Si votre bases de données utilise l'encodage SQL_ASCII, les caractères non ASCII seront représentés par des séquences UTF-8.

  Le résultat ressemble à ceci : `/CN=Somebody /C=Some country/O=Some organization`.

`ssl_issuer_dn() returns text`
: Renvoie le nom complet du créateur du certificat actuel du client, convertissant les données caractères dans l'encodage actuel de la base de données. Les conversions d'encodage sont gérées de la même façon que pour `ssl_client_dn`.

  La combinaison de la valeur en retour de cette fonction avec le numéro de série du certificat identifie de façon unique le certificat.

  Cette fonction est réellement utile si vous avez plus d'un certificat d'un CA de confiance dans le fichier `root.crt` de votre serveur, ou si ce CA a envoyé quelques certificats intermédiaires d'autorité.

`ssl_client_dn_field(fieldname text) returns text`
: Cette fonction renvoie la valeur du champ spécifié dans le sujet du certificat, ou NULL si le champ n'est pas présent. Les noms du champ sont des constantes de chaîne qui sont converties dans des identifieurs d'objet ASN1 en utilisant la base de données des objets OpenSSL. Les valeurs suivantes sont acceptables :

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

  Tous ces champs sont optionnels, sauf *`commonName`*. L'inclusion des champs dépend entièrement de la politique de votre CA. Par contre, la signification des champs est strictement définie par les standards X.500 et X.509, donc vous ne pouvez pas

leur donner des significations arbitraires.

`ssl_issuer_field(fieldname text) returns text`
Identique à `ssl_client_dn_field`, mais pour le créateur du certificat, plutôt que pour le sujet du certificat.

F.35.2. Auteur

Victor Wagner `<vitus@cryptocom.ru>`, Cryptocom LTD

E-Mail du groupe de développement Cryptocom OpenSSL : `<openssl@cryptocom.ru>`

F.36. tablefunc

Le module `tablefunc` inclut plusieurs fonctions permettant de renvoyer des tables (c'est-à-dire plusieurs lignes). Ces fonctions sont utiles directement et comme exemples sur la façon d'écrire des fonctions C qui renvoient plusieurs lignes.

F.36.1. Fonctions

Tableau F.27, « Fonctions `tablefunc` » montre les fonctions fournies par le module `tablefunc`.

Tableau F.27. Fonctions `tablefunc`

Fonction	Retour	Description
`normal_rand(int numvals, float8 mean, float8 stddev)`	setof float8	Renvoie un ensemble de valeurs float8 normalement distribuées
`crosstab(text sql)`	setof record	Renvoie une « table pivot » contenant les noms des lignes ainsi que *N* colonnes de valeur, où *N* est déterminé par le type de ligne spécifié par la requête appelant
`crosstab`*N*`(text sql)`	setof table_crosstab_*N*	Produit une « table pivot » contenant les noms des lignes ainsi que *N* colonnes de valeurs. `crosstab2`, `crosstab3` et `crosstab4` sont prédéfinies mais vous pouvez créer des fonctions `crosstab`*N* supplémentaires de la façon décrite ci-dessous
`crosstab(text source_sql, text category_sql)`	setof record	Produit une « table pivot » avec les colonnes des valeurs spécifiées par une autre requête
`crosstab(text sql, int N)`	setof record	Version obsolète de `crosstab(text)`. Le paramètre *N* est ignoré car le nombre de colonnes de valeurs est toujours déterminé par la requête appelante
`connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld ], text start_with, int max_depth [, text branch_delim ])`	setof record	Produit une représentation d'une structure hiérarchique en arbre

F.36.1.1. `normal_rand`

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

`normal_rand` produit un ensemble de valeurs distribuées au hasard (distribution gaussienne).

`numvals` est le nombre de valeurs que la fonction doit renvoyer. *`mean`* est la moyenne de la distribution normale des valeurs et *`stddev`* est la déviation standard de la distribution normale des valeurs.

Par exemple, cet appel demande 1000 valeurs avec une moyenne de 5 et une déviation standard de 3 :

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
---------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
   -0.369151492880995
    0.283600703686639
       .
       .
       .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.36.1.2. `crosstab(text)`

```
crosstab(text sql)
crosstab(text sql, int N)
```

La fonction `crosstab` est utilisé pour créer un affichage « pivot » où les données sont listées de gauche à droite plutôt que de haut en bas. Par exemple, avec ces données

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

l'affiche ressemble à ceci

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

La fonction `crosstab` prend un paramètre texte qui est une requête SQL produisant des données brutes formatées de la façon habituelle et produit une table avec un autre formatage.

Le paramètre *`sql`* est une instruction SQL qui produit l'ensemble source des données. Cette instruction doit renvoyer une colonne *`row_name`*, une colonne *`category`* et une colonne *`value`*. *`N`* est un paramètre obsolète, ignoré quand il est fourni (auparavant, il devait correspondre au nombre de colonnes de valeurs en sortie, mais maintenant ceci est déterminé par la requête appelant).

Par exemple, la requête fournie peut produire un ensemble ressemblant à ceci :

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

La fonction `crosstab` déclare renvoyer un setof record, donc les noms et types réels des colonnes doivent être définis dans la clause `FROM` de l'instruction **SELECT** appelante. Par exemple : statement, for example:

```
    SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2
text);
```

Cet exemple produit un ensemble ressemblant à ceci :

```
           <== value  columns  ==>
row_name   category_1   category_2
 ---------+------------+------------
   row1        val1         val2
   row2        val5         val6
```

La clause `FROM` doit définir la sortie comme une colonne *`row_name`* (du même type que la première colonne du résultat de la requête SQL) suivie par N colonnes *`value`* (tous du même type de données que la troisième colonne du résultat de la requête SQL). Vous pouvez configurer autant de colonnes de valeurs en sortie que vous voulez. Les noms des colonnes en sortie n'ont pas d'importance en soi.

La fonction `crosstab` produit une ligne en sortie pour chaque groupe consécutif de lignes en entrée avec la même valeur *`row_name`*. Elle remplit les colonnes de *`value`*, de gauche à droite, avec les champs *`value`* provenant de ces lignes. S'il y a moins de lignes dans un groupe que de colonnes *`value`* en sortie, les colonnes supplémentaires sont remplies avec des valeurs NULL ; s'il y a trop de ligne, les colonnes en entrée supplémentaires sont ignorées.

En pratique, la requête SQL devrait toujours spécifier `ORDER BY 1,2` pour s'assurer que les lignes en entrée sont bien ordonnées, autrement dit que les valeurs de même *`row_name`* sont placées ensemble et son correctement ordonnées dans la ligne. Notez que `crosstab` ne fait pas attention à la deuxième colonne du résultat de la requête ; elle est là pour permettre le tri, pour contrôler l'ordre dans lequel les valeurs de la troisième colonne apparaissent dans la page.

Voici un exemple complet :

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

Vous pouvez toujours éviter d'avoir à écrire une clause `FROM` pour définir les colonnes en sortie, en définissant une fonction crosstab personnalisée qui a le type de ligne désiré en sortie en dur dans sa définition. Ceci est décrit dans la prochaine section. Une autre possibilité est d'embarquer la clause `FROM` requise dans la définition d'une vue.

F.36.1.3. `crosstab`*`N`*`(text)`

```
crosstabN(text sql)
```

Les fonctions `crosstabN` sont des exemples de configuration de fonctions d'emballage pour la fonction généraliste `crosstab`. Cela vous permet de ne pas avoir à écrire les noms et types des colonnes dans la requête **SELECT** appelante. Le module `tablefunc` inclut `crosstab2`, `crosstab3` et `crosstab4`, dont les types de ligne en sortie sont définis ainsi :

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
);
```

Du coup, ces fonctions peuvent être utilisées directement quand la requête en entrée produit des colonnes *`row_name`* et *`value`* de type text, et que vous voulez 2, 3 ou 4 colonnes de valeur en sortie. Autrement, elles se comportent exactement la fonction `crosstab` décrite précédemment.

L'exemple de la section précédente pourrait aussi fonctionner ainsi :

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

Ces fonctions sont fournies principalement comme exemples. Vous pouvez créer vos propres types de retour et fonctions basées sur la fonction `crosstab()`. Il existe deux façons de le faire :

- Créer un type composite décrivant les colonnes désirées en sortie, similaire aux exemples du script d'installation. Ensuite, définir un nom de fonction unique acceptant un paramètre de type text et renvoyant setof nom_de_votre_type, mais renvoyant à la fonction C `crosstab`. Par exemple, si votre source de données produit des noms de ligne qui sont de type text, et des valeurs qui sont de type float8, et que vous voulez cinq colonnes de valeurs :

  ```
  CREATE TYPE my_crosstab_float8_5_cols AS (
      my_row_name text,
      my_category_1 float8,
      my_category_2 float8,
      my_category_3 float8,
      my_category_4 float8,
      my_category_5 float8
  );

  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
    RETURNS setof my_crosstab_float8_5_cols
    AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
  ```

- Utiliser des paramètres `OUT` pour définir implicitement le type en retour. Le même exemple pourrait s'écrire ainsi :

  ```
  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
      IN text,
      OUT my_row_name text,
      OUT my_category_1 float8,
      OUT my_category_2 float8,
      OUT my_category_3 float8,
      OUT my_category_4 float8,
      OUT my_category_5 float8)
    RETURNS setof record
    AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
  ```

F.36.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

La limite principale de la forme à un paramètre de `crosstab` est qu'elle traite toutes les valeurs d'un groupe de la même façon, en insérant chaque valeur dans la première colonne disponible. Si vous voulez les colonnes de valeur correspondant à des catégories spécifiques de données, et que certains groupes n'ont pas de données pour certaines des catégories, alors cela ne fonctionne pas. La forme à deux paramètres de la fonction `crosstab` gère ce cas en fournissant une liste explicite des catégories correspondant aux colonnes en sortie.

`source_sql` est une instruction SQL qui produit l'ensemble source des données. Cette instruction doit renvoyer une colonne *`row_name`*, une colonne *`category`* et une colonne *`value`*. Elle pourrait aussi avoir une ou plusieurs colonnes « extra ». La colonne *`row_name`* doit être la première. Les colonnes *`category`* et *`value`* doivent être les deux dernières colonnes, dans cet ordre. Toutes les colonnes entre *`row_name`* et *`category`* sont traitées en « extra ». Les colonnes « extra » doivent être les mêmes pour toutes les lignes avec la même valeur *`row_name`*.

Par exemple, *`source_sql`* produit un ensemble ressemblant à ceci :

```
 SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

     row_name    extra_col   cat    value
    ----------+------------+-----+---------
      row1         extra1    cat1    val1
      row1         extra1    cat2    val2
      row1         extra1    cat4    val4
      row2         extra2    cat1    val5
      row2         extra2    cat2    val6
      row2         extra2    cat3    val7
      row2         extra2    cat4    val8
```

`category_sql` est une instruction SQL qui produit l'ensemble des catégories. Cette instruction doit renvoyer seulement une colonne. Cela doit produire au moins une ligne, sinon une erreur sera générée. De plus, cela ne doit pas produire de valeurs dupliquées, sinon une erreur sera aussi générée. *`category_sql`* doit ressembler à ceci :

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

La fonction `crosstab` déclare renvoyer setof record, donc les noms et types réels des colonnes en sortie doivent être définis dans la clause `FROM` de la requête **SELECT** appelante, par exemple :

```
    SELECT * FROM crosstab('...', '...')
      AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

Ceci produira un résultat ressemblant à ceci :

```
                              <==  value  columns   ==>
           row_name   extra   cat1   cat2   cat3   cat4
           ---------+-------+------+------+------+------
             row1     extra1  val1   val2          val4
             row2     extra2  val5   val6   val7   val8
```

La clause `FROM` doit définir le bon nombre de colonnes en sortie avec les bon types de données. S'il y a *N* colonnes dans le résultat de la requête *source_sql*, les *N*-2 premiers d'entre eux doivent correspondre aux *N*-2 premières colonnes en sortie. Les colonnes restantes en sortie doivent avoir le type de la dernière colonne du résultat de la requête The remaining output columns *source_sql*, et il doit y en avoir autant que de lignes dans le résultat de la requête *category_sql*.

La fonction `crosstab` produit une ligne en sortie pour chaque groupe consécutif de lignes en entrée avec la même valeur *row_name*. La colonne en sortie *row_name* ainsi que toutes colonnes « extra » sont copiées à partir de la première ligne du groupe. Les colonnes *value* en sortie sont remplies avec les champs *value* à partir des lignes ayant une correspondance avec des valeurs *category*. Si la *category* d'une ligne ne correspond pas à une sortie de la requête *category_sql*, sa *value* est ignorée. Les colonnes en sortie dont la catégorie correspondante est absente de toute ligne en entrée du groupe sont remplies avec des valeurs NULL.

En pratique, la requête *source_sql* doit toujours spécifier `ORDER BY 1` pour s'assurer ques les valeurs du même *row_name* sont assemblées. Néanmoins, l'ordre des catégories dans un groupe n'est pas important. De plus, il est essentiel que l'ordre du résultat de la requête *category_sql* corresponde à l'ordre des colonnes spécifiées en sortie.

Voici deux exemples complets :

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)
```

```
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
```

```
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate
| volts
-------+--------------------------+-------------+-------------+--------------------------+--
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
| 2.6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00
2003 | 3.1234
(2 rows)
```

Vous pouvez créer des fonctions prédéfinies pour éviter d'avoir à écrire les noms et types des colonnes en résultat dans chaque requête. Voir les exemples dans la section précédente. La fonction C sous-jacente pour cette forme de `crosstab` est appelée `crosstab_hash`.

F.36.1.5. `connectby`

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

La fonction `connectby` réalise un affichage de données hiérarchiques stockées dans une table. La table doit avoir un champ clé qui identifie de façon unique les lignes et un champ clé qui référence le parent de chaque ligne. `connectby` peut afficher le sous-arbre à partir de n'importe quelle ligne.

Tableau F.28, « Paramètres `connectby` » explique les paramètres.

Tableau F.28. Paramètres `connectby`

Paramètre	Description
`relname`	Nom de la relation source
`keyid_fld`	Nom du champ clé
`parent_keyid_fld`	Nom du champ clé du parent
`orderby_fld`	Nom du champ des autres relations (optionnel)
`start_with`	Valeur de la clé de la ligne de début
`max_depth`	Profondeur maximum pour la descente, ou zéro pour une profondeur illimitée
`branch_delim`	Chaîne pour séparer les clés des branches (optionnel)

Les champs clé et clé du parent peuvent être de tout type mais ils doivent être du même type. Notez que la valeur *`start_with`* doit être saisi comme une chaîne de caractères, quelque soit le type du champ clé.

La fonction `connectby` déclare renvoyer un setof record, donc les noms et types réels des colonnes en sortie doivent être définis dans la clause `FROM` de l'instruction **SELECT** appelante, par exemple :

```
    SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2',
0, '~')
      AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

Des deux premières colonnes en sortie sont utilisées pour la clé de la ligne en cours et la clé de son parent ; elles doivent correspondre au type du champ clé de la table. La troisième colonne est la profondeur de l'arbre est doit être du type integer. Si un paramètre *`branch_delim`* est renseigné, la prochaine colonne en sortie est l'affichage de la branche et doit être de type text. Enfin, si le paramètre *`orderby_fld`* est renseigné, la dernière colonne en sortie est un numéro de série et doit être de type integer.

La colonne « branch » en sortie affiche le chemin des clés utilisé pour atteindre la ligne actuelle. Les clés sont séparées par la chaîne *branch_delim* spécifiée. Si l'affichage des branches n'est pas voulu, omettez le paramètre *branch_delim* et la colonne branche dans la liste des colonnes en sortie. and the branch column in the output column list.

Si l'ordre des relations du même parent est important, incluez le paramètre *orderby_fld* pour indiquer par quel champ ordonner les relations. Ce champs doit être de tout type de données triable. La liste des colonnes en sortie doit inclure une colonne numéro de série de type integer si, et seulement si, *orderby_fld* est spécifiée.

Les paramètres représentant table et noms de champs sont copiés tels quel dans les requêtes SQL que `connectby` génère en interne. Du coup, ajoutez des guillemets doubles si les noms utilisent majuscules et minuscules ou s'ils contiennent des caractères spéciaux. Vous pouvez aussi avoir besoin de qualifier le nom de la table avec le nom du schéma.

Dans les grosses tables, les performances seront faibles sauf si un index est créé sur le champ clé parent.

Il est important que la chaîne *branch_delim* n'apparaisse pas dans les valeurs des clés, sinon `connectby` pourrait rapporter des erreurs de récursion infinie totalement erronées. Notez que si *branch_delim* n'est pas fourni, une valeur par défaut ~ est utilisé pour des raisons de détection de récursion.

Voici un exemple :

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- with branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+---------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- without branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- with branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0,
'~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
```

```
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- without branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.36.2. Auteur

Joe Conway

F.37. test_parser

`test_parser` est un exemple d'analyseur personnalisé pour la recherche plein texte. Il ne fait rien de particulièrement utile mais peut servir comme point de départ pour développer votre propre analyseur.

`test_parser` reconnaît les mots séparés par des espaces blancs, et renvoie simplement deux types de jeton :

```
mydb=# SELECT * FROM ts_token_type('testparser');
 tokid | alias |  description
-------+-------+---------------
     3 | word  | Word
    12 | blank | Space symbols
(2 rows)
```

Ces nombres jeton ont été choisis pour être compatible avec la numération par défaut de l'analyseur. Ceci nous permet d'utiliser sa fonction `headline()`, conservant du coup l'exemple simple.

F.37.1. Usage

Exécuter le script d'installation crée un analyseur de recherche plein texte nommé `testparser`. Il n'utilise pas de paramètres configurables.

Vous pouvez tester l'analyseur avec, par exemple :

```
mydb=# SELECT * FROM ts_parse('testparser', 'That''s my first own parser');
 tokid | token
-------+--------
     3 | That's
    12 |
     3 | my
    12 |
     3 | first
    12 |
     3 | own
    12 |
     3 | parser
```

Une utilisation réelle nécessite le paramétrage d'une configuration de recherche plein texte qui utilise cet analyseur. Par exemple :

```
mydb=# CREATE TEXT SEARCH CONFIGURATION testcfg ( PARSER = testparser );
CREATE TEXT SEARCH CONFIGURATION

mydb=# ALTER TEXT SEARCH CONFIGURATION testcfg
mydb-#   ADD MAPPING FOR word WITH english_stem;
ALTER TEXT SEARCH CONFIGURATION
```

```
mydb=#  SELECT to_tsvector('testcfg', 'That''s my first own parser');
          to_tsvector
-------------------------------
 'that':1 'first':3 'parser':5
(1 row)

mydb=# SELECT ts_headline('testcfg', 'Supernovae stars are the brightest phenomena in
galaxies',
mydb(#                    to_tsquery('testcfg', 'star'));
                           ts_headline
----------------------------------------------------------------
 Supernovae <b>stars</b> are the brightest phenomena in galaxies
(1 row)
```

F.38. tsearch2

Le module `tsearch2` fournit une compatibilité ascendante pour la fonctionnalité de recherche plein texte avec les applications qui ont utilisé `contrib/tsearch2` avant que la recherche plein texte ne soit intégré au cœur de PostgreSQL™ dans la version 8.3.

F.38.1. Problèmes de portabilité

Bien que les fonctionnalités de recherche plein texte intégrées au moteur sont basées sur `contrib/tsearch2` et sont largement similaires à ce dernier, il existe un grand nombre de petites différences qui créent des problèmes de portabilités pour les applications existantes :

- Certains noms de fonctions ont été changés, par exemple `rank` en `ts_rank`. Le remplacement du module `tsearch2` fournit des alias utilisant les anciens noms.
- Les types de données et les fonctions de recherche existent tous dans le schéma système `pg_catalog`. Dans une installation utilisant `contrib/tsearch2`, ces objets auraient été créé dans le schéma `public` bien que certains utilisateurs ont choisi de les placer dans un schéma propre. Les références explicites de schéma échoueront donc dans tous les cas. Le remplacement du module `tsearch2` fournit des objets qui sont stockés dans `public` (ou un autre schéma si nécessaire) pour que les références en question fonctionnent.
- Le concept d'« analyseur courant » et de « dictionnaire courant » n'existe pas dans les fonctionnalités intégrées. Seule la configuration courante est disponible (via le paramètre `default_text_search_config`). Bien que l'analyseur courant et le dictionnaire courant étaient seulement utilisés par les fonctions de déboguage, ceci pourrait être un obstacle de portage dans certains cas. Le module de remplacement `tsearch2` émule ces variables d'état et fournit des fonctions de compatibilité ascendante pour leur initialisation et leur récupération.

Il existe des problèmes qui ne sont pas adressés par le module de remplacement `tsearch2`, ce qui réclamera des modifications dans le code des applications :

- L'ancienne fonction trigger `tsearch2` permettait l'utilisation d'une liste d'arguments indiquant le nom des fonctions à appeler sur la donnée de type texte avant la conversion au format tsvector. Ceci a été supprimé car c'est une faille de sécurité. Il n'était pas possible de garantir que la fonction appelée était celle qui était voulue. L'approche recommendée si la donnée doit être traitée avant d'être indexée est d'écrire un trigger personnalisé qui fait le boulot lui-même.
- Les informations sur la configuration de la recherche plein texte ont été déplacées dans les catalogues système qui sont vraiment différents des tables utilisées par `contrib/tsearch2`. Toutes applications qui examinent ou modifient ces tables ont besoin d'être modifiées.
- Si une application a utilisé des configurations personnalisées de recherche plein texte, ces dernières devront être revues pour être placées dans les catalogues système en utilisant les nouvelles commandes SQL de configuration de la recherche plein texte. Le module de remplacement `tsearch2` offre un peu de support pour cela en permettant le chargement des anciennes tables de configuration de `contrib/tsearch2` avec PostgreSQL™ 8.3. (Sans ce module, il n'est pas possible de charger les données de configuration car les valeurs des colonnes regprocedure ne peuvent utiliser des fonctions.) Bien que ces tables de configuration ne *font* vraiment rien, au moins leur contenu sera disponible à la consultation lors de la configuration d'une configuration personnalisée dans 8.3.
- Les anciennes fonctions `reset_tsearch()` et `get_covers()` ne sont pas supportées.
- Le module de remplacement `tsearch2` ne définit pas d'alias d'opérateurs, se reposant entièrement sur les opérateurs internes. Ceci pose un problème seulement si une application utilise les noms d'opérateurs en les qualifiant du schéma, ce qui est très

rare.

F.38.2. Convertir une installation pré-8.3

La façon recommandée de mettre à jour une installation pré-8.3 qui utilise `contrib/tsearch2` est :

1. Faire une sauvegarde de l'ancienne installation de la façon habituelle, mais en s'assurant de ne pas utiliser l'option `-c` (option `--clean`)de pg_dump et pg_dumpall.
2. Dans la nouvelle installation, créez une ou plusieurs bases de donnée vides et installez le module de remplacement `tsearch2` dans chaque base qui utilise la recherche plein texte. Ceci doit être fait *avant* le chargement des données ! Si votre ancienne installation a placé les objets de `contrib/tsearch2` dans un autre schéma que `public`, assurez-vous d'ajuster le script d'installation `tsearch2` pour que les objets de remplacement soient créés dans ce même schéma.
3. Rechargez la sauvegarde. Il y aura quelques erreurs dûes à la restauration impossible d'objets du `contrib/tsearch2` original. Ces erreurs peuvent être ignorées mais cela signifie que vous ne pouvez pas restaurer la sauvegarde dans une transaction complète (autrement dit, vous ne pouvez pas utiliser l'option `-1` de pg_restore).
4. Examinez le contenu des tables de configuration restaurées de `contrib/tsearch2` (pg_ts_cfg et ainsi de suite), et créez les configurations internes de recherche plein texte nécessaire. Vous pouvez supprimer les anciennes tables de configuration une fois que vous avez extrait toutes les informations utiles.
5. Testez votre application.

Plus tard, si vous le souhaitez, vous pourrez renommer les références de l'application aux alias des objets internes pour que vous puissez éventuellement déinstaller le module de remplacement `tsearch2`.

F.38.3. Références

Site de développement de Tsearch2

F.39. unaccent

`unaccent` is a text search dictionary that supprime les accents d'un lexeme. C'est un dictionnaire de filtre, ce qui signifie que sa sortie est passée au prochain dictionnaire (s'il y en a un), contrairement au comportement normal des dictionnaires. Cela permet le traitement des accents pour la recherche plein texte.

L'implémentation actuelle d'`unaccent` ne peut pas être utilisée comme un dictionnaire de normalisation pour un dictionnaire `thesaurus`.

F.39.1. Configuration

Le dictionnaire `unaccent` accepte les options suivantes :

- `RULES` est le nom de base du fichier contenant la liste des règles de traduction. Ce fichier doit être stocké dans le répertoire `$SHAREDIR/tsearch_data/` (`$SHAREDIR` étant le répertoire des données partagées de PostgreSQL™). Son nom doit se terminer avec l'extension `.rules` (qui ne doit pas être inclus dans le paramètre `RULES`).

Le fichier des règles a le format suivant :

- Chaque ligne représente une paire, consistant en un caractère avec accent suivi par un caractère sans accent. Le premier est traduit par le second. Par exemple :

```
À           A
Á           A
Â             A
Ã           A
Ä               A
Å             A
Æ             A
```

Un exemple plus complet, qui est directement utile pour les langages européens, se trouve dans `unaccent.rules`, qui est installé dans le répertoire `$SHAREDIR/tsearch_data/` une fois le module `unaccent` installé.

F.39.2. Utilisation

L'exécution du script d'installation `unaccent.sql` crée un modèle de recherche de texte appelé `unaccent` et un dictionnaire basé sur ce modèle, appelé lui-aussi `unaccent`. Le dictionnaire `unaccent` a le paramètre par défaut `RULES='unaccent'`, qui le rend directement utilisable avec le fichier standard `unaccent.rules`. Si vous le souhaitez, vous pouvez modifier le paramètre. Par exemple :

```
ma_base=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='mes_regles');
```

Vous pouvez aussi créer des nouveaux dictionnaires basés sur le modèle.

Pour tester le dictionnaire, vous pouvez essayer la requête suivante :

```
ma_base=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

Voici un exemple montrant comment installer le dictionnaire `unaccent` dans une configuration de recherche plein texte :

```
ma_base=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
ma_base=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
ma_base=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

ma_base=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)
ma_base=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
  <b>Hôtel</b>de la Mer
(1 row)
```

F.39.3. Fonctions

La fonction `unaccent()` supprime les accents d'une chaîne de caractères donnée. Il utilise le dictionnaire `unaccent` mais il peut être utilisé en dehors du contexte normal de la recherche plein texte.

```
unaccent([dictionary, ] string) returns text
```

```
SELECT unaccent('unaccent','Hôtel');
SELECT unaccent('Hôtel');
```

F.40. uuid-ossp

Le module `uuid-ossp` fournit des fonctions qui permettent de créer des identifiants uniques universels (UUIDs) à l'aide d'algorithmes standard. Ce module fournit aussi des fonctions pour produire certaines constantes UUID spéciales.

Ce module dépend de la bibliothèque OSSP UUID, disponible sur *http://www.ossp.org/pkg/lib/uuid/*.

F.40.1. Fonctions de `uuid-ossp`

Tableau F.29, « Fonctions pour la génération d'UUID » montre les fonctions disponibles pour générer des UUIDs. Les standards en question, ITU-T Rec. X.667, ISO/IEC 9834-8:2005 et RFC 4122, spécifient quatre algorithmes pour produire des UUID identifiés par les numéros de version 1, 3, 4 et 5. (Il n'existe pas d'algorithme version 2.) Chacun de ces algorithmes peut convenir pour un ensemble différent d'applications.

Tableau F.29. Fonctions pour la génération d'UUID

Fonction	Description
`uuid_generate_v1()`	Cette fonction crée un UUID version 1. Ceci implique l'adresse MAC de l'ordinateur et un horodatage. Les UUID de ce type révèlent l'identité de l'ordinateur qui a créé l'identifiant et l'heure de création de cet identifiant, ce qui peut ne pas convenir pour certaines applications sensibles à la sécurité.
`uuid_generate_v1mc()`	Cette fonction crée un UUID version 1, mais utilise une adresse MAC multicast à la place de la vraie adresse de l'ordinateur.
`uuid_generate_v3(namespace uuid, name text)`	Cette fonction crée un UUID version 3 dans l'espace de nom donné en utilisant le nom indiqué en entrée. L'espace de nom doit être une des constantes spéciales produites par les fonctions `uuid_ns_*()` indiquées dans ???. (En théorie, cela peut être tout UUID.) Le nom est un identifiant dans l'espace de nom sélectionné. Par exemple : `SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');` Le paramètre name sera haché avec MD5, donc la version claire ne peut pas être récupérée à partir de l'UUID généré. La génération des UUID par cette méthode ne comprend aucun élément au hasard ou dépendant de l'environnement et est du coup reproductible.
`uuid_generate_v4()`	Cette fonction crée un UUID version 4 qui est entièrement réalisé à partir de nombres aléatoires.
`uuid_generate_v5(namespace uuid, name text)`	Cette fonction crée un UUID version 5 qui fonctionne comme un UUID version 3 sauf que SHA-1 est utilisé comme méthode de hachage. La version 5 devrait être préférée à la version 3 car SHA-1 est considéré plus sécurisé que MD5.

Tableau F.30. Fonctions renvoyant des constantes UUID

`uuid_nil()`	Une constante UUID « nil », qui ne correspond pas à un UUID réel.
`uuid_ns_dns()`	Constante désignant l'espace de nom pour les UUID.
`uuid_ns_url()`	Constante désignant l'espace de nom URL pour les UUID.
`uuid_ns_oid()`	Constante désignant l'espace de nom des identifiants d'objets ISO pour les UUIDs. (Ceci aboutit aux OID ASN.1, mais n'a pas de relation avec les OID de PostgreSQL™.)
`uuid_ns_x500()`	Constante désignant l'espace de nom « X.500 distinguished name » (DN) pour les UUID.

F.40.2. Auteur

Peter Eisentraut <peter_e@gmx.net>

F.41. vacuumlo

vacuumlo est un outil simple qui supprimera tous les « Large Objects » « orphelins » d'une base de données PostgreSQL™. Un « Large Object » orphelin est tout « Large Object » dont l'OID n'apparaît dans aucune colonne oid ou lo de la base de données.

Si vous l'utilisez, vous pourriez être intéressé par le trigger `lo_manage` du module contrib `contrib/lo` (voir Section F.17, « lo »). `lo_manage` est utile pour tenter d'éviter la création de « Large Object » orphelins.

F.41.1. Usage

```
vacuumlo [options] base [base2 ... basen]
```

Toutes les bases de données indiquées sur la ligne de commande sont traitées. Les options disponibles sont :

`-v`
: Écrit beaucoup de messages de progression.

`-n`
: Ne supprime rien, affiche simplement ce qu'il aurait fait.

`-l` *`limite`*
: Ne supprime pas plus que ce nombre (définie par le paramètre *`limite`*) de Large Objects par transaction (par défaut 1000). Comme le serveur récupère un verrou par Large Object supprimé, supprimer un grand nombre de Large Objects en une transaction risque de prendre trop de verrous par rapport à la configuration de max_locks_per_transaction. Configurez la limite à 0 si vous voulez tous les supprimer en une transaction.

`-U` *`nom_utilisateur`*
: Nom d'utilisateur pour la connexion.

`-w`, `--no-password`
: Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
: Force vacuumlo à demander un mot de passe avant la connexion à une base de données.

  Cette option n'est jamais obligatoire car vacuumlo demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, vacuumlo perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`-h` *`nom_hote`*
: Serveur de la base de données.

`-p` *`port`*
: Port du serveur de la base de données.

F.41.2. Méthode

Tout d'abord, vacuumlo construit une table temporaire contenant tous les OID des « Large Objects » dans la base de donnée sélectionnée.

Ensuite, il parcourt toutes les colonnes de la base qui sont de type oid ou lo, et supprime les entrées correspondantes de la table temporaire. (Notez que seuls ces types sont pris en considération ; les domaines définies à partir de ces types ne le sont pas.)

Les entrées restantes de la table temporaire identifient les « Large Objects » orphelins. Ils sont supprimés.

F.41.3. Auteur

Peter Mount <peter@retep.org.uk>

http://www.retep.org.uk

F.42. xml2

Le module `xml2` fournit des fonctionnalités pour les requêtes XPath et pour XSLT.

F.42.1. Notice d'obsolescence

À partir de PostgreSQL™ 8.3, les fonctionnalités XML basées sur le standard SQL/XML sont dans le cœur du serveur. Cela couvre la vérification de la syntaxe XML et les requêtes XPath, ce que fait aussi ce module (en dehors d'autres choses) mais l'API n'est pas du tout compatible. Il est prévu que ce module soit supprimé dans une future version de PostgreSQL pour faire place à une nouvelle API standard, donc vous êtes encouragés à convertir vos applications. Si vous trouvez que des fonctionnalités de ce module ne sont pas disponibles dans un format adéquat avec la nouvelle API, merci d'expliquer votre problème sur la liste `<pgsql-hackers@postgresql.org>` pour que ce problème soit corrigé.

F.42.2. Description des fonctions

Tableau F.31, « Fonctions » montre les fonctions fournies par ce module. Ces fonctions fournissent une analyse XML et les requêtes XPath. Tous les arguments sont du type text, ce n'est pas affiché pour que ce soit plus court.

Tableau F.31. Fonctions

Fonction	Retour	Description
`xml_is_well_formed(document)`	bool	Ceci analyse un document fourni comme argument au format text et renvoie true si le document est du XML bien formé (notez qu'avant PostgreSQL 8.2, cette fonction était appelée `xml_valid()`. C'est un mauvais nom car un document bien formé n'est pas forcément un document valide en XML. L'ancien nom est toujours disponible mais est obsolète.)
`xpath_string(document, query)`	text	Ces fonctions évaluent la requête XPath à partir du document fourni, et convertie le résultat dans le type spécifié.
`xpath_number(document, query)`	float4	
`xpath_bool(document, query)`	bool	

`xpath_nodeset(document, query, toptag, itemtag)`	text	Cette fonction évalue la requête sur le document et enveloppe le résultat dans des balises XML. Si le résultat a plusieurs valeurs, la sortie ressemblera à ceci :
		`<toptag>`
		`<itemtag>Valeur 1 qui pourrait être un fragment XML</itemtag>`
		`<itemtag>Valeur 2....</itemtag>`
		`</toptag>`
		Si `toptag` et/ou `itemtag` sont des chaînes vides, la balise adéquate est omise.

`xpath_nodeset(document, query)`	text	Comme `xpath_nodeset(document,query,toptag,itemtag)` mais le résultat omet les balises.
`xpath_nodeset(document, query, itemtag)`	text	Comme `xpath_nodeset(document,query,toptag,itemtag)` mais le résultat

Fonction	Retour	Description
		omet les balises.
`xpath_list(document, query, separator)`	text	Cette fonction renvoie plusieurs valeurs séparées par le caractère indiqué, par exemple `Valeur 1,Valeur 2,Valeur 3` si le séparateur est une virgule (`,`).
`xpath_list(document, query)`	text	Ceci est un emballage de la fonction ci-dessus avec la virgule comme séparateur.

F.42.3. `xpath_table`

```
   xpath_table(text key, text document, text relation, text xpaths, text criteria)
returns setof record
```

`xpath_table` est une fonction SRF qui évalue un ensemble de requêtes XPath sur chaque ensemble de documents et renvoie les résultats comme une table. Le champ de clé primaire de la table des documents est renvoyé comme première colonne des résultats pour que les résultats puissent être utilisés dans des jointures. Les pamraètres sont décrits dans Tableau F.32, « Paramètres de `xpath_table` ».

Tableau F.32. Paramètres de `xpath_table`

Paramètre	Description
`key`	Le nom du champ de la clé primaire (« key »). C'est simplement le champ à utiliser comme première colonne de la table en sortie, autrement dit celle qui identifie l'enregistrement (voir la note ci-dessous sur les valeurs multiples).
`document`	Le nom du champ contenant le document XML.
`relation`	Le nom de la table ou de la vue contenant les documents.
`xpaths`	Une ou plusieurs expressions XPath séparées par des `|`
`criteria`	Le contenu de la clause WHERE. Elle doit être spécifiée, donc utilisez `true` ou `1=1` si vous voulez traiter toutes les lignes de la relation.

Ces paramètres (en dehors des chaînes XPath) sont simplement substitués dans une instruction SELECT, donc vous avez de la flexibilité. L'instruction est celle qui suit :

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

Donc les paramètres peuvent être *tout* ce qui est valide dans ces emplacements particuliers. Le résultat de ce SELECT a besoin de renvoyer exactement deux colonnes (ce qu'il fera sauf si vous essayez d'indiquer plusieurs champs pour la clé ou le document). Cette approche simpliste implique que vous validiez avant tout valeur fournie par un utilisateur pour éviter les attaques par injection de code SQL.

La fonction doit être utilisée dans une expression `FROM` avec une clause `AS` pour indiquer les colonnes en sortie. Par exemple :

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

La clause AS définit les noms et types des colonnes de la table en sortie. La première est le champ « key » et le reste correspond à la requête XPath. S'il y a plus de requêtes XPath que de colonnes résultats, les requêtes supplémentaires seront ignorées, S'il y a plus de colonnes résultats que de requêtes XPath, les colonnes supplémentaires seront NULL.

Notez que cet exemple définit la colonne résultat page_count en tant qu'entier (integer). La fonction gère en interne les représentations textes, donc quand vous dites que vous voulez un entier en sortie, il prendra la représentation texte du résultat XPath et utilisera les fonctions en entrée de PostgreSQL pour la transformer en entier (ou tout type que la clause AS réclame). Vous obtiendrez une erreur s'il ne peut pas le faire -- par exemple si le résultat est vide -- donc rester sur du texte est préférable si vous pensez que vos données peuvent poser problème.

L'instruction SELECT n'a pas besoin d'être un `SELECT *`. Elle peut référencer les colonnes par nom ou les joindre à d'autres tables. La fonction produit une table virtuelle avec laquelle vous pouvez réaliser toutes les opérations que vous souhaitez (c'est-à-dire agrégation, jointure, tri, etc.) Donc nous pouvons aussi avoir :

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

comme exemple plus compliqué. Bien sûr, vous pouvez placer tout ceci dans une vue pour une utilisation plus simple.

F.42.3.1. Résultats à plusieurs valeurs

La fonction `xpath_table` suppose que les résultats de chaque requête XPath ramènent plusieurs valeurs, donc le nombre de lignes renvoyées par la fonction pourrait ne pas être le même que le nombre de documents en entrée. La première ligne renvoyée contient le premier résultat de chaque requête, la deuxième le second résultat de chaque requête. Si une res requêtes a moins de valeur que les autres, des valeurs NULL seront renvoyées.

Dans certains cas, un utilisateur saura qu'une requête XPath renverra seulement un seul résultat, peut-être un identifiant unique de document) -- si elle est utilisée avec une requête XPath renvoyant plusieurs résultats, le résultat sur une ligne apparaîtra seulement sur la première ligne du résultat. La solution à cela est d'utiliser le champ clé pour une jointure avec une requête XPath. Comme exemple :

```
    CREATE TABLE test (
        id int PRIMARY KEY,
        xml text
    );

    INSERT INTO test VALUES (1, '<doc num="C1">
    <line num="L1"><a>1</a><b>2</b><c>3</c></line>
    <line num="L2"><a>11</a><b>22</b><c>33</c></line>
    </doc>');

    INSERT INTO test VALUES (2, '<doc num="C2">
    <line num="L1"><a>111</a><b>222</b><c>333</c></line>
    <line num="L2"><a>111</a><b>222</b><c>333</c></line>
    </doc>');

    SELECT * FROM
      xpath_table('id','xml','test',
                  '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
                  'true')
      AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int4, val2 int4,
val3 int4)
    WHERE id = 1 ORDER BY doc_num, line_num

     id | doc_num | line_num | val1 | val2 | val3
    ----+---------+----------+------+------+------
      1 | C1      | L1       |    1 |    2 |    3
      1 |         | L2       |   11 |   22 |   33
```

Pour obtenir `doc_num` sur chaque ligne, la solution est d'utiliser deux appels à `xpath_table` et joindre les résultats :

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int4, val2 int4, val3 int4),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.42.4. Fonctions XSLT

Les fonctions suivantes sont disponibles si libxslt est installé.

F.42.4.1. `xslt_process`

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

Cette fonction applique la feuille de style XSLT au document et renvoie le résultat transformé. Le paramètre `paramlist` est une liste de paramètres à utiliser dans la transformation, spécifiée sous la forme 'a=1,b=2'. Notez que l'analyse des paramètres est simpliste : les valeurs des paramètres ne peuvent pas contenir de virgules !

Il existe aussi une version de `xslt_process` à deux paramètres qui ne passe pas de paramètres pour la transformation.

F.42.5. Auteur

John Gray `<jgray@azuli.co.uk>`

Le développement de ce module a été sponsorisé par Torchbox Ltd. (www.torchbox.com) Il utilise la même licence BSD que PostgreSQL.

Annexe G. Projets externes

PostgreSQL™ est un projet complexe et difficile à gérer. Il est souvent plus efficace de développer des améliorations à l'extérieur du projet principal.

Pour aider la communauté à développer ses projets externes, *PgFoundry*, site web qui héberge les projets liés à PostgreSQL™ a été créé. PgFoundry est construit à partir du projet GForge similaire à *SourceForge.net* dans ses fonctionnalités, puisqu'il fournit des listes de diffusion, des forums, un outil de gestion des bogues, un système de versionnage et un site web. Tout développeur d'un projet libre en relation avec PostgreSQL™, qui souhaite l'héberger sur PgFoundry, peut créer un nouveau projet.

G.1. Interfaces client

Il n'existe que deux interfaces clients dans la distribution de base de PostgreSQL™ :

- libpq, car il s'agit de l'interface principal pour le langage C et parce que de nombreux interfaces clients sont construits par dessus ;
- ECPG, car il dépend de la grammaire SQL côté serveur et est donc sensible aux modifications internes de PostgreSQL™.

Tous les autres interfaces sont des projets externes et sont distribués séparément. Tableau G.1, « Interfaces clients maintenus en externe » présente certains de ces projets. Ils peuvent ne pas être distribués sous la même licence que PostgreSQL™. Pour obtenir plus d'informations sur chaque interface, avec les termes de la licence, on se référera au site web et à la documentation.

Tableau G.1. Interfaces clients maintenus en externe

Nom	Langage	Commentaires	Site web
DBD::Pg	Perl	Pilote DBI Perl	*http://search.cpan.org/dist/DBD-Pg/*
JDBC	JDBC	Pilote JDBC Type 4	*http://jdbc.postgresql.org/*
libpqxx	C++	Interface C++, nouveau style	*http://pqxx.org/*
Npgsql	.NET	Fournisseur de données .NET	*http://npgsql.projects.postgresql.org/*
pgtclng	Tcl		*http://sourceforge.net/projects/pgtclng/*
psqlODBC	ODBC	Pilote ODBC	*http://psqlodbc.projects.postgresql.org/*
psycopg	Python	Compatible DB API 2.0	*http://initd.org/psycopg/*

G.2. Langages procéduraux

PostgreSQL™ inclut plusieurs langages procéduraux avec la distribution de base : PL/PgSQL, PL/Tcl, PL/Perl et PL/Python.

Il existe également d'autres langages procéduraux développés et maintenus en dehors de la distribution principale de PostgreSQL™. Tableau G.2, « Langages procéduraux maintenus en externe » liste certains de ces langages. Ils peuvent ne pas être distribués sous la même licence que PostgreSQL™. Pour obtenir plus d'informations sur chaque langage, avec les termes de la licence, on se référera au site web et à la documentation.

Tableau G.2. Langages procéduraux maintenus en externe

Nom	Langage	Site web
PL/Java	Java	*http://pljava.projects.postgresql.org/*
PL/PHP	PHP	*http://www.commandprompt.com/community/plphp/*
PL/Py	Python	*http://python.projects.postgresql.org/*
PL/R	R	*http://www.joeconway.com/plr/*
PL/Ruby	Ruby	*http://raa.ruby-lang.org/project/pl-ruby/*

Nom	Langage	Site web
PL/Scheme	Scheme	*http://plscheme.projects.postgresql.org/*
PL/sh	Unix shell	*http://plsh.projects.postgresql.org/*

G.3. Extensions

PostgreSQL™ est conçu pour être facilement extensible. C'est pour cette raison que les extensions chargées dans la base de données peuvent fonctionner comme les fonctionnalités intégrées au SGBD. Le répertoire `contrib/` livré avec le code source contient un grand nombre d'extensions. Le fichier `README` dans ce répertoire les résume. On y trouve des outils de conversion, d'indexation en texte intégral (*full text indexing*), des outils XML et des types de données et méthodes d'indexation supplémentaires. D'autres extensions sont développées indépendamment, comme *PostGIS*. Même les solutions de réplication de PostgreSQL™ sont développées en externe. Ainsi, *Slony-I*, solution populaire de réplication maître/esclave, est développée indépendamment du projet principal.

Il existe plusieurs outils d'administration pour PostgreSQL™. Le plus populaire est *pgAdmin III*. On trouve également des outils propriétaires.

Annexe H. Dépôt du code source

Le code source de PostgreSQL™ est stocké et géré en utilisant le système de contrôle de version appelé Git™. Un miroir public du dépôt maître est disponible ; il est mis à jour une minute après chaque changement du dépôt maître.

Notre wiki, *http://wiki.postgresql.org/wiki/Working_with_Git*, contient des informations sur l'utilisation de Git.

Notez que la construction de PostgreSQL™ à partir du dépôt des sources nécessite des versions raisonnablement récentes de bison, flex et Perl. Ces outils ne sont pas nécessaires pour construire à partir d'une archive tar distribuée car les fichiers qu'ils sont sensés construire sont inclus dans l'archive tar. Les autres besoins en outillage sont les mêmes que ceux exposés dans Chapitre 15, Procédure d'installation de PostgreSQL™ du code source.

H.1. Récupérer les sources via Git™

Avec Git™, vous devez avoir une copie du dépôt de code sur votre machine locale, pour que vous ayez accès à tout l'historique et les branches sans avoir besoin d'être en ligne. C'est le moyen le plus rapide et le plus flexible pour développer ou tester des patchs.

Procédure H.1. Git

1. Vous aurez besoin d'une version installée de Git™, que vous pouvez obtenir sur *http://git-scm.com*. La plupart des systèmes ont actuellement une version récente de Git installée par défaut ou disponible dans le système de package.
2. Pour commencer à utiliser le dépôt Git, commencez par faire un clone du miroir officiel :

   ```
   git clone git://git.postgresql.org/git/postgresql.git
   ```

   Ceci va copier le dépôt complet sur votre machine locale, donc ça prend un peu de temps à se faire, tout spécialement si vous avez une connexion lente. Les fichiers seront placés dans un nouveau sous-répertoire nommé `postgresql` de votre répertoire courant.

   Le miroir Git peut aussi être atteint avec le protocole HTTP si, par exemple, un pare-feu bloquait l'accès au protocole Git. Changez simplement le préfixe de l'URL par `http`, comme par exemple :

   ```
   git clone http://git.postgresql.org/git/postgresql.git
   ```

   Le protocole HTTP est moins efficace que le protocole Git, donc il sera plus lent à utiliser.
3. À chaque fois que vous voulez obtenir les dernières mises à jour, allez dans le dépôt avec la commande **cd** et exécutez la commande qui suit :

   ```
   git fetch
   ```

Git™ peut faire bien plus de choses que de simplement récupérer les sources. Pour plus d'informations, consultez les pages man de Git™ ou visitez le site *http://git-scm.com*.

Annexe I. Documentation

PostgreSQL™ fournit quatre formats principaux de documentation :

- le texte brut, pour les information de pré-installation ;
- HTML, pour la lecture en ligne et les références ;
- PDF ou PostScript, pour l'impression ;
- les pages man (de manuel), pour la référence rapide.

De plus, un certain nombre de fichiers `README` peuvent être trouvés à divers endroits de l'arbre des sources de PostgreSQL™. Ils renseignent l'utilisateur sur différents points d'implantation.

La documentation HTML et les pages de manuel font parties de la distribution standard et sont installées par défaut. Les documents au format PDF et PostScript sont disponibles indépendemment par téléchargement.

I.1. DocBook

Les sources de la documentation sont écrites en *DocBook*, langage assez semblable au HTML. Ces deux langages sont des applications du *Standard Generalized Markup Language*, SGML, qui est essentiellement un langage de description d'autres langages. Dans ce qui suit, les termes DocBook et SGML sont tous deux utilisés, mais ils ne sont pas techniquement interchangeables.

DocBook™ permet à l'auteur de spécifier la structure et le contenu d'un document technique sans qu'il ait à se soucier du détail de la présentation. Un style de document définit le rendu du contenu dans un des formats de sortie finaux. DocBook est maintenu par le groupe *OASIS* . Le *site officiel de DocBook* présente une bonne documentation d'introduction et de référence ainsi qu'un livre complet de chez O'Reilly disponible à la lecture en ligne. Le *guide DocBook des nouveaux venus* est très utile pour les débutants. Le *projet de documentation FreeBSD* utilise également DocBook et fournit également de bonnes informations, incluant un certain nombre de lignes directrices qu'il peut être bon de prendre en considération.

I.2. Ensemble d'outils

Les outils qui suivent sont utilisés pour produire la documentation. Certains sont optionnels (comme mentionné).

DTD DocBook
: Il s'agit de la définition de DocBook elle-même. C'est actuellement la version 4.2 qui est utilisée. Vous avez besoin de la variante SGML de la DTD DocBook mais, pour construire les pages man, vous avez aussi besoin de la variante XML de la même version.

Les entités de caractère ISO 8879
: Celles-ci sont nécessaires à DocBook mais sont distribuées à part car maintenues par l'ISO.

DocBook DSSSL Stylesheets
: Ils contiennent les instructions de traitement pour convertir les sources DocBook vers d'autres formats, comme par exemple le HTML.

DocBook XSL Stylesheets
: C'est une autre feuille de style pour convertir DocBook vers d'autres formats. Nous l'utilisons actuellement pour produire les pages man et les pages HTMLHelp en option. Vous pouvez aussi utiliser cette chaîne de production pour produire une sortie HTML ou PDF, mais les versions officielles de PostgreSQL utilisent les feuilles de style DSSSL pour cela.

OpenJade
: C'est le paquetage de base pour le traitement de SGML. Il contient un analyseur SGML, un processeur DSSSL (programme qui permet la conversion de documents SGML en d'autres formats à l'aide de feuilles de styles DSSSL), ainsi qu'un certain nombre d'autres outils. Jade™ est actuellement maintenu par le groupe OpenJade et non plus par James Clark.

Libxslt for **xsltproc**
: C'est l'outil de traitement à utiliser avec les feuilles de style XSLT (alors que **jade** est l'outil de traitement des feuilles de style DSSSL).

JadeTeX
: JadeTeX™ peut être installé pour utiliser TeX™ comme outil de formatage pour Jade™. JadeTeX est capable de créer des fichiers au formats PostScript ou PDF (ce dernier avec les signets).

  Cependant, une sortie JadeTeX est de moindre qualité qu'une sortie RTF. Les principaux problèmes que l'on peut rencontrer

concernent les tables et les éléments de placements verticaux et horizontaux. Il n'y a aucune possibilité de corriger manuellement le résultat.

Différentes méthodes d'installation sont détaillées ci-après pour les divers outils nécessaires au traitement de la documentation. Il peut exister d'autres types de distributions empaquetées de ces outils. Tout changement du statut d'un paquetage peut être rapportée auprés de la liste de discussion de la documentation, afin d'inclure ces informations ici-même.

I.2.1. Installation par RPM Linux™

La plupart des revendeurs mettent à disposition des utilisateurs un ensemble complet de paquetages RPM pour le traitement de DocBook au sein de leur distribution. Lors de l'installation, il faut recherche une option « SGML » ou les paquetages suivants : `sgml-common`, `docbook`, `stylesheets`, `openjade`. `sgml-tools` est probablement requis. Si le fournisseur de la distribution ne les fournit pas, il doit être possible d'utiliser des paquetages issus d'une distribution compatible.

I.2.2. Installation sous FreeBSD

Le projet de documentation FreeBSD (FreeBSD Documentation Project) est lui-même un utilisateur intensif de DocBook, et c'est sans surprise que l'on retrouve en son sein un ensemble complet de « portages » des outils de documentation sur FreeBSD. Les portages suivants doivent être installés afin de produire la documentation sur FreeBSD :

- `textproc/sp` ;
- `textproc/openjade` ;
- `textproc/iso8879` ;
- `textproc/dsssl-docbook-modular`.
- `textproc/docbook-420`

Un intérêt particulier peut également être porté aux différents éléments de `/usr/ports/print` (`tex`, `jadetex`).

Il est probable que les portages ne mettent pas à jour le fichier de catalogue général dans `/usr/local/share/sgml/catalog.ports` ou que l'ordre ne soit pas le bon. Les lignes suivantes doivent figurent au début :

```
CATALOG "openjade/catalog"
CATALOG "iso8879/catalog"
CATALOG "docbook/dsssl/modular/catalog"
CATALOG "docbook/4.2/catalog"
```

Pour ne pas éditer ce fichier, il est possible de positionner la variable d'environnement `SGML_CATALOG_FILES` en y mettant une liste de fichiers catalogues séparés par des caractères « deux points ».

De plus amples informations sur les outils dédiés à la documentation de FreeBSD se trouvent dans les *instructions du projet de documentation de FreeBSD*.

I.2.3. Paquetages Debian

Un ensemble complet de paquetages d'outils de documentation est disponible pour Debian GNU/Linux™. Pour l'installer, il suffit de taper :

```
apt-get install docbook docbook-dsssl docbook-xsl openjade xsltproc
```

I.2.4. Installation manuelle à partir des sources

L'installation manuelle des outils DocBook est quelque peu complexe. Il est donc préférable d'utiliser des paquetages précompilés. Seule une procédure de mise en œuvre standard, qui utilise des répertoires d'installation standard et sans fonctionnalités particulières, est ici décrite. Pour les détails, on peut étudier la documentation respective de chaque paquetage et lire les documents d'introduction à SGML.

I.2.4.1. Installer OpenJade

1. L'installation d'OpenJade se fait par l'intermédiaire des outils de construction `./configure; make; make install` classiques de GNU. Les détails sont dans la distribution des sources d'OpenJade. En quelques mots :

```
./configure --enable-default-catalog=/usr/local/share/sgml/catalog
make
make install
```

L'emplacement du « catalogue par défaut » a son importance ; il est utilisé par la suite. Il est possible de ne pas s'en souvenir, mais dans ce cas, la variable d'environnement `SGML_CATALOG_FILES` doit être définie de façon à pointer vers le bon fichier à chaque fois que jade est exécuté. (Cette méthode est également possible si OpenJade est déjà installé et que le reste de l'environnement de publication est installé localement.)

2. De même, les fichiers `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd` et `catalog` du répertoire `dsssl` doivent être installés quelque part, dans `/usr/local/share/sgml/dsssl`, par exemple. Il est certainement plus facile de copier le répertoire entier.

```
cp -R dsssl /usr/local/share/sgml
```

3. Enfin, le fichier `/usr/local/share/sgml/catalog` doit être créé pour y ajouter la ligne suivante :

```
CATALOG "dsssl/catalog"
```

(Il s'agit d'un chemin relatif référençant le fichier installé dans l'Étape 2. Ce chemin doit être correctement renseigné si un répertoire d'installation différent est utilisé.)

I.2.4.2. Installation du kit DTD de DocBook™

1. Récupérer la distribution *DocBook V4.2*.
2. Créer le répertoire `/usr/local/share/sgml/docbook-4.2` et s'y placer. (L'emplacement exact importe peu mais celui-ci a le bénéfice d'être cohérent avec le schéma d'installation proposé ici.)

```
$ mkdir /usr/local/share/sgml/docbook-4.2
$ cd /usr/local/share/sgml/docbook-4.2
```

3. Décompresser l'archive.

```
$ unzip -a ...../docbook-4.2.zip
```

(L'archive décompresse ses fichier dans le répertoire courant.)

4. Éditer le fichier `/usr/local/share/sgml/catalog` (ou celui précisé à jade lors de l'installation) et y placer une ligne similaire à celle-ci :

```
CATALOG "docbook-4.2/docbook.cat"
```

5. Télécharger l'archive contenant les *entités de caractères ISO 8879*, la décompresser et placer les fichiers dans le même répertoire que celui des fichiers de DocBook.

```
$ cd /usr/local/share/sgml/docbook-4.2
$ unzip ...../ISOEnts.zip
```

6. Lancer la commande suivante dans le répertoire contenant les fichiers DocBook et ISO :

```
perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
```

(Cette opération permet de corriger les mélanges entre le fichier de catalogue de DocBook et les noms réels des fichiers contenant les entités de caractères ISO.)

I.2.4.3. Installation des feuilles de style DSSSL DocBook

Pour installer les feuilles de style, décompresser et dépaqueter la distribution, la déplacer à un endroit convenable de l'aborescence comme, par exemple, `/usr/local/share/sgml`. (L'archive crée automatiquement un sous-répertoire.)

```
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

L'entrée de catalogue communément admise dans `/usr/local/share/sgml/catalog` peut également être réalisée :

```
CATALOG "docbook-dsssl-1.xx/catalog"
```

Comme les feuilles de styles changent assez souvent et qu'il est parfois avantageux d'essayer des versions alternatives, PostgreSQL™ n'utilise pas cette entrée du catalogue. Regarder dans la Section I.2.5, « Détection par **configure** » pour tout renseignement sur la manière de sélectionner une feuille de style.

I.2.4.4. Intallation de JadeTeX™

Pour installer et utiliser JadeTeX™, il faut une installation fonctionnelle de TeX™ et de LaTeX2e™, incluant également les paquetages tools™ et graphics™, Babel™, les polices AMS™ et AMS-LaTeX™, l'extension PSNFSS™ et le kit d'accompagnement de « 35 polices », le programme dvips™ pour la production de PostScript™, le paquetage de macros fancyhdr™, hyperref™, minitoc™, url™ et enfin ot2enc™. Tous peuvent être trouvés sur le *site web du CTAN*. L'installation du système TeX de base est en dehors du périmètre de cette introduction. Des paquetages binaires sont probablement disponibles pour tout système pouvant exécuter TeX.

Avant de pouvoir utiliser JadeTeX avec les sources de la documentation de PostgreSQL™, il va falloir augmenter la taille des structures de données internes de TeX. Des explications plus détaillées sont fournies dans les instructions d'installation de JadeTeX.

Lorsque tout cela est réalisé, JadeTeX peut être installé :

```
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

Les deux dernières commandes doivent être exécutées en `root`.

I.2.5. Détection par configure

Avant de pouvoir engendrer la documentation, le script `configure` doit être lancé, comme cela se fait pour la génération des programmes PostgreSQL™ eux-mêmes. La fin de l'affichage de l'exécution de ce script doit ressembler à :

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/share/sgml/docbook/stylesheet/dsssl/modular
checking for collateindex.pl... /usr/bin/collateindex.pl
checking for xsltproc... xsltproc
checking for osx... osx
```

Si ni `onsgmls` ni `nsgmls` n'ont été trouvés, certains des tests suivants seront ignorés. `onsgmls` fait partie du paquetage Jade. Les variables d'environnement `JADE` et `NSGMLS` peuvent être renseignées pour indiquer les emplacements des programmes s'ils ne sont pas trouvés automatiquement. Si « DocBook V4.2 » n'a pas été trouvé, c'est que le kit de la DTD DocBook n'est pas installé à un endroit où Jade peut le trouver ou que les fichiers catalogues ne sont pas correctement configurés. Il convient, dans ce cas, de se reporter aux conseils donnés plus haut. Les feuilles de style DocBook sont recherchées dans un certain nombre d'endroits standard, mais si elles sont placées en un autre endroit, il faut configurer la variable d'environnement `DOCBOOKSTYLE` avec cet emplacement et relancer `configure`.

I.3. Construire la documentation

Lorsque tout est en place, se placer dans le répertoire `doc/src/sgml` et lancer une des commandes décrites dans les sections suivantes afin de produire la documentation. (Il est impératif d'utiliser la version GNU de make.)

I.3.1. HTML

Pour engendrer la version HTML de la documentation, effectuer :

```
doc/src/sgml$ gmake html
```

Il s'agit également de la cible par défaut. La sortie apparaît dans le sous-répertoire `html`.

Pour créer un bon index, la construction doit se faire en plusieurs étapes identiques. Si l'index n'est pas utile et qu'il s'agit simplement de vérifier le résultat, on peut utiliser `draft` :

```
doc/src/sgml$ gmake draft
```

Pour construire la documentation sous la forme d'une seule page HTML, utilisez :

```
doc/src/sgml$ gmake postgres.html
```

I.3.2. Pages man (de manuel)

Nous utilisons les feuilles de style XSL DocBook pour convertir les pages de références DocBook™ dans un format *roff compatible avec les pages man. Les pages man sont également distribuées sous la forme d'une archive tar, à l'instar de la version HTML. Pour créer les pages man, utiliser les commandes :

```
cd doc/src/sgml
gmake man
```

I.3.3. Sortie pour l'impression via JadeTex

Si JadeTex est utilisé pour produire une documentation pour impression, une des commandes suivantes peut être utilisée :

- pour produire un fichier PostScript à partir du DVI au format A4 :

  ```
  doc/src/sgml$ gmake postgres-A4.ps
  ```

  Au format letter :

  ```
  doc/src/sgml$ gmake postgres-US.ps
  ```

- pour créer un PDF :

  ```
  doc/src/sgml$ gmake postgres-A4.pdf
  ```

  ou

  ```
  doc/src/sgml$ gmake postgres-US.pdf
  ```

  (Bien entendu, la version PDF peut être obtenue à partir d'un document PostScript, mais la génération directe permet de bénéficier des liens et autres fonctionnalités avancées dans le PDF.)

Lors de l'utilisation de JadeTeX pour la construction de la documentation PostgreSQL, il est probablement nécessaire d'augmenter la valeur de certains paramètres internes de TeX. Ils sont configurables dans le fichier `texmf.cnf`. Les paramètres suivants ont fonctionné quand ce texte a été écrit :

```
hash_extra.jadetex  = 200000
hash_extra.pdfjadetex  = 200000
pool_size.jadetex = 2000000
pool_size.pdfjadetex = 2000000
string_vacancies.jadetex = 150000
string_vacancies.pdfjadetex = 150000
max_strings.jadetex = 300000
max_strings.pdfjadetex = 300000
save_size.jadetex = 15000
save_size.pdfjadetex = 15000
```

I.3.4. Texte dépassant du cadre

Quelque fois, le texte est trop long pour tenir dans les marges d'impression. Dans certains cas extrêmes, il est trop long pour tenir sur la page imprimée. Cela arrive principalement avec du code et des gros tableaux. Ce texte génère des messages « Overfull hbox » dans les traces de TeX, c'est-à-dire dans `postgres-US.log` ou dans `postgres-A4.log`. Un pouce contient 72 points, donc tout ce qui dépasse 72 points ne tiendra probablement pas dans la page imprimée (en supposant une marge d'un pouce). Pour retrouver le texte SGML causant cette surcharge, recherchez le premier numéro de page mentionné au-dessus du message d'erreur, par exemple `[50 ###]` (page 50), puis regardez à la page suivante (toujours dans notre exemple, la page 51) dans le fichier PDF pour voir le texte dépassant et ajustez ainsi le fichier SGML.

I.3.5. Version imprimable via RTF

Une version imprimable de la documentation de PostgreSQL™ peut être obtenue en la convertissant en RTF et en y appliquant des corrections de formatage mineures à l'aide d'une suite de logiciels de bureau. En fonction des capacités de cette dernière, la do-

cumentation peut ensuite être convertie dans les formats PostScript et/ou PDF. La procédure ci-dessous décrit cette procédure pour Applixware™.

> **Note**
>
> Il apparaît que la version actuelle de la documentation de PostgreSQL™ produit quelques bogues au niveau d'OpenJade, dépassant notamment la taille maximale de traitement. Si le processus de génération de la version RTF reste suspendu un long moment et que le fichier de sortie reste vide, c'est que ce problème est survenu. (Une génération normale doit prendre 5 à 10 minutes, ne pas s'avouer vaincu trop vite.)

Procédure I.1. Nettoyage du RTF avec Applixware™

OpenJade ne précise pas de style par défaut pour le corps du texte. Dans le passé, ce problème non diagnostiqué engendrait un long processus de génération du sommaire. Grâce à la grande aide des gens d'Applixware™, le symptôme a été diagnostiqué et un contournement est disponible.

1. Produire la version RTF en saisissant :

   ```
   doc/src/sgml$ gmake postgres.rtf
   ```

2. Réparer le fichier RTF afin de préciser correctement tous les styles et en particulier le style par défaut (default). Si le document contient des sections `refentry`, il faut remplacer les éléments de formatage qui relient le paragraphe précédent au paragraphe courant par un lien entre le paragraphe courant et le paragraphe suivant. Un utilitaire, **fixrtf**, est disponible dans `doc/src/sgml` afin de permettre ces corrections :

   ```
   doc/src/sgml$ ./fixrtf --refentry postgres.rtf
   ```

   Le script ajoute `{\s0 Normal;}` comme style de rang zéro dans le document. D'après Applixware™, le standard RTF prohibe l'utilisation implicite d'un style de rang 0 alors que Microsoft Word est capable de gérer ce cas. Afin de réparer les sections `refentry`, le script remplace les balises `\keepn` par `\keep`.

3. Ouvrir un nouveau document dans Applixware Words™ et y importer le fichier RTF.
4. Produire une nouvelle table des matières avec Applixware™.

   a. sélectionner les lignes existantes de la table des matières, du premier caractère de la table au dernier caractère de la dernière ligne ;

   b. construire une nouvelle table des matières en allant dans le menu Tools → Book Building → Create Table of Contents. Sélectionner les trois premiers niveaux de titres pour l'inclusion dans la table des matières. Cela remplace les lignes existantes importées du RTF en une table des matières issue d'Applixware™ ;

   c. Ajuster le formatage de la table des matières à l'aide de Format → Style, en sélectionnant chacun des trois styles de la table des matières et en ajustant les indentations pour `First` et `Left`. Utiliser les valeurs suivantes :

Style	Indentation de First (pouces)	Indentation de Left (pouces)
`TOC-Heading 1`	`0.4`	`0.4`
`TOC-Heading 2`	`0.8`	`0.8`
`TOC-Heading 3`	`1.2`	`1.2`

5. Sur l'ensemble du document :

   - ajuster les sauts de page ;
   - ajuster la largeur des colonnes des tables.

6. Remplacer les numéros de pages justifiés à droite dans les portions d'exemple et de figures de la table des matières avec les bonnes valeurs. Cela ne prend que quelques minutes.
7. Supprimer la section d'index du document si elle est vide.
8. Régénérer et ajuster la table des matières.

a. sélectionner un champ de la table des matières ;

b. sélectionner le menu Tools → Book Building → Create Table of Contents ;

c. délier la table des matières en sélectionnant le menu Tools → Field Editing → Unprotect ;

d. supprimer la première ligne de la table des matières, qui est un des champs de la table elle-même.

9. Sauvegarder le document au format natif Applixware Words™ pour que les modifications de dernière minute soient plus faciles par la suite.

10. « Imprimer » le document dans un fichier au format PostScript.

I.3.6. Fichiers texte

Les instructions d'installation sont aussi distribuées sous la forme d'un fichier texte, au cas où cela se révélerait nécessaire (par exemple si des outils plus avancés n'étaient pas disponibles). Le fichier `INSTALL` correspond au Chapitre 15, Procédure d'installation de PostgreSQL™ du code source, avec quelques changements mineurs pour tenir compte de contextes différents. Pour recréer le fichier, se placer dans le répertoire `doc/src/sgml` et entrer la commande **`gmake INSTALL`**.

Dans le passé, les notes de versions et les instructions pour les tests de régression étaient aussi distribuées sous la forme de fichiers textes mais ce n'est plus le cas.

I.3.7. Vérification syntaxique

Fabriquer la documentation peut prendre beaucoup de temps. Il existe cependant une méthode, qui ne prend que quelques secondes, permettant juste de vérifier que la syntaxe est correcte dans les fichiers de documentation :

```
doc/src/sgml$ gmake check
```

I.4. Écriture de la documentation

SGML et DocBook™ ne souffrent pas du foisonnement d'outils d'édition OpenSource. Le plus commun d'entre eux est l'éditeur Emacs™/XEmacs™ qui dispose d'un mode d'édition approprié. Sur certains systèmes, ces outils sont fournis sous la forme d'une installation complète (un même paquetage ou ensemble).

I.4.1. Emacs/PSGML

PSGML™ est le mode d'édition de documents SGML le plus répandu et le plus puissant. S'il est correctement configuré, il permet d'utiliser Emacs pour insérer des balises et en vérifier la cohérence. Il peut aussi être utilisé pour le HTML. Visiter le *site web de PSGML* pour le télécharger, avoir des informations sur l'installation et pour disposer d'une documentation détaillée.

L'auteur de PSGML™ suppose que le répertoire principal pour la DTD SGML est `/usr/local/lib/sgml`. Si, comme c'est le cas dans les exemples de ce chapitre, le répertoire `/usr/local/share/sgml`est utilisé, il faut modifier la variable d'environnement `SGML_CATALOG_FILES` ou personnaliser l'installation de PSGML™ (son manuel indique la procédure).

Ajouter les lignes suivantes dans le fichier d'environnement `~/.emacs` (en ajustant les noms des répertoires pour qu'il convienne au système de fichiers) :

```
; ********** pour le mode SGML (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "./reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))
(setq sgml-ecat-files nil)

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

et dans le même fichier ajouter l'entrée pour le SGML dans la définition (existante) pour `auto-mode-alist` :

```
(setq
  auto-mode-alist
  '(("\\.xml$" . sgml-mode)
   ))
```

La distribution PostgreSQL™ inclut un fichier DTD pré-traité contenant des définitions appelé `reference.ced`. Il peut s'évérer plus confortable pour gérer des fichiers séparés d'insérer une déclaration `DOCTYPE` propre lors de l'édition d'un fichier avec PSGML™. Si l'on considère le fichier source de cette annexe, par exemple, il est possible de préciser que le document est une instance « appendix » d'un document DocBook en ajoutant en première ligne ceci :

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

Ceci signifie que, quelque soit l'outil lisant le SGML, il le fait correctement et que la validité du document peut être vérifiée avec **nsgmls -s docguide.xml**. (Cependant, il faut supprimer la ligne avant de fabriquer le document complet à partir des différents fichiers.)

I.4.2. Autres modes pour Emacs

GNU Emacs™ dispose d'un mode SGML natif différent qui n'est pas aussi puissant que PSGML™, mais qui a le bénéfice d'être plus simple et léger. Il offre la coloration syntaxique (en mode font lock), ce qui peut être utile.

Norm Walsh propose *un mode majeur spécifique à DocBook* qui dispose également de la coloration syntaxique et d'un certain nombre de fonctions permettant de réduire le temps de saisie.

I.5. Guide des styles

I.5.1. Pages de références

Les pages de références obéissent à des règles de standardisation. De cette façon, les utilisateurs retrouvent plus rapidement l'information souhaitée, et cela encourage également les rédacteurs à documenter tous les aspects relatifs à une commande. Cette cohérence n'est pas uniquement souhaitée pour les pages de références PostgreSQL™, mais également pour les pages de références fournies par le système d'exploitation et les autres paquetages. C'est pour cela que les règles suivantes ont été développées. Elles sont, pour la plupart, cohérentes avec les règles similaires établies pour différents systèmes d'exploitation.

Les pages de référence qui décrivent des commandes exécutables doivent contenir les sections qui suivent dans l'ordre indiqué. Les sections qui ne sont pas applicables peuvent être omises. Des sections de premier niveau additionnelles ne doivent être utilisées que dans des circonstances particulières ; dans la plupart des cas, les informations qui y figureraient relèvent de la section « Usage ».

Nom
: Cette section est produite automatiquement. Elle contient le nom de la commande et une courte phrase résumant sa fonctionnalité.

Synopsis
: Cette section contient le schéma syntaxique de la commande. Le synopsis ne doit en général pas lister toutes les options de la commande, cela se fait juste au dessous. À la place, il est important de lister les composantes majeures de la ligne de commande comme, par exemple, l'emplacement des fichiers d'entrée et sortie.

Description
: Plusieurs paragraphes décrivant ce que permet de faire la commande.

Options
: Une liste décrivant chacune des options de la ligne de commande. S'il y a beaucoup d'options, il est possible d'utiliser des sous-sections.

Code de sortie
: Si le programme utilise 0 en cas de succès et une valeur non-nulle dans le cas contraire, il n'est pas nécessaire de le documenter. S'il y a une signification particulière au code de retour différent de zéro, c'est ici qu'ils faut décrire les codes de retour.

Utilisation
: Décrire ici tout sous-programme ou interface de lancement du programme. Si le programme n'est pas interactif, cette section peut être omise. Dans les autres cas, cette section est un fourre-tout pour les fonctionnalités disponibles lors de l'utilisation du programme. Utiliser des sous-sections si cela est approprié.

Environnement
: Lister ici toute variable d'environnement utilisable. Il est préférable de ne rien omettre. Même des variables qui semblent triviales, comme `SHELL`, peuvent être d'un quelconque intérêt pour l'utilisateur.

Fichiers
: Lister tout fichier que le programme peut accéder, même implicitement. Les fichiers d'entrée ou de sortie indiqués sur la ligne de commande ne sont pas listés, mais plutôt les fichiers de configuration, etc.

Diagnostiques
: C'est ici que l'ont trouve l'explication de tout message inhabituel produit par le programme. Il est inutile de lister tous les messages d'erreur possibles. C'est un travail considérable et cela n'a que peu d'intérêt dans la pratique. En revanche, si les messages d'erreurs ont un format particulier, que l'utilisateur peut traiter, c'est dans cette section que ce format doit être décrit.

Notes
: Tout ce qui ne peut être contenu dans les autres sections peut être placé ici. En particulier les bogues, les carences d'une implantation, les considérations de sécurité et les problèmes de compatibilité.

Exemples
: Les exemples.

Historique
: S'il y a eu des échéances majeures dans l'histoire du programme, elles peuvent être listées ici. Habituellement, cette section peut être omise.

Voir aussi
: Des références croisées, listées dans l'ordre suivant : pages de référence vers d'autres commandes PostgreSQL™, pages de référence de commmandes SQL de PostgreSQL™, citation des manuels PostgreSQL™, autres pages de référence (système d'exploitation, autres paquetages, par exemple), autre documentation. Les éléments d'un même groupe sont listés dans l'ordre alphabétique.

Les pages de référence qui décrivent les commandes SQL doivent contenir les sections suivantes : « Nom », « Synopsis », « Description », « Paramètres », « Sorties », « Notes », « Exemples », « Compatibilité », « Historique », « Voir aussi ». La section « Paramètres » est identique à la section « Options » mais elle offre plus de liberté sur les clauses qui peuvent être listées. La section « Sorties » n'est nécessaire que si la commande renvoie autre chose qu'un complément de commande par défaut. La section « Compatibilité » doit expliquer dans quelle mesure une commande se conforme au standard SQL, ou avec quel autre système de gestion de base de données elle est compatible. La section « Voir aussi » des commandes SQL doit lister les commandes SQL avant de faire de faire référence aux programmes.

Annexe J. Acronymes

La suite présente la liste des acronymes habituellement utilisés dans la documentation de PostgreSQL™ et les discussions qui tournent autour de PostgreSQL™.

ANSI
: *American National Standards Institute*, l'Institut National Américain des Standards

API
: *Application Programming Interface*, interface de programmation applicative

ASCII
: *American Standard Code for Information Interchange*, Code Standard Américain pour l'échange d'informations

BKI
: Backend Interface, interface serveur

CA
: *Certificate Authority*, autorité de certification

CIDR
: *Classless Inter-Domain Routing*, routage inter-domaine dépourvu de classe

CPAN
: *Comprehensive Perl Archive Network*, réseau d'archives Perl

CRL
: *Certificate Revocation List*, liste de révocation de certificats

CSV
: *Comma Separated Values*, valeurs séparées par des virgules (format de fichier présentant les données en ligne en séparant les colonnes par des virgules ou point-virgules)

CTE
: Common Table Expression

CVE
: *Common Vulnerabilities and Exposures*, vulnérabilités usuelles

DBA
: *Database Administrator*, administrateur de bases de données

DBI
: *Database Interface (Perl)*, interface avec la base de données

DBMS
: *Database Management System*, système de gestion de bases de données (SGBD)

DDL
: *Data Definition Language*, langage de définition des données, qui regroupe les commandes SQL telles que **CREATE TABLE**, **ALTER USER**

DML
: *Data Manipulation Language*, langage de manipulation des données, qui regroupe les commandes SQL telles que **INSERT**, **UPDATE**, **DELETE**

DST
: *Daylight Saving Time*, gestion des changements d'heure instaurés pour des raisons d'économie d'énergie

ECPG
: Embedded C for PostgreSQL, C embarqué pour PostgreSQL

ESQL
: *Embedded SQL*, SQL embarqué

FAQ
: *Frequently Asked Questions*, Foire aux Questions

FSM
: Free Space Map, cartographie de l'espace libre

GEQO
: Genetic Query Optimizer, optimiseur de requête génétique

GIN
: Generalized Inverted Index, index inversé généralisé

GiST
: Generalized Search Tree, arbre de recherche généralisé

Git
: *Git*

GMT
: *Greenwich Mean Time*, heure au méridien de Greenwich

GSSAPI
: *Generic Security Services Application Programming Interface*, Interface de programmation applicative pour les services de sécurité génériques

GUC
: Grand Unified Configuration, Configuration générale unifiée le sous-système PostgreSQL™ qui gère la configuration du serveur

HBA
: Host-Based Authentication, authentification fondée sur l'hôte

HOT
: *Heap-Only Tuples*, tuples en mémoire seule

IEC
: *International Electrotechnical Commission*, Commission internationale d'électrotechnique

IEEE
: *Institute of Electrical and Electronics Engineers*, institut des ingénieurs en électrique et électronique

IPC
: *Inter-Process Communication*, communication inter-processus

ISO
: *International Organization for Standardization*, Organisation de standardisation internationale

ISSN
: *International Standard Serial Number*, numéro de série international standardisé

JDBC
: *Java Database Connectivity*, connecteur de bases de données en Java

LDAP
: *Lightweight Directory Access Protocol*, protocole léger d'accès aux annuaires

MSVC
: *Microsoft Visual C™*

MVCC
: Multi-Version Concurrency Control, Contrôle de concurrence par multi-versionnage

NLS
: *National Language Support*, support des langages nationaux

ODBC
: *Open Database Connectivity*, connecteur ouvert de bases de données

OID
: Object Identifier, identifiant objet

OLAP
: *Online Analytical Processing*, traitement analytique en ligne

OLTP
: *Online Transaction Processing*, traitement transactionnel en ligne

ORDBMS
: *Object-Relational Database Management System*, système de gestion de bases de données relationnelles objet (SGBR/O)

PAM
Pluggable Authentication Modules, modules d'authentification connectables

PGSQL
PostgreSQL™

PGXS
PostgreSQL™ Extension System, système d'extension de PostgreSQL

PID
Process Identifier, identifiant processus

PITR
Point-In-Time Recovery, Restauration d'un instantané

PL
Programming Languages, langages de programmation côté serveur

POSIX
Portable Operating System Interface, interface portable au système d'exploitation

RDBMS
Relational Database Management System, système de gestion de bases de données relationnelles (SGBDR)

RFC
Request For Comments

SGML
Standard Generalized Markup Language

SPI
Server Programming Interface, interface de programmation serveur

SQL
Structured Query Language

SRF
Set-Returning Function, fonction retournant un ensemble

SSH
Secure Shell

SSL
Secure Sockets Layer

SSPI
Security Support Provider Interface

SYSV
Unix System V

TCP/IP
Transmission Control Protocol (TCP) / Internet Protocol (IP)

TID
Tuple Identifier, identifiant de tuple

TOAST
The Oversized-Attribute Storage Technique, technique de stockage des données surdimensionnées

TPC
Transaction Processing Performance Council

URL
Uniform Resource Locator

UTC
Coordinated Universal Time

UTF
Unicode Transformation Format

UTF8

Eight-Bit Unicode Transformation Format

UUID
Universally Unique Identifier, identifiant universel unique

WAL
Write-Ahead Log

XID
Transaction Identifier, identifiant de transaction

XML
Extensible Markup Language

Annexe K. Traduction française

Ce manuel est une traduction de la version originale. La dernière version à jour est disponible sur le site *docs.postgresql.fr*. Ce site est hébergé par l'*association PostgreSQLFr*, dont le but est la promotion du logiciel PostgreSQL. Si cette documentation vous a plu, *adhérez à l'association* et, si vous en avez la possibilité, contribuez.

Cette annexe supplémentaire présente la liste des contributeurs, un historique limité aux dix dernières modifications et la procédure pour remonter des informations aux traducteurs.

Voici les contributeurs par ordre alphabétique (prénom puis nom) :

- Antoine
- Bruno Levêque
- Christophe 'nah-ko' Truffier
- Christophe Bredel
- Christophe 'KrysKool' Chauvet
- Claude Castello
- Claude Thomassin
- Emmanuel Magin
- Emmanuel Seyman
- Erwan Duroselle
- Fabien Foglia
- Fabien Grumelard
- Florence Cousin
- François Suter
- Frédéric Andres
- Guillaume 'gleu' Lelarge
- Hervé Dumont
- Jacques Massé
- Jean-Christophe 'jca' Arnu
- Jean-Christophe Weis
- Jean-Max Reymond
- Jean-Michel Poure
- Jérôme Seyler
- Marc Blanc
- Marc Cousin
- Matthieu Clavier
- Philippe Rimbault
- Pierre Jarillon
- Stéphane 'SAS' Schildknecht
- Thomas Reiss
- Thomas Silvi
- Vincent Picavet
- Yves Darmaillac

Tous ont participé en traduisant, en relisant ou en rédigeant un rapport de bogue. N'hésitez pas à nous signaler toute personne qui aurait été oubliée.

De même que tout logiciel peut contenir des erreurs de programmation, toute traduction n'est pas exempte d'erreurs : faute d'orthographe, faute de grammaire, erreur de saisie, voire, bien pire, contresens. Bien que l'équipe de traduction passe beaucoup de temps à relire la documentation traduite, il lui arrive de laisser passer des erreurs. Elle a donc besoin de vous.

Si vous découvrez une erreur ou si un passage n'est pas compréhensible, l'équipe de traduction souhaite le savoir. Pour cela, vous pouvez envoyer un mail au coordinateur (*Guillaume Lelarge*). C'est le moyen le plus simple et le plus sûr. Tout problème remonté sera pris en considération.

Bibliographie

Références sélectionnées et lectures autour du SQL et de PostgreSQL™.

Quelques livres blancs et rapports techniques réalisés par l'équipe d'origine de développement de POSTGRES™ sont disponibles sur le *site web* du département des sciences informatiques de l'université de Californie.

Livres de référence sur SQL

Textes de référence sur les fonctionnalités de SQL.

[bowman01] *The Practical SQL Handbook*. Bowman et al, 2001. Using SQL Variants. Fourth Edition. Judith Bowman, Sandra Emerson, et Marcy Darnovsky. 0-201-70309-2. 2001. Addison-Wesley Professional. Copyright © 2001 Addison-Wesley Longman, Inc..

[date97] *A Guide to the SQL Standard*. Date and Darwen, 1997. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date et Hugh Darwen. 0-201-96426-0. 1997. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

[date04] *An Introduction to Database Systems*. Date, 2004. Eighth Edition. C. J. Date. 0-321-19784-4. 2003. Addison-Wesley. Copyright © 2004 Pearson Education, Inc..

[elma04] *Fundamentals of Database Systems*. Fourth Edition. Ramez Elmasri et Shamkant Navathe. 0-321-12226-7. 2003. Addison-Wesley. Copyright © 2004.

[melt93] *Understanding the New SQL*. Melton and Simon, 1993. A complete guide. Jim Melton et Alan R. Simon. 1-55860-245-3. 1993. Morgan Kaufmann. Copyright © 1993 Morgan Kaufmann Publishers, Inc..

[ull88] *Principles of Database and Knowledge*. Base Systems. Ullman, 1988. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

Documentation spécifique sur PostgreSQL

Cette section concerne la documentation relative à PostgreSQL.

[sim98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Simkovics, 1998. Stefan Simkovics. 29 novembre 1998. Département des systèmes d'informations, université de technologie de Vienne. Vienne, Autriche.

[yu95] *The Postgres95. User Manual*. Yu and Chen, 1995. A. Yu et J. Chen. . 5 septembre 1995. Université de Californie. Berkeley, Californie.

[fong] *The design and implementation of the POSTGRES™ query optimizer*. Zelaine Fong. Université de Californie, Berkeley, département des sciences informatiques.

Procédures et articles

Cette section recense les articles et brèves.

[olson93] *Partial indexing in POSTGRES: research project*. Olson, 1993. Nels Olson. 1993. UCB Engin T7.49.1993 O676. Université de Californie. Berkeley, Californie.

« A Unified Framework for Version Modeling Using Production Rules in a Database System ». Ong and Goh, 1990. L. Ong et J. Goh. *ERL Technical Memorandum M90/33*. Avril 1990. Université de Californie. Berkely, Californie.

« *The POSTGRES™ data model* ». Rowe and Stonebraker, 1987. L. Rowe et M. Stonebraker. VLDB Conference. Septembre 1987. Brighton, Angleterre. .

« Generalized Partial Indexes *(version en cache)* ». Seshardri, 1995. P. Seshadri et A. Swami. Eleventh International Conference on Data Engineering. 6-10 mars 1995. Taipeh, Taiwan. . 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, Californie. 420-7.

« *The design of POSTGRES™* ». Stonebraker and Rowe, 1986. M. Stonebraker et L. Rowe. ACM-SIGMOD Conference on Management of Data. Mai 1986. Washington, DC. .

« The design of the POSTGRES. rules system ». Stonebraker, Hanson, Hong, 1987. M. Stonebraker, E. Hanson, et C. H. Hong. IEEE Conference on Data Engineering. Février 1987. Los Angeles, Californie. .

« *The design of the POSTGRES™ storage system* ». Stonebraker, 1987. M. Stonebraker. VLDB Conference. Septembre 1987.

Brighton, Angleterre. .

« *A commentary on the POSTGRES™ rules system* ». Stonebraker et al, 1989. M. Stonebraker, M. Hearst, et S. Potamianos. *SIGMOD Record 18(3)*. Septembre 1989.

« *The case for partial indexes* ». Stonebraker, M, 1989b. M. Stonebraker. *SIGMOD Record 18(4)*. 4-11. Décembre 1989.

« *The implementation of POSTGRES™* ». Stonebraker, Rowe, Hirohama, 1990. M. Stonebraker, L. A. Rowe, et M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. Mars 1990.

« *On Rules, Procedures, Caching and Views in Database Systems* ». Stonebraker et al, ACM, 1990. M. Stonebraker, A. Jhingran, J. Goh, et S. Potamianos. ACM-SIGMOD Conference on Management of Data. Juin 1990. .