Formation DEV42

Développement avancé

23.09

Table des matières

Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 1
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 2
Sur ce document . . . 2

1/ SQL pour l'analyse de données **5**
1.1 Préambule . . . 6
1.1.1 Menu . . . 6
1.1.2 Objectifs . . . 6
1.2 Agrégats . . . 7
1.2.1 Agrégats avec GROUP BY . . . 8
1.2.2 GROUP BY : principe . . . 10
1.2.3 GROUP BY : exemples . . . 10
1.2.4 Agrégats et ORDER BY . . . 11
1.2.5 Utiliser ORDER BY avec un agrégat . . . 12
1.3 Clause FILTER . . . 13
1.3.1 Filtrer avec CASE . . . 13
1.3.2 Filtrer avec FILTER . . . 14
1.4 Fonctions de fenêtrage . . . 15
1.4.1 Regroupement . . . 16
1.4.2 Regroupement : exemple . . . 17
1.4.3 Regroupement : principe . . . 17
1.4.4 Regroupement : syntaxe . . . 18
1.4.5 Tri . . . 18
1.4.6 Tri : exemple . . . 19
1.4.7 Tri : exemple avec une somme . . . 20
1.4.8 Tri : principe . . . 21
1.4.9 Tri : syntaxe . . . 21
1.4.10 Regroupement et tri . . . 22
1.4.11 Regroupement et tri : exemple . . . 22
1.4.12 Regroupement et tri : principe . . . 24
1.4.13 Regroupement et tri : syntaxe . . . 24
1.4.14 Fonctions analytiques . . . 25
1.4.15 lead() et lag() . . . 26
1.4.16 lead() et lag() : exemple . . . 26
1.4.17 lead() et lag() : principe . . . 27
1.4.18 first/last/nth_value . . . 27
1.4.19 first/last/nth_value : exemple . . . 28
1.4.20 Clause WINDOW . . . 29
1.4.21 Clause WINDOW : syntaxe . . . 30

1.4.22 Définition de la fenêtre . . . 30
1.4.23 Définition de la fenêtre : RANGE . . . 31
1.4.24 Définition de la fenêtre : ROWS . . . 31
1.4.25 Définition de la fenêtre : GROUPS . . . 32
1.4.26 Définition de la fenêtre : EXCLUDE . . . 32
1.4.27 Définition de la fenêtre : exemple . . . 33
1.5 WITHIN GROUP . . . 34
1.5.1 WITHIN GROUP : exemple . . . 34
1.6 Grouping Sets . . . 36
1.6.1 GROUPING SETS : jeu de données . . . 36
1.6.2 GROUPING SETS : exemple visuel . . . 38
1.6.3 GROUPING SETS : exemple ordre sql . . . 38
1.6.4 GROUPING SETS : équivalent . . . 39
1.6.5 ROLLUP . . . 40
1.6.6 ROLLUP : exemple visuel . . . 40
1.6.7 ROLLUP : exemple ordre sql . . . 41
1.6.8 CUBE . . . 43
1.6.9 CUBE : exemple visuel . . . 44
1.6.10 CUBE : exemple ordre sql . . . 44
1.7 Travaux pratiques . . . 47
1.8 Travaux pratiques (solutions) . . . 49

2/ Types avancés **59**
2.1 Types composés : généralités . . . 60
2.2 hstore . . . 62
2.2.1 hstore : exemple . . . 62
2.3 JSON . . . 64
2.3.1 Type json . . . 65
2.3.2 Type jsonb . . . 66
2.3.3 JSON : Exemple d'utilisation . . . 67
2.3.4 JSON : Affichage de champs . . . 68
2.3.5 Conversions jsonb / relationnel . . . 69
2.3.6 JSON : performances . . . 70
2.3.7 jsonb : indexation (1/2) . . . 71
2.3.8 jsonb : indexation (2/2) . . . 72
2.3.9 SQL/JSON & JSONpath . . . 73
2.3.10 Extension jsQuery . . . 74
2.4 XML . . . 76
2.5 Objets binaires . . . 78
2.5.1 bytea . . . 79
2.5.2 Large Object . . . 80
2.6 Quiz . . . 82
2.7 Travaux pratiques . . . 83
2.7.1 Hstore (Optionnel) . . . 83
2.7.2 jsonb . . . 84

2.7.3 Large Objects . . . 85
2.8 Travaux pratiques (solutions) . . . 87
2.8.1 Hstore (Optionnel) . . . 87
2.8.2 jsonb . . . 90
2.8.3 Large Objects . . . 94

3/ PL/pgSQL : les bases 95
3.1 Préambule . . . 96
3.1.1 Au menu . . . 96
3.1.2 Objectifs . . . 97
3.2 Introduction . . . 98
3.2.1 Qu'est-ce qu'un PL ? . . . 98
3.2.2 Quels langages PL sont disponibles ? . . . 98
3.2.3 Langages *trusted* vs *untrusted* . . . 99
3.2.4 Les langages PL de PostgreSQL . . . 100
3.2.5 Intérêts de PL/pgSQL en particulier . . . 101
3.2.6 Les autres langages PL ont toujours leur intérêt . . . 102
3.2.7 Routines / Procédures stockées / Fonctions . . . 103
3.3 Installation . . . 105
3.3.1 Installation des binaires nécessaires . . . 105
3.3.2 Activer/désactiver un langage . . . 106
3.3.3 Langage déjà installé ? . . . 106
3.4 Exemples de fonctions & procédures . . . 108
3.4.1 Fonction PL/pgSQL simple . . . 108
3.4.2 Exemple de fonction SQL . . . 108
3.4.3 Exemple de fonction PL/pgSQL utilisant la base . . . 110
3.4.4 Exemple de fonction PL/Perl complexe . . . 111
3.4.5 Exemple de fonction PL/pgSQL complexe . . . 112
3.4.6 Exemple de procédure . . . 113
3.4.7 Exemple de bloc anonyme en PL/pgSQL . . . 114
3.5 Utiliser une fonction ou une procédure . . . 116
3.5.1 Invocation d'une fonction ou procédure . . . 116
3.6 Création et maintenance des fonctions et procédures . . . 118
3.6.1 Création . . . 118
3.6.2 Langage . . . 119
3.6.3 Structure d'une routine PL/pgSQL . . . 119
3.6.4 Structure d'une routine PL/pgSQL (suite) . . . 120
3.6.5 Blocs nommés . . . 120
3.6.6 Modification du code d'une routine . . . 121
3.6.7 Modification des méta-données d'une routine . . . 121
3.6.8 Suppression d'une routine . . . 122
3.6.9 Utilisation des guillemets . . . 122
3.7 Paramètres et retour des fonctions et procédures . . . 124
3.7.1 Version minimaliste . . . 124
3.7.2 Paramètres IN, OUT & retour . . . 124

3.7.3 Type en retour : 1 valeur simple . . . 125
3.7.4 Type en retour : 1 lignes, plusieurs champs . . . 126
3.7.5 Retour multi-lignes . . . 127
3.7.6 Gestion des valeurs NULL . . . 129
3.8 Variables en PL/pgSQL . . . 131
3.8.1 Clause DECLARE . . . 131
3.8.2 Constantes . . . 132
3.8.3 Types de variables . . . 132
3.8.4 Type ROW - 1 . . . 132
3.8.5 Type ROW - 2 . . . 133
3.8.6 Type RECORD . . . 133
3.8.7 Type RECORD : exemple . . . 134
3.9 Exécution de requête dans un bloc PL/pgSQL . . . 135
3.9.1 Requête dans un bloc PL/pgSQL . . . 135
3.9.2 Affectation d'une valeur à une variable . . . 135
3.9.3 Exécution d'une requête . . . 136
3.9.4 Exécution d'une requête sans besoin du résultat . . . 137
3.10 SQL dynamique . . . 139
3.10.1 EXECUTE d'une requête . . . 139
3.10.2 EXECUTE & requête dynamique : injection SQL . . . 139
3.10.3 EXECUTE & requête dynamique : 3 possibilités . . . 140
3.10.4 EXECUTE & requête dynamique (suite) . . . 142
3.10.5 Outils pour construire une requête dynamique . . . 142
3.11 Structures de contrôle en PL/pgSQL . . . 144
3.11.1 Tests conditionnels - 2 . . . 144
3.11.2 Tests conditionnels : CASE . . . 144
3.11.3 Boucle LOOP/EXIT/CONTINUE : syntaxe . . . 145
3.11.4 Boucle LOOP/EXIT/CONTINUE : exemple . . . 146
3.11.5 Boucle WHILE . . . 146
3.11.6 Boucle FOR : syntaxe . . . 146
3.11.7 Boucle FOR … IN … LOOP : parcours de résultat de requête . . . 147
3.11.8 Boucle FOREACH . . . 148
3.12 Autres propriétés des fonctions . . . 149
3.12.1 Politique de sécurité . . . 149
3.12.2 Optimisation des fonctions . . . 150
3.12.3 Parallélisation . . . 150
3.13 Utilisation de fonctions dans les index . . . 152
3.14 Conclusion . . . 157
3.14.1 Pour aller plus loin . . . 157
3.14.2 Questions . . . 157
3.15 Quiz . . . 158
3.16 Travaux pratiques . . . 159
3.16.1 Hello . . . 159
3.16.2 Division . . . 159
3.16.3 SELECT sur des tables dans les fonctions . . . 159

3.16.4 Multiplication . . . 160
3.16.5 Salutations . . . 160
3.16.6 Inversion de chaîne . . . 161
3.16.7 Jours fériés . . . 161
3.17 Travaux pratiques (solutions) . . . 163
3.17.1 Hello . . . 163
3.17.2 Division . . . 163
3.17.3 SELECT sur des tables dans les fonctions . . . 165
3.17.4 Multiplication . . . 166
3.17.5 Salutations . . . 170
3.17.6 Inversion de chaîne . . . 171
3.17.7 Jours fériés . . . 172

4/ PL/pgSQL avancé **179**
4.1 Préambule . . . 180
4.1.1 Au menu . . . 180
4.1.2 Objectifs . . . 180
4.2 Routines variadic . . . 181
4.2.1 Routines variadic : introduction . . . 181
4.2.2 Routines variadic : exemple . . . 181
4.2.3 Routines variadic : exemple PL/pgSQL . . . 182
4.3 Routines polymorphes . . . 183
4.3.1 Routines polymorphes : introduction . . . 183
4.3.2 Routines polymorphes : anyelement . . . 184
4.3.3 Routines polymorphes : anyarray . . . 184
4.3.4 Routines polymorphes : exemple . . . 184
4.3.5 Routines polymorphes : tests . . . 185
4.3.6 Routines polymorphes : problème . . . 185
4.4 Fonctions trigger . . . 187
4.4.1 Fonctions trigger : introduction . . . 187
4.4.2 Fonctions trigger : variables (1/5) . . . 188
4.4.3 Fonctions trigger : variables (2/5) . . . 188
4.4.4 Fonctions trigger : variables (3/5) . . . 189
4.4.5 Fonctions trigger : variables (4/5) . . . 189
4.4.6 Fonctions trigger : variables (5/5) . . . 190
4.4.7 Fonctions trigger : retour . . . 191
4.4.8 Fonctions trigger : exemple - 1 . . . 192
4.4.9 Fonctions trigger : exemple - 2 . . . 192
4.4.10 Options de CREATE TRIGGER . . . 192
4.4.11 Tables de transition . . . 193
4.5 Curseurs . . . 196
4.5.1 Curseurs : introduction . . . 196
4.5.2 Curseurs : déclaration d'un curseur . . . 196
4.5.3 Curseurs : ouverture d'un curseur . . . 197
4.5.4 Curseurs : ouverture d'un curseur lié . . . 197

4.5.5 Curseurs : récupération des données . . . 198
4.5.6 Curseurs : récupération des données . . . 198
4.5.7 Curseurs : modification des données . . . 199
4.5.8 Curseurs : fermeture d'un curseur . . . 199
4.5.9 Curseurs : renvoi d'un curseur . . . 199
4.6 Contrôle transactionnel . . . 201
4.7 Gestion des erreurs . . . 203
4.7.1 Gestion des erreurs : introduction . . . 203
4.7.2 Gestion des erreurs : une exception . . . 203
4.7.3 Gestion des erreurs : flot dans une fonction . . . 204
4.7.4 Gestion des erreurs : flot dans une exception . . . 204
4.7.5 Gestion des erreurs : codes d'erreurs . . . 205
4.7.6 Messages d'erreurs : RAISE - 1 . . . 207
4.7.7 Messages d'erreurs : RAISE - 2 . . . 207
4.7.8 Messages d'erreurs : configuration des logs . . . 208
4.7.9 Messages d'erreurs : RAISE EXCEPTION - 1 . . . 208
4.7.10 Messages d'erreurs : RAISE EXCEPTION - 2 . . . 208
4.7.11 Flux des erreurs dans du code PL . . . 209
4.7.12 Flux des erreurs dans du code PL - 2 . . . 210
4.7.13 Flux des erreurs dans du code PL - 3 . . . 210
4.7.14 Flux des erreurs dans du code PL - 4 . . . 211
4.8 Sécurité . . . 213
4.8.1 Sécurité : droits . . . 213
4.8.2 Sécurité : ajout . . . 213
4.8.3 Sécurité : suppression . . . 213
4.8.4 Sécurité : SECURITY INVOKER/DEFINER . . . 214
4.8.5 Sécurité : LEAKPROOF . . . 214
4.8.6 Sécurité : visibilité des sources - 1 . . . 215
4.8.7 Sécurité : visibilité des sources - 2 . . . 216
4.8.8 Sécurité : Injections SQL . . . 216
4.9 Optimisation . . . 218
4.9.1 Fonctions immutables, stables ou volatiles - 1 . . . 218
4.9.2 Fonctions immutables, stables ou volatiles - 2 . . . 218
4.9.3 Fonctions immutables, stables ou volatiles - 3 . . . 220
4.9.4 Optimisation : rigueur . . . 220
4.9.5 Optimisation : EXCEPTION . . . 221
4.9.6 Requête statique ou dynamique ? . . . 222
4.9.7 Requête statique ou dynamique ? - 2 . . . 223
4.9.8 Requête statique ou dynamique ? -3 . . . 223
4.10 Outils . . . 224
4.10.1 pldebugger . . . 224
4.10.2 pldebugger - Compilation . . . 225
4.10.3 pldebugger - Activation . . . 226
4.10.4 auto_explain . . . 226
4.10.5 pldebugger - Utilisation . . . 230

4.10.6 log_functions . . . 232
4.10.7 log_functions - Compilation . . . 232
4.10.8 log_functions - Activation . . . 233
4.10.9 log_functions - Configuration . . . 234
4.10.10 log_functions - Utilisation . . . 234
4.11 Conclusion . . . 236
4.11.1 Pour aller plus loin . . . 236
4.11.2 Questions . . . 236
4.12 Travaux pratiques . . . 237
4.13 Travaux pratiques (solutions) . . . 238

5/ Extensions PostgreSQL pour l'utilisateur 247
5.1 Qu'est-ce qu'une extension ? . . . 248
5.2 Administration des extensions . . . 249
5.2.1 Installation des extensions . . . 249
5.3 Contribs - Fonctionnalités . . . 251
5.4 Quelques extensions . . . 252
5.4.1 pgcrypto . . . 252
5.4.2 hstore : stockage clé/valeur . . . 253
5.4.3 PostgreSQL Anonymizer . . . 253
5.4.4 PostGIS . . . 255
5.4.5 Mais encore… . . . 256
5.4.6 Autres extensions connues . . . 257
5.5 Extensions pour de nouveaux langages . . . 258
5.6 Accès distants . . . 259
5.7 Contribs orientés DBA . . . 260
5.8 PGXN . . . 261
5.9 Créer son extension . . . 265
5.10 Conclusion . . . 266
5.10.1 Questions . . . 266
5.11 Travaux pratiques . . . 267
5.11.1 Masquage statique de données avec PostgreSQL Anonymizer . . . 267
5.11.2 Masquage dynamique de données avec PostgreSQL Anonymizer . . . 267
5.11.3 Masquage statique de données avec PostgreSQL Anonymizer . . . 268
5.11.4 Masquage dynamique de données avec PostgreSQL Anonymizer . . . 269

6/ Partitionnement sous PostgreSQL 273
6.1 Principe & intérêts du partitionnement . . . 274
6.2 Partitionnement applicatif . . . 276
6.3 Méthodes de partitionnement intégrées à PostgreSQL . . . 277
6.4 Partitionnement par héritage . . . 278
6.5 Partitionnement déclaratif . . . 282
6.5.1 Partitionnement par liste . . . 283
6.5.2 Partitionnement par intervalle . . . 284
6.5.3 Partitionnement par hachage . . . 286
6.5.4 Clé de partitionnement multi-colonnes . . . 287

6.5.5 Performances en insertion . . . 289
6.5.6 Partition par défaut . . . 290
6.5.7 Attacher une partition . . . 292
6.5.8 Détacher une partition . . . 293
6.5.9 Supprimer une partition . . . 293
6.5.10 Fonctions de gestion et vues système . . . 294
6.5.11 Indexation . . . 295
6.5.12 Opérations de maintenance . . . 296
6.5.13 Intérêts du partitionnement déclaratif . . . 297
6.5.14 Limitations du partitionnement déclaratif et versions . . . 298
6.6 Extensions & outils . . . 300
6.7 Conclusion . . . 301
6.8 Quiz . . . 302
6.9 Travaux pratiques . . . 303
6.9.1 Partitionnement . . . 303
6.9.2 Partitionner pendant l'activité . . . 304
6.10 Travaux pratiques (solutions) . . . 307
6.10.1 Partitionnement . . . 307
6.10.2 Partitionner pendant l'activité . . . 310

7/ Connexions distantes 319
7.1 Accès à distance à d'autres sources de données . . . 320
7.2 SQL/MED . . . 321
7.2.1 Objets proposés par SQL/MED . . . 322
7.2.2 Foreign Data Wrapper . . . 323
7.2.3 Fonctionnalités disponibles pour un FDW (1/2) . . . 324
7.2.4 Fonctionnalités disponibles pour un FDW (2/2) . . . 325
7.2.5 Foreign Server . . . 325
7.2.6 User Mapping . . . 326
7.2.7 Foreign Table . . . 326
7.2.8 Exemple : file_fdw . . . 327
7.2.9 Exemple : postgres_fdw . . . 328
7.2.10 SQL/MED : Performances . . . 332
7.2.11 SQL/MED : héritage . . . 332
7.3 dblink . . . 339
7.4 PL/Proxy . . . 341
7.5 Conclusion . . . 342
7.6 Travaux pratiques . . . 343
7.6.1 Foreign Data Wrapper sur un fichier . . . 343
7.6.2 Foreign Data Wrapper sur une autre base . . . 343
7.7 Travaux pratiques (solutions) . . . 344
7.7.1 Foreign Data Wrapper sur un fichier . . . 344
7.7.2 Foreign Data Wrapper sur une autre base . . . 344

8/ Fonctionnalités avancées pour la performance **347**
8.1 Préambule . . . 348
8.1.1 Au menu . . . 348
8.2 Tables temporaires . . . 349
8.3 Tables non journalisées (unlogged) . . . 352
8.3.1 Tables non journalisées : mise en place . . . 353
8.3.2 Bascule d'une table en/depuis unlogged . . . 353
8.4 JIT : la compilation à la volée . . . 355
8.4.1 JIT : qu'est-ce qui est compilé ? . . . 356
8.4.2 JIT : algorithme « naïf » . . . 357
8.4.3 Quand le JIT est-il utile ? . . . 358
8.5 Recherche Plein Texte . . . 359
8.5.1 Full Text Search : exemple . . . 360
8.5.2 Full Text Search : dictionnaires . . . 362
8.5.3 Full Text Search : stockage & indexation . . . 365
8.5.4 Full Text Search sur du JSON . . . 367
8.6 Quiz . . . 369
8.7 Travaux pratiques . . . 370
8.7.1 Tables non journalisées . . . 370
8.7.2 Indexation Full Text . . . 371
8.8 Travaux pratiques (solutions) . . . 372
8.8.1 Tables non journalisées . . . 372
8.8.2 Indexation Full Text . . . 375

9/ Masquage de données & postgresql_anonymizer **377**
9.1 Cas d'usage . . . 378
9.1.1 Objectifs . . . 378

10/ PostgreSQL Anonymizer **379**
10.0.1 Principe . . . 379
10.0.2 Masquages . . . 379
10.0.3 Pré-requis . . . 380
10.0.4 Base d'exemple . . . 381

11/ Masquage statique avec postgresql_anonymizer **383**
11.1 L'histoire . . . 384
11.2 Comment ça marche . . . 385
11.3 Objectifs . . . 386
11.4 Table « customer » . . . 387
11.4.1 Quelques clients . . . 387
11.5 Table « payout » . . . 388
11.5.1 Quelques données . . . 388
11.5.2 Activer l'extension . . . 388
11.6 Déclarer les règles de masquage . . . 389
11.7 Appliquer les règles de manière permanente . . . 390

11.8 Exercices . . . 391
11.8.1 E101 - Masquer les prénoms des clients . . . 391
11.8.2 E102 - Masquer les 3 derniers chiffres du code postal . . . 391
11.8.3 E103 - Compter le nombre de clients dans chaque département. . . . 391
11.8.4 E104 - Ne garder que l'année dans les dates de naissance . . . 391
11.8.5 E105 - Identifier un client particulier . . . 391
11.9 Solutions . . . 393
11.9.1 S101 . . . 393
11.9.2 S102 . . . 393
11.9.3 S103 . . . 393
11.9.4 S104 . . . 394
11.9.5 S105 . . . 394

12/ Masquage dynamique avec postgresql_anonymizer 395
12.1 Principe du masquage dynamique . . . 396
12.2 L'histoire . . . 397
12.3 Comment ça marche . . . 398
12.4 Objectifs de la section . . . 399
12.5 Table « company » . . . 400
12.5.1 Quelques données . . . 400
12.6 Table « supplier » . . . 401
12.6.1 Quelques données . . . 401
12.7 Activer l'extension . . . 402
12.8 Activer le masquage dynamique . . . 403
12.9 Rôle masqué . . . 404
12.10 Masquer le nom des fournisseurs . . . 405
12.11 Exercices . . . 406
12.11.1 E201 - Deviner qui est le PDG de « Johnny's Shoe Store » . . . 406
12.11.2 E202 - Anonymiser les sociétés . . . 406
12.11.3 E203 - Pseudonymiser le nom des sociétés . . . 406
12.12 Solutions . . . 408
12.12.1 S201 . . . 408
12.12.2 S202 . . . 408
12.12.3 S203 . . . 408

13/ Sauvegardes anonymes avec postgresql_anonymizer 411
13.1 L'histoire . . . 412
13.2 Comment ça marche ? . . . 413
13.3 Objectifs . . . 414
13.4 Table « website_comment » . . . 415
13.4.1 Quelques données . . . 415
13.5 Activer l'extension . . . 416
13.6 Masquer une colonne de type JSON . . . 417
13.6.1 Fonctions de masquage personnalisées . . . 417
13.6.2 Utilisation de la fonction de masquage personnalisée . . . 418
13.6.3 Sauvegarde anonymisée . . . 418

13.7 Exercices . . . 420
13.7.1 E301 - Exporter les données anonymisées dans une nouvelle base de données 420
13.7.2 E302 - Pseudonymiser les métadonnées du commentaire . . . 420
13.8 Solutions . . . 421
13.8.1 S301 . . . 421
13.8.2 S302 . . . 421

14/ Généralisation avec postgresql_anonymizer 423
14.1 Principe . . . 424
14.2 L'histoire . . . 425
14.3 Comment ça marche ? . . . 426
14.4 Objectifs . . . 427
14.5 Table « employee » . . . 428
14.6 Quelques données . . . 429
14.7 Suppression de données . . . 430
14.8 Calculer le k-anonymat . . . 431
14.9 Fonctions d'intervalle et de généralisation . . . 432
14.9.1 Déclarer les identifiants indirects . . . 433
14.10 Exercices . . . 434
14.10.1 E401 - Simplifier la vue v_staff_per_month pour en réduire la granularité. . . . 434
14.10.2 E402 - Progression du personnel au fil des années . . . 434
14.10.3 E403 - Atteindre le facteur *2-anonymity* sur la vue v_staff_per_year . . . 434
14.11 Solutions . . . 435
14.11.1 S401 . . . 435
14.11.2 S402 . . . 435
14.11.3 S403 . . . 435

15/ Conclusion sur postgresql_anonymizer 437
15.1 Beaucoup de stratégies de masquage . . . 438
15.2 Beaucoup de fonctions de masquage . . . 439
15.3 Avantages . . . 440
15.4 Inconvénients . . . 441
15.5 Pour aller plus loin . . . 442
15.6 Contribuez ! . . . 443
15.6.1 Questions . . . 443

16/ Pooling 445
16.1 Au menu . . . 446
16.1.1 Objectifs . . . 446
16.2 Pool de connexion . . . 447
16.2.1 Serveur de pool de connexions . . . 447
16.2.2 Serveur de pool de connexions . . . 448
16.2.3 Intérêts du pool de connexions . . . 449
16.2.4 Inconvénients du pool de connexions . . . 450
16.3 Pooling de sessions . . . 451
16.3.1 Intérêts du pooling de sessions . . . 451

16.4 Pooling de transactions . . . 453
16.4.1 Avantages & inconvénients du pooling de transactions . . . 454
16.5 Pooling de requêtes . . . 456
16.5.1 Avantages & inconvénients du pooling de requêtes . . . 456
16.6 Pooling avec PgBouncer . . . 458
16.6.1 PgBouncer : Fonctionnalités . . . 459
16.6.2 PgBouncer : Installation . . . 459
16.6.3 PgBouncer : Fichier de configuration . . . 460
16.6.4 PgBouncer : Connexions . . . 461
16.6.5 PgBouncer : Définition des accès aux bases . . . 462
16.6.6 PgBouncer : Authentification par fichier de mots de passe . . . 463
16.6.7 PgBouncer : Authentification par délégation . . . 464
16.6.8 PgBouncer : Nombre de connexions . . . 465
16.6.9 PgBouncer : types de connexions . . . 467
16.6.10 PgBouncer : Durée de vie . . . 468
16.6.11 PgBouncer : Traces . . . 469
16.6.12 PgBouncer : Administration . . . 470
16.7 Conclusion . . . 473
16.7.1 Questions . . . 473
16.8 Travaux pratiques . . . 474
16.8.1 Pooling par session . . . 474
16.8.2 Pooling par transaction . . . 474
16.8.3 Pooling par requête . . . 474
16.8.4 pgbench . . . 474
16.9 Travaux pratiques (solutions) . . . 476
16.9.1 Pooling par session . . . 478
16.9.2 Pooling par transaction . . . 479
16.9.3 Pooling par requête . . . 482
16.9.4 Pgbench . . . 482

Les formations Dalibo **487**
Cursus des formations . . . 487
Les livres blancs . . . 488
Téléchargement gratuit . . . 488

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

[1] mailto:formation@dalibo.com

Licence Creative Commons CC-BY-NC-SA

Marques déposées

Sur ce document

Formation	Formation DEV42
Titre	Développement avancé
Révision	23.09
ISBN	N/A
PDF	https://dali.bo/dev42_pdf

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode
[3] https://www.postgresql.org/about/policies/trademarks/

EPUB	https://dali.bo/dev42_epub
HTML	https://dali.bo/dev42_html
Slides	https://dali.bo/dev42_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

1/ SQL pour l’analyse de données

1.1 PRÉAMBULE

- Analyser des données est facile avec PostgreSQL
  - opérations d'agrégation disponibles
  - fonctions OLAP avancées

1.1.1 Menu

- Agrégation de données
- Clause FILTER
- Fonctions window
- GROUPING SETS, ROLLUP, CUBE
- WITHIN GROUPS

1.1.2 Objectifs

- Écrire des requêtes encore plus complexes
- Analyser les données en amont
  - pour ne récupérer que le résultat

1.2 AGRÉGATS

- SQL dispose de fonctions de calcul d'agrégats
- Utilité :
  - calcul de sommes, moyennes, valeur minimale et maximale
  - nombreuses fonctions statistiques disponibles

À l'aide des fonctions de calcul d'agrégats, on peut réaliser un certain nombre de calculs permettant d'analyser les données d'une table.

La plupart des exemples utilisent une table `employes` définie telle que :

```
CREATE TABLE employes (
  matricule char(8) primary key,
  nom       text    not null,
  service   text,
  salaire   numeric(7,2)
);

INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000001', 'Dupuis', 'Direction', 10000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000004', 'Fantasio', 'Courrier', 4500.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000006', 'Prunelle', 'Publication', 4000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000020', 'Lagaffe', 'Courrier', 3000.00);
INSERT INTO employes (matricule, nom, service, salaire)
    VALUES ('00000040', 'Lebrac', 'Publication', 3000.00);

SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

Ainsi, on peut déduire le salaire moyen avec la fonction `avg()`, les salaires maximum et minimum versés par la société avec les fonctions `max()` et `min()`, ainsi que la somme totale des salaires versés avec la fonction `sum()` :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
```

```
       sum(salaire) AS somme_salaires
  FROM employes;
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 4900.0000000000000000 |        10000.00 |         3000.00 |       24500.00
```

La base de données réalise les calculs sur l'ensemble des données de la table et n'affiche que le résultat du calcul.

Si l'on applique un filtre sur les données, par exemple pour ne prendre en compte que le service *Courrier*, alors PostgreSQL réalise le calcul uniquement sur les données issues de la lecture :

```
SELECT avg(salaire) AS salaire_moyen,
       max(salaire) AS salaire_maximum,
       min(salaire) AS salaire_minimum,
       sum(salaire) AS somme_salaires
  FROM employes
 WHERE service = 'Courrier';
     salaire_moyen     | salaire_maximum | salaire_minimum | somme_salaires
-----------------------+-----------------+-----------------+----------------
 3750.0000000000000000 |         4500.00 |         3000.00 |        7500.00
(1 ligne)
```

En revanche, il n'est pas possible de référencer d'autres colonnes pour les afficher à côté du résultat d'un calcul d'agrégation à moins de les utiliser comme critère de regroupement :

```
SELECT avg(salaire), nom FROM employes;
ERROR:  column "employes.nom" must appear in the GROUP BY clause or be used in
        an aggregate function
LIGNE 1 : SELECT avg(salaire), nom FROM employes;
                               ^
```

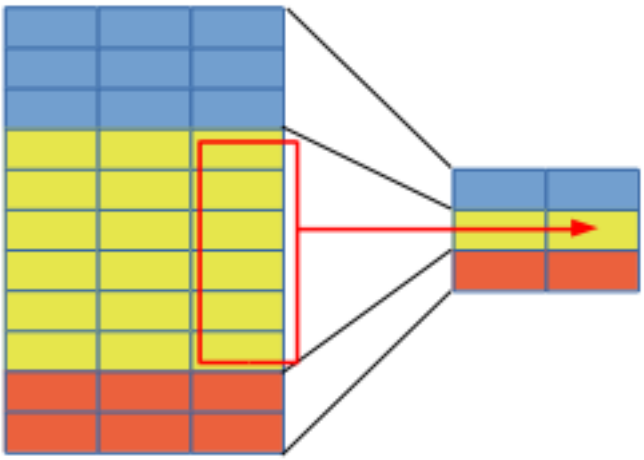
1.2.1 Agrégats avec GROUP BY

- agrégat + GROUP BY
- Utilité
  - effectue des calculs sur des regroupements : moyenne, somme, comptage, etc.
  - regroupement selon un critère défini par la clause GROUP BY
  - exemple : calcul du salaire moyen de chaque service

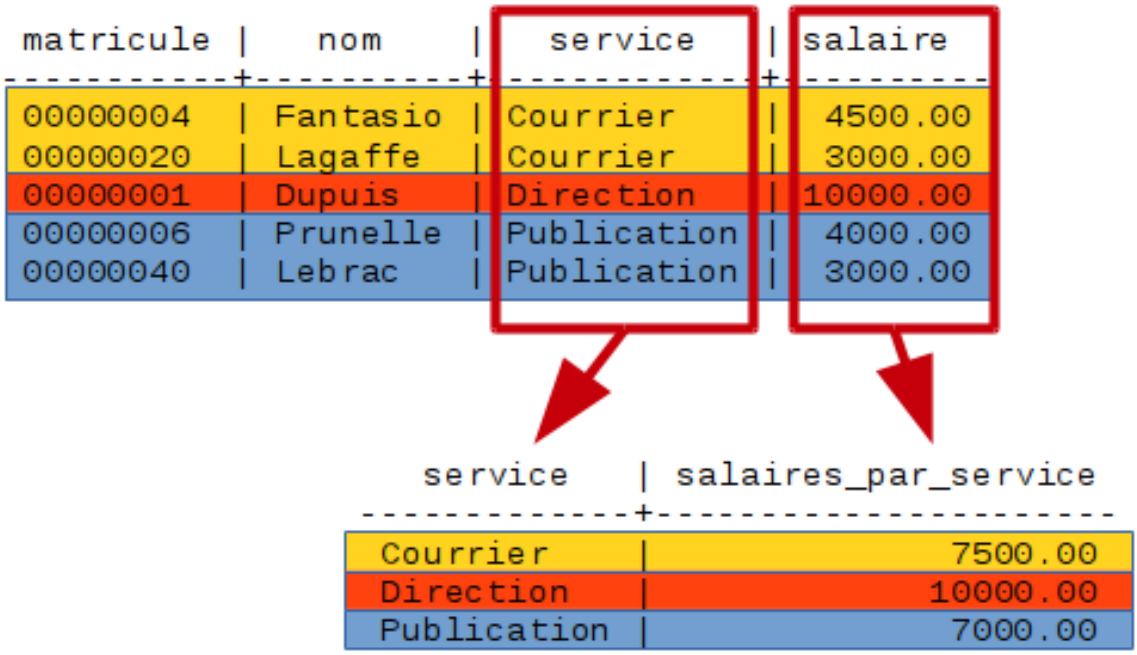
L'opérateur d'agrégat GROUP BY indique à la base de données que l'on souhaite regrouper les données selon les mêmes valeurs d'une colonne.

matricule	nom	service	salaire
00000004	Fantasio	Courrier	4500.00
00000020	Lagaffe	Courrier	3000.00
00000001	Dupuis	Direction	10000.00
00000006	Prunelle	Publication	4000.00
00000040	Lebrac	Publication	3000.00

Des calculs pourront être réalisés sur les données agrégées selon le critère de regroupement donné. Le résultat sera alors représenté en n'affichant que les colonnes de regroupement puis les valeurs calculées par les fonctions d'agrégation :



1.2.2 GROUP BY : principe



L'agrégation est ici réalisée sur la colonne `service`. En guise de calcul d'agrégation, une somme est réalisée sur les salaires payés dans chaque service.

1.2.3 GROUP BY : exemples

```
SELECT service,
       sum(salaire) AS salaires_par_service
  FROM employes
 GROUP BY service;

   service    | salaires_par_service
--------------+----------------------
 Courrier     |              7500.00
 Direction    |             10000.00
 Publication  |              7000.00
(3 lignes)
```

SQL permet depuis le début de réaliser des calculs d'agrégation. Pour cela, la base de données observe les critères de regroupement définis dans la clause GROUP BY de la requête et effectue l'opération sur l'ensemble des lignes qui correspondent au critère de regroupement.

On peut bien entendu combiner plusieurs opérations d'agrégations :

```
SELECT service,
       sum(salaire) salaires_par_service,
       avg(salaire) AS salaire_moyen_service
  FROM employes
 GROUP BY service;
```

```
   service    | salaires_par_service | salaire_moyen_service
--------------+----------------------+-----------------------
 Courrier     |              7500.00 |  3750.0000000000000000
 Direction    |             10000.00 | 10000.0000000000000000
 Publication  |              7000.00 |  3500.0000000000000000
(3 lignes)
```

On peut combiner le résultat de deux requêtes d'agrégation avec `UNION ALL`, si les ensembles retournées sont de même type :

```
SELECT service,
       sum(salaire) AS salaires_par_service
  FROM employes GROUP BY service
UNION ALL
SELECT 'Total' AS service,
       sum(salaire) AS salaires_par_service
  FROM employes;
```

```
   service    | salaires_par_service
--------------+----------------------
 Courrier     |              7500.00
 Direction    |             10000.00
 Publication  |              7000.00
 Total        |             24500.00
(4 lignes)
```

On le verra plus loin, cette dernière requête peut être écrite plus simplement avec les `GROUPING SETS`, mais qui nécessitent au minimum PostgreSQL 9.5.

1.2.4 Agrégats et ORDER BY

- Extension propriétaire de PostgreSQL
  - ORDER BY dans la fonction d'agrégat
- Utilité :
  - ordonner les données agrégées
  - surtout utile avec `array_agg`, `string_agg` et `xmlagg`

Les fonctions `array_agg`, `string_agg` et `xmlagg` permettent d'agréger des éléments dans un tableau, dans une chaîne ou dans une arborescence XML. Autant l'ordre dans lequel les données sont utilisées n'a pas d'importance lorsque l'on réalise un calcul d'agrégat classique, autant cet ordre va influencer la façon dont les données seront produites par les trois fonctions citées plus haut. En effet, le tableau généré par `array_agg` est composé d'éléments ordonnés, de même que la chaîne de caractères ou l'arborescence XML.

1.2.5 Utiliser ORDER BY avec un agrégat

```
SELECT service,
       string_agg(nom, ', ' ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |  liste_employes
--------------+-------------------
 Courrier     | Fantasio, Lagaffe
 Direction    | Dupuis
 Publication | Lebrac, Prunelle
(3 lignes)
```

La requête suivante permet d'obtenir, pour chaque service, la liste des employés dans un tableau, trié par ordre alphabétique :

```
SELECT service,
       string_agg(nom, ', ' ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |  liste_employes
--------------+-------------------
 Courrier     | Fantasio, Lagaffe
 Direction    | Dupuis
 Publication | Lebrac, Prunelle
(3 lignes)
```

Il est possible de réaliser la même chose mais pour obtenir un tableau plutôt qu'une chaîne de caractère :

```
SELECT service,
       array_agg(nom ORDER BY nom) AS liste_employes
  FROM employes
 GROUP BY service;
   service    |    liste_employes
--------------+---------------------
 Courrier     | {Fantasio,Lagaffe}
 Direction    | {Dupuis}
 Publication | {Lebrac,Prunelle}
```

1.3 CLAUSE FILTER

- Clause `FILTER`
- Utilité :
  - filtrer les données sur les agrégats
  - évite les expressions CASE complexes
- SQL:2003
- Intégré dans la version 9.4

La clause `FILTER` permet de remplacer des expressions complexes écrites avec CASE et donc de simplifier l'écriture de requêtes réalisant un filtrage dans une fonction d'agrégat.

1.3.1 Filtrer avec CASE

- La syntaxe suivante était utilisée :

```
SELECT count(*) AS compte_pays,
       count(CASE WHEN r.nom_region='Europe' THEN 1
                  ELSE NULL
             END) AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

Avec cette syntaxe, dès que l'on a besoin d'avoir de multiples filtres ou de filtres plus complexes, la requête devient très rapidement peu lisible et difficile à maintenir. Le risque d'erreur est également élevé.

1.3.2 Filtrer avec FILTER

- La même requête écrite avec la clause `FILTER` :

```
SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe')
                AS compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
```

L'exemple suivant montre l'utilisation de la clause `FILTER` et son équivalent écrit avec une expression CASE :

```
sql=# SELECT count(*) AS compte_pays,
       count(*) FILTER (WHERE r.nom_region='Europe') AS compte_pays_europeens,
       count(CASE WHEN r.nom_region='Europe' THEN 1 END)
           AS oldschool_compte_pays_europeens
  FROM pays p
  JOIN regions r
    ON (p.region_id = r.region_id);
 compte_pays | compte_pays_europeens | oldschool_compte_pays_europeens
-------------+-----------------------+---------------------------------
          25 |                     5 |                               5
(1 ligne)
```

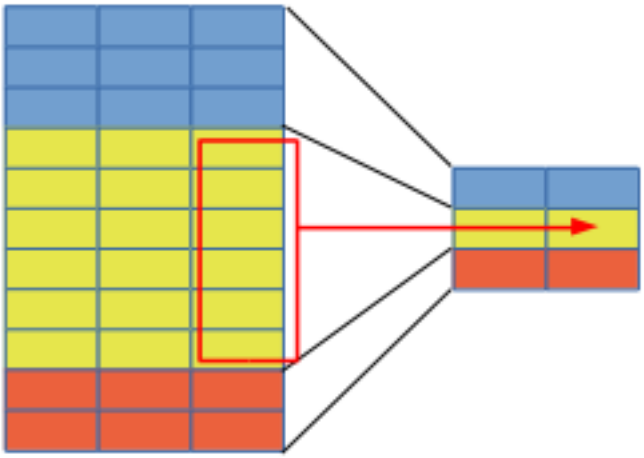
1.4 FONCTIONS DE FENÊTRAGE

- Fonctions *window*
  - travaille sur des ensembles de données regroupés et triés indépendamment de la requête principale
- Utilisation :
  - utiliser plusieurs critères d'agrégation dans la même requête
  - utiliser des fonctions de classement
  - faire référence à d'autres lignes de l'ensemble de données

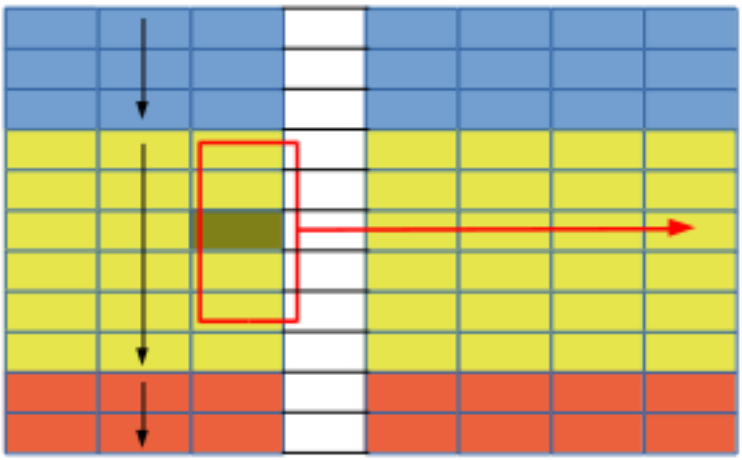
PostgreSQL supporte les fonctions de fenêtrage depuis la version 8.4. Elles apportent des fonctionnalités analytiques à PostgreSQL, et permettent d'écrire beaucoup plus simplement certaines requêtes.

Prenons un exemple.

```
SELECT service, AVG(salaire)
FROM employe
GROUP BY service
```



```
SELECT service, id_employe, salaire,
  AVG(salaire) OVER (
    PARTITION BY service
    ORDER BY age
    ROWS BETWEEN 2 PRECEEDING AND 2 FOLLOWING
  )
FROM employes
```



1.4.1 Regroupement

- Regroupement
  - clause `OVER (PARTITION BY ...)`
- Utilité :
  - plusieurs critères de regroupement différents
  - avec des fonctions de calcul d'agrégats

La clause `OVER` permet de définir la façon dont les données sont regroupées - uniquement pour la colonne définie - avec la clause `PARTITION BY`.

Les exemples vont utiliser cette table `employes` :

```
exemple=# SELECT * FROM employes ;
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   | Direction   | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

1.4.2 Regroupement : exemple

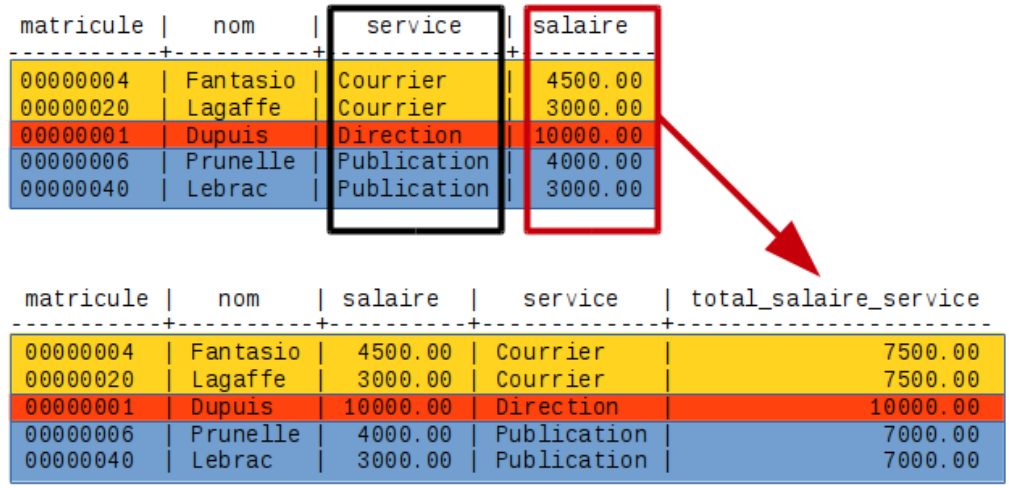
```
SELECT matricule, salaire, service,
       SUM(salaire) OVER (PARTITION BY service)
                 AS total_salaire_service
  FROM employes;
```

```
 matricule | salaire  |   service   | total_salaire_service
-----------+----------+-------------+-----------------------
 00000004  |  4500.00 | Courrier    |               7500.00
 00000020  |  3000.00 | Courrier    |               7500.00
 00000001  | 10000.00 | Direction   |              10000.00
 00000006  |  4000.00 | Publication |               7000.00
 00000040  |  3000.00 | Publication |               7000.00
```

Les calculs réalisés par cette requête sont identiques à ceux réalisés avec une agrégation utilisant GROUP BY. La principale différence est que l'on évite de ici de perdre le détail des données tout en disposant des données agrégées dans le résultat de la requête.

1.4.3 Regroupement : principe

```
SUM(salaire) OVER (PARTITION BY service)
```



Entouré de noir, le critère de regroupement et entouré de rouge, les données sur lesquelles sont appliqués le calcul d'agrégat.

1.4.4 Regroupement : syntaxe

```
SELECT ...
 agregation OVER (PARTITION BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Le terme `PARTITION BY` permet d'indiquer les critères de regroupement de la fenêtre sur laquelle on souhaite travailler.

1.4.5 Tri

- Tri
  - `OVER (ORDER BY …)`
- Utilité :
  - numéroter les lignes : `row_number()`
  - classer des résultats : `rank()`, `dense_rank()`
  - faire appel à d'autres lignes du résultat : `lead()`, `lag()`

1.4.6 Tri : exemple

- Pour numéroter des lignes :

```
SELECT row_number() OVER (ORDER BY matricule),
       matricule, nom
  FROM employes;

 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          3 | 00000006  | Prunelle
          4 | 00000020  | Lagaffe
          5 | 00000040  | Lebrac
(5 lignes)
```

La fonction `row_number()` permet de numéroter les lignes selon un critère de tri défini dans la clause OVER.

L'ordre de tri de la clause OVER n'influence pas l'ordre de tri explicite d'une requête :

```
SELECT row_number() OVER (ORDER BY matricule),
       matricule, nom
  FROM employes
 ORDER BY nom;
 row_number | matricule |   nom
------------+-----------+----------
          1 | 00000001  | Dupuis
          2 | 00000004  | Fantasio
          4 | 00000020  | Lagaffe
          5 | 00000040  | Lebrac
          3 | 00000006  | Prunelle
(5 lignes)
```

On dispose aussi de fonctions de classement, pour déterminer par exemple les employés les moins bien payés :

```
SELECT matricule, nom, salaire, service,
       rank() OVER (ORDER BY salaire),
       dense_rank() OVER (ORDER BY salaire)
  FROM employes ;
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

La fonction de fenêtrage `rank()` renvoie le classement en autorisant des trous dans la numérotation, et `dense_rank()` le classement sans trous.

1.4.7 Tri : exemple avec une somme

- Calcul d'une somme glissante :

```
SELECT matricule, salaire,
       SUM(salaire) OVER (ORDER BY matricule)
  FROM employes;

 matricule | salaire  |   sum
-----------+----------+----------
 00000001  | 10000.00 | 10000.00
 00000004  |  4500.00 | 14500.00
 00000006  |  4000.00 | 18500.00
 00000020  |  3000.00 | 21500.00
 00000040  |  3000.00 | 24500.00
```

1.4.8 Tri : principe

```
SUM(salaire) OVER (ORDER BY matricule)
```

```
  matricule | salaire
-----------+----------
 00000001   | 10000.00
 00000004   |  4500.00
 00000006   |  4000.00
 00000020   |  3000.00
 00000040   |  3000.00
```

Fenêtre de calcul pour la ligne courante

SUM(salaire)

```
  matricule | salaire  |   sum
-----------+----------+----------
 00000001   | 10000.00 | 10000.00
 00000004   |  4500.00 | 14500.00
 00000006   |  4000.00 | 18500.00
 00000020   |  3000.00 | 21500.00
 00000040   |  3000.00 | 24500.00
```

Lorsque l'on utilise une clause de tri, la portion de données visible par l'opérateur d'agrégat correspond aux données comprises entre la première ligne examinée et la ligne courante. La fenêtre est définie selon le critère `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Nous verrons plus loin que nous pouvons modifier ce comportement.

1.4.9 Tri : syntaxe

```
SELECT ...
 agregation OVER (ORDER BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Le terme ORDER BY permet d'indiquer les critères de tri de la fenêtre sur laquelle on souhaite travailler.

1.4.10 Regroupement et tri

- On peut combiner les deux
  - `OVER (PARTITION BY .. ORDER BY ..)`
- Utilité :
  - travailler sur des jeux de données ordonnés et isolés les uns des autres

Il est possible de combiner les clauses de fenêtrage `PARTITION BY` et `ORDER BY`. Cela permet d'isoler des jeux de données entre eux avec la clause `PARTITION BY`, tout en appliquant un critère de tri avec la clause `ORDER BY`. Beaucoup d'applications sont possibles si l'on associe à cela les nombreuses fonctions analytiques disponibles.

1.4.11 Regroupement et tri : exemple

```
SELECT continent, pays, population,
       rank() OVER (PARTITION BY continent
                    ORDER BY population DESC)
              AS rang
  FROM population;

    continent     |       pays          | population | rang
------------------+---------------------+------------+------
 Afrique          | Nigéria             |      173.6 |    1
 Afrique          | Éthiopie            |       94.1 |    2
 Afrique          | Égypte              |       82.1 |    3
 Afrique          | Rép. dém. du Congo  |       67.5 |    4
(...)
 Amérique du Nord | États-Unis          |      320.1 |    1
 Amérique du Nord | Canada              |       35.2 |    2
(...)
```

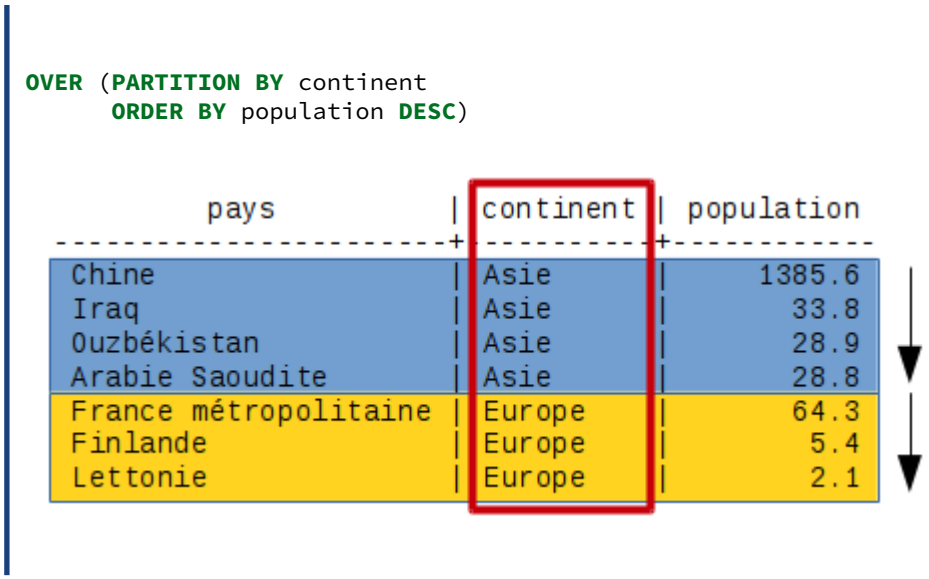
Si l'on applique les deux clauses `PARTITION BY` et `ORDER BY` à une fonction de fenêtrage, alors le critère de tri est appliqué dans la partition et chaque partition est indépendante l'une de l'autre.

Voici un extrait plus complet du résultat de la requête présentée ci-dessus :

```
         continent          |         pays        | population | rang_pop
----------------------------+---------------------+------------+----------
```

```
 Afrique                  | Nigéria            |      173.6 |        1
 Afrique                  | Éthiopie           |       94.1 |        2
 Afrique                  | Égypte             |       82.1 |        3
 Afrique                  | Rép. dém. du Congo |       67.5 |        4
 Afrique                  | Afrique du Sud     |       52.8 |        5
 Afrique                  | Tanzanie           |       49.3 |        6
 Afrique                  | Kenya              |       44.4 |        7
 Afrique                  | Algérie            |       39.2 |        8
 Afrique                  | Ouganda            |       37.6 |        9
 Afrique                  | Maroc              |       33.0 |       10
 Afrique                  | Ghana              |       25.9 |       11
 Afrique                  | Mozambique         |       25.8 |       12
 Afrique                  | Madagascar         |       22.9 |       13
 Afrique                  | Côte-d'Ivoire      |       20.3 |       14
 Afrique                  | Niger              |       17.8 |       15
 Afrique                  | Burkina Faso       |       16.9 |       16
 Afrique                  | Zimbabwe           |       14.1 |       17
 Afrique                  | Soudan             |       14.1 |       17
 Afrique                  | Tunisie            |       11.0 |       19
 Amérique du Nord         | États-Unis         |      320.1 |        1
 Amérique du Nord         | Canada             |       35.2 |        2
 Amérique latine. Caraïbes | Brésil            |      200.4 |        1
 Amérique latine. Caraïbes | Mexique           |      122.3 |        2
 Amérique latine. Caraïbes | Colombie          |       48.3 |        3
 Amérique latine. Caraïbes | Argentine         |       41.4 |        4
 Amérique latine. Caraïbes | Pérou             |       30.4 |        5
 Amérique latine. Caraïbes | Venezuela         |       30.4 |        5
 Amérique latine. Caraïbes | Chili             |       17.6 |        7
 Amérique latine. Caraïbes | Équateur          |       15.7 |        8
 Amérique latine. Caraïbes | Guatemala         |       15.5 |        9
 Amérique latine. Caraïbes | Cuba              |       11.3 |       10
(...)
```

1.4.12 Regroupement et tri : principe



```
OVER (PARTITION BY continent
      ORDER BY population DESC)
```

pays	continent	population
Chine	Asie	1385.6
Iraq	Asie	33.8
Ouzbékistan	Asie	28.9
Arabie Saoudite	Asie	28.8
France métropolitaine	Europe	64.3
Finlande	Europe	5.4
Lettonie	Europe	2.1

1.4.13 Regroupement et tri : syntaxe

```
SELECT ...
 <agregation> OVER (PARTITION BY <colonnes>
                    ORDER BY <colonnes>)
  FROM <liste_tables>
 WHERE <predicats>
```

Cette construction ne pose aucune difficulté syntaxique. La norme impose de placer la clause PARTITION BY avant la clause ORDER BY, c'est la seule chose à retenir au niveau de la syntaxe.

1.4.14 Fonctions analytiques

- PostgreSQL dispose d'un certain nombre de fonctions analytiques
- Utilité :
  - faire référence à d'autres lignes du même ensemble
  - évite les auto-jointures complexes et lentes

Sans les fonctions analytiques, il était difficile en SQL d'écrire des requêtes nécessitant de faire appel à des données provenant d'autres lignes que la ligne courante.

Par exemple, pour renvoyer la liste détaillée de tous les employés ET le salaire le plus élevé du service auquel il appartient, on peut utiliser la fonction `first_value()` :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
 matricule |   nom    | salaire  |   service   | salaire_maximum_service
-----------+----------+----------+-------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier    |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier    |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction   |                10000.00
 00000006  | Prunelle |  4000.00 | Publication |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication |                 4000.00
(5 lignes)
```

Il existe également les fonctions suivantes :

- `last_value(colonne)` : renvoie la dernière valeur pour la colonne ;
- `nth(colonne, n)` : renvoie la n-ème valeur (en comptant à partir de **1**) pour la colonne ;
- `lag(colonne, n)` : renvoie la valeur située en n-ème position **avant** la ligne en cours pour la colonne ;
- `lead(colonne, n)` : renvoie la valeur située en n-ème position **après** la ligne en cours pour la colonne ;

  - pour ces deux fonctions, le n est facultatif et vaut **1** par défaut ;
  - ces deux fonctions acceptent un 3ème argument facultatif spécifiant la valeur à renvoyer si aucune valeur n'est trouvée en n-ème position avant ou après. Par défaut, `NULL` sera renvoyé.

1.4.15 lead() et lag()

- `lead(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **après** la ligne courante
- `lag(colonne, n)`
  - retourne la valeur d'une colonne, n lignes **avant** la ligne courante

La construction `lead(colonne)` est équivalente à `lead(colonne, 1)`. De même, la construction `lag(colonne)` est équivalente à `lag(colonne, 1)`. Il s'agit d'un raccourci pour utiliser la valeur précédente ou la valeur suivante d'une colonne dans la fenêtre définie.

1.4.16 lead() et lag() : exemple

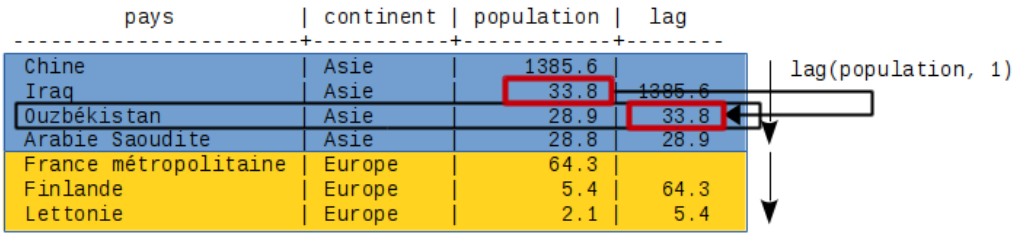
```
SELECT pays, continent, population,
       lag(population) OVER (PARTITION BY continent
                             ORDER BY population DESC)
  FROM population;
```

```
         pays          | continent | population |  lag
-----------------------+-----------+------------+--------
 Chine                 | Asie      |     1385.6 |
 Iraq                  | Asie      |       33.8 | 1385.6
 Ouzbékistan           | Asie      |       28.9 |   33.8
 Arabie Saoudite       | Asie      |       28.8 |   28.9
 France métropolitaine | Europe    |       64.3 |
 Finlande              | Europe    |        5.4 |   64.3
 Lettonie              | Europe    |        2.1 |    5.4
```

La requête présentée en exemple ne s'appuie que sur un jeu réduit de données afin de montrer un résultat compréhensible.

1.4.17 lead() et lag() : principe

```
lag(population) OVER (PARTITION BY continent
                      ORDER BY population DESC)
```



NULL est renvoyé lorsque la valeur n'est pas accessible dans la fenêtre de données, comme par exemple si l'on souhaite utiliser la valeur d'une colonne appartenant à la ligne précédant la première ligne de la partition.

1.4.18 first/last/nth_value

- `first_value(colonne)`
  - retourne la première valeur pour la colonne
- `last_value(colonne)`
  - retourne la dernière valeur pour la colonne
- `nth_value(colonne, n)`
  - retourne la n-ème valeur (en comptant à partir de 1) pour la colonne

Utilisé avec `ORDER BY` et `PARTITION BY`, la fonction `first_value()` permet par exemple d'obtenir le salaire le plus élevé d'un service :

```
SELECT matricule, nom, salaire, service,
       first_value(salaire) OVER (PARTITION BY service ORDER BY salaire DESC)
       AS salaire_maximum_service
  FROM employes ;
```

```
 matricule |   nom    | salaire  |   service    | salaire_maximum_service
-----------+----------+----------+--------------+-------------------------
 00000004  | Fantasio |  4500.00 | Courrier     |                 4500.00
 00000020  | Lagaffe  |  3000.00 | Courrier     |                 4500.00
 00000001  | Dupuis   | 10000.00 | Direction    |                10000.00
 00000006  | Prunelle |  4000.00 | Publication  |                 4000.00
 00000040  | Lebrac   |  3000.00 | Publication  |                 4000.00
(5 lignes)
```

1.4.19 first/last/nth_value : exemple

```
SELECT pays, continent, population,
       first_value(population)
           OVER (PARTITION BY continent
                 ORDER BY population DESC)
  FROM population;
```

```
       pays       | continent | population | first_value
------------------+-----------+------------+-------------
 Chine            | Asie      |     1385.6 |      1385.6
 Iraq             | Asie      |       33.8 |      1385.6
 Ouzbékistan      | Asie      |       28.9 |      1385.6
 Arabie Saoudite  | Asie      |       28.8 |      1385.6
 France           | Europe    |       64.3 |        64.3
 Finlande         | Europe    |        5.4 |        64.3
 Lettonie         | Europe    |        2.1 |        64.3
```

Lorsque que la clause ORDER BY est utilisée pour définir une fenêtre, la fenêtre visible depuis la ligne courante commence par défaut à la première ligne de résultat et s'arrête à la ligne courante.

Par exemple, si l'on exécute la même requête en utilisant `last_value()` plutôt que `first_value()`, on récupère à chaque fois la valeur de la colonne sur la ligne courante :

```
SELECT pays, continent, population,
       last_value(population) OVER (PARTITION BY continent
                         ORDER BY population DESC)
  FROM population;
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Chine                 | Asie      |     1385.6 |     1385.6
 Iraq                  | Asie      |       33.8 |       33.8
 Ouzbékistan           | Asie      |       28.9 |       28.9
```

```
 Arabie Saoudite       | Asie      |       28.8 |       28.8
 France métropolitaine | Europe    |       64.3 |       64.3
 Finlande              | Europe    |        5.4 |        5.4
 Lettonie              | Europe    |        2.1 |        2.1
(7 rows)
```

Il est alors nécessaire de redéfinir le comportement de la fenêtre visible pour que la fonction se comporte comme attendu, en utilisant `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` - cet aspect sera décrit dans la section sur les possibilités de modification de la définition de la fenêtre.

1.4.20 Clause WINDOW

- Pour factoriser la définition d'une fenêtre :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
 WINDOW w AS (ORDER BY salaire);
```

Il arrive que l'on ait besoin d'utiliser plusieurs fonctions de fenêtrage au sein d'une même requête qui utilisent la même définition de fenêtre (même clause `PARTITION BY` et/ou `ORDER BY`). Afin d'éviter de dupliquer cette clause, il est possible de définir une fenêtre nommée et de l'utiliser à plusieurs endroits de la requête. Par exemple, l'exemple précédent des fonctions de classement pourrait s'écrire :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w,
       dense_rank() OVER w
  FROM employes
  WINDOW w AS (ORDER BY salaire);
 matricule |   nom    | salaire  |   service   | rank | dense_rank
-----------+----------+----------+-------------+------+------------
 00000020  | Lagaffe  |  3000.00 | Courrier    |    1 |          1
 00000040  | Lebrac   |  3000.00 | Publication |    1 |          1
 00000006  | Prunelle |  4000.00 | Publication |    3 |          2
 00000004  | Fantasio |  4500.00 | Courrier    |    4 |          3
 00000001  | Dupuis   | 10000.00 | Direction   |    5 |          4
(5 lignes)
```

À noter qu'il est possible de définir de multiples définitions de fenêtres au sein d'une même requête, et qu'une définition de fenêtre peut surcharger la clause `ORDER BY` si la définition parente ne l'a pas définie. Par exemple, la requête SQL suivante est correcte :

```
SELECT matricule, nom, salaire, service,
       rank() OVER w_asc,
```

```
      dense_rank() OVER w_desc
FROM employes
WINDOW w AS (PARTITION BY service),
       w_asc AS (w ORDER BY salaire),
       w_desc AS (w ORDER BY salaire DESC);
```

1.4.21 Clause WINDOW : syntaxe

```
SELECT fonction_agregat OVER nom,
       fonction_agregat_2 OVER nom ...
       ...
  FROM <liste_tables>
 WHERE <predicats>
 WINDOW nom AS (PARTITION BY ... ORDER BY ...)
```

1.4.22 Définition de la fenêtre

- La fenêtre de travail par défaut est :

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

- Trois modes possibles :
  - RANGE
  - ROWS
  - GROUPS (v11+)
- Nécessite une clause ORDER BY

1.4.23 Définition de la fenêtre : RANGE

- Indique un intervalle à bornes *flou*
- Borne de départ :
  - `UNBOUNDED PRECEDING`: depuis le début de la partition
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `UNBOUNDED FOLLOWING` : jusqu'à la fin de la partition
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
      ORDER BY ...
      RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

1.4.24 Définition de la fenêtre : ROWS

- Indique un intervalle borné par un nombre de ligne défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx valeurs devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx valeurs derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante

```
OVER (PARTITION BY ...
      ORDER BY ...
      ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

1.4.25 Définition de la fenêtre : GROUPS

- Indique un intervalle borné par un groupe de lignes de valeurs identiques défini avant et après la ligne courante
- Borne de départ :
  - `xxx PRECEDING` : depuis les xxx groupes de valeurs identiques devant la ligne courante
  - `CURRENT ROW` : depuis la ligne courante ou le premier élément identique dans le tri réalisé par ORDER BY
- Borne de fin :
  - `xxx FOLLOWING` : depuis les xxx groupes de valeurs identiques derrière la ligne courante
  - `CURRENT ROW` : jusqu'à la ligne courante ou le dernier élément identique dans le tri réalisé par ORDER BY

```
OVER (PARTITION BY ...
      ORDER BY ...
      GROUPS BETWEEN 2 PRECEDING AND 1 FOLLOWING
```

Ceci n'est disponible que depuis la version 11.

1.4.26 Définition de la fenêtre : EXCLUDE

- Indique des lignes à exclure de la fenêtre de données (v11+)
- `EXCLUDE CURRENT ROW` : exclut la ligne courante
- `EXCLUDE GROUP` : exclut la ligne courante et le groupe de valeurs identiques dans l'ordre
- `EXCLUDE TIES` exclut et le groupe de valeurs identiques à la ligne courante dans l'ordre mais pas la ligne courante
- `EXCLUDE NO OTHERS` : pas d'exclusion (valeur par défaut)

Ceci n'est disponible que depuis la version 11.

1.4.27 Définition de la fenêtre : exemple

```
SELECT pays, continent, population,
       last_value(population)
        OVER (PARTITION BY continent ORDER BY population
              RANGE BETWEEN UNBOUNDED PRECEDING
                        AND UNBOUNDED FOLLOWING)
  FROM population;
```

```
         pays          | continent | population | last_value
-----------------------+-----------+------------+------------
 Arabie Saoudite       | Asie      |       28.8 |     1385.6
 Ouzbékistan           | Asie      |       28.9 |     1385.6
 Iraq                  | Asie      |       33.8 |     1385.6
 Chine (4)             | Asie      |     1385.6 |     1385.6
 Lettonie              | Europe    |        2.1 |       64.3
 Finlande              | Europe    |        5.4 |       64.3
 France métropolitaine | Europe    |       64.3 |       64.3
```

1.5 WITHIN GROUP

- WITHIN GROUP
  - PostgreSQL 9.4
- Utilité :
  - calcul de médianes, centiles

La clause WITHIN GROUP est une nouvelle clause pour les agrégats utilisant des fonctions dont les données doivent être triées. Quelques fonctions ont été ajoutées pour profiter au mieux de cette nouvelle clause.

1.5.1 WITHIN GROUP : exemple

```
SELECT continent,
  percentile_disc(0.5)
    WITHIN GROUP (ORDER BY population) AS "mediane",
  percentile_disc(0.95)
    WITHIN GROUP (ORDER BY population) AS "95pct",
  ROUND(AVG(population), 1) AS moyenne
FROM population
 GROUP BY continent;

         continent          | mediane | 95pct  | moyenne
----------------------------+---------+--------+---------
 Afrique                    |    33.0 |  173.6 |    44.3
 Amérique du Nord           |    35.2 |  320.1 |   177.7
 Amérique latine. Caraïbes  |    30.4 |  200.4 |    53.3
 Asie                       |    53.3 | 1252.1 |   179.9
 Europe                     |     9.4 |   82.7 |    21.8
```

Cet exemple permet d'afficher le continent, la médiane de la population par continent et la population du pays le moins peuplé parmi les 5% de pays les plus peuplés de chaque continent.

Pour rappel, la table contient les données suivantes :

```
postgres=# SELECT * FROM population ORDER BY continent, population;
         pays           | population | superficie | densite | continent
------------------------+------------+------------+---------+----------
```

```
 Tunisie                |       11.0 |        164 |      67 | Afrique
 Zimbabwe               |       14.1 |        391 |      36 | Afrique
 Soudan                 |       14.1 |        197 |      72 | Afrique
 Burkina Faso           |       16.9 |        274 |      62 | Afrique
(...)
```

En ajoutant le support de cette clause, PostgreSQL améliore son support de la norme SQL 2008 et permet le développement d'analyses statistiques plus élaborées.

1.6 GROUPING SETS

- `GROUPING SETS/ROLLUP/CUBE`
- Extension de `GROUP BY`
- PostgreSQL 9.5
- Utilité :
  - présente le résultat de plusieurs agrégations différentes
  - réaliser plusieurs agrégations différentes dans la même requête

Les `GROUPING SETS` permettent de définir plusieurs clauses d'agrégation `GROUP BY`. Les résultats seront présentés comme si plusieurs requêtes d'agrégation avec les clauses `GROUP BY` mentionnées étaient assemblées avec `UNION ALL`.

1.6.1 GROUPING SETS : jeu de données

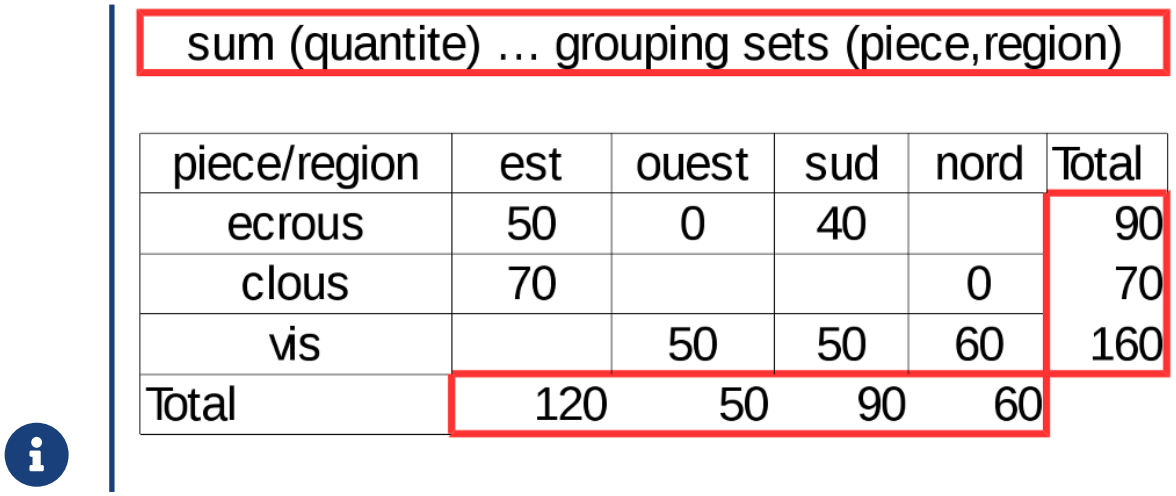
stock		
piece	region	quantite
ecrous	est	50
ecrous	ouest	0
ecrous	sud	40
clous	est	70
clous	nord	40
vis	ouest	50
vis	sud	50
vis	nord	60

stock				
piece/region	est	ouest	sud	nord
ecrous	50	0	40	
clous	70			0
vis		50	50	60

```
CREATE TABLE stock AS SELECT * FROM (
    VALUES ('ecrous', 'est',   50),
           ('ecrous', 'ouest',  0),
```

```
            ('ecrous', 'sud',   40),
            ('clous',  'est',   70),
            ('clous',  'nord',   0),
            ('vis',    'ouest', 50),
            ('vis',    'sud',   50),
            ('vis',    'nord',  60)
) AS VALUES(piece, region, quantite);
```

1.6.2 GROUPING SETS : exemple visuel



sum (quantite) … grouping sets (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total	120	50	90	60	

1.6.3 GROUPING SETS : exemple ordre sql

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
 piece  | region | sum
--------+--------+-----
 clous  |        |  70
 ecrous |        |  90
 vis    |        | 160
        | est    | 120
        | nord   |  60
        | ouest  |  50
        | sud    |  90
```

1.6.4 GROUPING SETS : équivalent

- On peut se passer de la clause GROUPING SETS
  - mais la requête sera plus lente

```
SELECT piece,NULL as region,sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region,sum(quantite)
  FROM STOCK
  GROUP BY region;
```

Le comportement de la clause GROUPING SETS peut être émulée avec deux requêtes utilisant chacune une clause GROUP BY sur les colonnes de regroupement souhaitées.

Cependant, le plan d'exécution de la requête équivalente conduit à deux lectures et peut être particulièrement coûteux si le jeu de données sur lequel on souhaite réaliser les agrégations est important :

```
EXPLAIN SELECT piece,NULL as region,sum(quantite)
  FROM stock
  GROUP BY piece
UNION ALL
SELECT NULL, region,sum(quantite)
  FROM STOCK
  GROUP BY region;

                                QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=1.12..2.38 rows=7 width=44)
   ->  HashAggregate  (cost=1.12..1.15 rows=3 width=45)
         Group Key: stock.piece
         ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=9)
   ->  HashAggregate  (cost=1.12..1.16 rows=4 width=44)
         Group Key: stock_1.region
         ->  Seq Scan on stock stock_1  (cost=0.00..1.08 rows=8 width=8)
```

La requête utilisant la clause GROUPING SETS propose un plan bien plus efficace :

```
EXPLAIN SELECT piece,region,sum(quantite)
FROM stock GROUP BY GROUPING SETS (piece,region);
                            QUERY PLAN
-------------------------------------------------------------------
 GroupAggregate  (cost=1.20..1.58 rows=14 width=17)
   Group Key: piece
   Sort Key: region
     Group Key: region
   ->  Sort  (cost=1.20..1.22 rows=8 width=13)
```

```
        Sort Key: piece
        ->  Seq Scan on stock  (cost=0.00..1.08 rows=8 width=13)
```

1.6.5 ROLLUP

- ROLLUP
- PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête

La clause ROLLUP est une fonctionnalité d'analyse type OLAP du langage SQL. Elle s'utilise dans la clause GROUP BY, tout comme GROUPING SETS

1.6.6 ROLLUP : exemple visuel

sum (quantite) … ROLLUP (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total					320

1.6.7 ROLLUP : exemple ordre sql

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY ROLLUP (piece,region);
```

Cette requête est équivalente à la requête suivante utilisant `GROUPING SETS` :

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY GROUPING SETS ((),(piece),(piece,region));
```

Sur une requête un peu plus intéressante, effectuant des statistiques sur des ventes :

```
SELECT type_client, code_pays, SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           |           |  111557177.00
(...)
 P           | RU        |  287605812.99
 P           | US        |  296424154.49
 P           |           | 4692152751.08
             |           | 5217862160.65
```

Une fonction `GROUPING`, associée à ROLLUP, permet de déterminer si la ligne courante correspond à un regroupement donné. Elle est de la forme d'un masque de bit converti au format décimal :

```
SELECT row_number()
         OVER ( ORDER BY grouping(piece,region)) AS ligne,
       grouping(piece,region)::bit(2) AS g,
```

```
       piece,
       region,
       sum(quantite)
FROM stock
GROUP BY CUBE (piece,region)
ORDER BY g ;
```

```
 ligne | g  | piece  | region | sum
-------+----+--------+--------+-----
     1 | 00 | clous  | est    | 150
     2 | 00 | clous  | nord   |  10
     3 | 00 | ecrous | est    | 110
     4 | 00 | ecrous | ouest  |  10
     5 | 00 | ecrous | sud    |  90
     6 | 00 | vis    | nord   | 130
     7 | 00 | vis    | ouest  | 110
     8 | 00 | vis    | sud    | 110
     9 | 01 | vis    |        | 350
    10 | 01 | ecrous |        | 210
    11 | 01 | clous  |        | 160
    12 | 10 |        | ouest  | 120
    13 | 10 |        | sud    | 200
    14 | 10 |        | est    | 260
    15 | 10 |        | nord   | 140
    16 | 11 |        |        | 720
```

Voici un autre exemple :

```
SELECT COALESCE(service,
    CASE
        WHEN GROUPING(service) = 0 THEN 'Unknown' ELSE 'Total'
    END) AS service,
       sum(salaire) AS salaires_service, count(*) AS nb_employes
  FROM employes
 GROUP BY ROLLUP (service);
  service    | salaires_service | nb_employes
-------------+------------------+-------------
 Courrier    |          7500.00 |           2
 Direction   |         50000.00 |           1
 Publication |          7000.00 |           2
 Total       |         64500.00 |           5
(4 rows)
```

Ou appliqué à l'exemple un peu plus complexe :

```
SELECT COALESCE(type_client,
       CASE
         WHEN GROUPING(type_client) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS type_client,
       COALESCE(code_pays,
       CASE
         WHEN GROUPING(code_pays) = 0 THEN 'Unknown' ELSE 'Total'
       END) AS code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
```

```
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY ROLLUP (type_client, code_pays);
```

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           | Total     |  111557177.00
(...)
 P           | US        |  296424154.49
 P           | Total     | 4692152751.08
 Total       | Total     | 5217862160.65
```

1.6.8 CUBE

- CUBE
  - PostgreSQL 9.5
- Utilité :
  - calcul de totaux dans la même requête
  - sur toutes les clauses de regroupement

La clause CUBE est une autre fonctionnalité d'analyse type OLAP du langage SQL. Tout comme ROLLUP, elle s'utilise dans la clause GROUP BY.

1.6.9 CUBE : exemple visuel

sum (quantite) … CUBE (piece,region)

piece/region	est	ouest	sud	nord	Total
ecrous	50	0	40		90
clous	70			0	70
vis		50	50	60	160
Total	120	50	90	60	320

1.6.10 CUBE : exemple ordre sql

```
SELECT piece,region,sum(quantite)
FROM stock GROUP BY CUBE (piece,region);
```

Cette requête est équivalente à la requête suivante utilisant `GROUPING SETS` :

```
SELECT piece,region,sum(quantite)
FROM stock
GROUP BY GROUPING SETS (
  (),
  (piece),
  (region),
  (piece,region)
  );
```

Elle permet de réaliser des regroupements sur l'ensemble des combinaisons possibles des clauses de regroupement indiquées. Pour de plus amples détails, se référer à cet article Wikipédia[1].

En reprenant la requête de l'exemple précédent :

```
SELECT type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
```

[1] https://en.wikipedia.org/wiki/OLAP_cube

```
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle retournera le résultat suivant :

```
 type_client | code_pays |    montant
-------------+-----------+---------------
 A           | CA        |    6273168.32
 A           | CN        |    7928641.50
 A           | DE        |    6642061.57
 A           | DZ        |    6404425.16
 A           | FR        |   55261295.52
 A           | IN        |    7224008.95
 A           | PE        |    7356239.93
 A           | RU        |    6766644.98
 A           | US        |    7700691.07
 A           |           |  111557177.00
 E           | CA        |   28457655.81
 E           | CN        |   25537539.68
 E           | DE        |   25508815.68
 E           | DZ        |   24821750.17
 E           | FR        |  209402443.24
 E           | IN        |   26788642.27
 E           | PE        |   24541974.54
 E           | RU        |   25397116.39
 E           | US        |   23696294.79
 E           |           |  414152232.57
 P           | CA        |  292975985.52
 P           | CN        |  287795272.87
 P           | DE        |  287337725.21
 P           | DZ        |  302501132.54
 P           | FR        | 2341977444.49
 P           | IN        |  295256262.73
 P           | PE        |  300278960.24
 P           | RU        |  287605812.99
 P           | US        |  296424154.49
 P           |           | 4692152751.08
             |           | 5217862160.65
             | CA        |  327706809.65
             | CN        |  321261454.05
             | DE        |  319488602.46
             | DZ        |  333727307.87
             | FR        | 2606641183.25
             | IN        |  329268913.95
             | PE        |  332177174.71
             | RU        |  319769574.36
             | US        |  327821140.35
```

Dans ce genre de contexte, lorsque le regroupement est réalisé sur l'ensemble des valeurs d'un critère de regroupement, alors la valeur qui apparaît est NULL pour la colonne correspondante. Si la colonne

possède des valeurs NULL légitimes, il est alors difficile de les distinguer. On utilise alors la fonction GROUPING() qui permet de déterminer si le regroupement porte sur l'ensemble des valeurs de la colonne. L'exemple suivant montre une requête qui exploite cette fonction :

```
SELECT GROUPING(type_client,code_pays)::bit(2),
       GROUPING(type_client)::boolean g_type_cli,
       GROUPING(code_pays)::boolean g_code_pays,
       type_client,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients cl
    ON (c.client_id = cl.client_id)
  JOIN contacts co
    ON (cl.contact_id = co.contact_id)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
GROUP BY CUBE (type_client, code_pays);
```

Elle produit le résultat suivant :

```
 grouping | g_type_cli | g_code_pays | type_client | code_pays |    montant
----------+------------+-------------+-------------+-----------+---------------
 00       | f          | f           | A           | CA        |    6273168.32
 00       | f          | f           | A           | CN        |    7928641.50
 00       | f          | f           | A           | DE        |    6642061.57
 00       | f          | f           | A           | DZ        |    6404425.16
 00       | f          | f           | A           | FR        |   55261295.52
 00       | f          | f           | A           | IN        |    7224008.95
 00       | f          | f           | A           | PE        |    7356239.93
 00       | f          | f           | A           | RU        |    6766644.98
 00       | f          | f           | A           | US        |    7700691.07
 01       | f          | t           | A           |           |  111557177.00
(...)
 01       | f          | t           | P           |           | 4692152751.08
 11       | t          | t           |             |           | 5217862160.65
 10       | t          | f           |             | CA        |  327706809.65
 10       | t          | f           |             | CN        |  321261454.05
 10       | t          | f           |             | DE        |  319488602.46
 10       | t          | f           |             | DZ        |  333727307.87
 10       | t          | f           |             | FR        | 2606641183.25
 10       | t          | f           |             | IN        |  329268913.95
 10       | t          | f           |             | PE        |  332177174.71
 10       | t          | f           |             | RU        |  319769574.36
 10       | t          | f           |             | US        |  327821140.35
(40 rows)
```

L'application sera alors à même de gérer la présentation des résultats en fonction des valeurs de `grouping` ou `g_type_client` et `g_code_pays`.

1.7 TRAVAUX PRATIQUES

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

```
     Table "public.brno_2015"
  Column   |   Type   | Modifiers
-----------+----------+-----------
 no_tour   | integer  |
 no_pilote | integer  |
 lap_time  | interval |
```

Une table `pilotes` permet de connaître les détails d'un pilote :

```
      Table "public.pilotes"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 no          | integer |
 nom         | text    |
 nationalite | text    |
 ecurie      | text    |
 moto        | text    |
```

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

Agrégation

1. Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?
2. Déterminer quel est le pilote le plus régulier (écart-type).

Window Functions

3. Afficher la place sur le podium pour chaque coureur.
4. À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.
5. Pour chaque tour, afficher :

- le nom du pilote ;
- son rang dans le tour ;
- son temps depuis le début de la course ;
- dans le tour, la différence de temps par rapport au premier.

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?
7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.
8. En quelle position a terminé le coureur qui a doublé le plus de personnes ? Combien de personnes a-t-il doublées ?

Grouping Sets

Ce TP nécessite PostgreSQL 9.5 ou supérieur. Il s'appuie sur les tables présentes dans le schéma `magasin`.

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.
10. Ajouter également le montant total des commandes depuis le début de l'activité.
11. Ajouter également le montant total des commandes par pays.

1.8 TRAVAUX PRATIQUES (SOLUTIONS)

Le schéma `brno2015` dispose d'une table pilotes ainsi que les résultats tour par tour de la course de MotoGP de Brno (CZ) de la saison 2015.

La table `brno2015` indique pour chaque tour, pour chaque pilote, le temps réalisé dans le tour :

```
     Table "public.brno_2015"
  Column   |   Type   | Modifiers
-----------+----------+-----------
 no_tour   | integer  |
 no_pilote | integer  |
 lap_time  | interval |
```

Une table `pilotes` permet de connaître les détails d'un pilote :

```
      Table "public.pilotes"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 no          | integer |
 nom         | text    |
 nationalite | text    |
 ecurie      | text    |
 moto        | text    |
```

Précisions sur les données à manipuler : la course est réalisée en plusieurs tours; certains coureurs n'ont pas terminé la course, leur relevé de tours s'arrête donc brutalement.

Agrégation

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

1. Quel est le pilote qui a le moins gros écart entre son meilleur tour et son moins bon tour ?

Le coureur :

```
SELECT nom, max(lap_time) - min(lap_time) as ecart
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 GROUP BY 1
 ORDER BY 2
 LIMIT 1;
```

La requête donne le résultat suivant :

```
       nom       |    ecart
-----------------+-------------
 Jorge LORENZO | 00:00:04.661
```

2. Déterminer quel est le pilote le plus régulier (écart-type).

Nous excluons le premier tour car il s'agit d'une course avec départ arrêté, donc ce tour est plus lent que les autres, ici d'au moins 8 secondes :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 WHERE no_tour > 1
 GROUP BY 1
 ORDER BY 2
 LIMIT 1;
```

Le résultat montre le coureur qui a abandonné en premier :

```
       nom       |      stddev
-----------------+-------------------
 Alex DE ANGELIS | 0.130107647741847
```

On s'aperçoit qu'Alex De Angelis n'a pas terminé la course. Il semble donc plus intéressant de ne prendre en compte que les pilotes qui ont terminé la course et toujours en excluant le premier tour (il y a 22 tours sur cette course, on peut le positionner soit en dur dans la requête, soit avec un sous-select permettant de déterminer le nombre maximum de tours) :

```
SELECT nom, stddev(extract (epoch from lap_time)) as stddev
  FROM brno_2015
  JOIN pilotes
    ON (no_pilote = no)
 WHERE no_tour > 1
   AND no_pilote in (SELECT no_pilote FROM brno_2015 WHERE no_tour=22)
GROUP BY 1
ORDER BY 2
LIMIT 1;
```

Le pilote 19 a donc été le plus régulier :

```
       nom       |      stddev
-----------------+-------------------
 Alvaro BAUTISTA | 0.222825823492654
```

Window Functions

Si ce n'est pas déjà fait, nous positionnons le search_path pour chercher les objets du schéma `brno2015` :

```
SET search_path = brno2015;
```

3. Afficher la place sur le podium pour chaque coureur.

Les coureurs qui ne franchissent pas la ligne d'arrivée sont dans le classement malgré tout. Il faut donc tenir compte de cela dans l'affichage des résultats.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
       nom, ecurie, total_time
  FROM (SELECT no_pilote,
               sum(lap_time) over (PARTITION BY no_pilote) as total_time,
               max(no_tour) over (PARTITION BY no_pilote) as max_lap
          FROM brno_2015
       ) AS race_data
  JOIN pilotes
```

```
    ON (race_data.no_pilote = pilotes.no)
 GROUP BY nom, ecurie, max_lap, total_time
 ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

```
 rang |       nom        |           ecurie             |  total_time
------+------------------+------------------------------+--------------
    1 | Jorge LORENZO    | Movistar Yamaha MotoGP       | 00:42:53.042
    2 | Marc MARQUEZ     | Repsol Honda Team            | 00:42:57.504
    3 | Valentino ROSSI  | Movistar Yamaha MotoGP       | 00:43:03.439
    4 | Andrea IANNONE   | Ducati Team                  | 00:43:06.113
    5 | Dani PEDROSA     | Repsol Honda Team            | 00:43:08.692
    6 | Andrea DOVIZIOSO | Ducati Team                  | 00:43:08.767
    7 | Bradley SMITH    | Monster Yamaha Tech 3        | 00:43:14.863
    8 | Pol ESPARGARO    | Monster Yamaha Tech 3        | 00:43:16.282
    9 | Aleix ESPARGARO  | Team SUZUKI ECSTAR           | 00:43:36.826
   10 | Danilo PETRUCCI  | Octo Pramac Racing           | 00:43:38.303
   11 | Yonny HERNANDEZ  | Octo Pramac Racing           | 00:43:43.015
   12 | Scott REDDING    | EG 0,0 Marc VDS              | 00:43:43.216
   13 | Alvaro BAUTISTA  | Aprilia Racing Team Gresini  | 00:43:47.479
   14 | Stefan BRADL     | Aprilia Racing Team Gresini  | 00:43:47.666
   15 | Loris BAZ        | Forward Racing               | 00:43:53.358
   16 | Hector BARBERA   | Avintia Racing               | 00:43:54.637
   17 | Nicky HAYDEN     | Aspar MotoGP Team            | 00:43:55.43
   18 | Mike DI MEGLIO   | Avintia Racing               | 00:43:58.986
   19 | Jack MILLER      | CWM LCR Honda                | 00:44:04.449
   20 | Claudio CORTI    | Forward Racing               | 00:44:43.075
   21 | Karel ABRAHAM    | AB Motoracing                | 00:44:55.697
   22 | Maverick VIÑALES | Team SUZUKI ECSTAR           | 00:29:31.557
   23 | Cal CRUTCHLOW    | CWM LCR Honda                | 00:27:38.315
   24 | Eugene LAVERTY   | Aspar MotoGP Team            | 00:08:04.096
   25 | Alex DE ANGELIS  | E-Motion IodaRacing Team     | 00:06:05.782
(25 rows)
```

4. À partir de la requête précédente, afficher également la différence du temps de chaque coureur par rapport à celui de la première place.

La requête n'est pas beaucoup modifiée, seule la fonction `first_value()` est utilisée pour déterminer le temps du vainqueur, temps qui sera ensuite retranché au temps du coureur courant.

```
SELECT rank() OVER (ORDER BY max_lap desc, total_time asc) AS rang,
       nom, ecurie, total_time,
       total_time - first_value(total_time)
         OVER (ORDER BY max_lap desc, total_time asc) AS difference
  FROM (SELECT no_pilote,
               sum(lap_time) over (PARTITION BY no_pilote) as total_time,
               max(no_tour) over (PARTITION BY no_pilote) as max_lap
          FROM brno_2015
       ) AS race_data
  JOIN pilotes
    ON (race_data.no_pilote = pilotes.no)
 GROUP BY nom, ecurie, max_lap, total_time
 ORDER BY max_lap desc, total_time asc;
```

La requête affiche le résultat suivant :

```
 r|       nom       |          ecurie      | total_time  |  difference
--+-----------------+----------------------+-------------+---------------
 1| Jorge LORENZO   | Movistar Yamaha [...]|00:42:53.042 | 00:00:00
 2| Marc MARQUEZ    | Repsol Honda Team    |00:42:57.504 | 00:00:04.462
 3| Valentino ROSSI | Movistar Yamaha [...]|00:43:03.439 | 00:00:10.397
 4| Andrea IANNONE  | Ducati Team          |00:43:06.113 | 00:00:13.071
 5| Dani PEDROSA    | Repsol Honda Team    |00:43:08.692 | 00:00:15.65
 6| Andrea DOVIZIOSO| Ducati Team          |00:43:08.767 | 00:00:15.725
 7| Bradley SMITH   | Monster Yamaha Tech 3|00:43:14.863 | 00:00:21.821
 8| Pol ESPARGARO   | Monster Yamaha Tech 3|00:43:16.282 | 00:00:23.24
 9| Aleix ESPARGARO | Team SUZUKI ECSTAR   |00:43:36.826 | 00:00:43.784
10| Danilo PETRUCCI | Octo Pramac Racing   |00:43:38.303 | 00:00:45.261
11| Yonny HERNANDEZ | Octo Pramac Racing   |00:43:43.015 | 00:00:49.973
12| Scott REDDING   | EG 0,0 Marc VDS      |00:43:43.216 | 00:00:50.174
13| Alvaro BAUTISTA | Aprilia Racing [...] |00:43:47.479 | 00:00:54.437
14| Stefan BRADL    | Aprilia Racing [...] |00:43:47.666 | 00:00:54.624
15| Loris BAZ       | Forward Racing       |00:43:53.358 | 00:01:00.316
16| Hector BARBERA  | Avintia Racing       |00:43:54.637 | 00:01:01.595
17| Nicky HAYDEN    | Aspar MotoGP Team    |00:43:55.43  | 00:01:02.388
18| Mike DI MEGLIO  | Avintia Racing       |00:43:58.986 | 00:01:05.944
19| Jack MILLER     | CWM LCR Honda        |00:44:04.449 | 00:01:11.407
20| Claudio CORTI   | Forward Racing       |00:44:43.075 | 00:01:50.033
21| Karel ABRAHAM   | AB Motoracing        |00:44:55.697 | 00:02:02.655
22| Maverick VIÑALES| Team SUZUKI ECSTAR   |00:29:31.557 | -00:13:21.485
23| Cal CRUTCHLOW   | CWM LCR Honda        |00:27:38.315 | -00:15:14.727
24| Eugene LAVERTY  | Aspar MotoGP Team    |00:08:04.096 | -00:34:48.946
25| Alex DE ANGELIS | E-Motion Ioda[...]    |00:06:05.782 | -00:36:47.26
(25 rows)
```

5. Pour chaque tour, afficher :

   - le nom du pilote ;
   - son rang dans le tour ;
   - son temps depuis le début de la course ;
   - dans le tour, la différence de temps par rapport au premier.

Pour construire cette requête, nous avons besoin d'obtenir le temps cumulé tour après tour pour chaque coureur. Nous commençons donc par écrire une première requête :

```
SELECT *,
       SUM(lap_time)
         OVER (PARTITION BY no_pilote ORDER BY no_tour) AS temps_tour_glissant
 FROM brno_2015
```

Elle retourne le résultat suivant :

```
 no_tour | no_pilote |   lap_time   | temps_tour_glissant
---------+-----------+--------------+---------------------
       1 |         4 | 00:02:02.209 | 00:02:02.209
       2 |         4 | 00:01:57.57  | 00:03:59.779
       3 |         4 | 00:01:57.021 | 00:05:56.8
       4 |         4 | 00:01:56.943 | 00:07:53.743
       5 |         4 | 00:01:57.012 | 00:09:50.755
       6 |         4 | 00:01:57.011 | 00:11:47.766
       7 |         4 | 00:01:57.313 | 00:13:45.079
```

```
       8 |          4 | 00:01:57.95   | 00:15:43.029
       9 |          4 | 00:01:57.296  | 00:17:40.325
      10 |          4 | 00:01:57.295  | 00:19:37.62
      11 |          4 | 00:01:57.185  | 00:21:34.805
      12 |          4 | 00:01:57.45   | 00:23:32.255
      13 |          4 | 00:01:57.457  | 00:25:29.712
      14 |          4 | 00:01:57.362  | 00:27:27.074
      15 |          4 | 00:01:57.482  | 00:29:24.556
      16 |          4 | 00:01:57.358  | 00:31:21.914
      17 |          4 | 00:01:57.617  | 00:33:19.531
      18 |          4 | 00:01:57.594  | 00:35:17.125
      19 |          4 | 00:01:57.412  | 00:37:14.537
      20 |          4 | 00:01:57.786  | 00:39:12.323
      21 |          4 | 00:01:58.087  | 00:41:10.41
      22 |          4 | 00:01:58.357  | 00:43:08.767
(…)
```

Cette requête de base est ensuite utilisée dans une CTE qui sera utilisée par la requête répondant à la question de départ. La colonne `temps_tour_glissant` est utilisée pour calculer le rang du pilote dans la course, est affiché et le temps cumulé du meilleur pilote est récupéré avec la fonction `first_value` :

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time)
      OVER (PARTITION BY no_pilote
            ORDER BY no_tour
            )  as temps_tour_glissant
    FROM brno_2015
    ORDER BY no_pilote, no_tour
)

SELECT no_tour, nom,
rank() OVER (PARTITION BY no_tour
             ORDER BY temps_tour_glissant ASC
            ) as place_course,
temps_tour_glissant,
temps_tour_glissant - first_value(temps_tour_glissant)
OVER (PARTITION BY no_tour
      ORDER BY temps_tour_glissant asc
    ) AS difference
FROM temps_glissant t
JOIN pilotes p ON p.no = t.no_pilote;
```

On pouvait également utiliser une simple sous-requête pour obtenir le même résultat :

```
SELECT no_tour,
  nom,
  rank()
    OVER (PARTITION BY no_tour
          ORDER BY temps_tour_glissant ASC
          ) AS place_course,
  temps_tour_glissant,
  temps_tour_glissant - first_value(temps_tour_glissant)
    OVER (PARTITION BY no_tour
          ORDER BY temps_tour_glissant asc
```

```
        ) AS difference
FROM (
  SELECT *, SUM(lap_time)
    OVER (PARTITION BY no_pilote
          ORDER BY no_tour)
          AS temps_tour_glissant
  FROM brno_2015) course
  JOIN pilotes
    ON (pilotes.no = course.no_pilote)
ORDER BY no_tour;
```

La requête fournit le résultat suivant :

```
no.|       nom        | place_c. | temps_tour_glissant |  difference
---+------------------+----------+---------------------+--------------
 1 | Jorge LORENZO    |        1 | 00:02:00.83         | 00:00:00
 1 | Marc MARQUEZ     |        2 | 00:02:01.058        | 00:00:00.228
 1 | Andrea DOVIZIOSO |        3 | 00:02:02.209        | 00:00:01.379
 1 | Valentino ROSSI  |        4 | 00:02:02.329        | 00:00:01.499
 1 | Andrea IANNONE   |        5 | 00:02:02.597        | 00:00:01.767
 1 | Bradley SMITH    |        6 | 00:02:02.861        | 00:00:02.031
 1 | Pol ESPARGARO    |        7 | 00:02:03.239        | 00:00:02.409
( ..)
 2 | Jorge LORENZO    |        1 | 00:03:57.073        | 00:00:00
 2 | Marc MARQUEZ     |        2 | 00:03:57.509        | 00:00:00.436
 2 | Valentino ROSSI  |        3 | 00:03:59.696        | 00:00:02.623
 2 | Andrea DOVIZIOSO |        4 | 00:03:59.779        | 00:00:02.706
 2 | Andrea IANNONE   |        5 | 00:03:59.9          | 00:00:02.827
 2 | Bradley SMITH    |        6 | 00:04:00.355        | 00:00:03.282
 2 | Pol ESPARGARO    |        7 | 00:04:00.87         | 00:00:03.797
 2 | Maverick VIÑALES |        8 | 00:04:01.187        | 00:00:04.114
(…)
(498 rows)
```

6. Pour chaque coureur, quel est son meilleur tour et quelle place avait-il sur ce tour ?

Il est ici nécessaire de sélectionner pour chaque tour le temps du meilleur tour. On peut alors sélectionner les tours pour lequels le temps du tour est égal au meilleur temps :

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time)
        OVER (PARTITION BY no_pilote
              ORDER BY no_tour
             ) as temps_tour_glissant
    FROM brno_2015
    ORDER BY no_pilote, no_tour
),

classement_tour AS (
    SELECT no_tour, no_pilote, lap_time,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant,
```

```
    min(lap_time) OVER (PARTITION BY no_pilote) as meilleur_temps
    FROM temps_glissant
)

SELECT no_tour, nom, place_course, lap_time
FROM classement_tour t
JOIN pilotes p ON p.no = t.no_pilote
WHERE lap_time = meilleur_temps;
```

Ce qui donne le résultat suivant :

```
 no_tour |       nom        | place_course |   lap_time
---------+------------------+--------------+--------------
       4 | Jorge LORENZO    |            1 | 00:01:56.169
       4 | Marc MARQUEZ     |            2 | 00:01:56.048
       4 | Valentino ROSSI  |            3 | 00:01:56.747
       6 | Andrea IANNONE   |            5 | 00:01:56.86
       6 | Dani PEDROSA     |            7 | 00:01:56.975
       4 | Andrea DOVIZIOSO |            4 | 00:01:56.943
       3 | Bradley SMITH    |            6 | 00:01:57.25
      17 | Pol ESPARGARO    |            8 | 00:01:57.454
       4 | Aleix ESPARGARO  |           12 | 00:01:57.844
       4 | Danilo PETRUCCI  |           11 | 00:01:58.121
       9 | Yonny HERNANDEZ  |           14 | 00:01:58.53
       2 | Scott REDDING    |           14 | 00:01:57.976
       3 | Alvaro BAUTISTA  |           21 | 00:01:58.71
       3 | Stefan BRADL     |           16 | 00:01:58.38
       3 | Loris BAZ        |           19 | 00:01:58.679
       2 | Hector BARBERA   |           15 | 00:01:58.405
       2 | Nicky HAYDEN     |           16 | 00:01:58.338
       3 | Mike DI MEGLIO   |           18 | 00:01:58.943
       4 | Jack MILLER      |           22 | 00:01:59.007
       2 | Claudio CORTI    |           24 | 00:02:00.377
      14 | Karel ABRAHAM    |           23 | 00:02:01.716
       3 | Maverick VIÑALES |            8 | 00:01:57.436
       3 | Cal CRUTCHLOW    |           11 | 00:01:57.652
       3 | Eugene LAVERTY   |           20 | 00:01:58.977
       3 | Alex DE ANGELIS  |           23 | 00:01:59.257
(25 rows)
```

7. Déterminer quels sont les coureurs ayant terminé la course qui ont gardé la même position tout au long de la course.

```
WITH nb_tour AS (
    SELECT max(no_tour) FROM brno_2015
),
temps_glissant AS (
    SELECT  no_tour, no_pilote, lap_time,
    sum(lap_time) OVER (
        PARTITION BY no_pilote
        ORDER BY no_tour
    ) as temps_tour_glissant,
    max(no_tour) OVER (PARTITION BY no_pilote) as total_tour
    FROM brno_2015
),
classement_tour AS (
```

```
    SELECT no_tour, no_pilote, lap_time, total_tour,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course
    FROM temps_glissant
)
SELECT no_pilote
FROM classement_tour t
JOIN nb_tour n ON n.max = t.total_tour
GROUP BY no_pilote
HAVING count(DISTINCT place_course) = 1;
```

Elle retourne le résultat suivant :

```
 no_pilote
-----------
        93
        99
```

8. En quelle position a terminé le coureur qui a doublé le plus de personnes. Combien de personnes a-t-il doublées ?

```
WITH temps_glissant AS (
    SELECT no_tour, no_pilote, lap_time,
    sum(lap_time) OVER (
        PARTITION BY no_pilote
        ORDER BY no_tour
    ) as temps_tour_glissant
    FROM brno_2015
),
classement_tour AS (
    SELECT no_tour, no_pilote, lap_time,
    rank() OVER (
        PARTITION BY no_tour
        ORDER BY temps_tour_glissant
    ) as place_course,
    temps_tour_glissant
    FROM temps_glissant
),
depassement  AS (
    SELECT no_pilote,
      last_value(place_course) OVER (PARTITION BY no_pilote) as rang,
    CASE
        WHEN lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course < 0
        THEN 0
        ELSE lag(place_course) OVER (
            PARTITION BY no_pilote
            ORDER BY no_tour
            ) - place_course
        END AS depasse
    FROM classement_tour t
)
```

```
SELECT no_pilote, rang, sum(depasse)
FROM depassement
GROUP BY no_pilote, rang
ORDER BY sum(depasse) DESC
LIMIT 1;
```

Grouping Sets

La suite de ce TP est maintenant réalisé avec la base de formation habituelle. Attention, ce TP nécessite l'emploi d'une version 9.5 ou supérieure de PostgreSQL.

Tout d'abord, nous positionnons le search_path pour chercher les objets du schéma `magasin` :

```
SET search_path = magasin;
```

9. En une seule requête, afficher le montant total des commandes par année et pays et le montant total des commandes uniquement par année.

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY GROUPING SETS (
   (extract('year' from date_commande), code_pays),
   (extract('year' from date_commande))
 );
```

Le résultat attendu est :

```
 annee | code_pays | montant_total_commande
-------+-----------+------------------------
  2003 | DE        |                49634.24
  2003 | FR        |                10003.98
  2003 |           |                59638.22
  2008 | CA        |              1016082.18
  2008 | CN        |               801662.75
  2008 | DE        |               694787.87
  2008 | DZ        |               663045.33
  2008 | FR        |              5860607.27
  2008 | IN        |               741850.87
  2008 | PE        |              1167825.32
  2008 | RU        |               577164.50
  2008 | US        |               928661.06
  2008 |           |             12451687.15
(...)
```

10. Ajouter également le montant total des commandes depuis le début de l'activité.

L'opérateur de regroupement ROLL UP amène le niveau d'agrégation sans regroupement :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY ROLLUP (extract('year' from date_commande), code_pays);
```

11. Ajouter également le montant total des commandes par pays.

Cette fois, l'opérateur CUBE permet d'obtenir l'ensemble de ces informations :

```
SELECT extract('year' from date_commande) AS annee, code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

12. À partir de la requête précédente, ajouter une colonne par critère de regroupement, de type booléen, qui est positionnée à `true` lorsque le regroupement est réalisé sur l'ensemble des valeurs de la colonne.

Ces colonnes booléennes permettent d'indiquer à l'application comment gérer la présentation des résultats.

```
SELECT grouping(extract('year' from date_commande))::boolean AS g_annee,
       grouping(code_pays)::boolean AS g_pays,
       extract('year' from date_commande) AS annee,
       code_pays,
       SUM(quantite*prix_unitaire) AS montant_total_commande
  FROM commandes c
  JOIN lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
  JOIN clients
    ON (c.client_id = clients.client_id)
  JOIN contacts co
    ON (clients.contact_id = co.contact_id)
 GROUP BY CUBE (extract('year' from date_commande), code_pays);
```

2/ Types avancés

PostgreSQL offre des types avancés :

- Composés :
  - `hstore`
  - JSON : `json`, `jsonb`
  - XML
- Pour les objets binaires :
  - `bytea`
  - Large Objects

2.1 TYPES COMPOSÉS : GÉNÉRALITÉS

- Un champ = plusieurs attributs
- De loin préférable à une table Entité/Attribut/Valeur
- Uniquement si le modèle relationnel n'est pas assez souple
- 3 types dans PostgreSQL :
  - `hstore` : clé/valeur
  - `json` : JSON, stockage texte, validation syntaxique, fonctions d'extraction
  - `jsonb` : JSON, stockage binaire, accès rapide, fonctions d'extraction, de requêtage, indexation avancée

Ces types sont utilisés quand le modèle relationnel n'est pas assez souple, donc s'il est nécessaire d'ajouter dynamiquement des colonnes à la table suivant les besoins du client, ou si le détail des attributs d'une entité n'est pas connu (modélisation géographique par exemple), etc.

La solution traditionnelle est de créer des tables entité/attribut de ce format :

```
CREATE TABLE attributs_sup (entite int, attribut text, valeur text);
```

On y stocke dans `entite` la clé de l'enregistrement de la table principale, dans `attribut` la colonne supplémentaire, et dans `valeur` la valeur de cet attribut. Ce modèle présente l'avantage évident de résoudre le problème. Les défauts sont par contre nombreux :

- Les attributs d'une ligne peuvent être totalement éparpillés dans la table `attributs_sup` : récupérer n'importe quelle information demandera donc des accès à de nombreux blocs différents.
- Il faudra plusieurs requêtes (au moins deux) pour récupérer le détail d'un enregistrement, avec du code plus lourd côté client pour fusionner le résultat des deux requêtes, ou bien une requête effectuant des jointures (autant que d'attributs, sachant que le nombre de jointures complexifie énormément le travail de l'optimiseur SQL) pour retourner directement l'enregistrement complet.

Toute recherche complexe est très inefficace : une recherche multi-critères sur ce schéma va être extrêmement peu performante. Les statistiques sur les valeurs d'un attribut deviennent nettement moins faciles à estimer pour PostgreSQL. Quant aux contraintes d'intégrité entre valeurs, elles deviennent pour le moins complexes à gérer.

Les types `hstore`, `json` et `jsonb` permettent de résoudre le problème autrement. Ils permettent de stocker les différentes entités dans un seul champ pour chaque ligne de l'entité. L'accès aux attributs se fait par une syntaxe ou des fonctions spécifiques.

Il n'y a même pas besoin de créer une table des attributs séparée : le mécanisme du « TOAST » permet de déporter les champs volumineux (texte, JSON, `hstore`…) dans une table séparée gérée par Post-

greSQL, éventuellement en les compressant, et cela de manière totalement transparente. On y gagne donc en simplicité de développement.

2.2 HSTORE

Stocker des données non structurées

- Extension
- Stockage Clé/Valeur, uniquement texte
- Binaire
- Indexable
- Plusieurs opérateurs disponibles

hstore est une extension, fournie en « contrib ». Elle est donc systématiquement disponible. L'installer permet d'utiliser le type de même nom. On peut ainsi stocker un ensemble de clés/valeurs, exclusivement textes, dans un unique champ.

Ces champs sont indexables et peuvent recevoir des contraintes d'intégrité (unicité, non recouvrement…).

Les `hstore` ne permettent par contre qu'un modèle « plat ». Il s'agit d'un pur stockage clé-valeur. Si vous avez besoin de stocker des informations davantage orientées document, vous devrez vous tourner vers un type JSON.

Ce type perd donc de son intérêt depuis que PostgreSQL 9.4 a apporté le type `jsonb`. Il lui reste sa simplicité d'utilisation.

2.2.1 hstore : exemple

```
CREATE EXTENSION hstore ;

CREATE TABLE animaux (nom text, caract hstore);
INSERT INTO animaux VALUES ('canari','pattes=>2,vole=>oui');
INSERT INTO animaux VALUES ('loup','pattes=>4,carnivore=>oui');
INSERT INTO animaux VALUES ('carpe','eau=>douce');

CREATE INDEX idx_animaux_donnees ON animaux
                                 USING gist (caract);

SELECT *, caract->'pattes' AS nb_pattes FROM animaux
WHERE caract@>'carnivore=>oui';

 nom  |              caract               | nb_pattes
------+-----------------------------------+-----------
 loup | "pattes"=>"4", "carnivore"=>"oui" | 4
```

Les ordres précédents installent l'extension, créent une table avec un champ de type `hstore`, insèrent trois lignes, avec des attributs variant sur chacune, indexent l'ensemble avec un index GiST, et enfin recherchent les lignes où l'attribut `carnivore` possède la valeur `t`.

```
# SELECT * FROM animaux ;

  nom   |              caract
--------+----------------------------------
 canari | "vole"=>"oui", "pattes"=>"2"
 loup   | "pattes"=>"4", "carnivore"=>"oui"
 carpe  | "eau"=>"douce"
```

Les différentes fonctions disponibles sont bien sûr dans la documentation[1].

Par exemple :

```
# UPDATE animaux SET caract = caract||'poil=>t'::hstore
  WHERE nom = 'loup' ;

# SELECT * FROM animaux WHERE caract@>'carnivore=>oui';

 nom  |                      caract
------+-------------------------------------------------
 loup | "poil"=>"t", "pattes"=>"4", "carnivore"=>"oui"
```

Il est possible de convertir un `hstore` en tableau :

```
# SELECT hstore_to_matrix(caract) FROM animaux
  WHERE caract->'vole' = 'oui';

    hstore_to_matrix
-------------------------
 { {vole,oui},{pattes,2} }
```

ou en JSON :

```
# SELECT caract::jsonb FROM animaux
  WHERE (caract->'pattes')::int > 2;

                       caract
-----------------------------------------------------
 {"pattes": "4", "poil": "t", "carnivore": "oui"}
```

L'indexation de ces champs peut se faire avec divers types d'index. Un index unique n'est possible qu'avec un index B-tree classique. Les index GIN ou GiST sont utiles pour rechercher des valeurs d'un attribut. Les index hash ne sont utiles que pour des recherches d'égalité d'un champ entier ; par contre ils sont très compacts. (Rappelons que les index hash sont inutilisables avant PostgreSQL 10).

[1] https://docs.postgresql.fr/current/hstore.html

2.3 JSON

```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "postalCode": "75009"
  }
}
```

- `json` : format texte
- `jsonb` : format binaire, à préférer
- `jsonpath` : SQL/JSON paths (PG 12+)

Le format JSON[2] est devenu extrêmement populaire. Au-delà d'un simple stockage clé/valeur, il permet de stocker des tableaux, ou des hiérarchies, de manière plus simple et lisible qu'en XML. Par exemple, pour décrire une personne, on peut utiliser cette structure :

```
{
  "firstName": "Jean",
  "lastName": "Dupont",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "state": "",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    {
      "type": "personnel",
      "number": "06 12 34 56 78"
    },
    {
      "type": "bureau",
      "number": "07 89 10 11 12"
    }
  ],
  "children": [],
  "spouse": null
}
```

Historiquement, le JSON est apparu dans PostgreSQL 9.2, mais n'est vraiment utilisable qu'avec

[2] https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

l'arrivée du type `jsonb` (binaire) dans PostgreSQL 9.4. Les opérateurs SQL/JSON path[3] ont été ajoutés dans PostgreSQL 12, suite à l'introduction du JSON dans le standard SQL:2016. Ils permettent de spécifier des parties d'un champ JSON.

2.3.1 Type json

- Type texte
- Validation du format JSON
- Fonctions de manipulation JSON
  - Mais ré-analyse du champ pour chaque appel de fonction
  - Indexation comme un simple texte
- => Réservé au stockage à l'identique
  - Sinon préférer `jsonb`

Le type natif `json`, dans PostgreSQL, n'est rien d'autre qu'un habillage autour du type texte. Il valide à chaque insertion/modification que la donnée fournie est une syntaxe JSON valide.

Le stockage est exactement le même qu'une chaîne de texte, et utilise le mécanisme du « TOAST », qui compresse au besoin les plus grands champs, de manière transparente pour l'utilisateur.

Toutefois, le fait que la donnée soit validée comme du JSON permet d'utiliser des fonctions de manipulation, comme l'extraction d'un attribut, la conversion d'un JSON en enregistrement, de façon systématique sur un champ sans craindre d'erreur.

On préférera généralement le type binaire `jsonb`, pour les performances et ses fonctionnalités supplémentaires. Au final, l'intérêt de `json` est surtout de conserver un objet JSON sous sa forme originale.

Les exemples suivants avec `jsonb` sont aussi applicables à `json`. La plupart des fonctions et opérateurs existent dans les deux versions.

[3] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

2.3.2 Type jsonb

- **JSON** au format **B**inaire
- Indexation GIN
- Gestion du langage JSON Path (v12+)

Le type `jsonb` permet de stocker les données dans un format binaire optimisé. Ainsi, il n'est plus nécessaire de désérialiser l'intégralité du document pour accéder à une propriété.

Pour un exemple extrême (document JSON d'une centaine de Mo), voici une comparaison des performances entre les deux types `json` et `jsonb` pour la récupération d'un champ sur 1300 lignes :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_json;

                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on test_json  (cost=0.00..26.38 rows=1310 width=32)
                        (actual time=893.454..912.168 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.021 ms
 Execution time: 912.194 ms

EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_jsonb;

                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on test_jsonb  (cost=0.00..26.38 rows=1310 width=32)
                         (actual time=77.707..84.148 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.026 ms
 Execution time: 84.177 ms
```

2.3.3 JSON : Exemple d'utilisation

```
CREATE TABLE personnes (datas jsonb );

INSERT INTO personnes  (datas) VALUES ('
{
  "firstName": "Jean",
  "lastName": "Valjean",
  "address": {
    "streetAddress": "43 rue du Faubourg Montmartre",
    "city": "Paris",
    "postalCode": "75002"
  },
  "phoneNumbers": [
    { "number": "06 12 34 56 78" },
    {"type": "bureau",
     "number": "07 89 10 11 12"}
  ],
  "children": [],
  "spouse": null
}
'),
('
{
  "firstName": "Georges",
  "lastName": "Durand",
  "address": {
    "streetAddress": "27 rue des Moulins",
    "city": "Châteauneuf",
    "postalCode": "45990"
  },
  "phoneNumbers": [
    { "number": "06 21 34 56 78" },
    {"type": "bureau",
     "number": "07 98 10 11 12"}
  ],
  "children": [],
  "spouse": null
}
');
```

Un champ de type `jsonb` (ou `json`) accepte tout champ JSON directement.

2.3.4 JSON : Affichage de champs

```
SELECT datas->>'firstName' AS prenom,    -- text
       datas->'address'    AS addr       -- jsonb
       FROM personnes ;

SELECT datas #>> '{address,city}' AS villes FROM personnes ; -- text

SELECT datas['address']['city'] as villes from personnes ;  -- jsonb, v14

SELECT datas['address']->>'city' as villes from personnes ;  -- text, v14

SELECT jsonb_array_elements (datas->'phoneNumbers')->>'number' FROM
 ↪  personnes;
```

Le type json dispose de nombreuses fonctions de manipulation et d'extraction. Les opérateurs ->> et -> renvoient respectivement une valeur au format texte, et au format JSON.

```
# SELECT datas->>'firstName' AS prenom,
       datas->'address'      AS addr
       FROM personnes
       \gdesc
```

```
 Column | Type
--------+-------
 prenom | text
 addr   | jsonb
```

```
# \g
```

```
 prenom  |                        addr
---------+-----------------------------------------------------
 Jean    | {                                                  +
         |     "streetAddress": "43 rue du Faubourg Montmartre",+
         |     "city": "Paris",                               +
         |     "state": "",                                   d
         |     "postalCode": "75002"                          +
         |   }
 Georges | {                                                  +
         |     "streetAddress": "27 rue des Moulins",         +
         |     "city": "Châteauneuf",                         +
         |     "postalCode": "45990"                          +
         |   }
```

L'équivalent existe avec des chemins, avec #> et #>> :

```
# SELECT datas #>> '{address,city}' AS villes FROM personnes ;
```

```
   villes
-------------
 Paris
 Châteauneuf
```

Depuis la version 14, une autre syntaxe plus claire est disponible, plus simple, et qui renvoie du JSON :

```
SELECT datas['address']['city'] as villes from personnes ;
```

```
    villes
---------------
 "Paris"
 "Châteauneuf"
```

`jsonb_array_elements` permet de parcourir des tableaux de JSON :

```
# SELECT jsonb_array_elements (datas->'phoneNumbers')->>'number'
         AS numero
  FROM personnes ;
```

```
     numero
----------------
 06 12 34 56 78
 07 89 10 11 12
 06 21 34 56 87
 07 98 10 11 13
```

2.3.5 Conversions jsonb / relationnel

- Construire un objet JSON depuis un ensemble :
  - `json_object_agg()`
- Construire un ensemble de tuples depuis un objet JSON :
  - `jsonb_each()`
  - `jsonb_to_record()`
- Manipuler des tableaux :
  - `jsonb_array_elements()`
  - `jsonb_to_recordset()`

Les fonctions permettant de construire du `jsonb`, ou de le manipuler de manière ensembliste permettent une très forte souplesse. Il est aussi possible de déstructurer des tableaux, mais il est compliqué de requêter sur leur contenu.

Par exemple, si l'on souhaite filtrer des documents de la sorte pour ne ramener que ceux dont une catégorie est `categorie` :

```
{
  "id": 3,
```

```
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
```

```
CREATE TABLE json_table (id serial, document jsonb);
INSERT INTO json_table (document) VALUES ('
{
  "id": 3,
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
');
```

```
SELECT document->'id'
FROM json_table j,
LATERAL jsonb_array_elements(document #> '{sous_document, mon_tableau}')
    AS elements_tableau
WHERE elements_tableau->>'categorie' = 'categorie';
```

Ce type de requête devient rapidement compliqué à écrire, et n'est pas indexable.

2.3.6 JSON : performances

Inconvénients par rapport à un modèle normalisé :

- Perte d'intégrité (types, contraintes, FK…)
- Complexité du code
- Pas de statistiques sur les clés JSON !
- Pas forcément plus léger en disque
    - clés répétées
- Lire 1 champ = lire tout le JSON
    - voire accès table TOAST
- Mise à jour : tout ou rien

Les champs JSON sont très pratiques quand le schéma est peu structuré. Mais la complexité supplémentaire de code nuit à la lisibilité des requêtes. En termes de performances, ils sont coûteux, pour

les raisons que nous allons voir.

Les contraintes d'intégrité sur les types, les tailles, les clés étrangères… ne sont pas disponibles. Les contraintes protègent de nombreux bugs, mais elles sont aussi une aide précieuse pour l'optimiseur.

Chaque JSON récupéré l'est en bloc. Si un seul champ est récupéré, PostgreSQL devra charger tout le JSON et le décomposer. Cela peut même coûter un accès supplémentaire à une table TOAST pour les gros JSON. Rappelons que le mécanisme du TOAST permet à PostgreSQL de compresser à la volée un grand champ texte, binaire, JSON… et/ou de le déporter dans une table annexe interne, le tout étant totalement transparent pour l'utilisateur. Pour les détails, voir cet extrait de la formation DBA2[4].

Il n'y a pas de mise à jour partielle : changer un champ implique de décomposer tout le JSON pour le réécrire entièrement (et parfois en le *détoastant*/*retoastant*).

Un gros point noir est l'absence de statistiques propres aux clés du JSON. Le planificateur va avoir beaucoup de mal à estimer les cardinalités des critères.

Suivant le modèle, il peut y avoir une perte de place, puisque les clés sont répétées entre chaque champ JSON, et non normalisées dans des tables séparées.

Ces inconvénients sont à mettre en balance avec les intérêts du JSON (surtout : éviter des lignes avec trop de champs toujours à NULL, si même on les connaît), les fréquences de lecture et mises à jour des JSON, et les modalités d'utilisation des champs.

Certaines de ces limites peuvent être réduites par les techniques ci-dessous.

2.3.7 jsonb : indexation (1/2)

- Index fonctionnel sur un champ précis
  - bonus : statistiques

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
ANALYZE personnes ;
```

- Champ dénormalisé:
  - champ normal, indexable, facile à utiliser
  - statistiques

```
ALTER TABLE personnes
ADD COLUMN lastname text
GENERATED ALWAYS AS  ((datas->>'lastName')) STORED ;
```

Index fonctionnel :

[4] https://dali.bo/m4_html#mécanisme-toast

L'extraction d'une partie d'un JSON est en fait une fonction immutable, donc indexable. Un index fonctionnel permet d'accéder directement à certaines propriétés, par exemple :

```
CREATE INDEX idx_prs_nom ON personnes ((datas->>'lastName')) ;
```

Mais il ne fonctionnera que s'il y a une clause WHERE avec cette expression exacte. Pour un champ fréquemment utilisé pour des recherches, c'est le plus efficace.

On n'oubliera pas de lancer un ANALYZE pour calculer les statistiques après création de l'index fonctionnel. Même si l'index est peu discriminant, on obtient ainsi de bonnes statistiques sur son critère. Un VACUUM permet aussi les *Index Only Scan* quand c'est possible.

Champ dénormalisé :

Une autre possibilité est de dénormaliser le champ JSON intéressant dans un champ séparé de la table, géré automatiquement, et indexable :

```
ALTER TABLE personnes
ADD COLUMN lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED ;

ANALYZE personnes ;

CREATE INDEX ON personnes (lastname) ;
```

Ce champ coûte un peu d'espace disque supplémentaire, mais il peut être intéressant pour la lisibilité du code, la facilité d'utilisation avec certains outils ou pour certains utilisateurs. Dans le cas des gros JSON, il peut aussi éviter quelques allers-retours vers la table TOAST.

2.3.8 jsonb : indexation (2/2)

- Indexation « schemaless » grâce au GIN :

```
CREATE INDEX idx_prs ON personnes USING gin(datas jsonb_path_ops) ;
```

Index GIN :

Les champs `jsonb` peuvent tirer parti de fonctionnalités avancées de PostgreSQL, notamment les index GIN, et ce via deux classes d'opérateurs.

Même si l'opérateur par défaut GIN pour `jsonb` supporte plus d'opérations, il est souvent suffisant, et surtout plus efficace, de choisir l'opérateur `jsonb_path_ops` (voir les détails[5]) :

[5] https://docs.postgresql.fr/current/datatype-json.html#JSON-INDEXING

```
CREATE INDEX idx_prs ON personnes USING gin (datas jsonb_path_ops) ;
```

`jsonb_path_ops` supporte notamment l'opérateur « contient » (@>) :

```
# EXPLAIN (ANALYZE) SELECT datas->>'firstName' FROM personnes
  WHERE datas @> '{"lastName": "Dupont"}'::jsonb ;

                            QUERY PLAN
------------------------------------------------------------------
Bitmap Heap Scan on personnes  (cost=2.01..3.02 rows=1 width=32)
                      (actual time=0.018..0.019 rows=1 loops=1)
  Recheck Cond: (datas @> '{"lastName": "Dupont"}'::jsonb)
  Heap Blocks: exact=1
  ->  Bitmap Index Scan on idx_prs  (cost=0.00..2.01 rows=1 width=0)
                           (actual time=0.010..0.010 rows=1 loops=1)
        Index Cond: (datas @> '{"lastName": "Dupont"}'::jsonb)
Planning Time: 0.052 ms
Execution Time: 0.104 ms
```

Ce type d'index est moins efficace qu'un index fonctionnel B-tree classique, mais il est idéal quand la clé de recherche n'est pas connue, et que n'importe quel champ du JSON est un critère. De plus il est compressé.

Un index GIN ne permet cependant pas d'*Index Only Scan*.

Surtout, un index GIN ne permet pas de recherches sur des opérateurs B-tree classiques (<, <=, >, >=), ou sur le contenu de tableaux. On est obligé pour cela de revenir au monde relationnel, ou se rabattre sur les index fonctionnels vus plus haut. Il est donc préférable d'utiliser les opérateurs spécifiques, comme « contient » (@>).

2.3.9 SQL/JSON & JSONpath

- SQL:2016 introduit SQL/JSON et le langage JSON Path
- JSON Path :
  - langage de recherche pour JSON
  - concis, flexible, plus rapide
  - inclus dans PostreSQL 12 pour l'essentiel
  - exemple :

```
SELECT jsonb_path_query (datas, '$.phoneNumbers[*] ? (@.type ==
↪  "bureau") ')
  FROM personnes ;
```

JSON path facilite la recherche et le parcours dans les documents JSON complexes. Il évite de parcourir manuellement les nœuds du JSON.

Par exemple, une recherche peut se faire ainsi :

```
SELECT datas->>'firstName' AS prenom
FROM personnes
WHERE  datas  @@ '$.lastName == "Durand"' ;

 prenom
----------
 Georges
```

Mais l'intérêt est d'extraire facilement des parties d'un tableau :

```
SELECT jsonb_path_query (datas, '$.phoneNumbers[*] ? (@.type == "bureau") ')
FROM   personnes ;

               jsonb_path_query
-----------------------------------------------
 {"type": "bureau", "number": "07 89 10 11 12"}
 {"type": "bureau", "number": "07 98 10 11 13"}
```

On trouvera d'autres exemples dans la présentation de Postgres Pro dédié à la fonctionnalité lors la parution de PostgreSQL 12[6], ou dans un billet de Michael Paquier[7].

2.3.10 Extension jsQuery

- Fournit un « langage de requête » sur JSON
  - similaire aux jsonpath (PG 12+), mais dès PG 9.4
- Indexation GIN

```
SELECT document->'id'
FROM json_table j
WHERE j.document @@ 'sous_document.mon_tableau.#.categorie = categorie' ;
```

L'extension jsquery fournit un opérateur @@ (« correspond à la requête jsquery »), similaire à l'opérateur @@ de la recherche plein texte. Celui-ci permet de faire des requêtes évoluées sur un document JSON, optimisable facilement grâce à la méthode d'indexation supportée.

jsquery permet de requêter directement sur des champs imbriqués, en utilisant même des jokers pour certaines parties.

Le langage en lui-même est relativement riche, et fournit un système de *hints* pour pallier à certains problèmes de la collecte de statistiques, qui devrait être améliorée dans le futur.

Il supporte aussi les opérateurs différents de l'égalité :

[6] https://www.postgresql.eu/events/pgconfeu2019/sessions/session/2555/slides/221/jsonpath-pgconfeu-2019.pdf
[7] https://paquier.xyz/postgresql-2/postgres-12-jsonpath/

```
SELECT *
FROM json_table j
WHERE j.document @@ 'ville.population > 10000';
```

Le périmètre est très proche des expressions jsonpath apparues dans PostgreSQL 12, qui, elles, se basent sur le standard SQL:2016. Les auteurs sont d'ailleurs les mêmes. Voir cet article pour les détails[8], ou le dépôt github[9]. La communauté fournit des paquets.

[8] https://habr.com/en/company/postgrespro/blog/500440/
[9] https://github.com/akorotkov/jsquery

2.4 XML

- Type `xml`
  - stocke un document XML
  - valide sa structure
- Quelques fonctions et opérateurs disponibles :
  - `XMLPARSE`, `XMLSERIALIZE`, `query_to_xml`, `xmlagg`
  - `xpath` (XPath 1.0 uniquement)

Le type `xml`, inclus de base, vérifie que le XML inséré est un document « bien formé », ou constitue des fragments de contenu (« content »). L'encodage UTF-8 est impératif. Il y a quelques limitations par rapport aux dernières versions du standard, XPath et XQuery[10]. Le stockage se fait en texte, donc bénéficie du mécanisme de compression TOAST.

Il existe quelques opérateurs et fonctions de validation et de manipulations, décrites dans la documentation du type xml[11] ou celle des fonctions[12]. Par contre, une simple comparaison est impossible et l'indexation est donc impossible directement. Il faudra passer par une expression XPath.

À titre d'exemple : `XMLPARSE` convertit une chaîne en document XML, `XMLSERIALIZE` procède à l'opération inverse.

```
CREATE TABLE liste_cd (catalogue xml) ;
\d liste_cd

                  Table « public.liste_cd »
  Colonne  | Type | Collationnement | NULL-able | Par défaut
-----------+------+-----------------+-----------+------------
 catalogue | xml  |                 |           |

INSERT INTO liste_cd
SELECT XMLPARSE ( DOCUMENT
$$<?xml version="1.0" encoding="UTF-8"?>
<CATALOG>
  <CD>
    <TITLE>The Times They Are a-Changin'</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <YEAR>1964</YEAR>
  </CD>
  <CD>
    <TITLE>Olympia 1961</TITLE>
```

[10] https://docs.postgresql.fr/current/xml-limits-conformance.html
[11] https://docs.postgresql.fr/current/datatype-xml.html
[12] https://docs.postgresql.fr/current/functions-xml.html

```
    <ARTIST>Jacques Brel</ARTIST>
    <COUNTRY>France</COUNTRY>
    <YEAR>1962</YEAR>
  </CD>
</CATALOG> $$ ) ;
--- Noter le $$ pour délimiter une chaîne contenant une apostrophe

SELECT XMLSERIALIZE (DOCUMENT catalogue AS text) FROM liste_cd;

                  xmlserialize
-------------------------------------------------
 <?xml version="1.0" encoding="UTF-8"?>          +
 <CATALOG>                                       +
   <CD>                                          +
     <TITLE>The Times They Are a-Changin'</TITLE>+
     <ARTIST>Bob Dylan</ARTIST>                  +
     <COUNTRY>USA</COUNTRY>                      +
     <YEAR>1964</YEAR>                           +
   </CD>                                         +
   <CD>                                          +
     <TITLE>Olympia 1961</TITLE>                 +
     <ARTIST>Jacques Brel</ARTIST>               +
     <COUNTRY>France</COUNTRY>                   +
     <YEAR>1962</YEAR>                           +
   </CD>                                         +
 </CATALOG>
(1 ligne)
```

Il existe aussi `query_to_xml` pour convertir un résultat de requête en XML, `xmlagg` pour agréger des champs XML, ou `xpath` pour extraire des nœuds suivant une expression XPath 1.0.

NB : l'extension xml2[13] est dépréciée et ne doit pas être utilisée dans les nouveaux projets.

[13] https://docs.postgresql.fr/current/xml2.html

2.5 OBJETS BINAIRES

- Souvent une mauvaise idée…
- 2 méthodes
  - `bytea` : type binaire
  - *Large Objects* : manipulation comme un fichier

PostgreSQL permet de stocker des données au format binaire, potentiellement de n'importe quel type, par exemple des images ou des PDF.

Il faut vraiment se demander si des binaires ont leur place dans une base de données relationnelle. Ils sont généralement beaucoup plus gros que les données classiques. La volumétrie peut donc devenir énorme, et encore plus si les binaires sont modifiés, car le mode de fonctionnement de PostgreSQL aura tendance à les dupliquer. Cela aura un impact sur la fragmentation, la quantité de journaux, la taille des sauvegardes, et toutes les opérations de maintenance. Ce qui est intéressant à conserver dans une base sont des données qu'il faudra rechercher, et l'on recherche rarement au sein d'un gros binaire. En général, l'essentiel des données binaires que l'on voudrait confier à une base peut se contenter d'un stockage classique, PostgreSQL ne contenant qu'un chemin ou une URL vers le fichier réel.

PostgreSQL donne le choix entre deux méthodes pour gérer les données binaires :

- `bytea` : un type comme un autre ;
- *Large Object* : des objets séparés, à gérer indépendamment des tables.

2.5.1 bytea

- Un type comme les autres
  - `bytea` : tableau d'octets
  - en texte : `bytea_output` = `hex` ou `escape`
- Récupération intégralement en mémoire !
- Toute modification entraîne la réécriture complète du `bytea`
- Maxi 1 Go (à éviter)
  - en pratique intéressant pour quelques Mo
- Import :

```
SELECT pg_read_binary_file ('/chemin/fichier');
```

Voici un exemple :

```
CREATE TABLE demo_bytea(a bytea);
INSERT INTO demo_bytea VALUES ('bonjour'::bytea);

SELECT * FROM demo_bytea ;

        a
------------------
 \x626f6e6a6f7572
```

Nous avons inséré la chaîne de caractère « bonjour » dans le champ bytea, en fait sa représentation binaire dans l'encodage courant (UTF-8). Si nous interrogeons la table, nous voyons la représentation textuelle du champ bytea. Elle commence par `\x` pour indiquer un encodage de type `hex`. Ensuite, chaque paire de valeurs hexadécimales représente un octet.

Un second format d'affichage est disponible : `escape` :

```
SET bytea_output = escape ;
SELECT * FROM demo_bytea ;

    a
---------
 bonjour

INSERT INTO demo_bytea VALUES ('journée'::bytea);
SELECT * FROM demo_bytea ;

       a
----------------
 bonjour
 journ\303\251e
```

Le format de sortie `escape` ne protège donc que les valeurs qui ne sont pas représentables en ASCII 7 bits. Ce format peut être plus compact pour des données textuelles essentiellement en alphabet latin sans accent, où le plus gros des caractères n'aura pas besoin d'être protégé.

Cependant, le format `hex` est bien plus efficace à convertir, et est le défaut depuis PostgreSQL 9.0.

Avec les vieilles applications, ou celles restées avec cette configuration, il faudra peut-être forcer `bytea_output` à `escape`, sous peine de corruption.)

Pour charger directement un fichier, on peut notamment utiliser la fonction `pg_read_binary_file`, exécutée par le serveur PostreSQL :

```
INSERT INTO demo_bytea (a)
SELECT pg_read_binary_file ('/chemin/fichier');
```

En théorie, un `bytea` peut contenir 1 Go. En pratique, on se limitera à nettement moins, ne serait-ce que parce `pg_dump` tombe en erreur quand il doit exporter des bytea de plus de 500 Mo environ (le décodage double le nombre d'octets et dépasse cette limite de 1 Go).

La documentation officielle[14] liste les fonctions pour encoder, décoder, extraire, hacher… les `bytea`.

2.5.2 Large Object

- Objet indépendant des tables
- Identifié par un OID
  - à stocker dans les tables
- Suppression manuelle
  - trigger
  - batch (extensions) : `lo_unlink` & `vacuumlo`
- Fonction de manipulation, modification
  - `lo_create`, `lo_import`
  - `lo_seek`, `lo_open`, `lo_read`, `lo_write`…
- Maxi 4 To (à éviter aussi…)

[14] https://docs.postgresql.fr/current/functions-binarystring.html

Un *large object* est un objet totalement décorrélé des tables. (il est stocké en fait dans la table système `pg_largeobject`). Le code doit donc gérer cet objet séparément :

- créer le *large object* et stocker ce qu'on souhaite dedans ;
- stocker la référence à ce *large object* dans une table (avec le type `lob`) ;
- interroger l'objet séparément de la table ;
- le supprimer explicitement quand il n'est plus référencé : il ne disparaîtra pas automatiquement !

Le *large object* nécessite donc un plus gros investissement au niveau du code.

En contrepartie, il a les avantages suivant :

- une taille jusqu'à 4 To, ce qui n'est tout de même pas conseillé ;
- la possibilité d'accéder à une partie directement (par exemple les octets de 152 000 à 153 020), ce qui permet de le transférer par parties sans le charger en mémoire (notamment, le driver JDBC de PostgreSQL fournit une classe LargeObject[15]) ;
- de ne modifier que cette partie sans tout réécrire.

Il y a plusieurs méthodes pour nettoyer les *large objets* devenu inutiles :

- appeler la fonction `lo_unlink` dans le code client — au risque d'oublier ;
- utiliser la fonction trigger `lo_manage` fournie par le module contrib `lo` : (voir documentation[16], si les *large objects* ne sont jamais référencés plus d'une fois ;
- appeler régulièrement le programme `vacuumlo` (là encore un contrib[17]) : il liste tous les *large objects* référencés dans la base, puis supprime les autres. Ce traitement est bien sûr un peu lourd.

Voir la documentation[18] pour les détails.

[15] https://jdbc.postgresql.org/documentation/binary-data/
[16] https://docs.postgresql.fr/current/lo.html
[17] https://docs.postgresql.fr/current/vacuumlo.html
[18] https://docs.postgresql.fr/current/largeobjects.html

2.6 QUIZ

https://dali.bo/s9_quiz

2.7 TRAVAUX PRATIQUES

Les TP sur les types hstore et JSON utilisent la base **cave**. La base **cave** peut être téléchargée depuis https://dali.bo/tp_cave (dump de 2,6 Mo, pour 71 Mo sur le disque au final) et importée ainsi :

```
$ psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
$ psql -c "CREATE DATABASE cave OWNER caviste"
$ pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

2.7.1 Hstore (Optionnel)

But : Découvrir hstore

Pour ce TP, il est fortement conseillé d'aller regarder la documentation officielle du type `hstore` sur https://docs.postgresql.fr/current/hstore.html.

But : Obtenir une version dénormalisée de la table `stock` : elle contiendra une colonne de type `hstore` contenant l'année, l'appellation, la région, le récoltant, le type, et le contenant :

```
vin_id      integer
nombre      integer
attributs   hstore
```

Ce genre de table n'est évidemment pas destiné à une application transactionnelle: on n'aurait aucun moyen de garantir l'intégrité des données de cette colonne. Cette colonne va nous permettre d'écrire une recherche multi-critères efficace sur nos stocks.

Afficher les informations à stocker avec la requête suivante :

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
LIMIT 10;
```

(`LIMIT 10` est là juste pour éviter de ramener tous les enregistrements).

> Créer une table `stock_denorm (vin_id int, nombre int, attributs hstore)` et y copier le contenu de la requête. Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de `hstore` sans se soucier de formatage de chaîne de caractères. (Voir la documentation.)

> Créer un index sur le champ `attributs` pour accélérer les recherches.

> Rechercher le nombre de bouteilles (attribut `bouteille`) en stock de vin blanc (attribut `type_vin`) d'Alsace (attribut `region`). Quel est le temps d'exécution de la requête ? Combien de buffers accédés ?

> Refaire la même requête sur la table initiale. Qu'en conclure ?

2.7.2 jsonb

> **But** : Découvrir JSON

Nous allons créer une table dénormalisée contenant uniquement un champ JSON.

Pour chaque vin, le document JSON aura la structure suivante :

```
{
  vin: {
      recoltant: {
        nom: text,
        adressse: text
      },
      appellation: {
        libelle: text,
        region: text
      },
      type_vin: text
  },
  stocks: [{
    contenant: {
      contenance: real,
      libelle: text
    },
    annee: integer,
    nombre: integer
  }]
}
```

Pour écrire une requête permettant de générer ces documents, nous allons procéder par étapes.

> La requête suivante permet de générer les parties `vin` du document, avec `recoltant` et `appellation`. Créer un document JSON pour chaque ligne de `vin` grâce à la fonction `jsonb_build_object`.

```
SELECT
  recoltant.nom,
  recoltant.adresse,
  appellation.libelle,
  region.libelle,
  type_vin.libelle
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;
```

> Écrire une requête permettant de générer la partie `stocks` du document, soit un document JSON pour chaque ligne de la table `stock`, incluant le `contenant`.

> Fusionner les requêtes précédentes pour générer un document complet pour chaque ligne de la table `vin`. Créer une table `stock_jsonb` avec un unique champ JSONB rassemblant ces documents.

> Calculer la taille de la table, et la comparer à la taille du reste de la base.

> Depuis cette nouvelle table, renvoyer l'ensemble des récoltants de la région Beaujolais.

> Renvoyer l'ensemble des vins pour lesquels au moins une bouteille entre 1992 et 1995 existe. (la difficulté est d'extraire les différents stocks d'une même ligne de la table)

> Indexer le document jsonb en utilisant un index de type GIN.

> Peut-on réécrire les deux requêtes précédentes pour utiliser l'index ?

2.7.3 Large Objects

But : Utilisation de Large Objects

- Créer une table `fichiers` avec un texte et une colonne permettant de référencer des *Large Objects*.

- Importer un fichier local à l'aide de psql dans un large object.
- Noter l'`oid` retourné.

- Importer un fichier du serveur à l'aide de psql dans un large object.

- Afficher le contenu de ces différents fichiers à l'aide de psql.

- Les sauvegarder dans des fichiers locaux.

2.8 TRAVAUX PRATIQUES (SOLUTIONS)

2.8.1 Hstore (Optionnel)

> Afficher les informations à stocker avec la requête suivante :

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
LIMIT 10;
```

(`LIMIT 10` est là juste pour éviter de ramener tous les enregistrements).

> Créer une table `stock_denorm (vin_id int, nombre  int, attributs hstore)` et y copier le contenu de la requête. Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de `hstore` sans se soucier de formatage de chaîne de caractères. (Voir la documentation.)

Une remarque toutefois : les éléments du tableau doivent tous être de même type, d'où la conversion en text des quelques éléments entiers. C'est aussi une limitation du type `hstore` : il ne supporte que les attributs texte.

Cela donne :

```
CREATE EXTENSION hstore;

CREATE TABLE stock_denorm AS
SELECT stock.vin_id,
       stock.nombre,
       hstore(ARRAY['annee', stock.annee::text,
                    'recoltant', recoltant.nom,
                    'appellation',appellation.libelle,
                    'region',region.libelle,
                    'type_vin',type_vin.libelle,
                    'contenance',contenant.contenance::text,
                    'contenant',contenant.libelle]) AS attributs
FROM stock
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
```

```
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id);
```

Et l'on n'oublie pas les statistiques :

```
ANALYZE stock_denorm;
```

> Créer un index sur le champ `attributs` pour accélérer les recherches.

```
CREATE INDEX idx_stock_denorm on stock_denorm USING gin (attributs );
```

> Rechercher le nombre de bouteilles (attribut `bouteille`) en stock de vin blanc (attribut `type_vin`) d'Alsace (attribut `region`). Quel est le temps d'exécution de la requête ? Combien de buffers accédés ?

Attention au A majuscule de Alsace, les hstore sont sensibles à la casse !

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT *
FROM stock_denorm
WHERE attributs @>
'type_vin=>blanc, region=>Alsace, contenant=>bouteille';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on stock_denorm (cost=64.70..374.93 rows=91 width=193)
                                  (actual time=64.370..68.526 rows=1680 loops=1)
   Recheck Cond: (attributs @> '"region"=>"Alsace", "type_vin"=>"blanc",
                               "contenant"=>"bouteille"'::hstore)
   Heap Blocks: exact=1256
   Buffers: shared hit=1353
   ->  Bitmap Index Scan on idx_stock_denorm
            (cost=0.00..64.68 rows=91 width=0)
            (actual time=63.912..63.912 rows=1680 loops=1)
         Index Cond: (attributs @> '"region"=>"Alsace", "type_vin"=>"blanc",
                                    "contenant"=>"bouteille"'::hstore)
         Buffers: shared hit=97
 Planning time: 0.210 ms
 Execution time: 68.927 ms
```

> Refaire la même requête sur la table initiale. Qu'en conclure ?

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
       recoltant.nom AS recoltant,
       appellation.libelle AS appellation,
       region.libelle AS region,
       type_vin.libelle AS type_vin,
       contenant.contenance,
       contenant.libelle as contenant
FROM stock
```

```
JOIN vin ON (stock.vin_id=vin.id)
JOIN recoltant ON (vin.recoltant_id=recoltant.id)
JOIN appellation ON (vin.appellation_id=appellation.id)
JOIN region ON (appellation.region_id=region.id)
JOIN type_vin ON (vin.type_vin_id=type_vin.id)
JOIN contenant ON (stock.contenant_id=contenant.id)
WHERE type_vin.libelle='blanc' AND region.libelle='Alsace'
AND contenant.libelle = 'bouteille';

                                QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=11.64..873.33 rows=531 width=75)
              (actual time=0.416..24.779 rows=1680 loops=1)
   Join Filter: (stock.contenant_id = contenant.id)
   Rows Removed by Join Filter: 3360
   Buffers: shared hit=6292
   ->  Seq Scan on contenant  (cost=0.00..1.04 rows=1 width=16)
                              (actual time=0.014..0.018 rows=1 loops=1)
         Filter: (libelle = 'bouteille'::text)
         Rows Removed by Filter: 2
         Buffers: shared hit=1
   ->  Nested Loop  (cost=11.64..852.38 rows=1593 width=67)
                    (actual time=0.392..22.162 rows=5040 loops=1)
         Buffers: shared hit=6291
         ->  Hash Join  (cost=11.23..138.40 rows=106 width=55)
                        (actual time=0.366..5.717 rows=336 loops=1)
               Hash Cond: (vin.recoltant_id = recoltant.id)
               Buffers: shared hit=43
               ->  Hash Join  (cost=10.07..135.78 rows=106 width=40)
                              (actual time=0.337..5.289 rows=336 loops=1)
                     Hash Cond: (vin.type_vin_id = type_vin.id)
                     Buffers: shared hit=42
                     ->  Hash Join  (cost=9.02..132.48 rows=319 width=39)
                                    (actual time=0.322..4.714 rows=1006 loops=1)
                           Hash Cond: (vin.appellation_id = appellation.id)
                           Buffers: shared hit=41
                           ->  Seq Scan on vin
                                 (cost=0.00..97.53 rows=6053 width=16)
                                 (actual time=0.011..1.384 rows=6053 loops=1)
                               Buffers: shared hit=37
                           ->  Hash  (cost=8.81..8.81 rows=17 width=31)
                                     (actual time=0.299..0.299 rows=53 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 4kB
                                 Buffers: shared hit=4
                                 ->  Hash Join
                                      (cost=1.25..8.81 rows=17 width=31)
                                      (actual time=0.033..0.257 rows=53 loops=1)
                                       Hash Cond:
                                            (appellation.region_id = region.id)
                                       Buffers: shared hit=4
                                       ->  Seq Scan on appellation
                                            (cost=0.00..6.19 rows=319 width=24)
                                            (actual time=0.010..0.074 rows=319
                                             loops=1)
                                             Buffers: shared hit=3
                                       ->  Hash
```

```
                                          (cost=1.24..1.24 rows=1 width=15)
                                          (actual time=0.013..0.013 rows=1
                                           loops=1)
                                        Buckets: 1024  Batches: 1
                                                 Memory Usage: 1kB
                                        Buffers: shared hit=1
                                        ->  Seq Scan on region
                                          (cost=0.00..1.24 rows=1 width=15)
                                          (actual time=0.005..0.012 rows=1
                                                  loops=1)
                                              Filter: (libelle =
                                                       'Alsace'::text)
                                              Rows Removed by Filter: 18
                                              Buffers: shared hit=1
                    ->  Hash  (cost=1.04..1.04 rows=1 width=9)
                              (actual time=0.008..0.008 rows=1 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 1kB
                          Buffers: shared hit=1
                          ->  Seq Scan on type_vin
                                    (cost=0.00..1.04 rows=1 width=9)
                                    (actual time=0.005..0.007 rows=1 loops=1)
                                Filter: (libelle = 'blanc'::text)
                                Rows Removed by Filter: 2
                                Buffers: shared hit=1
              ->  Hash  (cost=1.07..1.07 rows=7 width=23)
                        (actual time=0.017..0.017 rows=7 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 1kB
                    Buffers: shared hit=1
                    ->  Seq Scan on recoltant
                              (cost=0.00..1.07 rows=7 width=23)
                              (actual time=0.004..0.009 rows=7 loops=1)
                          Buffers: shared hit=1
        ->  Index Scan using idx_stock_vin_annee on stock
                    (cost=0.42..6.59 rows=15 width=16)
                    (actual time=0.013..0.038 rows=15 loops=336)
              Index Cond: (vin_id = vin.id)
              Buffers: shared hit=6248
 Planning time: 4.341 ms
 Execution time: 25.232 ms
(53 lignes)
```

La requête sur le schéma normalisé est ici plus rapide. On constate tout de même qu'elle accède à 6300 buffers, contre 1300 à la requête dénormalisée, soit 4 fois plus de données. Un test identique exécuté sur des données hors du cache donne environ 80 ms pour la requête sur la table dénormalisée, contre près d'une seconde pour les tables normalisées. Ce genre de transformation est très utile lorsque le schéma ne se prête pas à une normalisation, et lorsque le volume de données à manipuler est trop important pour tenir en mémoire. Les tables dénormalisées avec hstore se prêtent aussi bien mieux aux recherches multi-critères.

2.8.2 jsonb

Pour chaque vin, le document JSON aura la structure suivante :

```
{
  vin: {
      recoltant: {
        nom: text,
        adressse: text
      },
      appellation: {
        libelle: text,
        region: text
      },
      type_vin: text
  },
  stocks: [{
    contenant: {
      contenance: real,
      libelle: text
    },
    annee: integer,
    nombre: integer
  }]
}
```

> La requête suivante permet de générer les parties `vin` du document, avec `recoltant` et `appellation`. Créer un document JSON pour chaque ligne de `vin` grâce à la fonction `jsonb_build_object`.

```
SELECT
  recoltant.nom,
  recoltant.adresse,
  appellation.libelle,
  region.libelle,
  type_vin.libelle
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;

SELECT
  jsonb_build_object(
    'recoltant',
    json_build_object('nom', recoltant.nom, 'adresse',
                      recoltant.adresse
    ),
    'appellation',
    jsonb_build_object('libelle', appellation.libelle, 'region', region.libelle),
    'type_vin', type_vin.libelle
)
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id ;
```

> Écrire une requête permettant de générer la partie `stocks` du document, soit un document JSON pour chaque ligne de la table `stock`, incluant le `contenant`.

La partie `stocks` du document est un peu plus compliquée, et nécessite l'utilisation de fonctions d'agrégations.

```
SELECT json_build_object(
  'contenant',
  json_build_object('contenance', contenant.contenance, 'libelle',
                    contenant.libelle),
  'annee', stock.annee,
  'nombre', stock.nombre)
FROM stock join contenant on stock.contenant_id = contenant.id;
```

Pour un vin donné, le tableau `stock` ressemble à cela :

```
SELECT json_agg(json_build_object(
  'contenant',
  json_build_object('contenance', contenant.contenance, 'libelle',
                    contenant.libelle),
  'annee', stock.annee,
  'nombre', stock.nombre))
FROM stock
INNER JOIN contenant on stock.contenant_id = contenant.id
WHERE vin_id = 1
GROUP BY vin_id;
```

> Fusionner les requêtes précédentes pour générer un document complet pour chaque ligne de la table `vin`. Créer une table `stock_jsonb` avec un unique champ JSONB rassemblant ces documents.

On assemble les deux parties précédentes :

```
CREATE TABLE stock_jsonb  AS (
  SELECT
    json_build_object(
    'vin',
      json_build_object(
        'recoltant',
        json_build_object('nom', recoltant.nom, 'adresse', recoltant.adresse),
        'appellation',
        json_build_object('libelle', appellation.libelle, 'region',
                          region.libelle),
        'type_vin', type_vin.libelle),
    'stocks',
      json_agg(json_build_object(
        'contenant',
        json_build_object('contenance', contenant.contenance, 'libelle',
                          contenant.libelle),
        'annee', stock.annee,
        'nombre', stock.nombre)))::jsonb as document
  FROM vin
  INNER JOIN recoltant on vin.recoltant_id = recoltant.id
```

```
  INNER JOIN appellation on vin.appellation_id = appellation.id
  INNER JOIN region on region.id = appellation.region_id
  INNER JOIN type_vin on vin.type_vin_id = type_vin.id
  INNER JOIN stock on stock.vin_id = vin.id
  INNER JOIN contenant on stock.contenant_id = contenant.id
  GROUP BY vin_id, recoltant.id, region.id, appellation.id, type_vin.id
);
```

> Calculer la taille de la table, et la comparer à la taille du reste de la base.

La table avec JSON contient toutes les mêmes informations que l'ensemble des tables normalisées de la base cave (à l'exception des `id`). Elle occupe en revanche une place beaucoup moins importante, puisque les documents individuels vont pouvoir être compressés en utilisant le mécanisme TOAST. De plus, on économise les 26 octets par ligne de toutes les autres tables.

Elle est même plus petite que la seule table `stock` :

```
 \d+
                                        Liste des relations
 Schéma |        Nom          |   Type   | Propriétaire | Persistence | Taille | …
--------+---------------------+----------+--------------+-------------+--------+-
…
 public | stock               | table    | caviste      | permanent   | 36 MB  |
 public | stock_jsonb         | table    | postgres     | permanent   | 12 MB  |
…
```

> Depuis cette nouvelle table, renvoyer l'ensemble des récoltants de la région Beaujolais.

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document #>> '{vin, appellation, region}' = 'Beaujolais';
```

> Renvoyer l'ensemble des vins pour lesquels au moins une bouteille entre 1992 et 1995 existe. (la difficulté est d'extraire les différents stocks d'une même ligne de la table)

La fonction `jsonb_array_elements` permet de convertir les différents éléments du document `stocks` en autant de lignes. La clause `LATERAL` permet de l'appeler une fois pour chaque ligne :

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb,
LATERAL jsonb_array_elements(document #> '{stocks}') as stock
WHERE (stock->'annee')::text::integer BETWEEN 1992 AND 1995;
```

> Indexer le document jsonb en utilisant un index de type GIN.

```
CREATE INDEX ON stock_jsonb USING gin (document jsonb_path_ops);
```

> Peut-on réécrire les deux requêtes précédentes pour utiliser l'index ?

Pour la première requête, on peut utiliser l'opérateur « contient » pour passer par l'index :

```
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document @> '{"vin": {"appellation": {"region": "Beaujolais"}}}';
```

La seconde ne peut malheureusement pas être réécrite pour tirer partie de l'index.

La dénormalisation vers un champ externe n'est pas vraiment possible, puisqu'il y a plusieurs `stocks` par ligne.

2.8.3 Large Objects

- Créer une table `fichiers` avec un texte et une colonne permettant de référencer des *Large Objects*.

```
CREATE TABLE fichiers (nom text PRIMARY KEY, data OID);
```

- Importer un fichier local à l'aide de psql dans un large object.
- Noter l'`oid` retourné.

```
psql -c "\lo_import '/etc/passwd'"
```

```
lo_import 6821285
```

```
INSERT INTO fichiers VALUES ('/etc/passwd',6821285) ;
```

- Importer un fichier du serveur à l'aide de psql dans un large object.

```
INSERT INTO fichiers SELECT 'postgresql.conf',
lo_import('/var/lib/pgsql/15/data/postgresql.conf') ;
```

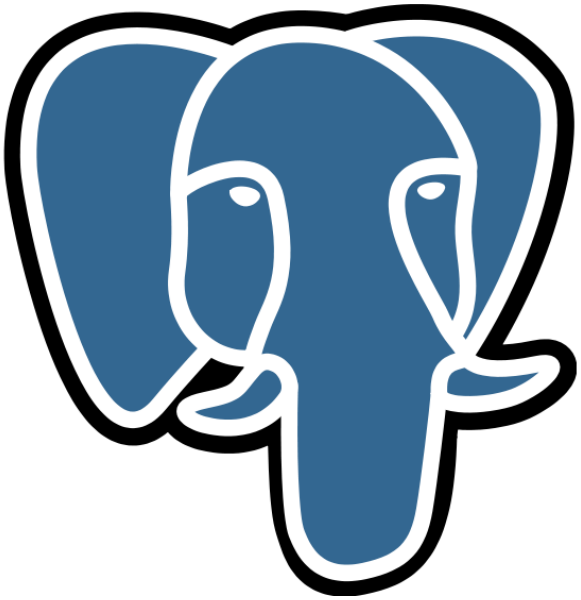
- Afficher le contenu de ces différents fichiers à l'aide de psql.

```
psql -c "SELECT nom,encode(l.data,'escape') \
         FROM fichiers f JOIN pg_largeobject l ON f.data = l.loid;"
```

- Les sauvegarder dans des fichiers locaux.

```
psql -c "\lo_export loid_retourné '/home/dalibo/passwd_serveur';"
```

3/ PL/pgSQL : les bases



3.1 PRÉAMBULE

- Vous apprendrez :
  - à choisir si vous voulez écrire du PL
  - à choisir votre langage PL
  - les principes généraux des langages PL autres que PL/pgSQL
  - les bases de PL/pgSQL

Ce module présente la programmation PL/pgSQL. Il commence par décrire les routines stockées et les différents langages disponibles. Puis il aborde les bases du langage PL/pgSQL, autrement dit :

- comment installer PL/pgSQL dans une base PostgreSQL ;
- comment créer un squelette de fonction ;
- comment déclarer des variables ;
- comment utiliser les instructions de base du langage ;
- comment créer et manipuler des structures ;
- comment passer une valeur de retour de la fonction à l'appelant.

3.1.1 Au menu

- Présentation du PL et des principes
- Présentations de PL/pgSQL et des autres langages PL
- Installation d'un langage PL
- Détails sur PL/pgSQL

3.1.2 Objectifs

- Comprendre les cas d'utilisation d'une routine PL/pgSQL
- Choisir son langage PL en connaissance de cause
- Comprendre la différence entre PL/pgSQL et les autres langages PL
- Écrire une routine simple en PL/pgSQL
    - et même plus complexe

3.2 INTRODUCTION

3.2.1 Qu'est-ce qu'un PL ?

- PL = *Procedural Language*
- 3 langages activés par défaut :
  - C
  - SQL
  - PL/pgSQL

PL est l'acronyme de « Procedural Languages ». En dehors du C et du SQL, tous les langages acceptés par PostgreSQL sont des PL.

Par défaut, trois langages sont installés et activés : C, SQL et PL/pgSQL.

3.2.2 Quels langages PL sont disponibles ?

- Installé par défaut :
  - PL/pgSQL
- Intégrés au projet :
  - PL/Perl
  - PL/Python
  - PL/Tcl
- Extensions tierces :
  - PL/java, PL/R, PL/v8 (Javascript), PL/sh …
  - extensible à volonté

Les quatre langages PL supportés nativement (en plus du C et du SQL bien sûr) sont décrits en détail dans la documentation officielle :

- PL/PgSQL[1] est intégré par défaut dans toute nouvelle base (de par sa présence dans la base

[1] https://docs.postgresql.fr/current/plpgsql.html

modèle **template1**) ;
- PL/Tcl[2] (existe en version *trusted* et *untrusted*) ;
- PL/Perl[3] (existe en version *trusted* et *untrusted*) ;
- PL/Python[4] (uniquement en version *untrusted*).

D'autres langages PL sont accessibles en tant qu'extensions tierces. Les plus stables sont mentionnés dans la documentation[5], comme PL/Java[6] ou PL/R[7]. Ils réclament généralement d'installer les bibliothèques du langage sur le serveur.

Une liste plus large est par ailleurs disponible sur le wiki PostgreSQL[8], Il en ressort qu'au moins 16 langages sont disponibles, dont 10 installables en production. De plus, il est possible d'en ajouter d'autres, comme décrit dans la documentation[9].

3.2.3 Langages *trusted* vs *untrusted*

- *Trusted* = langage de confiance :
  - ne permet que l'accès à la base de données
  - donc pas aux systèmes de fichiers, aux sockets réseaux, etc.
  - SQL, PL/pgSQL, PL/Perl, PL/Tcl
- *Untrusted*:
  - PL/Python, C…
  - PL/TclU, PL/PerlU

Les langages de confiance ne peuvent accéder qu'à la base de données. Ils ne peuvent pas accéder aux autres bases, aux systèmes de fichiers, au réseau, etc. Ils sont donc confinés, ce qui les rend moins facilement utilisables pour compromettre le système. PL/pgSQL est l'exemple typique. Mais de ce fait, ils offrent moins de possibilités que les autres langages.

Seuls les superutilisateurs peuvent créer une routine dans un langage *untrusted*. Par contre, ils peuvent ensuite donner les droits d'exécution à ces routines aux autres rôles dans la base :

```
GRANT EXECUTE ON FUNCTION nom_fonction TO un_role ;
```

[2] https://docs.postgresql.fr/current/pltcl.html
[3] https://docs.postgresql.fr/current/plperl.html
[4] https://docs.postgresql.fr/current/plpython.html
[5] https://docs.postgresql.fr/current/external-pl.html
[6] https://tada.github.io/pljava/
[7] https://github.com/postgres-plr/plr
[8] https://wiki.postgresql.org/wiki/PL_Matrix
[9] https://docs.postgresql.fr/current/plhandler.html

3.2.4 Les langages PL de PostgreSQL

- Les langages PL fournissent :
  - des fonctionnalités procédurales dans un univers relationnel
  - des fonctionnalités avancées du langage PL choisi
  - des performances de traitement souvent supérieures à celles du même code côté client

La question se pose souvent de placer la logique applicative du côté de la base, dans un langage PL, ou des clients. Il peut y avoir de nombreuses raisons en faveur de la première option. Simplifier et centraliser des traitements clients directement dans la base est l'argument le plus fréquent. Par exemple, une insertion complexe dans plusieurs tables, avec mise en place d'identifiants pour liens entre ces tables, peut évidemment être écrite côté client. Il est quelquefois plus pratique de l'écrire sous forme de PL. Les avantages sont :

Centralisation du code :

Si plusieurs applications ont potentiellement besoin d'opérer un même traitement, à fortiori dans des langages différents, porter cette logique dans la base réduit d'autant les risques de *bugs* et facilite la maintenance.

Une règle peut être que tout ce qui a trait à l'intégrité des données devrait être exécuté au niveau de la base.

Performances :

Le code s'exécute localement, directement dans le moteur de la base. Il n'y a donc pas tous les changements de contexte et échanges de messages réseaux dus à l'exécution de nombreux ordres SQL consécutifs. L'impact de la latence due au trafic réseau de la base au client est souvent sous-estimée.

Les langages PL permettent aussi d'accéder à leurs bibliothèques spécifiques (extrêmement nombreuses en python ou perl, entre autres).

Une fonction en PL peut également servir à l'indexation des données. Cela est impossible si elle se calcule sur une autre machine.

Simplicité :

Suivant le besoin, un langage PL peut être bien plus pratique que le langage client.

Il est par exemple très simple d'écrire un traitement d'insertion/mise à jour en PL/pgSQL, le langage étant créé pour simplifier ce genre de traitements, et la gestion des exceptions pouvant s'y produire. Si vous avez besoin de réaliser du traitement de chaîne puissant, ou de la manipulation de fichiers, PL/Perl ou PL/Python seront probablement des options plus intéressantes car plus performantes, là aussi utilisables dans la base.

La grande variété des différents langages PL supportés par PostgreSQL permet normalement d'en trouver un correspondant aux besoins et aux langages déjà maîtrisés dans l'entreprise.

Les langages PL permettent donc de rajouter une couche d'abstraction et d'effectuer des traitements avancés directement en base.

3.2.5 Intérêts de PL/pgSQL en particulier

- Inspiré de l'ADA, proche du Pascal
- Ajout de structures de contrôle au langage SQL
- **Dédié au traitement des données et au SQL**
- Peut effectuer des traitements complexes
- Hérite de tous les types, fonctions et opérateurs définis par les utilisateurs
- *Trusted*
- Facile à utiliser

Le langage étant assez ancien, proche du Pascal et de l'ADA, sa syntaxe ne choquera personne. Elle est d'ailleurs très proche de celle du PLSQL d'Oracle.

Le PL/pgSQL permet d'écrire des requêtes directement dans le code PL sans déclaration préalable, sans appel à des méthodes complexes, ni rien de cette sorte. Le code SQL est mélangé naturellement au code PL, et on a donc un sur-ensemble procédural de SQL.

PL/pgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux rajoutés par l'utilisateur. Il peut les manipuler de façon transparente.

PL/pgSQL est *trusted*. Tous les utilisateurs peuvent donc créer des routines dans ce langage (par défaut). Vous pouvez toujours soit supprimer le langage, soit retirer les droits à un utilisateur sur ce langage (via la commande SQL REVOKE).

PL/pgSQL est donc raisonnablement facile à utiliser : il y a peu de complications, peu de pièges, et il dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

3.2.6 Les autres langages PL ont toujours leur intérêt

- Avantages des autres langages PL par rapport à PL/pgSQL :
  - beaucoup plus de possibilités
  - souvent plus performants pour la résolution de certains problèmes
- Mais :
  - pas spécialisés dans le traitement de requêtes
  - types différents
  - interpréteur séparé

Les langages PL « autres », comme PL/perl[10] et PL/Python (les deux plus utilisés après PL/pgSQL), sont bien plus évolués que PL/PgSQL. Par exemple, ils sont bien plus efficaces en matière de traitement de chaînes de caractères, possèdent des structures avancées comme des tables de hachage, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leur version *untrusted*, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un service web par exemple, ou d'écrire des données dans un fichier externe.

Il existe des langages PL spécialisés. Le plus emblématique d'entre eux est PL/R[11]. R est un langage utilisé par les statisticiens pour manipuler de gros jeux de données. PL/R permet donc d'effectuer ces traitements R directement en base, traitements qui seraient très pénibles à écrire dans d'autres langages, et avec une latence dans le transfert des données.

Il existe aussi un langage qui est, du moins sur le papier, plus rapide que tous les langages cités précédemment : vous pouvez écrire des procédures stockées en C[12], directement. Elles seront compilées à l'extérieur de PostgreSQL, en respectant un certain formalisme, puis seront chargées en indiquant la bibliothèque C qui les contient et leurs paramètres et types de retour.

Mais attention : toute erreur dans le code C est susceptible d'accéder à toute la mémoire visible par le processus PostgreSQL qui l'exécute, et donc de corrompre les données. Il est donc conseillé de ne faire ceci qu'en dernière extrémité.

Le gros défaut est simple et commun à tous ces langages : ils ne sont pas spécialement conçus pour s'exécuter en tant que langage de procédures stockées. Ce que vous utilisez quand vous écrivez du PL/Perl est donc du code Perl, avec quelques fonctions supplémentaires (préfixées par `spi`) pour

[10] https://docs.postgresql.fr/current/plperl-builtins.html
[11] https://github.com/postgres-plr/plr/blob/master/userguide.md
[12] https://docs.postgresql.fr/current/xfunc-c.html

accéder à la base de données ; de même en C. L'accès aux données est assez fastidieux au niveau syntaxique, comparé à PL/pgSQL.

Un autre problème des langages PL (autre que C et PL/pgSQL), est que ces langages n'ont pas les mêmes types natifs que PostgreSQL, et s'exécutent dans un interpréteur relativement séparé. Les performances sont donc moindres que PL/pgSQL et C, pour les traitements dont le plus consommateur est l'accès aux données. Souvent, le temps de traitement dans un de ces langages plus évolués est tout de même meilleur grâce au temps gagné par les autres fonctionnalités (la possibilité d'utiliser un cache, ou une table de hachage par exemple).

3.2.7 Routines / Procédures stockées / Fonctions

- **Procédure** stockée
  - pas de retour
  - contrôle transactionnel : `COMMIT` / `ROLLBACK`
  - PostgreSQL 11 ou +
- **Fonction**
  - peut renvoyer des données (même des lignes)
  - utilisable dans un `SELECT`
  - peut être de type `TRIGGER`, agrégat, fenêtrage
- **Routine**
  - procédure ou fonction

Les programmes écrits à l'aide des langages PL sont habituellement enregistrés sous forme de « routines » :

- procédures ;
- fonctions ;
- fonctions *trigger* ;
- fonctions d'agrégat ;
- fonctions de fenêtrage (*window functions*).

Le code source de ces objets est stocké dans la table `pg_proc` du catalogue.

Les procédures, apparues avec PostgreSQL 11, sont très similaires aux fonctions. Les principales différences entre les deux sont :

- Les fonctions doivent déclarer des arguments en sortie (`RETURNS` ou arguments `OUT`). Elles peuvent renvoyer n'importe quel type de donnée, ou des ensembles de lignes. Il est possible

d'utiliser `void` pour une fonction sans argument de sortie ; c'était d'ailleurs la méthode utilisée pour émuler le comportement d'une procédure avant leur introduction avec PostgreSQL 11. Les procédures n'ont pas de code retour (on peut cependant utiliser des paramètres `OUT` ou `INOUT` /* selon version, voir plus bas */).

- Les procédures offrent le support du contrôle transactionnel, c'est-à-dire la capacité de valider (`COMMIT`) ou annuler (`ROLLBACK`) les modifications effectuées jusqu'à ce point par la procédure. L'intégralité d'une fonction s'effectue dans la transaction appelante.
- Les procédures sont appelées exclusivement par la commande SQL `CALL` ; les fonctions peuvent être appelées dans la plupart des ordres DML/DQL (notamment `SELECT`), mais pas par `CALL`.
- Les fonctions peuvent être déclarées de telle manière qu'elles peuvent être utilisées dans des rôles spécifiques (`TRIGGER`, agrégat ou fonction de fenêtrage).

3.3 INSTALLATION

3.3.1 Installation des binaires nécessaires

- SQL, C et PL/pgSQL
  - compilés et installés par défaut
- Paquets du PGDG pour la plupart des langages :

```
yum|dnf install postgresql14-plperl
apt     install postgresql-plpython3-14
```

- Autres langages :
  - à compiler soi-même

Pour savoir si PL/Perl ou PL/Python a été compilé, on peut demander à `pg_config` :

```
pg_config --configure
'--prefix=/usr/local/pgsql-10_icu' '--enable-thread-safety'
'--with-openssl' '--with-libxml' '--enable-nls' '--with-perl' '--enable-debug'
'ICU_CFLAGS=-I/usr/local/include/unicode/'
'ICU_LIBS=-L/usr/local/lib -licui18n -licuuc -licudata' '--with-icu'
```

Si besoin, les emplacements exacts d'installation des bibliothèques peuvent être récupérés à l'aide des options `--libdir` et `--pkglibdir` de `pg_config`.

Cependant, dans les paquets fournis par le PGDG, il faudra installer explicitement le paquet dédié à `plperl` pour la version majeure de PostgreSQL concernée. Pour PostgreSQL 14, les paquets sont `postgresql14-plperl` (depuis yum.postgresql.org) ou `postgresql-plperl-14` (depuis apt.postgresql.org).

Ainsi, la bibliothèque `plperl.so` que l'on trouvera dans ces répertoires contiendra les fonctions qui permettent l'utilisation du langage PL/Perl. Elle est chargée par le moteur à la première utilisation d'une procédure utilisant ce langage.

De même pour Python 3 (paquets `postgresql14-plpython3` ou `postgresql-plython3-14`).

La plupart des langages intéressants sont disponibles sous forme de paquets. Des versions très récentes, ou des langages plus exotiques, peuvent nécessiter une compilation de l'extension.

3.3.2 Activer/désactiver un langage

- Activer un langage passe par la création de l'extension :

```
CREATE EXTENSION plperl ;
```

- Supprimer l'extension désactive le langage :

```
DROP EXTENSION plperl ;
```

Le langage est activé uniquement dans la base dans laquelle la commande est lancée. S'il faut l'activer sur plusieurs bases, il sera nécessaire d'exécuter cet ordre SQL sur les différentes bases ciblées.

Activer un langage dans la base modèle `template1` l'activera aussi pour toutes les bases créées par la suite.

3.3.3 Langage déjà installé ?

- Interroger le catalogue système `pg_language`
  - ou `\dx` avec `psql`
- Une ligne par langage installé
- *Trusted* ou *untrusted* ?

Voici un exemple d'interrogation de `pg_language` :

```
SELECT lanname, lanpltrusted
FROM pg_language
WHERE lanname='plpgsql';
```

```
 lanname | lanpltrusted
---------+--------------
 plpgsql | t
```

Si un langage est *trusted*, tous les utilisateurs peuvent créer des procédures dans ce langage. Sinon seuls les superutilisateurs le peuvent. Il existe par exemple deux variantes de PL/Perl : PL/Perl et PL/PerlU. La seconde est la variante *untrusted* et est un Perl « complet ». La version *trusted* n'a pas le droit d'ouvrir des fichiers, des sockets, ou autres appels systèmes qui seraient dangereux.

SQL, PL/pgSQL, PL/Tcl, PL/Perl (mais pas PL/Python) sont *trusted*.

C, PL/TclU, PL/PerlU, et PL/PythonU (et les variantes spécifiques aux versions PL/Python2U et PL/Python3U) sont *untrusted*.

Les langages PL sont généralement installés par le biais d'extensions :

```
base=# \dx
                        Liste des extensions installées
     Nom     | Version |   Schéma   |            Description
-------------+---------+------------+---------------------------------------
 plpgsql     | 1.0     | pg_catalog | PL/pgSQL procedural language
```

3.4 EXEMPLES DE FONCTIONS & PROCÉDURES

3.4.1 Fonction PL/pgSQL simple

Une fonction simple en PL/pgSQL :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS '
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2;
  RETURN resultat;
END ' ;
```

```
SELECT addition (1,2);
```

```
 addition
----------
        3
```

3.4.2 Exemple de fonction SQL

Même fonction en SQL pur :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE sql
IMMUTABLE
AS '
    SELECT entier1 + entier2;
' ;
```

- Intérêt : planification !
- Syntaxe allégée possible en v14

Les fonctions simples peuvent être écrites en SQL pur. La syntaxe est plus claire, mais bien plus limitée qu'en PL/pgSQL (ni boucles, ni conditions, ni exceptions notamment).

À partir de PostgreSQL 14, il est possible de se passer des guillemets encadrants, pour les fonctions SQL uniquement. La même fonction devient donc :

```
CREATE OR REPLACE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE sql
IMMUTABLE
RETURN entier1 + entier2 ;
```

Cette nouvelle écriture respecte mieux le standard SQL. Surtout, elle autorise un *parsing* et une vérification des objets impliqués dès la déclaration, et non à l'utilisation. Les dépendances entre fonctions et objets utilisés sont aussi mieux tracées.

L'avantage principal des fonctions en pur SQL est, si elles sont assez simples, leur intégration lors de la réécriture interne de la requête (*inlining*) : elles ne sont donc pas pour l'optimiseur des « boîtes noires ». À l'inverse, l'optimiseur ne sait rien du contenu d'une fonction PL/pgSQL.

Dans l'exemple suivant, la fonction sert de filtre à la requête. Comme elle est en pur SQL, elle permet d'utiliser l'index sur la colonne `date_embauche` de la table `employes_big` :

```
CREATE OR REPLACE function employe_eligible_prime_sql (service int, date_embauche
 ↪  date)
RETURNS boolean
LANGUAGE sql
AS $$
  SELECT ( service !=3 AND date_embauche < '2003-01-01')  ; -- ancien employé, sauf
   ↪  un service
$$ ;

EXPLAIN (ANALYZE) SELECT matricule, num_service, nom, prenom
FROM    employes_big
WHERE   employe_eligible_prime_sql (num_service, date_embauche) ;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
       (cost=0.42..1.54 rows=1 width=22) (actual time=0.008..0.009 rows=1 loops=1)
   Index Cond: (date_embauche < '2003-01-01'::date)
   Filter: (num_service <> 3)
   Rows Removed by Filter: 1
 Planning Time: 0.102 ms
 Execution Time: 0.029 ms
```

Avec une version de la même fonction en PL/pgSQL, le planificateur ne voit pas le critère indexé. Il n'a pas d'autre choix que de lire toute la table et d'appeler la fonction pour chaque ligne, ce qui est bien sûr plus lent :

```
CREATE OR REPLACE function employe_eligible_prime_pl (service int, date_embauche
 ↪  date)
RETURNS boolean
LANGUAGE plpgsql AS $$
BEGIN
 RETURN ( service !=3 AND date_embauche < '2003-01-01') ;
END ;
$$ ;

EXPLAIN (ANALYZE) SELECT matricule, num_service, nom, prenom
FROM employes_big
WHERE  employe_eligible_prime_pl (num_service, date_embauche) ;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on employes_big   (cost=0.00..134407.90 rows=166338 width=22)
                            (actual time=0.069..269.121 rows=1 loops=1)
   Filter: employe_eligible_prime_pl(num_service, date_embauche)
   Rows Removed by Filter: 499014
 Planning Time: 0.038 ms
 Execution Time: 269.157 ms
```

Le wiki[13] décrit les conditions pour que l'*inlining* des fonctions SQL fonctionne : obligation d'un seul SELECT, interdiction de certains fonctionnalités…

3.4.3 Exemple de fonction PL/pgSQL utilisant la base

```
CREATE OR REPLACE FUNCTION nb_lignes_table (sch text, tbl text)
 RETURNS bigint
 STABLE
AS '
DECLARE     n bigint ;
BEGIN
    SELECT n_live_tup
    INTO n
    FROM pg_stat_user_tables
    WHERE schemaname = sch AND relname = tbl ;
    RETURN n ;
END ; '
LANGUAGE plpgsql ;
```

Dans cet exemple, on récupère l'estimation du nombre de lignes actives d'une table passée en paramètres.

L'intérêt majeur du PL/pgSQL et du SQL sur les autres langages est la facilité d'accès aux données. Ici, un simple `SELECT <champ> INTO <variable>` suffit à récupérer une valeur depuis une table dans une variable.

```
SELECT nb_lignes_table ('public', 'pgbench_accounts');

 nb_lignes_table
-----------------
        10000000
```

[13] https://wiki.postgresql.org/wiki/Inlining_of_SQL_functions

3.4.4 Exemple de fonction PL/Perl complexe

- Permet d'insérer une facture associée à un client
- Si le client n'existe pas, une entrée est créée
- Utilisation fréquente de `spi_exec`

Voici l'exemple de la fonction :

```
CREATE OR REPLACE FUNCTION
    public.demo_insert_perl(nom_client text, titre_facture text)
 RETURNS integer
 LANGUAGE plperl
 STRICT
AS $function$
  use strict;
  my ($nom_client, $titre_facture)=@_;
  my $rv;
  my $id_facture;
  my $id_client;

  # Le client existe t'il ?
  $rv = spi_exec_query('SELECT id_client FROM mes_clients WHERE nom_client = '
    . quote_literal($nom_client)
  );
  # Sinon on le crée :
  if ($rv->{processed} == 0)
  {
    $rv = spi_exec_query('INSERT INTO mes_clients (nom_client) VALUES ('
        . quote_literal($nom_client) . ') RETURNING id_client'
    );
  }
  # Dans les deux cas, l'id client est dans $rv :
  $id_client=$rv->{rows}[0]->{id_client};

  # Insérons maintenant la facture
  $rv = spi_exec_query(
    'INSERT INTO mes_factures (titre_facture, id_client) VALUES ('
    . quote_literal($titre_facture) . ", $id_client ) RETURNING id_facture"
  );

  $id_facture = $rv->{rows}[0]->{id_facture};

  return $id_facture;
$function$ ;
```

Cette fonction n'est pas parfaite, elle ne protège pas de tout. Il est tout à fait possible d'avoir une insertion concurrente entre le `SELECT` et le `INSERT` par exemple.

Il est clair que l'accès aux données est malaisé en PL/Perl, comme dans la plupart des langages,

puisqu'ils ne sont pas prévus spécifiquement pour cette tâche. Par contre, on dispose de toute la puissance de Perl pour les traitements de chaîne, les appels système…

PL/Perl, c'est :

- Perl, moins les fonctions pouvant accéder à autre chose qu'à PostgreSQL (il faut utiliser PL/PerlU pour passer outre cette limitation) ;
- un bloc de code anonyme appelé par PostgreSQL ;
- des fonctions d'accès à la base, `spi_*`

3.4.5 Exemple de fonction PL/pgSQL complexe

- Même fonction en PL/pgSQL que précédemment
- L'accès aux données est simple et naturel
- Les types de données SQL sont natifs
- La capacité de traitement est limitée par le langage
- **Attention** au nommage des variables et paramètres

Pour éviter les conflits avec les objets de la base, il est conseillé de préfixer les variables.

```
CREATE OR REPLACE FUNCTION
public.demo_insert_plpgsql(p_nom_client text, p_titre_facture text)
 RETURNS integer
 LANGUAGE plpgsql
 STRICT
AS $function$
DECLARE
  v_id_facture int;
  v_id_client int;
BEGIN
  -- Le client existe t'il ?
  SELECT id_client
  INTO v_id_client
  FROM mes_clients
  WHERE nom_client = p_nom_client;

  -- Sinon on le crée :
  IF NOT FOUND THEN
    INSERT INTO mes_clients (nom_client)
    VALUES (p_nom_client)
    RETURNING id_client INTO v_id_client;
  END IF;

  -- Dans les deux cas, l'id client est maintenant dans v_id_client

  -- Insérons maintenant la facture
  INSERT INTO mes_factures (titre_facture, id_client)
```

```
  VALUES (p_titre_facture, v_id_client)
  RETURNING id_facture INTO v_id_facture;

  return v_id_facture;
END;
$function$ ;
```

3.4.6 Exemple de procédure

```
CREATE OR REPLACE PROCEDURE vide_tables (dry_run BOOLEAN)
AS '
BEGIN
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
    IF dry_run THEN
        ROLLBACK ;
    END IF ;
END ;
 ' LANGUAGE plpgsql ;
```

Cette procédure tronque des tables de la base d'exemple **pgbench**, et annule si `dry_run` est vrai.

Les procédures sont récentes dans PostgreSQL (à partir de la version 11). Elles sont à utiliser quand on n'attend pas de résultat en retour. Surtout, elles permettent de gérer les transactions (`COMMIT`, `ROLLBACK`), ce qui ne peut se faire dans des fonctions, même si celles-ci peuvent modifier les données.

Une procédure ne peut utiliser le contrôle transactionnel que si elle est appelée en dehors de toute transaction.

Comme pour les fonctions, il est possible d'utiliser le SQL pur dans les cas les plus simples, sans contrôle transactionnel notamment :

```
CREATE OR REPLACE PROCEDURE vide_tables ()
AS '
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
' LANGUAGE sql;
```

Toujours pour les procédures en SQL, il existe une variante sans guillemets, à partir de PostgreSQL 14, mais qui ne supporte pas tous les ordres. Comme pour les fonctions, l'intérêt est la prise en compte des dépendances entre objets et procédures.

```
CREATE OR REPLACE PROCEDURE vide_tables ()
BEGIN ATOMIC
    DELETE FROM pgbench_history ;
    DELETE FROM pgbench_accounts ;
    DELETE FROM pgbench_tellers ;
    DELETE FROM pgbench_branches ;
END ;
```

3.4.7 Exemple de bloc anonyme en PL/pgSQL

- Bloc procédural anonyme en PL/pgSQL :

```
DO $$
DECLARE r record;
BEGIN
    FOR r IN (SELECT schemaname, relname
              FROM pg_stat_user_tables
              WHERE coalesce(last_analyze, last_autoanalyze) IS NULL
              ) LOOP
        RAISE NOTICE 'Analyze %.%', r.schemaname, r.relname ;
        EXECUTE 'ANALYZE ' || quote_ident(r.schemaname)
                           || '.' || quote_ident(r.relname) ;
    END LOOP;
END$$;
```

Les blocs anonymes sont utiles pour des petits scripts ponctuels qui nécessitent des boucles ou du conditionnel, voire du transactionnel, sans avoir à créer une fonction ou une procédure. Ils ne renvoient rien. Ils sont habituellement en PL/pgSQL mais tout langage procédural installé est possible.

L'exemple ci-dessus lance un ANALYZE sur toutes les tables où les statistiques n'ont pas été calculées d'après la vue système, et donne aussi un exemple de SQL dynamique. Le résultat est par exemple :

```
NOTICE:  Analyze public.pgbench_history
NOTICE:  Analyze public.pgbench_tellers
NOTICE:  Analyze public.pgbench_accounts
NOTICE:  Analyze public.pgbench_branches
DO
Temps : 141,208 ms
```

(Pour ce genre de SQL dynamique, si l'on est sous `psql` , il est souvent plus pratique d'utiliser `\gexec`[14].)

[14] https://docs.postgresql.fr/current/app-psql.html#R2-APP-PSQL-4

Noter que les ordres constituent une transaction unique, à moins de rajouter des `COMMIT` ou `ROLLBACK` explicitement (ce n'est autorisé qu'à partir de la version 11).

3.5 UTILISER UNE FONCTION OU UNE PROCÉDURE

3.5.1 Invocation d'une fonction ou procédure

- Appeler une procédure : ordre spécifique CALL

```
CALL ma_procedure('arg1');
```

- Appeler une fonction : dans une requête

```
SELECT ma_fonction('arg1', 'arg2') ;

SELECT * FROM ma_fonction('arg1', 'arg2') ;

INSERT INTO matable
SELECT ma_fonction( champ1, champ2 )   FROM ma_table2 ;

CALL ma_procedure( mafonction() );

CREATE INDEX ON ma_table ( ma_fonction(ma_colonne) );
```

Demander l'exécution d'une procédure se fait en utilisant un ordre SQL spécifique : CALL[15]. Il suffit de fournir les paramètres. Il n'y a pas de code retour.

Les fonctions ne sont quant à elles pas directement compatibles avec la commande CALL, il faut les invoquer dans le contexte d'une commande SQL. Elles sont le plus couramment appelées depuis des commandes de type DML (SELECT, INSERT, etc.), mais on peut aussi les trouver dans d'autres commandes.

Voici quelques exemples :

- dans un SELECT (la fonction ne doit renvoyer qu'une seule ligne) :

```
SELECT ma_fonction('arg1', 'arg2');
```

- dans un SELECT, en passant en argument les valeurs d'une colonne d'une table :

```
SELECT ma_fonction(ma_colonne) FROM ma_table;
```

- dans le FROM d'un SELECT, la fonction renvoit ici généralement plusieurs lignes (SETOF), et un résultat de type RECORD :

```
SELECT result FROM ma_fonction() AS f(result);
```

- dans un INSERT pour générer la valeur à insérer :

[15] https://docs.postgresql.fr/current/sql-call.html

```
INSERT INTO ma_table(ma_colonne) VALUES ( ma_fonction() );
```

- dans une création d'index (index fonctionnel, la fonction sera réellement appelée lors des mises à jour de l'index… attention la fonction doit être déclarée « immutable ») :

```
CREATE INDEX ON ma_table ( ma_fonction(ma_colonne) );
```

- appel d'une fonction en paramètre d'une autre fonction ou d'une procédure, par exemple ici le résultat de la fonction `ma_fonction()` (qui doit renvoyer une seule ligne) est passé en argument d'entrée de la procédure `ma_procedure()` :

```
CALL ma_procedure( ma_fonction() );
```

Par ailleurs, certaines fonctions sont spécialisées et ne peuvent être invoquées que dans le contexte pour lequel elles ont été conçues (fonctions trigger, d'agrégat, de fenêtrage, etc.).

3.6 CRÉATION ET MAINTENANCE DES FONCTIONS ET PROCÉDURES

3.6.1 Création

- CREATE FUNCTION
- CREATE PROCEDURE

Voici la syntaxe complète pour une fonction d'après la documentation[16] :

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, …] ] )
     [ RETURNS rettype
       | RETURNS TABLE ( column_name column_type [, …] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, … ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } …
```

Voici la syntaxe complète pour une procédure d'après la documentation[17] :

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, …] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, … ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } …
```

Nous allons décrire les clauses importantes ci-dessous.

[16] https://www.postgresql.org/docs/current/sql-createfunction.html
[17] https://www.postgresql.org/docs/current/sql-createprocedure.html

3.6.2 Langage

- Le langage de la routine doit être précisé
  `LANGUAGE nomlang`
- Nous étudierons SQL et `plpgsql`
- Aussi : `plpython3u`, `plperl`, `pl/R`...

Il n'y a pas de langage par défaut. Il est donc nécessaire de le spécifier à chaque création d'une routine.

Ici ce sera surtout : `LANGUAGE plpgsql`.

Une routine en pur SQL indiquera `LANGUAGE sql`. On rencontrera aussi `plperl`, `plpython3u`, etc. en fonction des besoins.

3.6.3 Structure d'une routine PL/pgSQL

- Reprenons le code montré plus haut :

```
CREATE FUNCTION addition(entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS '
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2 ;
  RETURN resultat ;
END';
```

Le langage PL/pgSQL n'est pas sensible à la casse, tout comme SQL (sauf les noms des objets ou variables, si vous les mettez entre des guillemets doubles).

L'opérateur de comparaison est =, l'opérateur d'affectation `:=`

3.6.4 Structure d'une routine PL/pgSQL (suite)

- DECLARE
  - déclaration des variables locales
- BEGIN
  - début du code de la routine
- END
  - la fin
- Instructions séparées par des points-virgules
- Commentaires commençant par `--` ou compris entre `/*` et `*/`

Une routine est composée d'un bloc de déclaration des variables locales et d'un bloc de code. Le bloc de déclaration commence par le mot clé DECLARE et se termine avec le mot clé BEGIN. Ce mot clé est celui qui débute le bloc de code. La fin est indiquée par le mot clé END.

Toutes les instructions se terminent avec des points-virgules. Attention, DECLARE, BEGIN et END ne sont pas des instructions.

Il est possible d'ajouter des commentaires. `--` indique le début d'un commentaire qui se terminera en fin de ligne. Pour être plus précis dans la délimitation, il est aussi possible d'utiliser la notation C : `/*` est le début d'un commentaire et `*/` la fin.

3.6.5 Blocs nommés

- Labels de bloc possibles
- Plusieurs blocs d'exception possibles dans une routine
- Permet de préfixer des variables avec le label du bloc
- De donner un label à une boucle itérative
- Et de préciser de quelle boucle on veut sortir, quand plusieurs d'entre elles sont imbriquées

Indiquer le nom d'un label ainsi :

```
<<mon_label>>
-- le code (blocs DECLARE, BEGIN-END, et EXCEPTION)
```

ou bien (pour une boucle)

```
[ <<mon_label>> ]
LOOP
    ordres …
END LOOP [ mon_label ];
```

Bien sûr, il est aussi possible d'utiliser des labels pour des boucles `FOR`, `WHILE`, `FOREACH`.

On sort d'un bloc ou d'une boucle avec la commande `EXIT`, on peut aussi utiliser `CONTINUE` pour passer à l'exécution suivante d'une boucle sans terminer l'itération courante.

Par exemple :

```
EXIT [mon_label] WHEN compteur > 1;
```

3.6.6 Modification du code d'une routine

- `CREATE OR REPLACE FUNCTION`
- `CREATE OR REPLACE PROCEDURE`
- Une routine est définie par son nom et ses arguments
- Si type de retour différent, la fonction doit d'abord être supprimée puis recréée

Une routine est surchargeable. La seule façon de les différencier est de prendre en compte les arguments (nombre et type). Les noms des arguments peuvent être indiqués mais ils seront ignorés.

Deux routines identiques aux arguments près (on parle de prototype) ne sont pas identiques, mais bien deux routines distinctes.

`CREATE OR REPLACE` a principalement pour but de modifier le code d'une routine, mais il est aussi possible de modifier les méta-données.

3.6.7 Modification des méta-données d'une routine

- `ALTER FUNCTION` / `ALTER PROCEDURE`
- Une routine est définie par son nom et ses arguments
- Permet de modifier nom, propriétaire, schéma et autres options

Toutes les méta-données discutées plus haut sont modifiables avec un `ALTER`.

3.6.8 Suppression d'une routine

- Une routine est définie par son nom et ses arguments :

```
DROP FUNCTION addition (integer, integer) ;

DROP PROCEDURE public.vide_tables (boolean);
DROP PROCEDURE public.vide_tables ();
```

La suppression se fait avec l'ordre DROP.

Une fonction pouvant exister en plusieurs exemplaires, avec le même nom et des arguments de type différents, il faudra parfois parfois préciser ces derniers.

3.6.9 Utilisation des guillemets

- Les guillemets deviennent très rapidement pénibles
  - préférer $$
  - ou $fonction$, $toto$...

Définir une fonction entre guillemets simples (') devient très pénible dès que la fonction doit en contenir parce qu'elle contient elle-même des chaînes de caractères. PostgreSQL permet de remplacer les guillemets par $$, ou tout mot encadré de $.

Par exemple, on peut reprendre la syntaxe de déclaration de la fonction `addition()` précédente en utilisant cette méthode :

```
CREATE FUNCTION addition(entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS $ma_fonction_addition$
DECLARE
  resultat integer;
BEGIN
  resultat := entier1 + entier2;
  RETURN resultat;
END
$ma_fonction_addition$;
```

Ce peut être utile aussi dans tout code réalisant une concaténation de chaînes de caractères contenant des guillemets. La syntaxe traditionnelle impose de les multiplier pour les protéger, et le code devient difficile à lire. :

```
requete := requete || '' AND vin LIKE ''''bordeaux%'''' AND xyz ''
```

En voilà une simplification grâce aux dollars :

```
requete := requete || $sql$ AND vin LIKE 'bordeaux%' AND xyz $sql$
```

Si vous avez besoin de mettre entre guillemets du texte qui inclut $$, vous pouvez utiliser Q, et ainsi de suite. Le plus simple étant de définir un marqueur de fin de routine plus complexe, par exemple incluant le nom de la fonction.

3.7 PARAMÈTRES ET RETOUR DES FONCTIONS ET PROCÉDURES

3.7.1 Version minimaliste

```
CREATE FUNCTION fonction (entier integer, texte text)
RETURNS int  AS …
```

Ceci une forme de fonction très simple (et très courante) : deux paramètres en entrée (implicitement en entrée seulement), et une valeur en retour.

Dans le corps de la fonction, il est aussi possible d'utiliser une notation numérotée au lieu des noms de paramètre : le premier argument a pour nom `$1`, le deuxième `$2`, etc. C'est à éviter.

Tous les types sont utilisables, y compris les types définis par l'utilisateur. En dehors des types natifs de PostgreSQL, PL/pgSQL ajoute des types de paramètres spécifiques pour faciliter l'écriture des routines.

3.7.2 Paramètres IN, OUT & retour

```
CREATE FUNCTION cree_utilisateur (nom text, type_id int DEFAULT 0)
 RETURNS id_utilisateur int AS …

CREATE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int, OUT
 ↪  annee int)
 AS …
```

- `IN` / `OUT` / `INOUT` : entrée/sortie/les 2
- `VARIADIC` : nombre variable
- nom (libre et optionnel)
- type (parmi tous les types de base et les types utilisateur)
- `DEFAULT` : valeur par défaut

Si le mode d'un argument est omis, `IN` est la valeur implicite : la valeur en entrée ne sera pas modifiée.

Un paramètre `OUT` sera modifié. S'il s'agit d'une variable d'un bloc PL appelant, sa valeur sera modifiée. Un paramètre `INOUT` est un paramètre en entrée mais sera également modifié.

Dans le corps d'une fonction, `RETURN` est inutile avec des paramètres `OUT` parce que c'est la valeur des paramètres `OUT` à la fin de la fonction qui est retournée, comme dans l'exemple plus bas.

L'option VARIADIC permet de définir une fonction avec un nombre d'arguments libres à condition de respecter le type de l'argument (comme `printf` en C par exemple). Seul un argument OUT peut suivre un argument VARIADIC : l'argument VARIADIC doit être le dernier de la liste des paramètres en entrée puisque tous les paramètres en entrée suivant seront considérées comme faisant partie du tableau variadic. Seuls les arguments IN et VARIADIC sont utilisables avec une fonction déclarée comme renvoyant une table (clause RETURNS TABLE, voir plus loin).

Jusque PostgreSQL 13 inclus, les procédures ne supportent pas les arguments OUT, seulement IN et INOUT.

La clause DEFAULT permet de rendre les paramètres optionnels. Après le premier paramètre ayant une valeur par défaut, tous les paramètres qui suivent doivent avoir une valeur par défaut. Pour rendre le paramètre optionnel, il doit être le dernier argument ou alors les paramètres suivants doivent aussi avoir une valeur par défaut.

3.7.3 Type en retour : 1 valeur simple

- Fonctions uniquement

```
RETURNS type      -- int, text, etc
```

- Tous les types de base & utilisateur
- Rien : `void`

Le type de retour (clause RETURNS dans l'entête) est obligatoire pour les fonctions et interdit pour les procédures.

Avant la version 11, il n'était pas possible de créer une procédure, mais il était possible de créer une fonction se comportant globalement comme une procédure en utilisant le type de retour `void`.

Des exemples plus haut utilisent des types simples, mais tous ceux de PostgreSQL ou les types créés par l'utilisateur sont utilisables.

Depuis le corps de la fonction, le résultat est renvoyé par un appel à RETURN (PL/pgSQL) ou SELECT (SQL).

3.7.4 Type en retour : 1 lignes, plusieurs champs

3 options :

- Type composé dédié

```
CREATE TYPE ma_structure AS ( … ) ;
CREATE FUNCTION …  RETURNS ma_structure ;
```

- Paramètres OUT

```
CREATE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int, OUT
 ↪  annee int) AS …
```

- RETURNS TABLE

```
CREATE FUNCTION explose_date_table (d date)
RETURNS TABLE (jour integer, mois integer, annee integer) AS…
```

S'il y a besoin de renvoyer plusieurs valeurs à la fois, une possibilité est de renvoyer un type composé défini auparavant.

Une alternative courante est d'utiliser plusieurs paramètres OUT (et pas de clause RETURN dans l'entête) pour obtenir un enregistrement composite :

```
CREATE OR REPLACE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int, OUT
 ↪  annee int)
AS $$
SELECT extract (day FROM d)::int, extract(month FROM d)::int, extract (year FROM
 ↪  d)::int
$$
LANGUAGE SQL;

SELECT * FROM explose_date ('31-12-2020');

 jour | mois | annee
------+------+-------
   31 |    0 |  2020
```

(Noter que l'exemple ci-dessus est en simple SQL.)

La clause TABLE est une autre alternative, sans doute plus claire. Cet exemple devient alors, toujours en pur SQL :

```
CREATE OR REPLACE FUNCTION explose_date_table (d date)
RETURNS TABLE (jour integer, mois integer, annee integer)
LANGUAGE sql
AS $$
  SELECT  extract (day FROM d)::int, extract(month FROM d)::int, extract (year FROM
   ↪  d)::int ;
$$ ;
```

3.7.5 Retour multi-lignes

- 1 seul champ ou plusieurs ?

```
RETURNS SETOF type   -- int, text, type personnalisé

RETURNS TABLE ( col1 type, col2 type … )
```

- Ligne à ligne ou en bloc ?

```
RETURN NEXT …

RETURN QUERY   SELECT …
RETURN QUERY   EXECUTE …
```

Pour renvoyer plusieurs lignes, la première possibilité est de déclarer un type de retour SETOF. Cet exemple utilise RETURN NEXT pour renvoyer les lignes une à une :

```
CREATE OR REPLACE FUNCTION liste_entiers_setof (limite int)
 RETURNS SETOF integer
 LANGUAGE plpgsql
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    RETURN NEXT i;
  END LOOP;
END
$$ ;

SELECT * FROM liste_entiers_setof (3) ;
 liste_entiers_setof
---------------------
                   1
                   2
                   3
(3 lignes)
```

S'il y plusieurs champs à renvoyer, une possibilité est d'utiliser un type dédié (composé), qu'il faudra cependant créer auparavant. L'exemple suivant utilise aussi un RETURN QUERY pour éviter d'itérer sur toutes les lignes du résultat :

```
CREATE TYPE pgt AS (schemaname text, tablename text) ;

CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
RETURNS SETOF pgt
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
```

```
                FROM pg_tables WHERE tableowner=p_owner
                ORDER BY tablename ;
END ; $$ ;

SELECT * FROM tables_by_owner ('pgbench');

 schemaname |    tablename
------------+------------------
 public     | pgbench_accounts
 public     | pgbench_branches
 public     | pgbench_history
 public     | pgbench_tellers
(4 lignes)
```

On a vu que la clause TABLE permet de renvoyer plusieurs champs. Or, elle implique aussi SETOF, et les deux exemples ci-dessus peuvent devenir :

```
CREATE OR REPLACE FUNCTION liste_entiers_table (limite int)
RETURNS TABLE (j int)
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    j = i ;
    RETURN NEXT ;  -- renvoie la valeur de j en cours
  END LOOP;
END $$ LANGUAGE plpgsql;

SELECT * FROM liste_entiers_table (3) ;
 j
 1
 2
 3
(3 lignes)
```

(Noter ici que le nom du champ retourné dépend du nom de la variable utilisée, et n'est pas forcément le nom de la fonction. En effet, chaque appel à RETURN NEXT retourne un enregistrement composé d'une copie de toutes les variables, au moment de l'appel à RETURN NEXT.)

```
CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
RETURNS TABLE (schemaname text, tablename text)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
                FROM pg_tables WHERE tableowner=p_owner
                ORDER BY tablename ;
END ; $$ ;
```

La variante RETURN QUERY EXECUTE … est destinée à des requêtes en SQL dynamique.

Les fonctions avec RETURN QUERY ou RETURN NEXT stockent tout le résultat avant de le retourner en bloc. Le paramètre work_mem permet de définir la mémoire utilisée avant l'utilisation d'un fichier temporaire, qui a bien sûr un impact sur les performances.

Si RETURNS TABLE est peut-être le plus souple et clair, le choix entre toutes ces méthodes est affaire de goût, ou de compatibilité avec du code ancien ou converti d'un produit concurrent.

Quand plusieurs lignes sont renvoyées, tout est conservé en mémoire jusqu'à la fin de la fonction. Donc, si beaucoup de données sont renvoyées, cela poser des problèmes de latence, voire de mémoire.

En général, l'appel se fait ainsi pour obtenir des lignes :

```
SELECT * FROM ma_fonction();
```

Une alternative est d'utiliser :

```
SELECT ma_fonction();
```

pour récupérer un résultat d'une seule colonne, scalaire, type composite ou RECORD suivant la fonction.

Cette différence concerne aussi les fonctions système :

```
# SELECT * FROM pg_control_system () ;

 pg_control_version | catalog_version_no |  system_identifier  |
 ↪  pg_control_last_modified
--------------------+--------------------+---------------------+--------------------
 ↪  ------
               1201 |          201909212 | 6744959735975969621 | 2021-09-17 18:24:05+02
(1 ligne)

# SELECT pg_control_system () ;

                      pg_control_system
---------------------------------------------------------------
 (1201,201909212,6744959735975969621,"2021-09-17 18:24:05+02")
 (1 ligne)
```

3.7.6 Gestion des valeurs NULL

- Comment gérer les paramètres à NULL ?
- STRICT :
  - 1 paramètre NULL : retourne NULL immédiatement
- Défaut :
  - gestion par la fonction

Si une fonction est définie comme STRICT et qu'un des arguments d'entrée est NULL, PostgreSQL n'exécute même pas la fonction et utilise NULL comme résultat.

Dans la logique relationnelle, `NULL` signifie « la valeur est inconnue ». La plupart du temps, il est logique qu'une fonction ayant un paramètre à une valeur inconnue retourne aussi une valeur inconnue, ce qui fait que cette optimisation est très souvent pertinente.

On gagne à la fois en temps d'exécution, mais aussi en simplicité du code (il n'y a pas à gérer les cas `NULL` pour une fonction dans laquelle `NULL` ne doit jamais être injecté).

Dans la définition d'une fonction, les options sont `STRICT` ou son synonyme `RETURNS NULL ON NULL INPUT`, ou le défaut implicite `CALLED ON NULL INPUT`.

3.8 VARIABLES EN PL/PGSQL

3.8.1 Clause DECLARE

- Dans le source, partie DECLARE :

```
DECLARE
  i  integer;
  j  integer := 5;
  k  integer NOT NULL DEFAULT 1;
  ch text    COLLATE "fr_FR";
```

- Blocs DECLARE/BEGIN/END imbriqués possible
  - restriction de scope de variable

En PL/pgSQL, pour utiliser une variable dans le corps de la routine (entre le BEGIN et le END), il est obligatoire de l'avoir déclarée précédemment :

- soit dans la liste des arguments (IN, INOUT ou OUT) ;
- soit dans la section DECLARE.

La déclaration doit impérativement préciser le nom et le type de la variable.

En option, il est également possible de préciser :

- sa valeur initiale (si rien n'est précisé, ce sera NULL par défaut) :

```
answer integer := 42;
```

- sa valeur par défaut, si on veut autre chose que NULL :

```
answer integer DEFAULT 42;
```

- une contrainte NOT NULL (dans ce cas, il faut impérativement un défaut différent de NULL, et toute éventuelle affectation ultérieure de NULL à la variable provoquera une erreur) :

```
answer integer NOT NULL DEFAULT 42;
```

- le collationnement à utiliser, pour les variables de type chaîne de caractères :

```
question text COLLATE "en_GB";
```

Pour les fonctions complexes, avec plusieurs niveaux de boucle par exemple, il est possible d'imbriquer les blocs DECLARE/BEGIN/END en y déclarant des variables locales à ce bloc. Si une variable est par erreur utilisée hors du *scope* prévu, une erreur surviendra.

3.8.2 Constantes

- Clause supplémentaire CONSTANT :

```
DECLARE
  eur_to_frf   CONSTANT numeric := 6.55957 ;
  societe_nom  CONSTANT text    := 'Dalibo SARL';
```

L'option CONSTANT permet de définir une variable pour laquelle il sera alors impossible d'assigner une valeur dans le reste de la routine.

3.8.3 Types de variables

- Récupérer le type d'une autre variable avec %TYPE :

```
quantite    integer ;
total       quantite%TYPE ;
```

- Récupérer le type de la colonne d'une table :

```
quantite    ma_table.ma_colonne%TYPE ;
```

Cela permet d'écrire des routines plus génériques.

3.8.4 Type ROW - 1

- Pour renvoyer plusieurs valeurs à partir d'une fonction
- Utiliser un type composite :

```
CREATE TYPE ma_structure AS (
    un_entier integer,
    une_chaine text,
    …);

CREATE FUNCTION ma_fonction () RETURNS ma_structure …;
```

3.8.5 Type ROW - 2

- Utiliser le type composite défini par la ligne d'une table

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
  ligne ma_table%ROWTYPE;
…
$$
```

L'utilisation de %ROWTYPE permet de définir une variable qui contient la structure d'un enregistrement de la table spécifiée. %ROWTYPE n'est pas obligatoire, il est néanmoins préférable d'utiliser cette forme, bien plus portable. En effet, dans PostgreSQL, toute création de table crée un type associé de même nom, le seul nom de la table est donc suffisant.

3.8.6 Type RECORD

- RECORD identique au type ROW
  - …sauf que son type n'est connu que lors de son affectation
- RECORD peut changer de type au cours de l'exécution de la routine
- Curseur et boucle sur une requête

RECORD est beaucoup utilisé pour manipuler des curseurs, ou dans des boucles FOR … LOOP : cela évite de devoir se préoccuper de déclarer un type correspondant exactement aux colonnes de la requête associée à chaque curseur.

3.8.7 Type RECORD : exemple

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
  ligne RECORD;
BEGIN
  -- récupération de la 1è ligne uniquement
  SELECT * INTO ligne FROM ma_première_table;
  -- ou : traitement ligne à ligne
  FOR ligne IN SELECT * FROM ma_deuxième_table  LOOP
    …
  END LOOP ;
  RETURN … ;
END $$ ;
```

Dans ces exemples, on récupère la première ligne de la fonction avec `SELECT … INTO`, puis on ouvre un curseur implicite pour balayer chaque ligne obtenue d'une deuxième table. Le type RECORD permet de ne pas déclarer une nouvelle variable de type ligne.

3.9 EXÉCUTION DE REQUÊTE DANS UN BLOC PL/PGSQL

3.9.1 Requête dans un bloc PL/pgSQL

- Toutes opérations sur la base de données
- Et calculs, comparaisons, etc.
- Toute expression écrite en PL/pgSQL sera passée à SELECT pour interprétation par le moteur
- PREPARE implicite, avec cache

Par expression, on entend par exemple des choses comme :

```
IF myvar > 0 THEN
    myvar2 := 1 / myvar;
END IF;
```

Dans ce cas, l'expression `myvar > 0` sera préparée par le moteur de la façon suivante :

```
PREPARE statement_name(integer, integer) AS SELECT $1 > $2;
```

Puis cette requête préparée sera exécutée en lui passant en paramètre la valeur de `myvar` et la constante `0`.

Si `myvar` est supérieur à `0`, il en sera ensuite de même pour l'instruction suivante :

```
PREPARE statement_name(integer, integer) AS SELECT $1 / $2;
```

Comme toute requête préparée, son plan sera mis en cache.

Pour les détails, voir les dessous de PL/pgSQL[18].

3.9.2 Affectation d'une valeur à une variable

- Utiliser l'opérateur `:=` :

```
un_entier := 5;
```

- Utiliser `SELECT INTO` :

```
SELECT 5 INTO un_entier;
```

[18] https://docs.postgresql.fr/current/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING

Privilégiez la première écriture pour la lisibilité, la seconde écriture est moins claire et n'apporte rien puisqu'il s'agit ici d'une affectation de constante.

À noter que l'écriture suivante est également possible pour une affectation :

```
ma_variable := une_colonne FROM ma_table WHERE id = 5;
```

Cette méthode profite du fait que toutes les expressions du code PL/pgSQL vont être passées au moteur SQL de PostgreSQL dans un SELECT pour être résolues. Cela va fonctionner, mais c'est très peu lisible, et donc non recommandé.

3.9.3 Exécution d'une requête

- Affectation de la ligne :

```
SELECT *
INTO ma_variable_ligne  -- type ROW ou RECORD
FROM …;
```

- INTO STRICT pour garantir unicité
  - INTO seul : juste 1è ligne !
- Plus d'un enregistrement :
  - écrire une boucle
- Ordre statique :
  - colonnes, clause WHERE, tables figées

Récupérer une ligne de résultat d'une requête dans une ligne de type ROW ou RECORD se fait avec SELECT … INTO. La première ligne est récupérée. Généralement on préférera utiliser INTO STRICT pour lever une de ces erreurs si la requête renvoie zéro ou plusieurs lignes :

```
ERROR:  query returned no rows
ERROR:  query returned more than one row
```

Dans le cas du type ROW, la définition de la ligne doit correspondre parfaitement à la définition de la ligne renvoyée. Utiliser un type RECORD permet d'éviter ce type de problème. La variable obtient directement le type ROW de la ligne renvoyée.

Il est possible d'utiliser SELECT INTO avec une simple variable si l'on n'a qu'un champ d'une ligne à récupérer.

Cette fonction compte les tables, et en trace la liste (les tables ne font pas partie du résultat) :

```
CREATE OR REPLACE FUNCTION compte_tables () RETURNS int LANGUAGE plpgsql AS $$
DECLARE
  n int ;
  t RECORD ;
BEGIN
  SELECT count(*) INTO STRICT n
  FROM pg_tables ;

  FOR t IN SELECT * FROM pg_tables LOOP
    RAISE NOTICE 'Table %.%', t.schemaname, t.tablename;
  END LOOP ;

  RETURN n ;
END ;
$$ ;
```

```
# SELECT compte_tables ();

NOTICE:  Table pg_catalog.pg_foreign_server
NOTICE:  Table pg_catalog.pg_type
…
NOTICE:  Table public.pgbench_accounts
NOTICE:  Table public.pgbench_branches
NOTICE:  Table public.pgbench_tellers
NOTICE:  Table public.pgbench_history
 compte_tables
---------------
           186
(1 ligne)
```

3.9.4 Exécution d'une requête sans besoin du résultat

- PERFORM : résultat ignoré

```
PERFORM * FROM ma_table WHERE une_colonne>0 ;
PERFORM mafonction (argument1) ;
```

- Variable FOUND
  - si une ligne est affectée par l'instruction
- Nombre de lignes :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

On peut déterminer qu'aucune ligne n'a été trouvée par la requête en utilisant la variable FOUND :

```
PERFORM * FROM ma_table WHERE une_colonne>0;
IF NOT FOUND THEN
```

```
…
END IF;
```

Pour appeler une fonction, il suffit d'utiliser PERFORM de la manière suivante :

```
PERFORM mafonction(argument1);
```

Pour récupérer le nombre de lignes affectées par l'instruction exécutée, il faut récupérer la variable de diagnostic ROW_COUNT :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

Il est à noter que le ROW_COUNT récupéré ainsi s'applique à l'ordre SQL précédent, quel qu'il soit :

- PERFORM ;
- EXECUTE ;
- ou même à un ordre statique directement dans le code PL/pgSQL.

3.10 SQL DYNAMIQUE

3.10.1 EXECUTE d'une requête

- `EXECUTE 'chaine' [INTO [STRICT] cible] [USING (paramètres)] ;`
- Exécute la requête dans `chaine`
- `chaine` peut être construite à partir d'autres variables
- `cible` : résultat (une seule ligne)

EXECUTE dans un bloc PL/pgSQL permet notamment du SQL dynamique : l'ordre peut être construit dans une variable.

3.10.2 EXECUTE & requête dynamique : injection SQL

Si nom vaut : « `'Robert' ; DROP TABLE eleves ;` »
que renvoie ceci ?

```
EXECUTE 'SELECT * FROM eleves WHERE nom = '|| nom ;
```

Un danger du SQL dynamique est de faire aveuglément confiance aux valeurs des variables en construisant un ordre SQL :

```
CREATE TEMP TABLE eleves (nom text, id int) ;
INSERT INTO eleves VALUES ('Robert', 0) ;

-- Mise à jour d'un ID
DO $f$
DECLARE
    nom text := $$'Robert' ; DROP TABLE eleves;$$  ;
    id  int ;
BEGIN
RAISE NOTICE 'A exécuter : %','SELECT * FROM eleves WHERE nom = '|| nom ;
EXECUTE 'UPDATE eleves SET id = 327 WHERE nom = '|| nom ;
END ;
$f$ LANGUAGE plpgsql ;

NOTICE:  A exécuter : SELECT * FROM eleves WHERE nom = 'Robert' ; DROP TABLE eleves;
```

```
\d+ eleves
Aucune relation nommée « eleves » n'a été trouvée.
```

Cet exemple est directement inspiré d'un dessin très connu de XKCD[19].

Dans la pratique, la variable nom (entrée ici en dur) proviendra par exemple d'un site web, et donc contient potentiellement des caractères terminant la requête dynamique et en insérant une autre, potentiellement destructrice.

Moins grave, une erreur peut être levée à cause d'une apostrophe (*quote*) dans une chaîne texte. Il existe effectivement des gens avec une apostrophe dans le nom.

Ce qui suit concerne le SQL dynamique dans des routines PL/pgSQL, mais le principe concerne tous les langages et clients, y compris `psql` et sa méta-commande `\gexec`[20]. En SQL pur, la protection contre les injections SQL est un argument pour utiliser les requêtes préparées[21], dont l'ordre EXECUTE diffère de celui-ci du PL/pgSQL ci-dessous.

3.10.3 EXECUTE & requête dynamique : 3 possibilités

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(nom_colonne)
    || ' = '
    || quote_literal(nouvelle_valeur)
    || ' WHERE cle = '
    || quote_literal(valeur_cle) ;

EXECUTE format('UPDATE matable SET %I = %L '
   'WHERE clef = %L', nom_colonne, nouvelle_valeur, valeur_clef);

EXECUTE format('UPDATE table SET %I = $1 '
   'WHERE clef = $2', nom_colonne) USING nouvelle_valeur, valeur_clef;
```

Les trois exemples précédents sont équivalents.

Le premier est le plus simple au premier abord. Il utilise `quote_ident` et `quote_literal` pour protéger des injections SQL[22] (voir plus loin).

[19] https://xkcd.com/327/
[20] https://docs.postgresql.fr/current/app-psql.html#APP-PSQL-META-COMMANDS
[21] https://docs.postgresql.fr/current/sql-prepare.html
[22] https://fr.wikipedia.org/wiki/Injection_SQL

Le second est plus lisible grâce à la fonction de formatage `format`[23] qui évite ces concaténations et appelle implicitement les fonctions `quote_%` Si un paramètre ne peut pas prendre la valeur NULL, utiliser `%L` (équivalent de `quote_nullable`) et non `%I` (équivalent de `quote_ident`).

La troisième alternative avec `USING` et les paramètres numériques `$1` et `$2` est considérée comme la plus performante.

(Voir les détails dans la documentation[24]).

L'exemple complet suivant tiré de la documentation officielle[25] utilise `EXECUTE` pour rafraîchir des vues matérialisées en masse.

```
CREATE FUNCTION rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Rafraichissement de toutes les vues matérialisées…';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP
        -- Maintenant "mviews" contient un enregistrement avec les informations sur
        ↪  la vue matérialisé
        RAISE NOTICE 'Rafraichissement de la vue matérialisée %.% (owner: %)…',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema,
        ↪  mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

[23] https://docs.postgresql.fr/current/functions-string.html#FUNCTIONS-STRING-FORMAT
[24] https://docs.postgresql.fr/current/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE
[25] https://www.postgresql.org/docs/current/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE

3.10.4 EXECUTE & requête dynamique (suite)

- `EXECUTE 'chaine' [INTO STRICT cible] [USING (paramètres)] ;`
- `STRICT` : **1** résultat
  - sinon `NO_DATA_FOUND` ou `TOO_MANY_ROWS`
- Sans `STRICT` :
  - 1ère ligne ou `NO_DATA_FOUND`
- Nombre de lignes :
  - `GET DIAGNOSTICS integer_var = ROW_COUNT`

De la même manière que pour `SELECT ... INTO`, utiliser `STRICT` permet de garantir qu'il y a exactement une valeur comme résultat de `EXECUTE`, ou alors une erreur sera levée.

Nous verrons plus loin comment traiter les exceptions.

3.10.5 Outils pour construire une requête dynamique

- `quote_ident ()`
  - pour mettre entre guillemets un identifiant d'un objet PostgreSQL (table, colonne, etc.)
- `quote_literal ()`
  - pour mettre entre guillemets une valeur (chaîne de caractères)
- `quote_nullable ()`
  - pour mettre entre guillemets une valeur (chaîne de caractères), sauf NULL qui sera alors renvoyé sans les guillemets
- `||` : concaténer
- Ou fonction `format(...)`, équivalent de `sprintf` en C

La fonction `format` est l'équivalent de la fonction `sprintf` en C : elle formate une chaîne en fonction d'un patron et de valeurs à appliquer à ses paramètres et la retourne. Les types de paramètre reconnus par `format` sont :

- `%I` : est remplacé par un identifiant d'objet. C'est l'équivalent de la fonction `quote_ident`. L'objet en question est entouré de guillemets doubles si nécessaire ;
- `%L` : est remplacé par une valeur littérale. C'est l'équivalent de la fonction `quote_literal`. Des guillemets simples sont ajoutés à la valeur et celle-ci est correctement échappée si nécessaire ;
- `%s` : est remplacé par la valeur donnée sans autre forme de transformation ;
- `%%` : est remplacé par un simple %.

Voici un exemple d'utilisation de cette fonction, utilisant des paramètres positionnels :

```
SELECT format(
    'SELECT %I FROM %I WHERE %1$I=%3$L',
    'MaColonne',
    'ma_table',
    $$l'été$$
);
```

```
                           format
-------------------------------------------------------------
 SELECT "MaColonne" FROM ma_table WHERE "MaColonne"='l''été'
```

3.11 STRUCTURES DE CONTRÔLE EN PL/PGSQL

- But du PL : les traitements procéduraux

3.11.1 Tests conditionnels - 2

Exemple :

```
IF nombre = 0 THEN
  resultat := 'zero';
ELSEIF nombre > 0 THEN
   resultat := 'positif';
ELSEIF nombre < 0 THEN
   resultat := 'négatif';
ELSE
   resultat := 'indéterminé';
END IF;
```

3.11.2 Tests conditionnels : CASE

```
    CASE nombre
    WHEN nombre = 0  THEN 'zéro'
    WHEN variable > 0  THEN 'positif'
    WHEN variable < 0  THEN 'négatif'
    ELSE 'indéterminé'
    END CASE
```

ou :

```
CASE current_setting ('server_version_num')::int/10000
    WHEN 8,9            THEN RAISE NOTICE 'Version non supportée !!' ;
    WHEN 10,11,12,13,14 THEN RAISE NOTICE 'Version supportée' ;
    ELSE                     RAISE NOTICE 'Version inconnue au 1/11/2021
    ↪  ?' ;
END CASE ;
```

L'instruction CASE WHEN est proche de l'expression CASE[26] des requêtes SQL dans son principe (à

[26] https://docs.postgresql.fr/current/functions-conditional.html#FUNCTIONS-CASE

part qu'elle se clôt par END en SQL, et END CASE en PL/pgSQL).

Elle est parfois plus légère à lire que des IF imbriqués.

Exemple complet :

```
DO $$
BEGIN
CASE current_setting ('server_version_num')::int/10000
    WHEN 8,9            THEN RAISE NOTICE 'Version non supportée !!' ;
    WHEN 10,11,12,13,14 THEN RAISE NOTICE 'Version supportée' ;
    ELSE                     RAISE NOTICE 'Version inconnue au 1/11/2021 ?' ;
END CASE ;
END ;
$$ LANGUAGE plpgsql ;
```

3.11.3 Boucle LOOP/EXIT/CONTINUE : syntaxe

- Boucle :
  - LOOP / END LOOP
  - label possible
- En sortir :
  - EXIT [label] [WHEN expression_booléenne]
- Commencer une nouvelle itération de la boucle
  - CONTINUE [label] [WHEN expression_booléenne]

Des boucles simples s'effectuent avec LOOP/END LOOP.

Pour les détails, voir la documentation officielle[27].

[27] https://docs.postgresql.fr/current/plpgsql-control-structures.html#PLPGSQL-CONTROL-STRUCTURES-LOOPS

3.11.4 Boucle LOOP/EXIT/CONTINUE : exemple

```
LOOP
  resultat := resultat + 1;
  EXIT WHEN resultat > 100;
  CONTINUE WHEN resultat < 50;
  resultat := resultat + 1;
END LOOP;
```

Cette boucle incrémente le résultat de 1 à chaque itération tant que la valeur du résultat est inférieure à 50. Ensuite, le résultat est incrémenté de 1 à deux reprises pour chaque tour de boucle. On incrémente donc de 2 par tour de boucle. Arrivée à 100, la procédure sort de la boucle.

3.11.5 Boucle WHILE

```
WHILE condition LOOP
  …
END LOOP;
```

- Boucle jusqu'à ce que la condition soit fausse
- Label possible

3.11.6 Boucle FOR : syntaxe

```
FOR variable in [REVERSE] entier1..entier2 [BY incrément]
LOOP
  …
END LOOP;
```

- `variable` va obtenir les différentes valeurs entre entier1 et entier2
- Label possible

La boucle FOR n'a pas d'originalité par rapport à d'autres langages.

L'option BY permet d'augmenter l'incrémentation :

```
FOR variable in 1..10 BY 5…
```

L'option REVERSE permet de faire défiler les valeurs en ordre inverse :

```
FOR variable in REVERSE 10..1 …
```

3.11.7 Boucle FOR … IN … LOOP : parcours de résultat de requête

```
FOR ligne IN ( SELECT * FROM ma_table ) LOOP
  …
END LOOP;
```

- Pour boucler dans les lignes résultats d'une requête
- `ligne` de type RECORD, ROW, ou liste de variables séparées par des virgules
- Utilise un curseur en interne
- Label possible

Cette syntaxe très pratique permet de parcourir les lignes résultant d'une requête sans avoir besoin de créer et parcourir un curseur. Souvent on utilisera une variable de type ROW ou RECORD (comme dans l'exemple de la fonction `rafraichir_vuemat` plus haut), mais l'utilisation directe de variables (déclarées préalablement) est possible :

```
FOR a, b, c, d IN
  (SELECT col_a, col_b, col_c, col_d FROM ma_table)
LOOP
  -- instructions utilisant ces variables
  …
END LOOP;
```

Attention de ne pas utiliser les variables en question hors de la boucle, elles auront gardé la valeur acquise dans la dernière itération.

3.11.8 Boucle FOREACH

```
FOREACH variable [SLICE n] IN ARRAY expression LOOP
    …
END LOOP ;
```

- Pour boucler sur les éléments d'un tableau
- `variable` va obtenir les différentes valeurs du tableau retourné par `expression`
- `SLICE` permet de jouer sur le nombre de dimensions du tableau à passer à la variable
- Label possible

Voici deux exemples permettant d'illustrer l'utilité de `SLICE` :

- sans `SLICE` :

```
DO $$
DECLARE a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int;
BEGIN
  FOREACH b IN ARRAY a LOOP
  RAISE INFO 'var: %', b;
END LOOP;
END $$ ;
```

```
INFO:  var: 1
INFO:  var: 2
INFO:  var: 3
INFO:  var: 4
INFO:  var: 5
INFO:  var: 6
```

- avec `SLICE` :

```
DO $$
DECLARE a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int[];
BEGIN
  FOREACH b SLICE 1 IN ARRAY a LOOP
  RAISE INFO 'var: %', b;
END LOOP;
END $$;
```

```
INFO:  var: {1,2}
INFO:  var: {3,4}
INFO:  var: {5,6}
```

et avec `SLICE 2`, on obtient :

```
INFO:  var: {{1,2},{3,4},{5,6}}
```

3.12 AUTRES PROPRIÉTÉS DES FONCTIONS

- Sécurité
- Optimisations
- Parallélisation

3.12.1 Politique de sécurité

- `SECURITY INVOKER` : défaut
- `SECURITY DEFINER`
  - « sudo de la base de données »
  - potentiellement dangereux
  - ne pas laisser à **public** !

Une fonction `SECURITY INVOKER` s'exécute avec les droits de l'appelant. C'est le mode par défaut.

Une fonction `SECURITY DEFINER` s'exécute avec les droits du créateur. Cela permet, au travers d'une fonction, de permettre à un utilisateur d'outrepasser ses droits de façon contrôlée.

Bien sûr, une fonction `SECURITY DEFINER` doit faire l'objet d'encore plus d'attention qu'une fonction normale. Elle peut facilement constituer un trou béant dans la sécurité de votre base.

Deux points importants sont à noter pour `SECURITY DEFINER` :

- Par défaut, toute fonction créée dans **public** est exécutable par le rôle **public**. La première chose à faire est donc de révoquer ce droit. Créer la fonction dans un schéma séparé permet aussi de gérer plus finalement les accès.
- Il faut se protéger des variables de session qui pourraient être utilisées pour modifier le comportement de la fonction, en particulier le *search_path* (qui pourrait faire pointer vers des tables de même nom dans un autre schéma). Il doit donc **impérativement** être positionné en dur dans cette fonction (soit d'emblée, avec un `SET` dans la fonction, soit en positionnant un `SET` dans le `CREATE FUNCTION`) ; ou bien les fonctions doivent préciser systématiquement le schéma (`SELECT … FROM nomschema.nomtable …`).

3.12.2 Optimisation des fonctions

- Fonctions uniquement
- À destination de l'optimiseur
- `COST cout_execution`
  - coût estimé pour l'exécution de la fonction
- `ROWS nb_lignes_resultat`
  - nombre estimé de lignes que la fonction renvoie

`COST` est un coût représenté en unité de `cpu_operator_cost` (100 par défaut).

`ROWS` vaut par défaut 1000 pour les fonctions `SETOF` ou `TABLE`, et 1 pour les autres.

Ces deux paramètres ne modifient pas le comportement de la fonction. Ils ne servent que pour aider l'optimiseur de requête à estimer le coût d'appel à la fonction, afin de savoir, si plusieurs plans sont possibles, lequel est le moins coûteux par rapport au nombre d'appels de la fonction et au nombre d'enregistrements qu'elle retourne.

3.12.3 Parallélisation

- Fonctions uniquement
- La fonction peut-elle être exécutée en parallèle ?
  - `PARALLEL UNSAFE` (défaut)
  - `PARALLEL RESTRICTED`
  - `PARALLEL SAFE`

`PARALLEL UNSAFE` indique que la fonction ne peut pas être exécutée dans le mode parallèle. La présence d'une fonction de ce type dans une requête SQL force un plan d'exécution en série. C'est la valeur par défaut.

Une fonction est non parallélisable si elle modifie l'état d'une base ou si elle fait des changements sur la transaction.

`PARALLEL RESTRICTED` indique que la fonction peut être exécutée en mode parallèle mais l'exécution est restreinte au processus principal d'exécution.

Une fonction peut être déclarée comme restreinte si elle accède aux tables temporaires, à l'état de connexion des clients, aux curseurs, aux requêtes préparées.

PARALLEL SAFE indique que la fonction s'exécute correctement dans le mode parallèle sans restriction.

En général, si une fonction est marquée sûre ou restreinte à la parallélisation alors qu'elle ne l'est pas, elle pourrait renvoyer des erreurs ou fournir de mauvaises réponses lorsqu'elle est utilisée dans une requête parallèle.

En cas de doute, les fonctions doivent être marquées comme UNSAFE, ce qui correspond à la valeur par défaut.

3.13 UTILISATION DE FONCTIONS DANS LES INDEX

- Fonctions uniquement !
- `IMMUTABLE | STABLE | VOLATILE`
- Ce mode précise la « volatilité » de la fonction.
- Permet de réduire le nombre d'appels
- Index : fonctions immutables uniquement (sinon problèmes !)

On peut indiquer à PostgreSQL le niveau de volatilité (ou de stabilité) d'une fonction. Ceci permet d'aider PostgreSQL à optimiser les requêtes utilisant ces fonctions, mais aussi d'interdire leur utilisation dans certains contextes.

Une fonction est « **immutable** » si son exécution ne dépend que de ses paramètres. Elle ne doit donc dépendre ni du contenu de la base (pas de `SELECT`, ni de modification de donnée de quelque sorte), ni d'**aucun** autre élément qui ne soit pas un de ses paramètres. Les fonctions arithmétiques simples (`+`, `*`, `abs`…) sont immutables.

À l'inverse, `now()` n'est évidemment pas immutable. Une fonction sélectionnant des données d'une table non plus. `to_char()` n'est pas non plus immutable, car son comportement dépend des paramètres de session, par exemple `to_char(timestamp with time zone, text)` dépend du paramètre de session `timezone`…

Une fonction est « **stable** » si son exécution donne toujours le même résultat sur toute la durée d'un ordre SQL, pour les mêmes paramètres en entrée. Cela signifie que la fonction ne modifie pas les données de la base. Une fonction n'exécutant que des `SELECT` sur des tables (pas des fonctions !) sera stable. `to_char()` est stable. L'optimiseur peut réduire ainsi le nombre d'appels sans que ce soit en pratique toujours le cas.

Une fonction est « **volatile** » dans tous les autres cas. `random()` est volatile. Une fonction volatile peut même modifier les donneés. Une fonction non déclarée comme stable ou immutable est volatile par défaut.

La volatilité des fonctions intégrées à PostgreSQL est déjà définie. C'est au développeur de préciser la volatilité des fonctions qu'il écrit. Ce n'est pas forcément évident. Une erreur peut poser des problèmes quand le plan est mis en cache, ou, on le verra, dans des index.

Quelle importance cela a-t-il ?

Prenons une table d'exemple sur les heures de l'année 2020 :

```
-- Une ligne par heure dans l année, 8784 lignes
CREATE TABLE heures
AS
SELECT i, '2020-01-01 00:00:00+01:00'::timestamptz + i * interval '1 hour'  AS t
FROM generate_series (1,366*24) i;
```

Définissons une fonction un peu naïve ramenant le premier jour du mois, volatile faute de mieux :

```
CREATE OR REPLACE FUNCTION premierjourdumois(t timestamptz)
RETURNS timestamptz
LANGUAGE plpgsql
VOLATILE
AS $$
BEGIN
    RAISE notice 'appel premierjourdumois' ; -- trace des appels
    RETURN date_trunc ('month', t);
END $$ ;
```

Demandons juste le plan d'un appel ne portant que sur le dernier jour :

```
EXPLAIN SELECT * FROM heures
WHERE t > premierjourdumois('2020-12-31 00:00:00+02:00'::timestamptz)
LIMIT 10 ;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
Limit  (cost=0.00..8.04 rows=10 width=12)
 ->  Seq Scan on heures  (cost=0.00..2353.80 rows=2928 width=12)
       Filter: (t > premierjourdumois(
                    '2020-12-30 23:00:00+01'::timestamp with time zone))
```

Le nombre de lignes attendues (2928) est le tiers de la table, alors que nous ne demandons que le dernier mois. Il s'agit de l'estimation forfaitaire que PostgreSQL utilise faute d'informations sur ce que va retourner la fonction.

Demander à voir le résultat mène à l'affichage de milliers de `NOTICE` : la fonction est appelée à chaque ligne pour calculer s'il faut filtrer la valeur. En effet, une fonction volatile sera systématiquement exécutée à chaque appel, et, selon le plan, ce peut être pour chaque ligne parcourue !

Cependant notre fonction ne fait que des calculs à partir du paramètre, sans effet de bord. Déclarons-la donc stable :

```
ALTER FUNCTION premierjourdumois(timestamp with time zone) STABLE ;
```

Une fonction stable peut en théorie être remplacée par son résultat pendant l'exécution de la requête. Mais c'est impossible de le faire plus tôt, car on ne sait pas forcément dans quel contexte la fonction va être appelée (par exemple, en cas de requête préparée, les paramètres de la session ou les données de la base peuvent même changer entre la planification et l'exécution).

Dans notre cas, le même `EXPLAIN` simple mène à ceci :

```
NOTICE:  appel premierjourdumois
                                  QUERY PLAN
------------------------------------------------------------------------------
Limit  (cost=0.00..32.60 rows=10 width=12)
 ->  Seq Scan on heures  (cost=0.00..2347.50 rows=720 width=12)
       Filter: (t > premierjourdumois(
                    '2020-12-30 23:00:00+01'::timestamp with time zone))
```

Comme il s'agit d'un simple `EXPLAIN`, la requête n'est pas exécutée. Or le message `NOTICE` est renvoyé : la fonction est donc exécutée pour une simple planification. Un appel unique suffit, puisque la valeur d'une fonction stable ne change pas pendant toute la durée de la requête pour les mêmes

paramètres (ici une constante). Cet appel permet d'affiner la volumétrie des valeurs attendues, ce qui peut avoir un impact énorme.

Cependant, à l'exécution, les NOTICE apparaîtront pour indiquer que la fonction est à nouveau appelée à chaque ligne. Pour qu'un seul appel soit effectué pour toute la requête, il faudrait déclarer la fonction comme immutable, ce qui serait faux, puisqu'elle dépend implicitement du fuseau horaire.

Dans l'idéal, une fonction immutable peut être remplacée par son résultat avant même la planification d'une requête l'utilisant. C'est le cas avec les calculs arithmétiques par exemple :

```
EXPLAIN SELECT * FROM heures
WHERE i > abs(364*24) AND t > '2020-06-01'::date + interval '57 hours' ;
```

La valeur est substituée très tôt, ce qui permet de les comparer aux statistiques :

```
Seq Scan on heures  (cost=0.00..179.40 rows=13 width=12)
  Filter: ((i > 8736) AND (t > '2020-06-03 09:00:00'::timestamp without time zone))
```

Pour forcer un appel unique quand on sait que la fonction renverra une constante, du moins le temps de la requête, même si elle est volatile, une astuce est de signifier à l'optimiseur qu'il n'y aura qu'une seule valeur de comparaison, même si on ne sait pas laquelle :

```
EXPLAIN (ANALYZE) SELECT * FROM heures
WHERE t > (SELECT premierjourdumois('2020-12-31 00:00:00+02:00'::timestamptz)) ;

NOTICE:  appel premierjourdumois
                                  QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on heures  (cost=0.26..157.76 rows=2920 width=12)
                     (actual time=1.090..1.206 rows=721 loops=1)
   Filter: (t > $0)
   Rows Removed by Filter: 8039
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.26 rows=1 width=8)
                 (actual time=0.138..0.139 rows=1 loops=1)
 Planning Time: 0.058 ms
 Execution Time: 1.328 ms
```

On note qu'il n'y a qu'un appel. On comprend donc l'intérêt de se poser la question à l'écriture de chaque fonction.

La volatilité est encore plus importante quand il s'agit de créer des fonctions sur index :

```
CREATE INDEX ON heures (premierjourdumois( t )) ;

ERROR:  functions in index expression must be marked IMMUTABLE
```

Ceci n'est possible que si la fonction est immutable. En effet, si le résultat de la fonction dépend de l'état de la base ou d'autres paramètres, la fonction exécutée au moment de la création de la clé d'index pourrait ne plus retourner le même résultat quand viendra le moment de l'interroger. PostgreSQL n'acceptera donc que les fonctions immutables dans la déclaration des index fonctionnels.

Déclarer hâtivement une fonction comme immutable juste pour pouvoir l'utiliser dans un index est dangereux : en cas d'erreur, les résultats d'une requête peuvent alors dépendre du plan d'exécution, selon que les index seront utilisés ou pas !

Cela est particulièrement fréquent quand les fuseaux horaires ou les dictionnaires sont impliqués. Vérifiez bien que vous n'utilisez que des fonctions immutables dans les index fonctionnels, les pièges sont nombreux.

Par exemple, si l'on veut une version immutable de la fonction précédente, il faut fixer le fuseau horaire dans l'appel à `date_trunc`. En effet, on peut voir avec `df+ date_trunc` que la seule version immutable de `date_trunc` n'accepte que des `timestamp` (sans fuseau), et en renvoie un. Notre fonction devient donc :

```
CREATE OR REPLACE FUNCTION premierjourdumois_utc(t timestamptz)
RETURNS timestamptz
LANGUAGE plpgsql
IMMUTABLE
AS $$
DECLARE
    jour1   timestamp ; --sans TZ
BEGIN
    jour1 := date_trunc ('month', (t at time zone 'UTC')::timestamp) ;
    RETURN jour1 AT TIME ZONE 'UTC';
END $$ ;
```

Testons avec une date dans les dernières heures de septembre en Alaska, qui correspond au tout début d'octobre en temps universel, et par exemple aussi au Japon :

```
\x

SET timezone TO 'US/Alaska';

SELECT d,
       d AT TIME ZONE 'UTC' AS d_en_utc,
       premierjourdumois_utc (d),
       premierjourdumois_utc (d) AT TIME ZONE 'UTC' as pjm_en_utc
FROM (SELECT '2020-09-30 18:00:00-08'::timestamptz AS d) x;

-[ RECORD 1 ]---------+----------------------
d                     | 2020-09-30 18:00:00-08
d_en_utc              | 2020-10-01 02:00:00
premierjourdumois_utc | 2020-09-30 16:00:00-08
pjm_en_utc            | 2020-10-01 00:00:00

SET timezone TO 'Japan';

SELECT d,
       d AT TIME ZONE 'UTC' AS d_en_utc,
       premierjourdumois_utc (d),
       premierjourdumois_utc (d) AT TIME ZONE 'UTC' as pjm_en_utc
FROM (SELECT '2020-09-30 18:00:00-08'::timestamptz AS d) x;
```

```
-[ RECORD 1 ]---------+-----------------------
d                     | 2020-10-01 11:00:00+09
d_en_utc              | 2020-10-01 02:00:00
premierjourdumois_utc | 2020-10-01 09:00:00+09
pjm_en_utc            | 2020-10-01 00:00:00
```

Malgré les différences d'affichage dues au fuseau horaire, c'est bien le même moment (la première seconde d'octobre en temps universel) qui est retourné par la fonction.

Pour une fonction aussi simple, la version SQL est même préférable :

```
CREATE OR REPLACE FUNCTION premierjourdumois_utc(t timestamptz)
RETURNS timestamptz
LANGUAGE sql
IMMUTABLE
AS $$
    SELECT (date_trunc ('month', (t at time zone 'UTC')::timestamp)) AT TIME ZONE
     ↪  'UTC';
$$ ;
```

Enfin, la volatilité a également son importance lors d'autres opérations d'optimisation, comme l'exclusion de partitions. Seules les fonctions immutables sont compatibles avec le *partition pruning* effectué à la planification, mais les fonctions stable sont éligibles au *dynamic partition pruning* (à l'exécution) apparu avec PostgreSQL 11.

3.14 CONCLUSION

- Grand nombre de structure de contrôle (test, boucle, etc.)
- Facile à utiliser et à comprendre
- Attention à la compatibilité ascendante

3.14.1 Pour aller plus loin

- Documentation officielle
  - « Chapitre 40. PL/pgSQL - Langage de procédures SQL »

La documentation officielle sur le langage PL/pgSQL peut être consultée en français à cette adresse[28].

3.14.2 Questions

```
FOR q IN (SELECT * FROM questions ) LOOP

  répondre (q) ;

END LOOP ;
```

[28] https://docs.postgresql.fr/current/plpgsql.html

3.15 QUIZ

https://dali.bo/p1_quiz

3.16 TRAVAUX PRATIQUES

3.16.1 Hello

But : Premières fonctions

Écrire une fonction `hello()` qui renvoie la chaîne de caractère « Hello World! » en SQL.

Écrire une fonction `hello_pl()` qui renvoie la chaîne de caractère « Hello World! » en PL/pgSQL.

Comparer les coûts des deux plans d'exécutions de ces requêtes. Expliquer ces coûts.

3.16.2 Division

But : Fonction avec calcul simple

Écrire en PL/pgSQL une fonction de division appelée `division`. Elle acceptera en entrée deux arguments de type entier et renverra un nombre réel (`numeric`).

Écrire cette même fonction en SQL.

Comment corriger le problème de la division par zéro ? Écrire cette nouvelle fonction dans les deux langages. (Conseil : dans ce genre de calcul impossible, il est possible d'utiliser la constante `NaN` (*Not A Number*)).

3.16.3 SELECT sur des tables dans les fonctions

But : Utiliser une table à l'intérieur d'une fonction

Ce TP utilise les tables de la base **employes_services**.

Le script de création de la base peut être téléchargé depuis https://dali.bo/tp_employes_services. Il ne fait que 3,5 ko. Le chargement se fait de manière classique :

```
$ psql < employes_services.sql
```

Les quelques tables occupent environ 80 Mo sur le disque.

> Créer une fonction qui ramène le nombre d'employés embauchés une année donnée (à partir du champ `employes.date_embauche`).

> Utiliser la fonction `generate_series()` pour lister le nombre d'embauches pour chaque année entre 2000 et 2010.

> Créer une fonction qui fait la même chose avec deux années en paramètres une boucle `FOR ... LOOP`, `RETURNS TABLE` et `RETURN NEXT`.

3.16.4 Multiplication

> **But** : Fonctions avec de nombreuses conditions, des manipulations de types, et un message.

> Écrire une fonction de multiplication dont les arguments sont des chiffres en toute lettre, inférieurs ou égaux à « neuf ». Par exemple, `multiplication ('deux','trois')` doit renvoyer 6.

> Si ce n'est déjà fait, faire en sorte que `multiplication` appelle une autre fonction pour faire la conversion de texte en chiffre, et n'effectue que le calcul.

> Essayer de multiplier « deux » par 4. Qu'obtient-on et pourquoi ?

> Corriger la fonction pour tomber en erreur si un argument est numérique (utiliser `RAISE EXCEPTION <message>`).

3.16.5 Salutations

> **But** : Fonction plus complexe

Écrire une fonction en PL/pgSQL qui prend en argument le nom de l'utilisateur, puis lui dit « Bonjour » ou « Bonsoir » suivant l'heure de la journée. Utiliser la fonction `to_char()`.

Écrire la même fonction avec un paramètre `OUT`.

Pour calculer l'heure courante, utiliser plutôt la fonction `extract`.

Réécrire la fonction en SQL.

3.16.6 Inversion de chaîne

But : Manipuler des chaînes

Écrire une fonction `inverser` qui inverse une chaîne (pour « toto » en entrée, afficher « otot » en sortie), à l'aide d'une boucle `WHILE` et des fonctions `char_length` et `substring`.

3.16.7 Jours fériés

But : Calculs complexes avec des dates

Le calcul de la date de Pâques est complexe[29]. On peut écrire la fonction suivante :

```
CREATE OR REPLACE FUNCTION paques (annee integer)
RETURNS date
AS $$
  DECLARE
    a integer ;
    b integer ;
    r date ;
  BEGIN
    a := (19*(annee % 19) + 24) % 30 ;
    b := (2*(annee % 4) + 4*(annee % 7) + 6*a + 5) % 7 ;
    SELECT (annee::text||'-03-31')::date + (a+b-9) INTO r ;
    RETURN r ;
  END ;
$$
LANGUAGE plpgsql ;
```

[29] https://fr.wikipedia.org/wiki/Calcul_de_la_date_de_P%C3%A2ques

Principe : Soit `m` l'année. On calcule successivement :

- le reste de `m/19` : c'est la valeur de a.
- le reste de `m/4` : c'est la valeur de b.
- le reste de `m/7` : c'est la valeur de c.
- le reste de `(19a + p)/30` : c'est la valeur de d.
- le reste de `(2b + 4c + 6d + q)/7` : c'est la valeur de e.

Les valeurs de p et de q varient de 100 ans en 100 ans. De 2000 à 2100, p vaut 24, q vaut 5. La date de Pâques est le `(22 + d + e)` mars ou le `(d + e − 9)` avril.

> Afficher les dates de Pâques de 2018 à 2025.

> Écrire une fonction qui calcule la date de l'Ascension, soit le jeudi de la sixième semaine après Pâques. Pour simplifier, on peut aussi considérer que l'Ascension se déroule 39 jours après Pâques.

> Pour écrire une fonction qui renvoie tous les jours fériés d'une année (libellé et date), en France métropolitaine :
>
> - Prévoir un paramètre supplémentaire pour l'Alsace-Moselle, où le Vendredi saint (précédant le dimanche de Pâques) et le 26 décembre sont aussi fériés.
> - Cette fonction doit renvoyer plusieurs lignes : utiliser `RETURN NEXT`.
> - Plusieurs variantes sont possibles : avec `SETOF record`, avec des paramètres `OUT`, ou avec `RETURNS TABLE (libelle, jour)`.
> - Enfin, il est possible d'utiliser `RETURN QUERY`.

3.17 TRAVAUX PRATIQUES (SOLUTIONS)

3.17.1 Hello

> Écrire une fonction `hello()` qui renvoie la chaîne de caractère « Hello World! » en SQL.

```
CREATE OR REPLACE FUNCTION hello()
RETURNS text
AS $BODY$
  SELECT 'hello world !'::text;
$BODY$
LANGUAGE SQL;
```

> Écrire une fonction `hello_pl()` qui renvoie la chaîne de caractère « Hello World! » en PL/pgSQL.

```
CREATE OR REPLACE FUNCTION hello_pl()
RETURNS text
AS $BODY$
  BEGIN
    RETURN 'hello world !';
  END
$BODY$
LANGUAGE plpgsql;
```

> Comparer les coûts des deux plans d'exécutions de ces requêtes. Expliquer ces coûts.

Requêtage :

```
EXPLAIN SELECT hello();

                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=32)

EXPLAIN SELECT hello_pl();

                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=32)
```

Par défaut, si on ne précise pas le coût (COST) d'une fonction, cette dernière a un coût par défaut de 100. Ce coût est à multiplier par la valeur du paramètre `cpu_operator_cost`, par défaut à 0,0025. Le coût total d'appel de la fonction `hello_pl` est donc par défaut de :

100*cpu_operator_cost + cpu_tuple_cost

Ce n'est pas valable pour la fonction en SQL pur, qui est ici intégrée à la requête.

3.17.2 Division

> Écrire en PL/pgSQL une fonction de division appelée `division`. Elle acceptera en entrée deux arguments de type entier et renverra un nombre réel (`numeric`).

Attention, sous PostgreSQL, la division de deux entiers est par défaut entière : il faut donc transtyper.

```
CREATE OR REPLACE FUNCTION division (arg1 integer, arg2 integer)
RETURNS numeric
AS $BODY$
  BEGIN
    RETURN arg1::numeric / arg2::numeric;
  END
$BODY$
LANGUAGE plpgsql;
```

```
SELECT division (3,2) ;
```

```
      division
--------------------
 1.5000000000000000
```

> Écrire cette même fonction en SQL.

```
CREATE OR REPLACE FUNCTION division_sql (a integer, b integer)
RETURNS numeric
AS $$
    SELECT a::numeric / b::numeric;
$$
LANGUAGE SQL;
```

> Comment corriger le problème de la division par zéro ? Écrire cette nouvelle fonction dans les deux langages. (Conseil : dans ce genre de calcul impossible, il est possible d'utiliser la constante NaN (*Not A Number*)).

Le problème se présente ainsi :

```
SELECT division(1,0);
```

```
ERROR:  division by zero
CONTEXTE : PL/pgSQL function division(integer,integer) line 3 at RETURN
```

Pour la version en PL :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS numeric
AS $BODY$
  BEGIN
    IF arg2 = 0 THEN
      RETURN 'NaN';
    ELSE
      RETURN arg1::numeric / arg2::numeric;
    END IF;
  END $BODY$
LANGUAGE plpgsql;
```

```
SELECT division (3,0) ;

 division
----------
      NaN
```

Pour la version en SQL :

```
CREATE OR REPLACE FUNCTION division_sql(a integer, b integer)
RETURNS numeric
AS $$
  SELECT CASE $2
    WHEN 0 THEN 'NaN'
    ELSE $1::numeric / $2::numeric
  END;
$$
LANGUAGE SQL;
```

3.17.3 SELECT sur des tables dans les fonctions

> Créer une fonction qui ramène le nombre d'employés embauchés une année donnée (à partir du champ `employes.date_embauche`).

```
CREATE OR REPLACE FUNCTION nb_embauches (v_annee integer)
RETURNS integer
AS $BODY$
  DECLARE
    nb integer;
  BEGIN
    SELECT  count(*)
    INTO    nb
    FROM    employes
    WHERE   extract (year from date_embauche) = v_annee ;
    RETURN nb;
  END
$BODY$
LANGUAGE plpgsql ;
```

Test :

```
SELECT nb_embauches (2006);

 nb_embauches
--------------
            9
```

> Utiliser la fonction `generate_series()` pour lister le nombre d'embauches pour chaque année entre 2000 et 2010.

```
SELECT n, nb_embauches (n)
FROM generate_series (2000,2010) n
ORDER BY n;
```

```
  n   | nb_embauches
------+--------------
 2000 |            2
 2001 |            0
 2002 |            0
 2003 |            1
 2004 |            0
 2005 |            2
 2006 |            9
 2007 |            0
 2008 |            0
 2009 |            0
 2010 |            0
```

> Créer une fonction qui fait la même chose avec deux années en paramètres une boucle FOR ... LOOP, RETURNS TABLE et RETURN NEXT.

```
CREATE OR REPLACE FUNCTION nb_embauches (v_anneedeb int, v_anneefin int)
RETURNS TABLE (annee int, nombre_embauches int)
AS $BODY$
  BEGIN
    FOR i in v_anneedeb..v_anneefin
    LOOP
      SELECT i, nb_embauches (i)
      INTO   annee, nombre_embauches ;
      RETURN NEXT ;
    END LOOP;
    RETURN;
  END
$BODY$
LANGUAGE plpgsql;
```

Le nom de la fonction a été choisi identique à la précédente, mais avec des paramètres différents. Cela ne gêne pas le requêtage :

```
SELECT * FROM  nb_embauches (2006,2010);
```

```
 annee | nombre_embauches
-------+------------------
  2006 |                9
  2007 |                0
  2008 |                0
  2009 |                0
  2010 |                0
```

3.17.4 Multiplication

> Écrire une fonction de multiplication dont les arguments sont des chiffres en toute lettre, inférieurs ou égaux à « neuf ». Par exemple, `multiplication ('deux','trois')` doit renvoyer 6.

```
CREATE OR REPLACE FUNCTION multiplication (arg1 text, arg2 text)
RETURNS integer
```

```
AS $BODY$
  DECLARE
    a1 integer;
    a2 integer;
  BEGIN
    IF arg1 = 'zéro' THEN
      a1 := 0;
    ELSEIF arg1 = 'un' THEN
      a1 := 1;
    ELSEIF arg1 = 'deux' THEN
      a1 := 2;
    ELSEIF arg1 = 'trois' THEN
      a1 := 3;
    ELSEIF arg1 = 'quatre' THEN
      a1 := 4;
    ELSEIF arg1 = 'cinq' THEN
      a1 := 5;
    ELSEIF arg1 = 'six' THEN
      a1 := 6;
    ELSEIF arg1 = 'sept' THEN
      a1 := 7;
    ELSEIF arg1 = 'huit' THEN
      a1 := 8;
    ELSEIF arg1 = 'neuf' THEN
      a1 := 9;
    END IF;

    IF arg2 = 'zéro' THEN
      a2 := 0;
    ELSEIF arg2 = 'un' THEN
      a2 := 1;
    ELSEIF arg2 = 'deux' THEN
      a2 := 2;
    ELSEIF arg2 = 'trois' THEN
      a2 := 3;
    ELSEIF arg2 = 'quatre' THEN
      a2 := 4;
    ELSEIF arg2 = 'cinq' THEN
      a2 := 5;
    ELSEIF arg2 = 'six' THEN
      a2 := 6;
    ELSEIF arg2 = 'sept' THEN
      a2 := 7;
    ELSEIF arg2 = 'huit' THEN
      a2 := 8;
    ELSEIF arg2 = 'neuf' THEN
      a2 := 9;
    END IF;

    RETURN a1*a2;
  END
$BODY$
LANGUAGE plpgsql;
```

Test :

```
SELECT multiplication('deux', 'trois');

 multiplication
----------------
              6

SELECT multiplication('deux', 'quatre');

 multiplication
----------------
              8
```

> Si ce n'est déjà fait, faire en sorte que `multiplication` appelle une autre fonction pour faire la conversion de texte en chiffre, et n'effectue que le calcul.

```
CREATE OR REPLACE FUNCTION texte_vers_entier(arg text)
RETURNS integer AS $BODY$
  DECLARE
    ret integer;
  BEGIN
    IF arg = 'zéro' THEN
      ret := 0;
    ELSEIF arg = 'un' THEN
      ret := 1;
    ELSEIF arg = 'deux' THEN
      ret := 2;
    ELSEIF arg = 'trois' THEN
      ret := 3;
    ELSEIF arg = 'quatre' THEN
      ret := 4;
    ELSEIF arg = 'cinq' THEN
      ret := 5;
    ELSEIF arg = 'six' THEN
      ret := 6;
    ELSEIF arg = 'sept' THEN
      ret := 7;
    ELSEIF arg = 'huit' THEN
      ret := 8;
    ELSEIF arg = 'neuf' THEN
      ret := 9;
    END IF;

    RETURN ret;
  END
$BODY$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION multiplication(arg1 text, arg2 text)
RETURNS integer
AS $BODY$
  DECLARE
    a1 integer;
    a2 integer;
  BEGIN
    a1 := texte_vers_entier(arg1);
    a2 := texte_vers_entier(arg2);
```

```
    RETURN a1*a2;
  END
$BODY$
LANGUAGE plpgsql;
```

> Essayer de multiplier « deux » par 4. Qu'obtient-on et pourquoi ?

```
SELECT multiplication('deux', 4::text);

 multiplication
----------------
```

Par défaut, les variables internes à la fonction valent NULL. Rien n'est prévu pour affecter le second argument, on obtient donc NULL en résultat.

> Corriger la fonction pour tomber en erreur si un argument est numérique (utiliser `RAISE EXCEPTION <message>`).

```
CREATE OR REPLACE FUNCTION texte_vers_entier(arg text)
RETURNS integer AS $BODY$
  DECLARE
    ret integer;
  BEGIN
    IF arg = 'zéro' THEN
      ret := 0;
    ELSEIF arg = 'un' THEN
      ret := 1;
    ELSEIF arg = 'deux' THEN
      ret := 2;
    ELSEIF arg = 'trois' THEN
      ret := 3;
    ELSEIF arg = 'quatre' THEN
      ret := 4;
    ELSEIF arg = 'cinq' THEN
      ret := 5;
    ELSEIF arg = 'six' THEN
      ret := 6;
    ELSEIF arg = 'sept' THEN
      ret := 7;
    ELSEIF arg = 'huit' THEN
      ret := 8;
    ELSEIF arg = 'neuf' THEN
      ret := 9;
    ELSE
      RAISE EXCEPTION 'argument "%" invalide', arg;
      ret := NULL;
    END IF;

    RETURN ret;
  END
$BODY$
LANGUAGE plpgsql;

SELECT multiplication('deux', 4::text);
```

```
ERROR:  argument "4" invalide
CONTEXTE : PL/pgSQL function texte_vers_entier(text) line 26 at RAISE
PL/pgSQL function multiplication(text,text) line 7 at assignment
```

3.17.5 Salutations

> Écrire une fonction en PL/pgSQL qui prend en argument le nom de l'utilisateur, puis lui dit « Bonjour » ou « Bonsoir » suivant l'heure de la journée. Utiliser la fonction `to_char()`.

```
CREATE OR REPLACE FUNCTION salutation(utilisateur text)
RETURNS text
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    heure := to_char(now(), 'HH24');
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    RETURN libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

Test :

```
SELECT salutation ('Guillaume');

   salutation
---------------------
 Bonsoir Guillaume !
```

> Écrire la même fonction avec un paramètre OUT.

```
CREATE OR REPLACE FUNCTION salutation(IN utilisateur text, OUT message text)
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    heure := to_char(now(), 'HH24');
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    message := libelle||' '||utilisateur||' !';
```

```
  END
$BODY$
LANGUAGE plpgsql;
```

Elle s'utilise de la même manière :

```
SELECT salutation ('Guillaume');

    salutation
--------------------
 Bonsoir Guillaume !
```

> Pour calculer l'heure courante, utiliser plutôt la fonction `extract`.

```
CREATE OR REPLACE FUNCTION salutation(IN utilisateur text, OUT message text)
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    SELECT INTO heure extract(hour from now())::int;
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    message := libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

> Réécrire la fonction en SQL.

Le CASE … WHEN remplace aisément un IF … THEN :

```
CREATE OR REPLACE FUNCTION salutation_sql(nom text)
RETURNS text
AS $$
  SELECT CASE extract(hour from now()) > 12
    WHEN 't' THEN 'Bonsoir '|| nom
    ELSE 'Bonjour '|| nom
  END::text;
$$ LANGUAGE SQL;
```

3.17.6 Inversion de chaîne

> Écrire une fonction `inverser` qui inverse une chaîne (pour « toto » en entrée, afficher « otot » en sortie), à l'aide d'une boucle `WHILE` et des fonctions `char_length` et `substring`.

```
CREATE OR REPLACE FUNCTION inverser(str_in varchar)
RETURNS varchar
```

```
AS $$
  DECLARE
    str_out varchar ;   -- à renvoyer
    position integer ;
  BEGIN
    -- Initialisation de str_out, sinon sa valeur reste à NULL
    str_out := '';
    -- Position initialisée ç la longueur de la chaîne
    position := char_length(str_in);
    -- La chaîne est traitée ç l'envers
    -- Boucle: Inverse l'ordre des caractères d'une chaîne de caractères
    WHILE position > 0 LOOP
    -- la chaîne donnée en argument est parcourue
    -- à l'envers,
    -- et les caractères sont extraits individuellement
      str_out := str_out || substring(str_in, position, 1);
      position := position - 1;
    END LOOP;
    RETURN str_out;
  END;
$$
LANGUAGE plpgsql;

SELECT inverser (' toto ') ;

 inverser
----------
  otot
```

3.17.7 Jours fériés

La fonction suivante calcule la date de Pâques d'une année :

```
CREATE OR REPLACE FUNCTION paques (annee integer)
RETURNS date
AS $$
  DECLARE
    a integer ;
    b integer ;
    r date ;
  BEGIN
    a := (19*(annee % 19) + 24) % 30 ;
    b := (2*(annee % 4) + 4*(annee % 7) + 6*a + 5) % 7 ;
    SELECT (annee::text||'-03-31')::date + (a+b-9) INTO r ;
    RETURN r ;
  END ;
$$
LANGUAGE plpgsql ;
```

> Afficher les dates de Pâques de 2018 à 2025.

```
SELECT paques (n) FROM generate_series (2018, 2025) n ;

   paques
------------
```

```
 2018-04-01
 2019-04-21
 2020-04-12
 2021-04-04
 2022-04-17
 2023-04-09
 2024-03-31
 2025-04-20
```

> Écrire une fonction qui calcule la date de l'Ascension, soit le jeudi de la sixième semaine après Pâques. Pour simplifier, on peut aussi considérer que l'Ascension se déroule 39 jours après Pâques.

Version complexe :

```
CREATE OR REPLACE FUNCTION ascension(annee integer)
RETURNS date
AS $$
  DECLARE
    r    date;
  BEGIN
    SELECT paques(annee)::date + 40 INTO r;
    SELECT r + (4 - extract(dow from r))::integer INTO r;
    RETURN r;
  END;
$$
LANGUAGE plpgsql;
```

Version simple :

```
CREATE OR REPLACE FUNCTION ascension(annee integer)
RETURNS date
AS $$
    SELECT (paques (annee) + INTERVAL '39 days')::date ;
$$
LANGUAGE sql;
```

Test :

```
SELECT paques (n), ascension(n) FROM generate_series (2018, 2025) n ;
```

```
   paques   | ascension
------------+------------
 2018-04-01 | 2018-05-10
 2019-04-21 | 2019-05-30
 2020-04-12 | 2020-05-21
 2021-04-04 | 2021-05-13
 2022-04-17 | 2022-05-26
 2023-04-09 | 2023-05-18
 2024-03-31 | 2024-05-09
 2025-04-20 | 2025-05-29
```

Pour écrire une fonction qui renvoie tous les jours fériés d'une année (libellé et date), en France métropolitaine :

- Prévoir un paramètre supplémentaire pour l'Alsace-Moselle, où le Vendredi saint (précédant le dimanche de Pâques) et le 26 décembre sont aussi fériés.
- Cette fonction doit renvoyer plusieurs lignes : utiliser `RETURN NEXT`.
- Plusieurs variantes sont possibles : avec `SETOF record`, avec des paramètres OUT, ou avec `RETURNS TABLE (libelle, jour)`.
- Enfin, il est possible d'utiliser `RETURN QUERY`.

Version avec SETOF record :

```
CREATE OR REPLACE FUNCTION vacances(
    annee integer, alsace_moselle boolean DEFAULT false )
RETURNS SETOF record
AS $$
  DECLARE
    f integer;
    r record;
  BEGIN
    SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Pâques'::text, paques(annee)::date + 1 INTO r;
    RETURN NEXT r;
    SELECT 'Ascension'::text, ascension(annee)::date INTO r;
    RETURN NEXT r;
    SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date INTO r;
    RETURN NEXT r;
    SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date INTO r;
    RETURN NEXT r;
    SELECT 'Assomption'::text, (annee::text||'-08-15')::date INTO r;
    RETURN NEXT r;
    SELECT 'La toussaint'::text, (annee::text||'-11-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date INTO r;
    RETURN NEXT r;
    SELECT 'Noël'::text, (annee::text||'-12-25')::date INTO r;
    RETURN NEXT r;
    IF alsace_moselle THEN
      SELECT 'Vendredi saint'::text, paques(annee)::date - 2 INTO r;
      RETURN NEXT r;
      SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date INTO r;
      RETURN NEXT r;
    END IF;

    RETURN;
  END;
$$
LANGUAGE plpgsql;
```

Le requêtage implique de nommer les colonnes :

```
SELECT *
FROM vacances(2020, true) AS (libelle text, jour date)
ORDER BY jour ;
```

```
      libelle       |    jour
--------------------+------------
 Jour de l'an       | 2020-01-01
 Vendredi saint     | 2020-04-10
 Pâques             | 2020-04-13
 Fête du travail    | 2020-05-01
 Victoire 1945      | 2020-05-08
 Ascension          | 2020-05-21
 Fête nationale     | 2020-07-14
 Assomption         | 2020-08-15
 La toussaint       | 2020-11-01
 Armistice 1918     | 2020-11-11
 Noël               | 2020-12-25
 Lendemain de Noël  | 2020-12-26
```

Version avec paramètres OUT :

Une autre forme d'écriture possible consiste à indiquer les deux colonnes de retour comme des paramètres OUT :

```
CREATE OR REPLACE FUNCTION vacances(
            annee integer,
            alsace_moselle boolean DEFAULT false,
            OUT libelle text,
            OUT jour date)
 RETURNS SETOF record
 LANGUAGE plpgsql
AS $function$
  DECLARE
    f integer;
    r record;
  BEGIN
    SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Pâques'::text, paques(annee)::date + 1 INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Ascension'::text, ascension(annee)::date INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Assomption'::text, (annee::text||'-08-15')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'La toussaint'::text, (annee::text||'-11-01')::date
```

```
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Noël'::text, (annee::text||'-12-25')::date INTO libelle, jour;
    RETURN NEXT;
    IF alsace_moselle THEN
      SELECT 'Vendredi saint'::text, paques(annee)::date - 2 INTO libelle, jour;
      RETURN NEXT;
      SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date
        INTO libelle, jour;
      RETURN NEXT;
    END IF;

    RETURN;
  END;
$function$;
```

La fonction s'utilise alors de façon simple :

```
SELECT *
FROM vacances(2020)
ORDER BY jour ;
```

```
     libelle     |    jour
-----------------+------------
 Jour de l'an    | 2020-01-01
 Pâques          | 2020-04-13
 Fête du travail | 2020-05-01
 Victoire 1945   | 2020-05-08
 Ascension       | 2020-05-21
 Fête nationale  | 2020-07-14
 Assomption      | 2020-08-15
 La toussaint    | 2020-11-01
 Armistice 1918  | 2020-11-11
 Noël            | 2020-12-25
```

Version avec RETURNS TABLE :

Seule la déclaration en début diffère de la version avec les paramètres OUT :

```
CREATE OR REPLACE FUNCTION vacances(
            annee integer,alsace_moselle boolean DEFAULT false)
 RETURNS TABLE (libelle text, jour date)
 LANGUAGE plpgsql
AS $function$
…
```

L'utilisation est aussi simple que la version précédente.

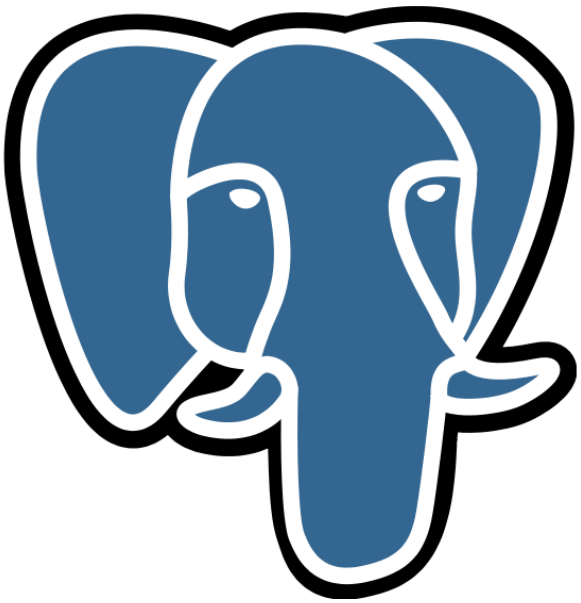
Version avec RETURN QUERY :

C'est peut-être la version la plus compacte :

```
CREATE OR REPLACE FUNCTION vacances(annee integer,alsace_moselle boolean DEFAULT
 ↪  false)
  RETURNS TABLE (libelle text, jour date)
```

```
  LANGUAGE plpgsql
  AS $function$
  BEGIN
    RETURN QUERY SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date ;
    RETURN QUERY SELECT 'Pâques'::text, paques(annee)::date + 1 ;
    RETURN QUERY SELECT 'Ascension'::text, ascension(annee)::date ;
    RETURN QUERY SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date ;
    RETURN QUERY SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date ;
    RETURN QUERY SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date ;
    RETURN QUERY SELECT 'Assomption'::text, (annee::text||'-08-15')::date ;
    RETURN QUERY SELECT 'La toussaint'::text, (annee::text||'-11-01')::date ;
    RETURN QUERY SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date ;
    RETURN QUERY SELECT 'Noël'::text, (annee::text||'-12-25')::date ;
    IF alsace_moselle THEN
      RETURN QUERY SELECT 'Vendredi saint'::text, paques(annee)::date - 2 ;
      RETURN QUERY SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date ;
    END IF;
    RETURN;
  END;
$function$;
```

4/ PL/pgSQL avancé



4.1 PRÉAMBULE

4.1.1 Au menu

- Routines « variadic » et polymorphes
- Fonctions trigger
- Curseurs
- Récupérer les erreurs
- Messages d'erreur dans les logs
- Sécurité
- Optimisation
- Problèmes fréquents

4.1.2 Objectifs

- Connaître la majorité des possibilités de PL/pgSQL
- Les utiliser pour étendre les fonctionnalités de la base
- Écrire du code robuste
- Éviter les pièges de sécurité
- Savoir optimiser une routine

4.2 ROUTINES VARIADIC

4.2.1 Routines variadic : introduction

- Permet de créer des routines avec un nombre d'arguments variables
- … mais du même type

L'utilisation du mot clé `VARIADIC` dans la déclaration des routines permet d'utiliser un nombre variable d'arguments dans la mesure où tous les arguments optionnels sont du même type de données. Ces arguments sont passés à la fonction sous forme de tableau d'arguments du même type.

```
VARIADIC tableau text[]
```

Il n'est pas possible d'utiliser d'autres arguments en entrée à la suite d'un paramètre `VARIADIC`.

4.2.2 Routines variadic : exemple

Récupérer le minimum d'une liste :

```
CREATE FUNCTION pluspetit(VARIADIC numeric[])
RETURNS numeric AS $$
SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT pluspetit(10, -1, 5, 4.4);
 pluspetit
-----------
        -1
(1 row)
```

Quelques explications sur cette fonction :

- SQL est un langage de routines stockées
  - une routine SQL ne contient que des ordres SQL exécutés séquentiellements
  - le résultat de la fonction est le résultat du dernier ordre
- `generate_subscript()` prend un tableau en premier paramètre et la dimension de ce tableau (un tableau peut avoir plusieurs dimensions), et elle retourne une série d'entiers allant du premier au dernier indice du tableau dans cette dimension
- g(i) est un alias : `generate_subscripts` est une SRF (`set-returning function`, retourne un SETOF), g est donc le nom de l'alias de table, et i le nom de l'alias de colonne.

4.2.3 Routines variadic : exemple PL/pgSQL

- En PL/pgSQL, cette fois-ci
- Démonstration de `FOREACH xxx IN ARRAY aaa LOOP`
- Précédemment, obligé de convertir le tableau en relation pour boucler (`unnest`)

En PL/pgSQL, il est possible d'utiliser une boucle FOREACH pour parcourir directement le tableau des arguments optionnels.

```
CREATE OR REPLACE FUNCTION pluspetit(VARIADIC liste numeric[])
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
DECLARE
  courant numeric;
  plus_petit numeric;
BEGIN
FOREACH courant IN ARRAY liste LOOP
  IF plus_petit IS NULL OR courant < plus_petit THEN
    plus_petit := courant;
  END IF;
END LOOP;
RETURN plus_petit;
END
$function$;
```

Auparavant, il fallait développer le tableau avec la fonction `unnest()` pour réaliser la même opération.

```
CREATE OR REPLACE FUNCTION pluspetit(VARIADIC liste numeric[])
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
DECLARE
  courant numeric;
  plus_petit numeric;
BEGIN
FOR courant IN SELECT unnest(liste) LOOP
  IF plus_petit IS NULL OR courant < plus_petit THEN
    plus_petit := courant;
  END IF;
END LOOP;
RETURN plus_petit;
END
$function$;
```

4.3 ROUTINES POLYMORPHES

4.3.1 Routines polymorphes : introduction

- Typer les variables oblige à dupliquer les routines communes à plusieurs types
- PostgreSQL propose des types polymorphes
- Le typage se fait à l'exécution

Pour pouvoir utiliser la même fonction en utilisant des types différents, il est nécessaire de la redéfinir avec les différents types autorisés en entrée. Par exemple, pour autoriser l'utilisation de données de type `integer` ou `float` en entrée et retournés par une même fonction, il faut la dupliquer.

```
CREATE OR REPLACE FUNCTION
  addition(var1 integer, var2 integer)
RETURNS integer
AS $$
DECLARE
  somme integer;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION
  addition(var1 float, var2 float)
RETURNS float
AS $$
DECLARE
  somme float;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

L'utilisation de types polymorphes permet d'éviter ce genre de duplications fastidieuses.

4.3.2 Routines polymorphes : anyelement

- Remplace tout type de données simple ou composite
  - pour les paramètres en entrée comme pour les paramètres en sortie
- Tous les paramètres et type de retour de type `anyelement` se voient attribués le même type
- Donc un seul type pour tous les `anyelement` autorisés
- Paramètre spécial `$0` : du type attribué aux éléments `anyelement`

4.3.3 Routines polymorphes : anyarray

- `anyarray` remplace tout tableau de type de données simple ou composite
  - pour les paramètres en entrée comme pour les paramètres en sortie
- Le typage se fait à l'exécution
- Tous les paramètres de type `anyarray` se voient attribués le même type

4.3.4 Routines polymorphes : exemple

L'addition est un exemple fréquent :

```
CREATE OR REPLACE FUNCTION
  addition(var1 anyelement, var2 anyelement)
RETURNS anyelement
AS $$
DECLARE
  somme ALIAS FOR $0;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

4.3.5 Routines polymorphes : tests

```
# SELECT addition(1, 3);
 addition
----------
        4
(1 row)

# SELECT addition(1.3, 3.5);
 addition
----------
      4.8
(1 row)
```

L'opérateur + étant défini pour les entiers comme pour les `numeric`, la fonction ne pose aucun problème pour ces deux types de données, et retourne une donnée du même type que les données d'entrée.

4.3.6 Routines polymorphes : problème

- Attention lors de l'utilisation de type polymorphe…

```
# SELECT addition('un'::text, 'mot'::text);
ERREUR:  L'opérateur n'existe pas : text + text
LIGNE 1 : SELECT   $1  +  $2
^
ASTUCE : Aucun opérateur correspond au nom donné et aux types d'arguments.
    Vous devez ajouter des conversions explicites de type.
REQUÊTE : SELECT   $1  +  $2
CONTEXTE : PL/pgSQL function "addition" line 4 at assignment
```

Le typage n'étant connu qu'à l'exécution, c'est aussi à ce moment que se déclenchent les erreurs.

De même, l'affectation du type unique pour tous les éléments se fait sur la base du premier élément, ainsi :

```
# SELECT addition(1, 3.5);
ERROR:  function addition(integer, numeric) does not exist
LIGNE 1 : SELECT addition(1, 3.5);
                 ^
ASTUCE : No function matches the given name and argument types.
         You might need to add explicit type casts.
```

génère une erreur car du premier argument est déduit le type `integer`, ce qui n'est évidement pas le cas du deuxième. Il peut donc être nécessaire d'utiliser une conversion explicite pour résoudre ce genre de problématique.

```
# SELECT addition(1::numeric, 3.5);
 addition
----------
      4.5
```

4.4 FONCTIONS TRIGGER

4.4.1 Fonctions trigger : introduction

- Fonction stockée
- Action déclenchée par `INSERT` (incluant `COPY`), `UPDATE`, `DELETE`, `TRUNCATE`
- Mode **par ligne** ou **par instruction**
- Exécution d'une fonction stockée codée à partir de tout langage de procédure activée dans la base de données

Un trigger est une spécification précisant que la base de données doit exécuter une fonction particulière quand un certain type d'opération est traité. Les fonctions trigger peuvent être définies pour s'exécuter avant ou après une commande `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE`.

La fonction trigger doit être définie avant que le trigger lui-même puisse être créé. La fonction trigger doit être déclarée comme une fonction ne prenant aucun argument et retournant un type `trigger`.

Une fois qu'une fonction trigger est créée, le trigger est créé avec `CREATE TRIGGER`. La même fonction trigger est utilisable par plusieurs triggers.

Un trigger `TRUNCATE` ne peut utiliser que le mode par instruction, contrairement aux autres triggers pour lesquels vous avez le choix entre « par ligne » et « par instruction ».

Enfin, l'instruction `COPY` est traitée comme s'il s'agissait d'une commande `INSERT`.

À noter que les problématiques de visibilité et de volatilité depuis un trigger sont assez complexes dès lors que l'on lit ou modifie les données. Voir la documentation[1] pour plus de détails à ce sujet.

[1] https://docs.postgresql.fr/current/trigger-datachanges.html

4.4.2 Fonctions trigger : variables (1/5)

- `OLD` :
  - type de données `RECORD` correspondant à la ligne avant modification
  - valable pour un `DELETE` et un `UPDATE`
- `NEW` :
  - type de données `RECORD` correspondant à la ligne après modification
  - valable pour un `INSERT` et un `UPDATE`

4.4.3 Fonctions trigger : variables (2/5)

- Ces deux variables sont valables uniquement pour les triggers en mode ligne
  - pour les triggers en mode instruction, la version 10 propose les tables de transition
- Accès aux champs par la notation pointée
  - `NEW.champ1` pour accéder à la nouvelle valeur de `champ1`

4.4.4 Fonctions trigger : variables (3/5)

- `TG_NAME`
  - nom du trigger qui a déclenché l'appel de la fonction
- `TG_WHEN`
  - chaîne valant `BEFORE`, `AFTER` ou `INSTEAD OF` suivant le type du trigger
- `TG_LEVEL`
  - chaîne valant `ROW` ou `STATEMENT` suivant le mode du trigger
- `TG_OP`
  - chaîne valant `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` suivant l'opération qui a déclenché le trigger

4.4.5 Fonctions trigger : variables (4/5)

- `TG_RELID`
  - `OID` de la table qui a déclenché le trigger
- `TG_TABLE_NAME`
  - nom de la table qui a déclenché le trigger
- `TG_TABLE_SCHEMA`
  - nom du schéma contenant la table qui a déclenché le trigger

Vous pourriez aussi rencontrer dans du code la variable `TG_RELNAME`. C'est aussi le nom de la table qui a déclenché le trigger. Attention, cette variable est obsolète, il est préférable d'utiliser maintenant `TG_TABLE_NAME`.

4.4.6 Fonctions trigger : variables (5/5)

- TG_NARGS
  - nombre d'arguments donnés à la fonction trigger
- TG_ARGV
  - les arguments donnés à la fonction trigger (le tableau commence à 0)

La fonction trigger est déclarée sans arguments mais il est possible de lui en passer dans la déclaration du trigger. Dans ce cas, il faut utiliser les deux variables ci-dessus pour y accéder. Attention, tous les arguments sont convertis en texte. Il faut donc se cantonner à des informations simples, sous peine de compliquer le code.

```
CREATE OR REPLACE FUNCTION verifier_somme()
RETURNS trigger AS $$
DECLARE
    fact_limit integer;
    arg_color varchar;
BEGIN
    fact_limit   := TG_ARGV[0];

    IF NEW.somme > fact_limit THEN
       RAISE NOTICE 'La facture % necessite une verification. '
                    'La somme % depasse la limite autorisee de %.',
                    NEW.idfact, NEW.somme, fact_limit;
    END IF;

    NEW.datecreate := current_timestamp;

    return NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER trig_verifier_debit
   BEFORE INSERT OR UPDATE ON test
   FOR EACH ROW
   EXECUTE PROCEDURE verifier_somme(400);

CREATE TRIGGER trig_verifier_credit
   BEFORE INSERT OR UPDATE ON test
   FOR EACH ROW
   EXECUTE PROCEDURE verifier_somme(800);
```

4.4.7 Fonctions trigger : retour

- Une fonction trigger a un type de retour spécial, `trigger`
- Trigger ROW, BEFORE :
  - si retour NULL, annulation de l'opération, sans déclencher d'erreur
  - sinon, poursuite de l'opération avec cette valeur de ligne
  - attention au `RETURN NEW;` avec trigger `BEFORE DELETE`
- Trigger ROW, AFTER : valeur de retour ignorée
- Trigger STATEMENT : valeur de retour ignorée
- Pour ces deux derniers cas, annulation possible dans le cas d'une erreur à l'exécution de la fonction (que vous pouvez déclencher dans le code du trigger)

Une fonction trigger retourne le type spécial `trigger`. Pour cette raison, ces fonctions ne peuvent être utilisées que dans le contexte d'un ou plusieurs triggers. Pour pouvoir être utilisée comme valeur de retour dans la fonction (avec RETURN), une variable doit être de structure identique à celle de la table sur laquelle le trigger a été déclenché. Les variables spéciales OLD (ancienne valeur avant application de l'action à l'origine du déclenchement) et NEW (nouvelle valeur après application de l'action) sont également disponibles, utilisables et même modifiables.

La valeur de retour d'un trigger de type ligne (ROW) déclenché avant l'opération (BEFORE) peut changer complètement l'effet de la commande ayant déclenché le trigger. Par exemple, il est possible d'annuler complètement l'action sans erreur (et d'empêcher également tout déclenchement ultérieur d'autres triggers pour cette même action) en retournant NULL. Il est également possible de changer les valeurs de la nouvelle ligne créée par une action INSERT ou UPDATE en retournant une des valeurs différentes de NEW (ou en modifiant NEW directement). Attention, dans le cas d'une fonction trigger BEFORE déclenchée par une action DELETE, in faut prendre en compte que NEW contient NULL, en conséquence `RETURN NEW;` provoquera l'annulation du DELETE ! Dans ce cas, si on désire laisser l'action inchangée, la convention est de faire un `RETURN OLD;`.

En revanche, la valeur de retour utilisée n'a pas d'effet dans les cas des triggers ROW et AFTER, et des triggers STATEMENT. À noter que bien que la valeur de retour soit ignorée dans ce cas, il est possible d'annuler l'action d'un trigger de type ligne intervenant après l'opération ou d'un trigger à l'instruction en remontant une erreur à l'exécution de la fonction.

4.4.8 Fonctions trigger : exemple - 1

- Horodater une opération sur une ligne

```
CREATE TABLE ma_table (
id serial,
-- un certain nombre de champs informatifs
date_ajout timestamp,
date_modif timestamp);
```

4.4.9 Fonctions trigger : exemple - 2

```
CREATE OR REPLACE FUNCTION horodatage() RETURNS trigger
AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    NEW.date_ajout := now();
  ELSEIF TG_OP = 'UPDATE' THEN
    NEW.date_modif := now();
  END IF;
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

4.4.10 Options de CREATE TRIGGER

`CREATE TRIGGER` permet quelques variantes :

- `CREATE TRIGGER name WHEN ( condition )`
- `CREATE TRIGGER name BEFORE UPDATE OF colx ON my_table`
- `CREATE CONSTRAINT TRIGGER` : exécuté qu'au moment de la validation de la transaction
- `CREATE TRIGGER view_insert INSTEAD OF INSERT ON my_view`

- On peut ne déclencher un trigger que si une condition est vérifiée. Cela simplifie souvent le code du trigger, et gagne en performances : plus besoin pour le moteur d'aller exécuter la fonction.

- On peut ne déclencher un trigger que si une colonne spécifique a été modifiée. Il ne s'agit donc que de triggers sur UPDATE. Encore un moyen de simplifier le code et de gagner en performances en évitant les déclenchements inutiles.
- On peut créer un trigger en le déclarant comme étant un trigger de contrainte. Il peut alors être *deferrable*, *deferred*, comme tout autre contrainte, c'est-à-dire n'être exécuté qu'au moment de la validation de la transaction, ce qui permet de ne vérifier les contraintes implémentées par le trigger qu'au moment de la validation finale.
- On peut créer un trigger sur une vue. C'est un trigger `INSTEAD OF`, qui permet de programmer de façon efficace les `INSERT`/`UPDATE`/`DELETE`/`TRUNCATE` sur les vues. Auparavant, il fallait passer par le système de règles (`RULES`), complexe et sujet à erreurs.

4.4.11 Tables de transition

- Pour les triggers de type AFTER et de niveau statement
- Possibilité de stocker les lignes avant et/ou après modification
  - REFERENCING OLD TABLE
  - REFERENCING NEW TABLE
- Par exemple :

```
CREATE TRIGGER tr1
AFTER DELETE ON t1
REFERENCING OLD TABLE AS oldtable
FOR EACH STATEMENT
EXECUTE PROCEDURE log_delete();
```

Dans le cas d'un trigger en mode instruction, il n'est pas possible d'utiliser les variables `OLD` et `NEW` car elles ciblent une seule ligne. Pour cela, le standard SQL parle de tables de transition.

La version 10 de PostgreSQL permet donc de rattraper le retard à ce sujet par rapport au standard SQL et SQL Server.

Voici un exemple de leur utilisation.

Nous allons créer une table t1 qui aura le trigger et une table archives qui a pour but de récupérer les enregistrements supprimés de la table t1.

```
CREATE TABLE t1 (c1 integer, c2 text);

CREATE TABLE archives (id integer GENERATED ALWAYS AS IDENTITY,
      dlog timestamp DEFAULT now(),
      t1_c1 integer, t1_c2 text);
```

Maintenant, il faut créer le code de la procédure stockée :

```
CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
      BEGIN
        INSERT INTO archives (t1_c1, t1_c2) SELECT c1, c2 FROM oldtable;
        RETURN null;
      END
      $$;
```

Et ajouter le trigger sur la table t1 :

```
CREATE TRIGGER tr1
      AFTER DELETE ON t1
      REFERENCING OLD TABLE AS oldtable
      FOR EACH STATEMENT
      EXECUTE PROCEDURE log_delete();
```

Maintenant, insérons un million de ligne dans t1 et supprimons-les :

```
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

DELETE FROM t1;
Time: 2141.871 ms (00:02.142)
```

La suppression avec le trigger prend 2 secondes. Il est possible de connaître le temps à supprimer les lignes et le temps à exécuter le trigger en utilisant l'ordre `EXPLAIN ANALYZE` :

```
TRUNCATE archives;

INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

EXPLAIN (ANALYZE) DELETE FROM t1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Delete on t1  (cost=0.00..14241.98 rows=796798 width=6)
               (actual time=781.612..781.612 rows=0 loops=1)
   ->  Seq Scan on t1  (cost=0.00..14241.98 rows=796798 width=6)
                       (actual time=0.113..104.328 rows=1000000 loops=1)
 Planning time: 0.079 ms
 Trigger tr1: time=1501.688 calls=1
 Execution time: 2287.907 ms
(5 rows)
```

Donc la suppression des lignes met 0,7 seconde alors que l'exécution du trigger met 1,5 seconde.

Pour comparer, voici l'ancienne façon de faire (configuration d'un trigger en mode ligne) :

```
CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
      BEGIN
        INSERT INTO archives (t1_c1, t1_c2) VALUES (old.c1, old.c2);
        RETURN null;
      END
      $$;

DROP TRIGGER tr1 ON t1;

CREATE TRIGGER tr1
      AFTER DELETE ON t1
      FOR EACH ROW
```

```
      EXECUTE PROCEDURE log_delete();

TRUNCATE archives;

TRUNCATE t1;

INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

DELETE FROM t1;
Time: 8445.697 ms (00:08.446)

TRUNCATE archives;

INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

EXPLAIN (ANALYZE) DELETE FROM t1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Delete on t1  (cost=0.00..14241.98 rows=796798 width=6)
               (actual time=1049.420..1049.420 rows=0 loops=1)
   ->  Seq Scan on t1  (cost=0.00..14241.98 rows=796798 width=6)
                       (actual time=0.061..121.701 rows=1000000 loops=1)
 Planning time: 0.096 ms
 Trigger tr1: time=7709.725 calls=1000000
 Execution time: 8825.958 ms
(5 rows)
```

Donc avec un trigger en mode ligne, la suppression du million de lignes met presque 9 secondes à s'exécuter, dont 7,7 pour l'exécution du trigger. Sur le trigger en mode instruction, il faut compter 2,2 secondes, dont 1,5 sur le trigger. Les tables de transition nous permettent de gagner en performance.

Le gros intérêt des tables de transition est le gain en performance que cela apporte.

4.5 CURSEURS

4.5.1 Curseurs : introduction

- Exécuter une requête en une fois peut ramener beaucoup de résultats
- Tout ce résultat est en mémoire
  - risque de dépassement mémoire
- La solution : les curseurs
- Un curseur permet d'exécuter la requête sur le serveur mais de ne récupérer les résultats que petit bout par petit bout
- Dans une transaction ou une routine

À noter que la notion de curseur existe aussi en SQL pur, sans passer par une routine PL/pgSQL. On les crée en utilisant la commande DECLARE, et les règles de manipulation sont légèrement différentes (on peut par exemple créer un curseur WITH HOLD, qui persistera après la fin de la transaction). Voir la documentation pour plus d'informations à ce sujet : https://docs.postgresql.fr/current/sql-declare.html

4.5.2 Curseurs : déclaration d'un curseur

- Avec le type refcursor
- Avec la pseudo-instruction CURSOR FOR
- Avec une requête paramétrée
- Exemples :

```
curseur1 refcursor;
curseur2 CURSOR FOR SELECT * FROM ma_table;
curseur3 CURSOR (param integer) IS
SELECT * FROM ma_table WHERE un_champ=param;
```

La première forme permet la création d'un curseur non lié à une requête.

4.5.3 Curseurs : ouverture d'un curseur

- Lier une requête à un curseur :

```
OPEN curseur FOR requete
```

- Plan de la requête mis en cache
- Lier une requête dynamique à un curseur

```
OPEN curseur FOR EXECUTE chaine_requete
```

Voici un exemple de lien entre une requête et un curseur :

```
OPEN curseur FOR SELECT * FROM ma_table;
```

Et voici un exemple d'utilisation d'une requête dynamique :

```
OPEN curseur FOR EXECUTE 'SELECT * FROM ' || quote_ident(TG_TABLE_NAME);
```

4.5.4 Curseurs : ouverture d'un curseur lié

- Instruction SQL : `OPEN curseur(arguments)`
- Permet d'ouvrir un curseur déjà lié à une requête
- Impossible d'ouvrir deux fois le même curseur
- Plan de la requête mise en cache
- Exemple

```
curseur CURSOR FOR SELECT * FROM ma_table;
...
OPEN curseur;
```

4.5.5 Curseurs : récupération des données

- Instruction SQL :

```
FETCH [ direction { FROM | IN } ] curseur INTO cible
```

- Récupère la prochaine ligne
- FOUND indique si cette nouvelle ligne a été récupérée
- Cible est
  - une variable RECORD
  - une variable ROW
  - un ensemble de variables séparées par des virgules

4.5.6 Curseurs : récupération des données

- `direction` du `FETCH` :
  - `NEXT`, `PRIOR`
  - `FIRST`, `LAST`
  - `ABSOLUTE nombre`, `RELATIVE nombre`
  - nombre
  - `ALL`
  - `FORWARD`, `FORWARD nombre`, `FORWARD ALL`
  - `BACKWARD`, `BACKWARD nombre`, `BACKWARD ALL`

4.5.7 Curseurs : modification des données

- Mise à jour d'une ligne d'un curseur :

```
UPDATE une_table SET ...
WHERE CURRENT OF curseur;
```

- Suppression d'une ligne d'un curseur :

```
DELETE FROM une_table
WHERE CURRENT OF curseur;
```

Attention, ces différentes syntaxes ne modifient pas les données dans le curseur en mémoire, mais font réellement la modification dans la table. L'emplacement actuel du curseur est utilisé ici pour identifier la ligne correspondante à mettre à jour.

4.5.8 Curseurs : fermeture d'un curseur

- Instruction SQL : `CLOSE curseur`
- Ferme le curseur
- Permet de récupérer de la mémoire
- Permet aussi de réouvrir le curseur

4.5.9 Curseurs : renvoi d'un curseur

- Fonction renvoyant une valeur de type `refcursor`
- Permet donc de renvoyer plusieurs valeurs

Voici un exemple d'utilisation d'une référence de curseur retournée par une fonction :

```
CREATE FUNCTION consult_all_stock(refcursor) RETURNS refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM stock;
    RETURN $1;
```

```
END;
$$ LANGUAGE plpgsql;

-- doit être dans une transaction pour utiliser les curseurs.
BEGIN;

SELECT * FROM consult_all_stock('cursor_a');

FETCH ALL FROM cursor_a;
COMMIT;
```

4.6 CONTRÔLE TRANSACTIONNEL

- Procédures uniquement !
- COMMIT et ROLLBACK
- Pas de BEGIN
  - automatique après la fin d'une transaction
- Ne fonctionne pas à l'intérieur d'une transaction
- Incompatible avec une clause EXCEPTION

Voici un exemple avec COMMIT ou ROLLBACK suivant que le nombre est pair ou impair :

```
CREATE TABLE test1 (a int) ;

CREATE OR REPLACE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
  FOR i IN 0..5 LOOP
    INSERT INTO test1 (a) VALUES (i);
    IF i % 2 = 0 THEN
      COMMIT;
    ELSE
      ROLLBACK;
    END IF;
  END LOOP;
END
$$;

CALL transaction_test1();

SELECT * FROM test1;
 a | b
---+---
 0 |
 2 |
 4 |
(3 lignes)
```

Une exemple plus fréquemment utilisé est celui d'une procédure effectuant un traitement de modification des données par lots, et donc faisant un COMMIT à intervalle régulier.

Noter qu'il n'y a pas de BEGIN explicite dans la gestion des transactions. Après un COMMIT ou un ROLLBACK, un BEGIN est immédiatement exécuté.

On ne peut pas imbriquer des transactions :

```
BEGIN ; CALL transaction_test1() ;
```

```
ERROR:  invalid transaction termination
CONTEXTE : PL/pgSQL function transaction_test1() line 6 at COMMIT
```

On ne peut pas utiliser en même temps une clause EXCEPTION et le contrôle transactionnel :

```
DO LANGUAGE plpgsql $$
BEGIN
  BEGIN
    INSERT INTO test1 (a) VALUES (1);
  COMMIT;
  INSERT INTO test1 (a) VALUES (1/0);
COMMIT;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'caught division_by_zero';
  END;
END;
$$;
```

```
ERREUR:  cannot commit while a subtransaction is active
CONTEXTE : fonction PL/pgSQL inline_code_block, ligne 5 à COMMIT
```

4.7 GESTION DES ERREURS

4.7.1 Gestion des erreurs : introduction

- Sans exceptions :
  - toute erreur provoque un arrêt de la fonction
  - toute modification suite à une instruction SQL (`INSERT`, `UPDATE`, `DELETE`) est annulée
  - d'où l'ajout d'une gestion personnalisée des erreurs avec le concept des exceptions

4.7.2 Gestion des erreurs : une exception

- La fonction comporte un bloc supplémentaire, `EXCEPTION` :

```
DECLARE
  -- déclaration des variables locales
BEGIN
  -- instructions de la fonction
EXCEPTION
WHEN condition THEN
  -- instructions traitant cette erreur
WHEN condition THEN
  -- autres instructions traitant cette autre erreur
  -- etc.
END
```

4.7.3 Gestion des erreurs : flot dans une fonction

- L'exécution de la fonction commence après le `BEGIN`
- Si aucune erreur ne survient, le bloc `EXCEPTION` est ignoré
- Si une erreur se produit
  - tout ce qui a été modifié dans la base dans le bloc est annulé
  - les variables gardent par contre leur état
  - l'exécution passe directement dans le bloc de gestion de l'exception

4.7.4 Gestion des erreurs : flot dans une exception

- Recherche d'une condition satisfaisante
- Si cette condition est trouvée
  - exécution des instructions correspondantes
- Si aucune condition n'est compatible
  - sortie du bloc `BEGIN/END` comme si le bloc d'exception n'existait pas
  - passage de l'exception au bloc `BEGIN/END` contenant (après annulation de ce que ce bloc a modifié en base)
- Dans un bloc d'exception, les instructions `INSERT`, `UPDATE`, `DELETE` de la fonction ont été annulées
- Dans un bloc d'exception, les variables locales de la fonction ont gardé leur ancienne valeur

4.7.5 Gestion des erreurs : codes d'erreurs

- SQLSTATE : code d'erreur
- SQLERRM : message d'erreur
- Par exemple :
  - `Data Exception` : division par zéro, overflow, argument invalide pour certaines fonctions, etc.
  - `Integrity Constraint Violation` : unicité, CHECK, clé étrangère, etc.
  - `Syntax Error`
  - `PL/pgSQL Error` : `RAISE EXCEPTION`, pas de données, trop de lignes, etc.
- Les erreurs sont contenues dans des classes d'erreurs plus génériques, qui peuvent aussi être utilisées

Toutes les erreurs sont référencées dans la documentation[2]

Attention, des codes d'erreurs nouveaux apparaissent à chaque version.

La classe data_exception contient de nombreuses erreurs, comme datetime_field_overflow, invalid_escape_character, invalid_binary_representation… On peut donc, dans la déclaration de l'exception, intercepter toutes les erreurs de type data_exception d'un coup, ou une par une.

L'instruction `GET STACKED DIAGNOSTICS` permet d'avoir une vision plus précise de l'erreur récupéré par le bloc de traitement des exceptions. La liste de toutes les informations que l'on peut collecter est disponible dans la documentation[3] .

La démonstration ci-dessous montre comment elle peut être utilisée.

```
# CREATE TABLE t5(c1 integer PRIMARY KEY);
CREATE TABLE
# INSERT INTO t5 VALUES (1);
INSERT 0 1
# CREATE OR REPLACE FUNCTION test(INT4) RETURNS void AS $$
DECLARE
    v_state   TEXT;
    v_msg     TEXT;
    v_detail  TEXT;
    v_hint    TEXT;
    v_context TEXT;
BEGIN
    BEGIN
        INSERT INTO t5 (c1) VALUES ($1);
```

[2] https://docs.postgresql.fr/current/errcodes-appendix.html
[3] https://docs.postgresql.fr/current/plpgsql-control-structures.html#plpgsql-exception-diagnostics-values

```
    EXCEPTION WHEN others THEN
        GET STACKED DIAGNOSTICS
            v_state   = RETURNED_SQLSTATE,
            v_msg     = MESSAGE_TEXT,
            v_detail  = PG_EXCEPTION_DETAIL,
            v_hint    = PG_EXCEPTION_HINT,
            v_context = PG_EXCEPTION_CONTEXT;
        raise notice E'Et une exception :
            state  : %
            message: %
            detail : %
            hint   : %
            context: %', v_state, v_msg, v_detail, v_hint, v_context;
    END;
    RETURN;
END;
$$ LANGUAGE plpgsql;
# SELECT test(2);
 test
------

(1 row)

# SELECT test(2);
NOTICE:  Et une exception :
            state  : 23505
            message: duplicate key value violates unique constraint "t5_pkey"
            detail : Key (c1)=(2) already exists.
            hint   :
            context: SQL statement "INSERT INTO t5 (c1) VALUES ($1)"
PL/pgSQL function test(integer) line 10 at SQL statement
 test
------

(1 row)
```

4.7.6 Messages d'erreurs : RAISE - 1

- Envoyer une trace dans les journaux applicatifs et/ou vers le client
  - `RAISE niveau message`
- Niveau correspond au niveau d'importance du message
  - `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `EXCEPTION`
- Message est la trace à enregistrer
- Message dynamique… tout signe `%` est remplacé par la valeur indiquée après le message
- Champs `DETAIL` et `HINT` disponibles

Il convient de noter qu'un message envoyé de cette manière ne fera pas partie de l'éventuel résultat d'une fonction, et ne sera donc pas exploitable en SQL. Pour cela, il faut utiliser l'instruction `RETURN` avec un type de retour approprié.

Le traitement des messages de ce type et leur destination d'envoi sont contrôlés par le serveur à l'aide des paramètres `log_min_messages` et `client_min_messages`.

4.7.7 Messages d'erreurs : RAISE - 2

Exemples :

```
RAISE WARNING 'valeur % interdite', valeur;

RAISE WARNING 'valeur % ambigue',
              valeur
              USING HINT = 'Controlez la valeur saisie en amont';
```

Les autres niveaux pour `RAISE` ne sont que des messages, sans déclenchement d'exception.

4.7.8 Messages d'erreurs : configuration des logs

- Deux paramètres importants pour les traces
- `log_min_messages`
  - niveau minimum pour que la trace soit enregistrée dans les journaux
- `client_min_messages`
  - niveau minimum pour que la trace soit envoyée au client
- Dans le cas d'un `RAISE NOTICE message`, il faut avoir soit `log_min_messages`, soit `client_min_messages`, soit les deux à la valeur `NOTICE` au minimum.

4.7.9 Messages d'erreurs : RAISE EXCEPTION - 1

- Annule le bloc en cours d'exécution
  - `RAISE EXCEPTION message`
- Sauf en cas de présence d'un bloc `EXCEPTION` gérant la condition `RAISE_EXCEPTION`
- message est la trace à enregistrer, et est dynamique… tout signe % est remplacé par la valeur indiquée après le message

4.7.10 Messages d'erreurs : RAISE EXCEPTION - 2

Exemple :

```
RAISE EXCEPTION 'erreur interne';
-- La chose à ne pas faire !
```

Le rôle d'une exception est d'intercepter une erreur pour exécuter un traitement permettant soit de corriger l'erreur, soit de remonter une erreur pertinente. Intercepter un problème pour retourner « erreur interne » n'est pas une bonne idée.

4.7.11 Flux des erreurs dans du code PL

- Les exceptions non traitées «remontent»
  - de bloc BEGIN/END imbriqués vers les blocs parents (fonctions appelantes comprises)
  - jusqu'à ce que personne ne puisse les traiter
  - voir note pour démonstration

Démonstration en plusieurs étapes :

```
# CREATE TABLE ma_table (
    id integer unique
);
CREATE TABLE

# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- Va déclencher une erreur de violation de contrainte d'unicité
  INSERT INTO ma_table VALUES (1);
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
ERROR:  duplicate key value violates unique constraint "ma_table_id_key"
DETAIL:  Key (id)=(1) already exists.
CONTEXT:  SQL statement "INSERT INTO ma_table VALUES (1)"
PL/pgSQL function demo_exception() line 6 at SQL statement
```

Une exception a été remontée avec un message explicite.

```
# SELECT * FROM ma_table ;
 a
(0 row)
```

La fonction a bien été annulée.

4.7.12 Flux des erreurs dans du code PL - 2

- Les erreurs remontent
- Cette fois-ci, on rajoute un bloc PL pour intercepter l'erreur

```
# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- Va déclencher une erreur de violation de contrainte d'unicité
  INSERT INTO ma_table VALUES (1);
EXCEPTION WHEN unique_violation THEN
  RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
  RAISE NOTICE 'erreur: %',sqlerrm;
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
NOTICE:  violation d'unicite, mais celle-ci n'est pas grave
NOTICE:  erreur: duplicate key value violates unique constraint "ma_table_id_key"
 demo_exception
----------------

(1 row)
```

L'erreur est bien devenue un message de niveau `NOTICE`.

```
# SELECT * FROM ma_table ;
 a
(0 row)
```

La table n'en reste pas moins vide pour autant puisque le bloc a été annulé.

4.7.13 Flux des erreurs dans du code PL - 3

- Cette fois-ci, on rajoute un bloc PL indépendant pour gérer le second `INSERT`

Voici une nouvelle version de la fonction :

```
# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- L'operation suivante pourrait échouer.
  -- Il ne faut pas perdre le travail effectué jusqu'à ici
  BEGIN
  -- Va déclencher une erreur de violation de contrainte d'unicité
    INSERT INTO ma_table VALUES (1);
  EXCEPTION WHEN unique_violation THEN
    -- Cette exception est bien celle du bloc imbriqué
    RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
    RAISE NOTICE 'erreur: %',sqlerrm;
  END; -- Fin du bloc imbriqué
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
NOTICE:  violation d'unicite, mais celle-ci n'est pas grave
NOTICE:  erreur: duplicate key value violates unique constraint "ma_table_id_key"
 demo_exception
----------------

(1 row)
```

En apparence, le résultat est identique.

```
# SELECT * FROM ma_table ;
 a
 1
(1 row)
```

Mais cette fois-ci, le bloc BEGIN parent n'a pas eu d'exception, il s'est donc bien terminé.

4.7.14 Flux des erreurs dans du code PL - 4

- Illustrons maintenant la remontée d'erreurs
- Nous avons deux blocs imbriqués
- Une erreur non prévue va se produire dans le bloc intérieur

On commence par ajouter une contrainte sur la colonne pour empêcher les valeurs supérieures ou égales à 10 :

```
# ALTER TABLE ma_table ADD CHECK (id < 10 ) ;
ALTER TABLE
```

Puis, on recrée la fonction de façon à ce qu'elle déclenche cette erreur dans le bloc le plus bas, et la gère uniquement dans le bloc parent :

```
CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- L'operation suivante pourrait échouer.
  -- Il ne faut pas perdre le travail effectué jusqu'à ici
  BEGIN
    -- Va déclencher une erreur de violation de check (col < 10)
    INSERT INTO ma_table VALUES (100);
  EXCEPTION WHEN unique_violation THEN
    -- Cette exception est bien celle du bloc imbriqué
    RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
    RAISE NOTICE 'erreur: %',sqlerrm;
  END; -- Fin du bloc imbriqué
EXCEPTION WHEN check_violation THEN
  RAISE NOTICE 'violation de contrainte check';
  RAISE EXCEPTION 'mais on va remonter une exception à l''appelant, '
                  'juste pour le montrer';
END
$function$;
```

Exécutons la fonction :

```
# SELECT demo_exception();
ERROR:  duplicate key value violates unique constraint "ma_table_id_key"
DETAIL:  Key (id)=(1) already exists.
CONTEXT:  SQL statement "INSERT INTO ma_table VALUES (1)"
PL/pgSQL function demo_exception() line 4 at SQL statement
```

C'est normal, nous avons toujours l'enregistrement à 1 du test précédent. L'exception se déclenche donc dans le bloc parent, sans espoir d'interception: nous n'avons pas d'exception pour lui.

Nettoyons donc la table, pour reprendre le test :

```
# TRUNCATE ma_table ;
TRUNCATE TABLE
# SELECT demo_exception();
NOTICE:  violation de contrainte check
ERREUR:  mais on va remonter une exception à l'appelant, juste pour le montrer
CONTEXT:  PL/pgSQL function demo_exception() line 17 at RAISE
```

Le gestionnaire d'exception qui intercepte l'erreur est bien ici celui de l'appelant. Par ailleurs, comme nous retournons nous-même une exception, la requête ne retourne pas de résultat, mais une erreur : il n'y a plus personne pour récupérer l'exception, c'est donc PostgreSQL lui-même qui s'en charge.

4.8 SÉCURITÉ

4.8.1 Sécurité : droits

- L'exécution de la routine dépend du droit EXECUTE
- Par défaut, ce droit est donné à la création de la routine
  - au propriétaire de la routine
  - au groupe spécial PUBLIC

4.8.2 Sécurité : ajout

- Ce droit peut être donné avec l'instruction SQL GRANT :

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name
    [ ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) ] [, ...  ]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA
↪   schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

4.8.3 Sécurité : suppression

- Un droit peut être révoqué avec l'instruction SQL REVOKE

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } function_name
    [ ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) ] [, ... ]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA
↪   schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

4.8.4 Sécurité : SECURITY INVOKER/DEFINER

- `SECURITY INVOKER`
  - la routine s'exécute avec les droits de l'utilisateur qui l'exécute
- `SECURITY DEFINER`
  - la routine s'exécute avec les droits de l'utilisateur qui en est le propriétaire
  - équivalent du `sudo` Unix
- Il faut impérativement sécuriser les variables d'environnement (surtout le *search path*) en `SECURITY DEFINER`

Exemple d'une fonction en `SECURITY DEFINER` avec un *search path* sécurisé :

```
CREATE OR REPLACE FUNCTION instance_is_in_backup ( )
RETURNS BOOLEAN AS $$
DECLARE is_exists BOOLEAN;
BEGIN
        -- Set a secure search_path: trusted schemas, then 'pg_temp'.
        PERFORM pg_catalog.set_config('search_path', 'pg_temp', true);
        SELECT ((pg_stat_file('backup_label')).modification IS NOT NULL)
        INTO is_exists;
        RETURN is_exists;
EXCEPTION
WHEN undefined_file THEN
  RETURN false;
END
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

4.8.5 Sécurité : LEAKPROOF

- `LEAKPROOF`
  - indique au planificateur que la routine ne peut pas faire fuiter d'information de contexte
  - réservé aux superutilisateurs
  - si on la déclare telle, s'assurer que la routine est véritablement sûre !
- Option utile lorsque l'on utilise des vues avec l'option `security_barrier`

Certains utilisateurs créent des vues pour filtrer des lignes, afin de restreindre la visibilité sur certaines données. Or, cela peut se révéler dangereux si un utilisateur malintentionné a la possibilité de créer une fonction car il peut facilement contourner cette sécurité si cette option n'est pas utilisée, notamment en jouant sur des paramètres de fonction comme `COST`, qui permet d'indiquer au planificateur un coût estimé pour la fonction.

En indiquant un coût extrêmement faible, le planificateur aura tendance à réécrire la requête, et à déplacer l'exécution de la fonction dans le code même de la vue, avant l'application des filtres restreignant l'accès aux données : la fonction a donc accès a tout le contenu de la table, et peut faire fuiter des données normalement inaccessibles, par exemple à travers l'utilisation de la commande `RAISE`.

L'option `security_barrier` des vues dans PostgreSQL bloque ce comportement du planificateur, mais en conséquence empêche le choix de plans d'exécutions potentiellement plus performants. Déclarer une fonction avec l'option `LEAKPROOF` permet d'indiquer à PostgreSQL que celle-ci ne peut pas occasionner de fuite d'informations. Ainsi, le planificateur de PostgreSQL sait qu'il peut en optimiser l'exécution. Cette option n'est accessible qu'aux superutilisateurs.

4.8.6 Sécurité : visibilité des sources - 1

- Le code d'une fonction est visible par tout le monde
  - y compris ceux qui n'ont pas le droit d'exécuter la fonction
- Vous devez donc écrire un code robuste
  - pas espérer que, comme personne n'en a le code, personne ne trouvera de faille
- **Surtout pour les fonctions SECURITY DEFINER**

4.8.7 Sécurité : visibilité des sources - 2

```
# SELECT proargnames, prosrc
FROM pg_proc WHERE proname='addition';

-[ RECORD 1 ]--------------------------
proargnames | {var1,var2}
prosrc      |
            :   DECLARE
            :     somme ALIAS FOR $0;
            :   BEGIN
            :     somme := var1 + var2;
            :     RETURN somme;
            :   END;
            :
```

La méta-commande psql `\df+ public.addition` permet également d'obtenir cette information.

4.8.8 Sécurité : Injections SQL

- Les paramètres d'une routine doivent être considérés comme hostiles :
  - ils contiennent des données non validées (qui appelle la routine ?)
  - ils peuvent, si l'utilisateur est imaginatif, être utilisés pour exécuter du code
- Utiliser `quote_ident`, `quote_literal` et `quote_nullable`
- Utiliser aussi `format`

Voici un exemple simple :

```
CREATE TABLE ma_table_secrete1 (b integer, a integer);
INSERT INTO ma_table_secrete1 SELECT i,i from generate_series(1,20) i;

CREATE OR REPLACE FUNCTION demo_injection ( param1 text, value1 text )
 RETURNS SETOF ma_table_secrete1
 LANGUAGE plpgsql
 SECURITY DEFINER
AS $function$
-- Cette fonction prend un nom de colonne variable
-- et l'utilise dans une clause WHERE
-- Il faut donc une requête dynamique
```

```
-- Par contre, mon utilisateur 'normal' qui appelle
-- n'a droit qu'aux enregistrements où a<10
DECLARE
  ma_requete text;
  ma_ligne record;
BEGIN
  ma_requete := 'SELECT * FROM ma_table_secrete1 WHERE ' || param1 || ' = ' ||
                value1 || ' AND a < 10';
    RETURN QUERY EXECUTE ma_requete;
END
$function$;
```

```
# SELECT * from demo_injection ('b','2');
 a | b
---+---
 2 | 2
(1 row)

# SELECT * from demo_injection ('a','20');
 a | b
---+---
(0 row)
```

Tout va bien, elle effectue ce qui est demandé.

Par contre, elle effectue aussi ce qui n'est pas prévu :

```
# SELECT * from demo_injection ('1=1 --','');
  a  |  b
-----+-----
   1 |   1
   2 |   2
   3 |   3
   4 |   4
   5 |   5
   6 |   6
   7 |   7
   8 |   8
   9 |   9
  10 |  10
  11 |  11
  12 |  12
  13 |  13
  14 |  14
  15 |  15
  16 |  16
  17 |  17
  18 |  18
  19 |  19
  20 |  20
(20 lignes)
```

Cet exemple est évidemment simplifié.

Une règle demeure : ne jamais faire confiance aux paramètres d'une fonction. Au minimum, un `quote_ident` pour param1 et un `quote_literal` pour val1 étaient obligatoires, pour se protéger de ce genre de problèmes.

4.9 OPTIMISATION

4.9.1 Fonctions immutables, stables ou volatiles - 1

- Par défaut, PostgreSQL considère que les fonctions sont `VOLATILE`
- `volatile` : fonction dont l'exécution ne peut ni ne doit être évitée

Les fonctions de ce type sont susceptibles de renvoyer un résultat différent à chaque appel, comme par exemple `random()` ou `setval()`.

Toute fonction ayant des effets de bords doit être qualifiée `volatile` dans le but d'éviter que PostgreSQL utilise un résultat intermédiaire déjà calculé et évite ainsi d'exécuter le code de la fonction.

À noter qu'il est possible de « forcer » le pré-calcul du résultat d'une fonction volatile dans une requête SQL en utilisant une sous-requête. Par exemple, dans l'exemple suivant, `random()` est exécutée pour chaque ligne de la table `ma_table`, et renverra donc une valeur différente par ligne :

```
SELECT random() FROM ma_table;
```

Par contre, en utilisant une sous-requête, l'optimiseur va pré-calculer le résultat de `random()`… l'exécution sera donc plus rapide, mais le résultat différent, puisque la même valeur sera affichée pour toutes les lignes !

```
SELECT ( SELECT random() ) FROM ma_table;
```

4.9.2 Fonctions immutables, stables ou volatiles - 2

- `immutable` : fonctions déterministes, dont le résultat peut être précalculé avant de planifier la requête.

Certaines fonctions que l'on écrit sont déterministes. C'est-à-dire qu'à paramètre(s) identique(s), le résultat est identique.

Le résultat de telles fonctions est alors remplaçable par son résultat avant même de commencer à planifier la requête.

Voici un exemple qui utilise cette particularité :

```
create function factorielle (a integer) returns bigint as
$$
```

```
declare
  result bigint;
begin
  if a=1 then
    return 1;
  else
    return a*(factorielle(a-1));
  end if;
end;
$$
language plpgsql immutable;

# CREATE TABLE test (a bigint UNIQUE);
CREATE TABLE
# INSERT INTO test SELECT generate_series(1,1000000);
INSERT 0 1000000
# ANALYZE test;
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                                  QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..16925.00 rows=1000000 width=8)
                   (actual time=0.032..130.921 rows=1000000 loops=1)
   Filter: (a < '479001600'::bigint)
 Planning time: 896.039 ms
 Execution time: 169.954 ms
(4 rows)
```

La fonction est exécutée une fois, remplacée par sa constante, et la requête est ensuite planifiée.

Si on déclare la fonction comme stable :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                       QUERY PLAN
---------------------------------------------------------
 Index Only Scan using test_a_key on test
        (cost=0.68..28480.67 rows=1000000 width=8)
              (actual time=0.137..115.592 rows=1000000 loops=1)
   Index Cond: (a < factorielle(12))
   Heap Fetches: 0
 Planning time: 4.682 ms
 Execution time: 153.762 ms
(5 rows)
```

La requête est planifiée sans connaître `factorielle(12)`, donc avec une hypothèse très approximative sur la cardinalité. `factorielle(12)` est calculé, et la requête est exécutée. Grâce au `Index Only Scan`, le requête s'effectue rapidement.

Si on déclare la fonction comme volatile :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                                  QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..266925.00 rows=333333 width=8)
                   (actual time=1.005..57519.702 rows=1000000 loops=1)
   Filter: (a < factorielle(12))
 Planning time: 0.388 ms
```

```
 Execution time: 57573.508 ms
(4 rows)
```

La requête est planifiée, et factorielle(12) est calculé pour chaque enregistrement de la table, car on ne sait pas si elle retourne toujours le même résultat.

4.9.3 Fonctions immutables, stables ou volatiles - 3

- `stable` : fonction ayant un comportement **stable** au sein d'un même ordre SQL.

Ces fonctions retournent la même valeur pour la même requête SQL, mais peuvent retourner une valeur différente dans la prochaine instruction.

Il s'agit typiquement de fonctions dont le traitement dépend d'autres valeurs dans la base de données, ou bien de réglages de configuration. Les fonctions comme `to_char()`, `to_date()` sont `STABLE` et non `IMMUTABLE` car des paramètres de configuration (*locale* utilisée pour `to_char()`, *timezone* pour les fonctions temporelles, etc.) pourraient influer sur le résultat.

À noter au passage que les fonctions de la famille de `current_timestamp` (et donc le fréquemment utilisé `now()`) renvoient de plus une valeur constante au sein d'une même transaction.

PostgreSQL refusera de déclarer comme `STABLE` toute fonction modifiant des données : elle ne peut pas être stable si elle modifie la base.

4.9.4 Optimisation : rigueur

- Fonction `STRICT`
- La fonction renvoie `NULL` si au moins un des arguments est `NULL`

Les fonctions définies comme `STRICT` ou `RETURNS NULL ON NULL INPUT` annule l'exécution de la requête si l'un des paramètres passés est `NULL`. Dans ce cas, la fonction est considérée comme ayant renvoyé `NULL`.

Si l'on reprend l'exemple de la fonction `factorielle()` :

```
create or replace function factorielle (a integer) returns bigint as
$$
declare
  result bigint;
```

```
begin
  if a=1 then
    return 1;
  else
    return a*(factorielle(a-1));
  end if;
end;
$$
language plpgsql immutable STRICT;
```

on obtient le résultat suivant si elle est exécutée avec la valeur NULL passée en paramètre :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(NULL);
                  QUERY PLAN
-----------------------------------------------
 Result  (cost=0.00..0.00 rows=0 width=8)
         (actual time=0.002..0.002 rows=0 loops=1)
   One-Time Filter: false
 Planning time: 0.100 ms
 Execution time: 0.039 ms
(4 rows)
```

4.9.5 Optimisation : EXCEPTION

- Un bloc contenant une clause EXCEPTION est plus coûteuse en entrée/sortie qu'un bloc sans
  - un SAVEPOINT est créé à chaque fois pour pouvoir annuler le bloc uniquement.
- À utiliser avec parcimonie
- Un bloc BEGIN imbriqué a un coût aussi
  - un SAVEPOINT est créé à chaque fois.

4.9.6 Requête statique ou dynamique ?

- Les requêtes statiques :
  - sont écrites « en dur » dans le code PL/pgSQL
  - donc pas d'EXECUTE ou PERFORM
  - sont préparées une fois par session, à leur première exécution
  - peuvent avoir un plan générique lorsque c'est jugé utile par le planificateur

Avant la version 9.2, un plan générique (indépendant des paramètres de l'ordre SQL) était systématiquement généré et utilisé. Ce système permet de gagner du temps d'exécution si la requête est réutilisée plusieurs fois, et qu'elle est coûteuse à planifier.

Toutefois, un plan générique n'est pas forcément idéal dans toutes les situations, et peut conduire à des mauvaises performances.

Par exemple :

```
SELECT * FROM ma_table WHERE col_pk = param_function ;
```

est un excellent candidat à être écrit statiquement : le plan sera toujours le même : on attaque l'index de la clé primaire pour trouver l'enregistrement.

```
SELECT * FROM ma_table WHERE col_timestamp > param_function ;
```

est un moins bon candidat : le plan, idéalement, dépend de param_function : on ne parcourt pas la même fraction de la table suivant la valeur de param_function.

Par défaut, un plan générique ne sera utilisé dès la première exécution d'une requête statique que si celle-ci ne dépend d'aucun paramètre. Dans le cas contraire, cela ne se produira qu'au bout de plusieurs exécutions de la requête, et seulement si le planificateur détermine que les plans spécifiques utilisés n'apportent pas d'avantage par rapport au plan générique.

4.9.7 Requête statique ou dynamique ? - 2

- Les requêtes dynamiques :
  - sont écrites avec un `EXECUTE`, `PERFORM`...
  - sont préparées à chaque exécution
  - ont un plan optimisé
  - sont donc plus coûteuses en planification
  - mais potentiellement plus rapides à l'exécution

L'écriture d'une requête dynamique est par contre un peu plus pénible, puisqu'il faut fabriquer un ordre SQL, puis le passer en paramètre à `EXECUTE`, avec tous les quote_* que cela implique pour en protéger les paramètres.

Pour se faciliter la vie, on peut utiliser `EXECUTE query USING param1, param2 …,` qui est même quelquefois plus lisible que la syntaxe en dur : les paramètres de la requête sont clairement identifiés dans cette syntaxe.

Par contre, la syntaxe `USING` n'est utilisable que si le nombre de paramètres est fixe.

4.9.8 Requête statique ou dynamique ? -3

- Alors, statique ou dynamique ?
- Si la requête est simple : statique
  - peu de `WHERE`
  - peu ou pas de jointure
- Sinon dynamique

La limite est difficile à placer, il s'agit de faire un compromis entre le temps de planification d'une requête (quelques dizaines de microsecondes pour une requête basique à potentiellement plusieurs secondes si on dépasse la dizaine de jointures) et le temps d'exécution.

Dans le doute, réalisez un test de performance de la fonction sur un jeu de données représentatif.

4.10 OUTILS

- Deux outils disponibles
  - un debugger
  - un pseudo-profiler

Tous les outils d'administration PostgreSQL permettent d'écrire des routines stockées en PL/pgSQL, la plupart avec les fonctionnalités habituelles (comme le surlignage des mots clés, l'indentation automatique, etc.).

Par contre, pour aller plus loin, l'offre est restreinte. Il existe tout de même un debugger qui fonctionne avec pgAdmin 4, sous la forme d'une extension.

4.10.1 pldebugger

- License Artistic 2.0
- Développé par EDB et intégrable dans pgAdmin
- Installé par défaut avec le one-click installer
  - mais non activé
- Compilation nécessaire pour les autres systèmes

pldebugger est un outil initialement créé par Dave Page et Korry Douglas au sein d'EnterpriseDB, repris par la communauté. Il est proposé sous license libre (Artistic 2.0).

Il fonctionne grâce à des hooks implémentés dans la version 8.2 de PostgreSQL.

Il est assez peu connu, ce qui explique que peu l'utilisent. Seul l'outil d'installation « one-click installer » l'installe par défaut. Pour tous les autres systèmes, cela réclame une compilation supplémentaire. Cette compilation est d'ailleurs peu aisée étant donné qu'il n'utilise pas le système pgxs.

4.10.2 pldebugger - Compilation

- Récupérer le source avec git
- Copier le répertoire dans le répertoire contrib des sources de PostgreSQL
- Et les suivre étapes standards
  - make
  - make install

Voici les étapes à réaliser pour compiler pldebugger en prenant pour hypothèse que les sources de PostgreSQL sont disponibles dans le répertoire `/usr/src/postgresql-10` et qu'ils ont été pré-configurés avec la commande `./configure` :

- Se placer dans le répertoire contrib des sources de PostgreSQL :

```
$ cd /usr/src/postgresql-10/contrib
```

- Cloner le dépôt git :

```
$ git clone git://git.postgresql.org/git/pldebugger.git
Cloning into 'pldebugger'...
remote: Counting objects: 441, done.
remote: Compressing objects: 100% (337/337), done.
remote: Total 441 (delta 282), reused 171 (delta 104)
Receiving objects: 100% (441/441), 170.24 KiB, done.
Resolving deltas: 100% (282/282), done.
```

- Se placer dans le nouveau répertoire `pldebugger` :

```
$ cd pldebugger
```

- Compiler pldebugger :

```
$ make
```

- Installer pldebugger :

```
# make install
```

L'installation copie le fichier `plugin_debugger.so` dans le répertoire des bibliothèques partagées de PostgreSQL. L'installation copie ensuite les fichiers SQL et de contrôle de l'extension `pldbgapi` dans le répertoire `extension` du répertoire `share` de PostgreSQL.

4.10.3 pldebugger - Activation

- Configurer `shared_preload_libraries`
  - `shared_preload_libraries = 'plugin_debugger'`
- Redémarrer PostgreSQL
- Installer l'extension `pldbgapi` :

```
CREATE EXTENSION pldbgapi;
```

La configuration du paramètre `shared_preload_libraries` permet au démarrage de PostgreSQL de laisser la bibliothèque `plugin_debugger` s'accrocher aux hooks de l'interpréteur PL/pgSQL. Du coup, pour que la modification de ce paramètre soit prise en compte, il faut redémarrer PostgreSQL.

L'interaction avec pldebugger se fait par l'intermédiaire de procédures stockées. Il faut donc au préalable créer ces procédures stockées dans la base contenant les procédures PL/pgSQL à débugguer. Cela se fait en créant l'extension :

```
$ psql
psql (13.0)
Type "help" for help.

postgres# create extension pldbgapi;
CREATE EXTENSION
```

4.10.4 auto_explain

- Mise en place globale (traces) :
  - `shared_preload_libraries='auto_explain'` si global
  - `ALTER DATABASE erp SET auto_explain.log_min_duration = '3s'`
- Ou par session :
  - `LOAD 'auto_explain'`
  - `SET auto_explain.log_analyze TO true;`
  - `SET auto_explain.log_nested_statements TO true;`

`auto_explain` est une « contrib » officielle de PostgreSQL (et non une extension). Il permet de tracer le plan d'une requête. En général, on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre`auto_explain.log_min_duration`. Par défaut, ce paramètre est à -1 pour ne tracer aucun plan.

Comme dans un `EXPLAIN` classique, on peut activer toutes les options (par exemple `ANALYZE` ou `TIMING` avec, respectivement `SET auto_explain.log_analyze TO true;` et `SET auto_explain.log_timing TO true;`) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'`EXPLAIN`, notamment `auto_explain.log_b` et `auto_explain.log_settings` (voir la documentation[4]).

L'exemple suivant utilise deux fonctions imbriquées mais cela marche pour une simple requête :

```
CREATE OR REPLACE FUNCTION table_nb_indexes (tabname IN text, nbi OUT int)
RETURNS int
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT  COUNT(*) INTO nbi
    FROM    pg_index i INNER JOIN pg_class c ON (c.oid=indrelid)
    WHERE   relname LIKE tabname ;
    RETURN ;
END ;
$$
;
CREATE OR REPLACE FUNCTION table_nb_col_indexes
                    (tabname IN text, nb_cols OUT int, nb_indexes OUT int)
RETURNS record
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT  COUNT(*) INTO nb_cols
    FROM    pg_attribute
    WHERE   attname LIKE tabname ;

    SELECT nbi INTO nb_indexes FROM table_nb_indexes (tabname) ;

    RETURN ;
END ;
$$
;
```

Chargement dans la session d'`auto_explain` (si pas déjà présent dans `shared_preload_libraries`) :

```
LOAD 'auto_explain' ;
```

Activation pour toutes les requêtes, avec les options `ANALYZE` et `BUFFERS`, puis affichage dans la console (si la sortie dans les traces ne suffit pas) :

```
SET auto_explain.log_min_duration TO 0 ;
SET auto_explain.log_analyze TO on ;
```

[4] https://docs.postgresql.fr/current/auto-explain.html

```
SET auto_explain.log_buffers TO on ;
SET client_min_messages TO log ;
```

Test de la première fonction : le plan s'affiche, mais les compteurs (ici juste *shared hit*), ne concernent que la fonction dans son ensemble.

```
postgres=# SELECT * FROM table_nb_col_indexes ('pg_class') ;

LOG:  duration: 2.208 ms  plan:
Query Text: SELECT * FROM table_nb_col_indexes ('pg_class') ;
Function Scan on table_nb_col_indexes  (cost=0.25..0.26 rows=1 width=8)
                               (actual time=2.203..2.203 rows=1 loops=1)
  Buffers: shared hit=294

 nb_cols | nb_indexes
---------+------------
       0 |          3
```

En activant `auto_explain.log_nested_statements`, on voit clairement les plans de chaque requête exécutée :

```
SET auto_explain.log_nested_statements TO on ;

postgres=# SELECT * FROM table_nb_col_indexes ('pg_class') ;

LOG:  duration: 0.235 ms  plan:
Query Text: SELECT  COUNT(*)                    FROM    pg_attribute
    WHERE   attname LIKE tabname
Aggregate  (cost=65.95..65.96 rows=1 width=8)
    (actual time=0.234..0.234 rows=1 loops=1)
  Buffers: shared hit=24
  ->  Index Only Scan using pg_attribute_relid_attnam_index on pg_attribute
                                      (cost=0.28..65.94 rows=1 width=0)
                                (actual time=0.233..0.233 rows=0 loops=1)
        Index Cond: ((attname >= 'pg'::text) AND (attname < 'ph'::text))
        Filter: (attname ~~ 'pg_class'::text)
        Heap Fetches: 0
        Buffers: shared hit=24

LOG:  duration: 0.102 ms  plan:
Query Text: SELECT  COUNT(*)                FROM    pg_index i
    INNER JOIN pg_class c ON (c.oid=indrelid)
    WHERE   relname LIKE tabname
Aggregate  (cost=24.48..24.49 rows=1 width=8)
    (actual time=0.100..0.100 rows=1 loops=1)
  Buffers: shared hit=18
  ->  Nested Loop  (cost=0.14..24.47 rows=1 width=0)
          (actual time=0.096..0.099 rows=3 loops=1)
        Buffers: shared hit=18
        ->  Seq Scan on pg_class c  (cost=0.00..23.30 rows=1 width=4)
                           (actual time=0.091..0.093 rows=1 loops=1)
              Filter: (relname ~~ 'pg_class'::text)
              Rows Removed by Filter: 580
              Buffers: shared hit=16
        ->  Index Only Scan using pg_index_indrelid_index on pg_index i
                                    (cost=0.14..1.16 rows=1 width=4)
```

```
                              (actual time=0.003..0.004 rows=3 loops=1)
              Index Cond: (indrelid = c.oid)
              Heap Fetches: 0
              Buffers: shared hit=2

LOG:  duration: 0.703 ms  plan:
Query Text: SELECT nbi                 FROM table_nb_indexes (tabname)
Function Scan on table_nb_indexes  (cost=0.25..0.26 rows=1 width=4)
                            (actual time=0.702..0.702 rows=1 loops=1)
  Buffers: shared hit=26

LOG:  duration: 1.524 ms  plan:
Query Text: SELECT * FROM table_nb_col_indexes ('pg_class') ;
Function Scan on table_nb_col_indexes  (cost=0.25..0.26 rows=1 width=8)
                               (actual time=1.520..1.520 rows=1 loops=1)
  Buffers: shared hit=59

 nb_cols | nb_indexes
---------+------------
       0 |          3
```

Cet exemple permet de mettre le doigt sur un petit problème de performance dans la fonction : le _ est interprété comme critère de recherche. En modifiant le paramètre on peut constater le changement de plan au niveau des index :

```
postgres=# SELECT * FROM table_nb_col_indexes ('pg\_class') ;

LOG:  duration: 0.141 ms  plan:
Query Text: SELECT  COUNT(*)                   FROM    pg_attribute
    WHERE   attname LIKE tabname
Aggregate  (cost=56.28..56.29 rows=1 width=8)
    (actual time=0.140..0.140 rows=1 loops=1)
  Buffers: shared hit=24
  ->  Index Only Scan using pg_attribute_relid_attnam_index on pg_attribute
                                       (cost=0.28..56.28 rows=1 width=0)
                                (actual time=0.138..0.138 rows=0 loops=1)
        Index Cond: (attname = 'pg_class'::text)
        Filter: (attname ~~ 'pg\_class'::text)
        Heap Fetches: 0
        Buffers: shared hit=24

LOG:  duration: 0.026 ms  plan:
Query Text: SELECT  COUNT(*)               FROM    pg_index i
    INNER JOIN pg_class c ON (c.oid=indrelid)
    WHERE   relname LIKE tabname
Aggregate  (cost=3.47..3.48 rows=1 width=8) (actual time=0.024..0.024 rows=1 loops=1)
  Buffers: shared hit=8
  ->  Nested Loop  (cost=0.42..3.47 rows=1 width=0) (…)
        Buffers: shared hit=8
        ->  Index Scan using pg_class_relname_nsp_index on pg_class c
                                                  (cost=0.28..2.29 rows=1 width=4)
                                           (actual time=0.017..0.018 rows=1 loops=1)
              Index Cond: (relname = 'pg_class'::text)
              Filter: (relname ~~ 'pg\_class'::text)
              Buffers: shared hit=6
        ->  Index Only Scan using pg_index_indrelid_index on pg_index i  (…)
```

```
              Index Cond: (indrelid = c.oid)
              Heap Fetches: 0
              Buffers: shared hit=2

LOG:  duration: 0.414 ms  plan:
Query Text: SELECT nbi                 FROM table_nb_indexes (tabname)
Function Scan on table_nb_indexes  (cost=0.25..0.26 rows=1 width=4)
                           (actual time=0.412..0.412 rows=1 loops=1)
  Buffers: shared hit=16

LOG:  duration: 1.046 ms  plan:
Query Text: SELECT * FROM table_nb_col_indexes ('pg\_class') ;
Function Scan on table_nb_col_indexes  (cost=0.25..0.26 rows=1 width=8)
                              (actual time=1.042..1.043 rows=1 loops=1)
  Buffers: shared hit=56

 nb_cols | nb_indexes
---------+------------
       0 |          3
```

Pour les procédures, il est possible de mettre en place cette trace avec `ALTER PROCEDURE … SET auto_explain.log_min_duration = 0`. Cela ne fonctionne pas pour les fonctions.

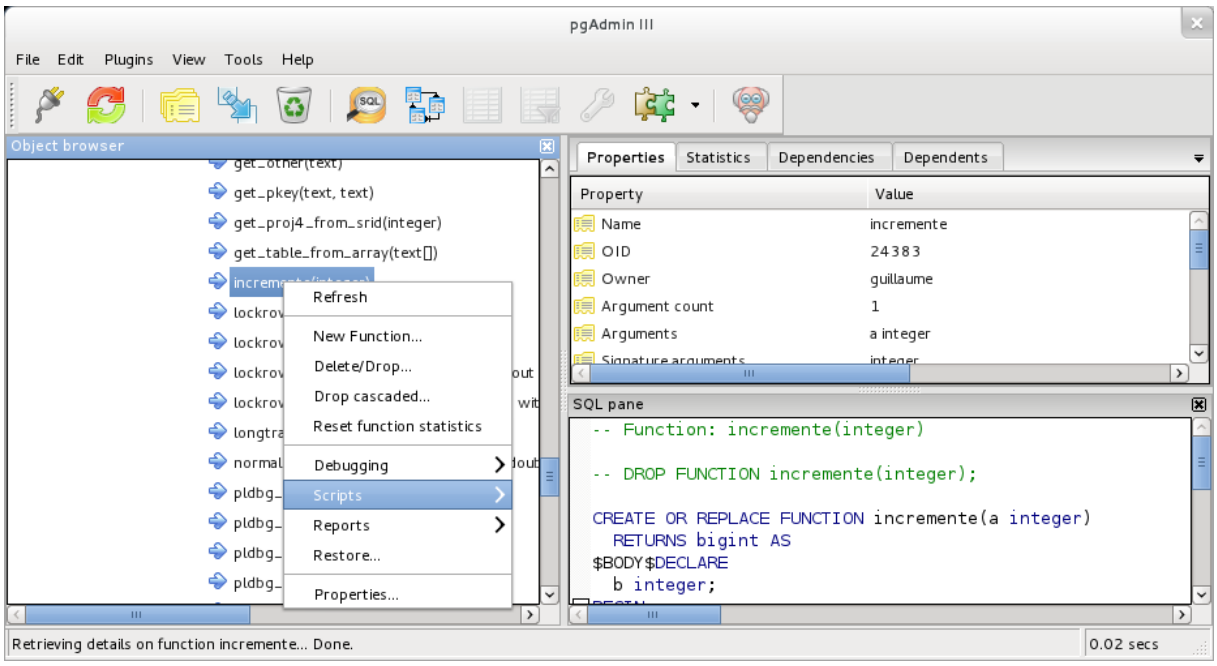
pgBadger est capable de lire les plans tracés par `auto_explain`, de les intégrer à son rapport et d'inclure un lien vers depesz.com[5] pour une version plus lisible.

4.10.5 pldebugger - Utilisation

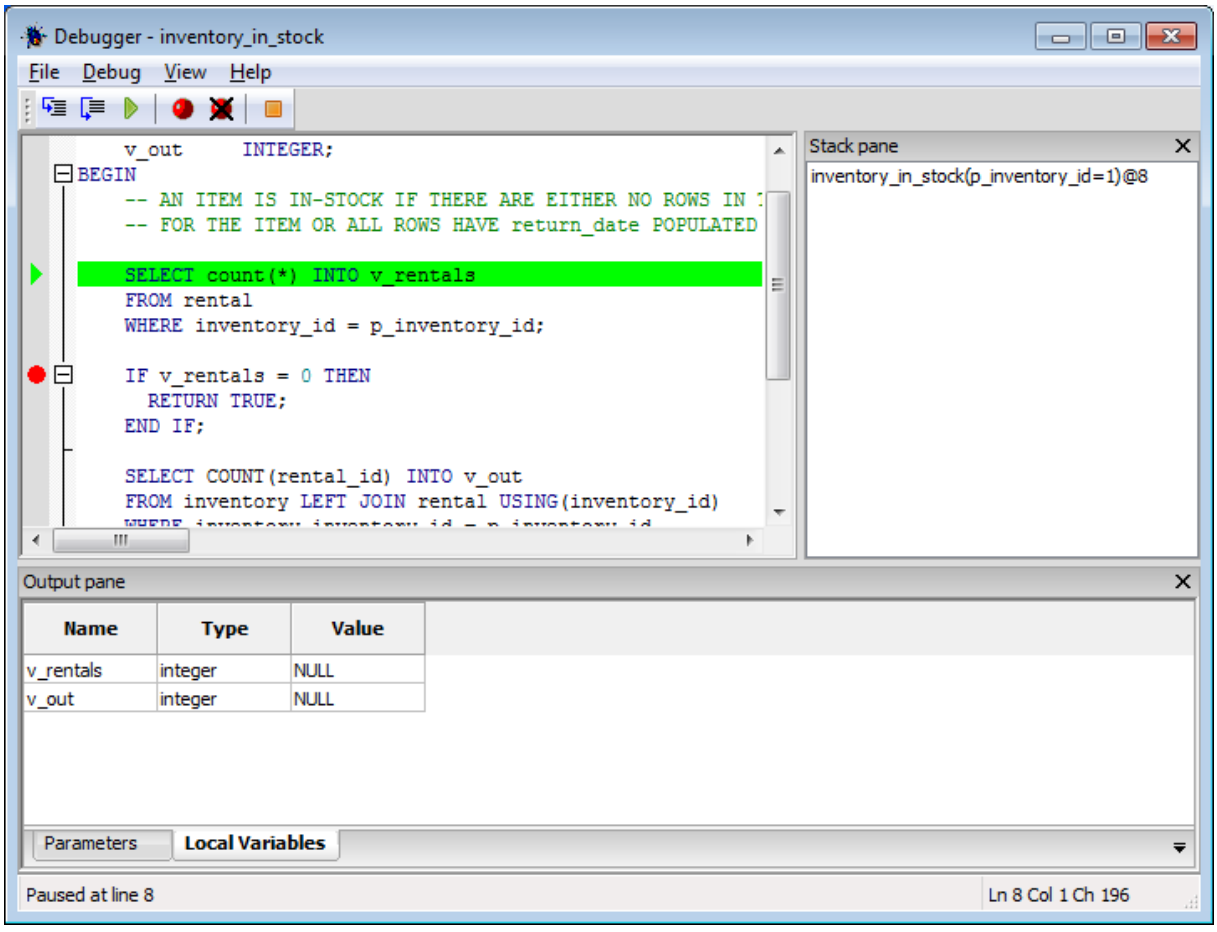
- Via pgAdmin

Le menu contextuel pour accéder au débuggage d'une fonction :

[5] https://explain.depesz.com/



La fenêtre du débugger :



4.10.6 log_functions

- Créé par Dalibo
- License BSD
- Compilation nécessaire

`log_functions` est un outil créé par Guillaume Lelarge au sein de Dalibo. Il est proposé sous license libre (BSD).

4.10.7 log_functions - Compilation

- Récupérer l'archive sur PGXN.org
- Décompresser l'archive puis : `make USE_PGXS=1 && make USE_PGXS=1 install`

Voici les étapes à réaliser pour compiler log_functions en prenant pour hypothèse que les sources de PostgreSQL sont disponibles dans le répertoire `/home/guillaume/postgresql-9.1.4` et qu'ils ont été préconfigurés avec la commande `./configure` :

- Se placer dans le répertoire contrib des sources de PostgreSQL :

```
$ cd /home/guillaume/postgresql-9.1.4/contrib
```

- Récupérer le dépôt git de log_functions :

```
$ git://github.com/gleu/log_functions.git

Cloning into 'log_functions'...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 24 (delta 8), reused 24 (delta 8)
Receiving objects: 100% (24/24), 11.71 KiB, done.
Resolving deltas: 100% (8/8), done.
```

- Se placer dans le nouveau répertoire `log_functions` :

```
$ cd log_functions
```

- Compiler log_functions :

```
$ make
```

- Installer log_functions :

```
$ make install
```

L'installation copie le fichier `log_functions.o` dans le répertoire des bibliothèques partagées de PostgreSQL.

Si la version de PostgreSQL est supérieure ou égale à la 9.2, alors l'installation est plus simple et les sources de PostgreSQL ne sont plus nécessaires.

Téléchargement de log_functions :

```
wget http://api.pgxn.org/dist/log_functions/1.0.0/log_functions-1.0.0.zip
```

puis décompression et installation de l'extension :

```
unzip log_functions-1.0.0.zip
cd log_functions-1.0.0/
make USE_PGXS=1 && make USE_PGXS=1 install
```

L'installation copie aussi le fichier `log_functions.so` dans le répertoire des bibliothèques partagées de PostgreSQL.

4.10.8 log_functions - Activation

- Permanente
  - `shared_preload_libraries = 'log_functions'`
  - Redémarrage de PostgreSQL
- Au cas par cas
  - `LOAD 'log_functions'`

Le module log_functions est activable de deux façons.

La première consiste à demander à PostgreSQL de le charger au démarrage. Pour cela, il faut configurer la variable `shared_preload_libraries`, puis redémarrer PostgreSQL pour que le changement soit pris en compte.

La deuxième manière de l'activer est de l'activer seulement au moment où son utilisation s'avère nécessaire. Il faut utiliser pour cela la commande LOAD en précisant le module à charger.

La première méthode a un coût en terme de performances car le module s'exécute à chaque exécution d'une procédure stockée écrite en PL/pgSQL. La deuxième méthode rend l'utilisation du profiler un peu plus complexe. Le choix est donc laissé à l'administrateur.

4.10.9 log_functions - Configuration

- 5 paramètres en tout
- À configurer
  - dans Postgresql.conf
  - ou avec SET

Les informations de profilage récupérées par log_functions sont envoyées dans les traces de PostgreSQL. Comme cela va générer plus d'écriture, et donc plus de lenteurs, il est possible de configurer chaque trace.

La configuration se fait soit dans le fichier `postgresql.conf` soit avec l'instruction SET.

Voici la liste des paramètres et leur utilité :

- `log_functions.log_declare`, à mettre à true pour tracer le moment où PL/pgSQL exécute la partie DECLARE d'une procédure stockée ;
- `log_functions.log_function_begin`, à mettre à true pour tracer le moment où PL/pgSQL exécute la partie BEGIN d'une procédure stockée ;
- `log_functions.log_function_end`, à mettre à true pour tracer le moment où PL/pgSQL exécute la partie END d'une procédure stockée ;
- `log_functions.log_statement_begin`, à mettre à true pour tracer le moment où PL/pgSQL commence l'exécution d'une instruction dans une procédure stockée ;
- `log_functions.log_statement_end`, à mettre à true pour tracer le moment où PL/pgSQL termine l'exécution d'une instruction dans une procédure stockée.

Par défaut, seuls `log_statement_begin` et `log_statement_end` sont à false pour éviter la génération de traces trop importantes.

4.10.10 log_functions - Utilisation

- Exécuter des procédures stockées en PL/pgSQL
- Lire les journaux applicatifs
  - `grep` très utile

Voici un exemple d'utilisation de cet outil :

```
b2# SELECT incremente(4);
 incremente
------------
          5
(1 row)

b2# LOAD 'log_functions';
LOAD
b2# SET client_min_messages TO log;
LOG:  duration: 0.136 ms  statement: set client_min_messages to log;
SET
b2# SELECT incremente(4);
LOG:  log_functions, DECLARE, incremente
LOG:  log_functions, BEGIN, incremente
CONTEXT:  PL/pgSQL function "incremente" during function entry
LOG:  valeur de b : 5
LOG:  log_functions, END, incremente
CONTEXT:  PL/pgSQL function "incremente" during function exit
LOG:  duration: 118.332 ms  statement: select incremente(4);
 incremente
------------
          5
(1 row)
```

4.11 CONCLUSION

- PL/pgSQL est un langage puissant
- Seul inconvénient
  - sa lenteur par rapport à d'autres PL comme PL/perl ou C
  - PL/perl est très efficace pour les traitements de chaîne notamment
- Permet néanmoins de traiter la plupart des cas, de façon simple et efficace

4.11.1 Pour aller plus loin

- Documentation officielle
  - « Chapitre 40. PL/pgSQL - Langage de procédures SQL »

Quelques liens utiles dans la documentation de PostgreSQL :

- Chapitre 40. PL/pgSQL - Langage de procédures SQL[6]
- Annexe A. Codes d'erreurs de PostgreSQL[7]

4.11.2 Questions

N'hésitez pas, c'est le moment !

[6] https://docs.postgresql.fr/current/plpgsql.html
[7] https://docs.postgresql.fr/current/errcodes-appendix.html

4.12 TRAVAUX PRATIQUES

TP2.1

Ré-écrire la fonction de division pour tracer le problème de division par zéro (vous pouvez aussi utiliser les exceptions).

TP2.2

Tracer dans une table toutes les modifications du champ `nombre` dans `stock`. On veut conserver l'ancienne et la nouvelle valeur. On veut aussi savoir qui a fait la modification et quand.

Interdire la suppression des lignes dans stock. Afficher un message dans les logs dans ce cas.

Afficher aussi un message `NOTICE` quand nombre devient inférieur à 5, et `WARNING` quand il vaut 0.

TP2.3

Interdire à tout le monde, sauf un compte admin, l'accès à la table des logs précédemment créée .

En conséquence, le trigger fonctionne-t-il ? Le cas échéant, le modifier pour qu'il fonctionne.

TP2.4

Lire toute la table `stock` avec un curseur.

Afficher dans les journaux applicatifs toutes les paires `(vin_id, contenant_id)` pour chaque nombre supérieur à l'argument de la fonction.

TP2.5

Ré-écrire la fonction `nb_bouteilles` du TP précédent de façon à ce qu'elle prenne désormais en paramètre d'entrée une liste variable d'années à traiter.

4.13 TRAVAUX PRATIQUES (SOLUTIONS)

TP2.1 Solution :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS float4 AS
$BODY$
  BEGIN
    RETURN arg1::float4/arg2::float4;
    EXCEPTION WHEN OTHERS THEN
      -- attention, division par zéro
      RAISE LOG 'attention, [%]: %', SQLSTATE, SQLERRM;
      RETURN 'NaN';
  END $BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage :

```
  cave=# SET client_min_messages TO log;
  SET
  cave=# SELECT division(1,5);
   division
  ----------
        0.2
  (1 ligne)

  cave=# SELECT division(1,0);
  LOG:  attention, [22012]: division par zéro
   division
  ----------
       NaN
  (1 ligne)
```

TP2.2 Solution :

1.

La table de log :

```
  CREATE TABLE log_stock (
  id serial,
  utilisateur text,
  dateheure timestamp,
  operation char(1),
  vin_id integer,
  contenant_id integer,
  annee integer,
  anciennevaleur integer,
  nouvellevaleur integer);
```

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    -- ce test a pour but de vérifier que le contenu de nombre a bien changé
    -- c'est forcément le cas dans une insertion et dans une suppression
    -- mais il faut tester dans le cas d'une mise à jour en se méfiant
    -- des valeurs NULL
    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'DELETE'
    THEN
      -- cas de la suppression
      v_atracer := true;
      v_vinid := OLD.vin_id;
      v_contenantid := OLD.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NULL;
    END IF;

    IF v_atracer
    THEN
      INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur);
```

```
    END IF;

    RETURN NEW;

  END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Le trigger :

```
  CREATE TRIGGER log_stock_nombre_trig
  AFTER INSERT OR UPDATE OR DELETE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE log_stock_nombre();
```

2.

On commence par supprimer le trigger :

```
  DROP TRIGGER log_stock_nombre_trig ON stock;
```

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
```

```
    END IF;

    IF v_atracer
    THEN
      INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_anciennevaleur, v_nouvellevaleur);
    END IF;

    RETURN NEW;

  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Le trigger :

```
  CREATE TRIGGER trace_nombre_de_stock
  AFTER INSERT OR UPDATE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE log_stock_nombre();
```

La deuxième fonction trigger :

```
CREATE OR REPLACE FUNCTION empeche_suppr_stock()
  RETURNS TRIGGER AS
$BODY$
  BEGIN

    IF TG_OP = 'DELETE'
    THEN
      RAISE WARNING 'Tentative de suppression du stock (%, %, %)',
                    OLD.vin_id, OLD.contenant_id, OLD.annee;
      RETURN NULL;
    ELSE
      RETURN NEW;
    END IF;

  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Le deuxième trigger :

```
  CREATE TRIGGER empeche_suppr_stock_trig
  BEFORE DELETE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE empeche_suppr_stock();
```

3.

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
    END IF;

    IF v_nouvellevaleur < 1
    THEN
      RAISE WARNING 'Il ne reste plus que % bouteilles dans le stock (%, %, %)',
                    v_nouvellevaleur, OLD.vin_id, OLD.contenant_id, OLD.annee;
    ELSEIF v_nouvellevaleur < 5
    THEN
      RAISE LOG 'Il ne reste plus que % bouteilles dans le stock (%, %, %)',
                v_nouvellevaleur, OLD.vin_id, OLD.contenant_id, OLD.annee;
    END IF;

    IF v_atracer
    THEN
      INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur);
    END IF;

    RETURN NEW;
```

```
  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage :

Faire des INSERT, DELETE, UPDATE pour jouer avec.

TP2.3 Solution :

```
 CREATE ROLE admin;
 ALTER TABLE log_stock OWNER TO admin;
 ALTER TABLE log_stock_id_seq OWNER TO admin;
 REVOKE ALL ON TABLE log_stock FROM public;

 cave=> insert into stock (vin_id, contenant_id, annee, nombre)
        values (3,1,2020,10);
ERROR:  permission denied for relation log_stock
CONTEXT:  SQL statement "INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur)"
PL/pgSQL function log_stock_nombre() line 45 at SQL statement

 ALTER FUNCTION log_stock_nombre() OWNER TO admin;
 ALTER FUNCTION log_stock_nombre() SECURITY DEFINER;

 cave=> insert into stock (vin_id, contenant_id, annee, nombre)
        values (3,1,2020,10);
  INSERT 0 1
```

Que constatez-vous dans `log_stock` ? (un petit indice : regardez l'utilisateur)

TP2.4 Solution :

```
CREATE OR REPLACE FUNCTION verif_nombre(maxnombre integer)
  RETURNS integer AS
$BODY$
  DECLARE
    v_curseur refcursor;
    v_resultat stock%ROWTYPE;
    v_index integer;
  BEGIN

    v_index := 0;
    OPEN v_curseur FOR SELECT * FROM stock WHERE nombre > maxnombre;
    LOOP
```

```
      FETCH v_curseur INTO v_resultat;
      IF NOT FOUND THEN
        EXIT;
      END IF;
      v_index := v_index + 1;
      RAISE NOTICE 'nombre de (%, %) : % (supérieur à %)',
        v_resultat.vin_id, v_resultat.contenant_id, v_resultat.nombre, maxnombre;
    END LOOP;

  RETURN v_index;

END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage:

```
  SELECT verif_nombre(16);
  INFO:  nombre de (6535, 3) : 17 (supérieur à 16)
  INFO:  nombre de (6538, 3) : 17 (supérieur à 16)
  INFO:  nombre de (6541, 3) : 17 (supérieur à 16)
  [...]
  INFO:  nombre de (6692, 3) : 18 (supérieur à 16)
  INFO:  nombre de (6699, 3) : 17 (supérieur à 16)
   verif_nombre
  --------------
         107935
  (1 ligne)
```

TP2.5

```
CREATE OR REPLACE FUNCTION
  nb_bouteilles(v_typevin text, VARIADIC v_annees integer[])
RETURNS SETOF record
AS $BODY$
  DECLARE
    resultat record;
    i integer;
  BEGIN
    FOREACH i IN ARRAY v_annees
    LOOP
      SELECT INTO resultat i, nb_bouteilles(v_typevin, i);
      RETURN NEXT resultat;
    END LOOP;
    RETURN;
  END
$BODY$
LANGUAGE plpgsql;
```

Exécution:

```
-- ancienne fonction
cave=# SELECT * FROM nb_bouteilles('blanc', 1990, 1995)
  AS (annee integer, nb integer);
```

```
 annee |  nb
-------+------
  1990 | 5608
  1991 | 5642
  1992 | 5621
  1993 | 5581
  1994 | 5614
  1995 | 5599
(6 lignes)

cave=# SELECT * FROM nb_bouteilles('blanc', 1990, 1992, 1994)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1990 | 5608
  1992 | 5621
  1994 | 5614
(3 lignes)

cave=# SELECT * FROM nb_bouteilles('blanc', 1993, 1991)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1993 | 5581
  1991 | 5642
(2 lignes)
```

5/ Extensions PostgreSQL pour l’utilisateur

5.1 QU'EST-CE QU'UNE EXTENSION ?

- Pour ajouter :
  - types de données
  - méthodes d'indexation
  - fonctions et opérateurs
  - tables, vues…
- Tous sujets, tous publics
- Intégrées (« contribs ») ou projets externes

Les extensions sont un gros point fort de PostgreSQL. Elles permettent de rajouter des fonctionnalités, aussi bien pour les utilisateurs que pour les administrateurs, sur tous les sujets : fonctions utilitaires, types supplémentaires, outils d'administration avancés, voire applications quasi-complètes. Certaines sont intégrées par le projet, mais n'importe qui peut en proposer et en intégrer une.

5.2 ADMINISTRATION DES EXTENSIONS

Techniquement :

- « packages » pour PostgreSQL, en C, SQL, PL/pgSQL…
- Langages : SQL, PL/pgSQL, C (!)…
- Ensemble d'objets livrés ensemble
- contrib <> extension

Une extension est un objet du catalogue, englobant d'autres objets. On peut la comparer à un paquetage Linux.

Une extension peut provenir d'un projet séparé de PostgreSQL (PostGIS, par exemple, ou le *Foreign Data Wrapper* Oracle).

Les extensions les plus simples peuvent se limiter à quelques objets en SQL, certaines sont en PL/pgSQL, beaucoup sont en C. Dans ce dernier cas, il faut être conscient que la stabilité du serveur est encore plus en jeu !

5.2.1 Installation des extensions

- Packagées ou à compiler
- Par base :
  - `CREATE EXTENSION … CASCADE`
  - `ALTER EXTENSION UPDATE`
  - `DROP EXTENSION`
  - `\dx`
- Listées dans `pg_available_extensions`

Au niveau du système d'exploitation, une extension nécessite des objets (binaires, scripts…) dans l'arborescence de PostgreSQL. De nombreuses extensions sont déjà fournies sous forme de paquets dans les distributions courantes ou par le PGDG, ou encore l'outil PGXN. Dans certains cas, il faudra aller sur le site du projet et l'installer soi-même, ce qui peut nécessiter une compilation.

L'extension doit être ensuite déclarée dans chaque base où elle est jugée nécessaire avec `CREATE EXTENSION nom_extension`. Les scripts fournis avec l'extension vont alors créer les objets nécessaires (vues, procédures, tables…). En cas de désinstallation avec `DROP EXTENSION`, ils seront supprimés. Une extension peut avoir besoin d'autres extensions : l'option `CASCADE` permet de les installer automatiquement.

Le mécanisme couvre aussi la mise à jour des extensions : `ALTER EXTENSION UPDATE` permet de mettre à jour une extension dans PostgreSQL suite à la mise à jour de ses binaires. Cela peut être nécessaire si elle contient des tables à mettre à jour, par exemple. Les versions des extensions disponibles sur le système et celles installées dans la base en cours sont visibles dans la vue `pg_available_extensions`.

Les extensions peuvent être exportées et importées par `pg_dump`/`pg_restore`. Un export par `pg_dump` contient un `CREATE EXTENSION nom_extension`, ce qui permettra de recréer d'éventuelles tables, et le *contenu* de ces tables. Une mise à jour de version majeure, par exemple, permettra donc de migrer les extensions dans leur dernière version installée sur le serveur (changement de prototypes de fonctions, nouvelles vues, etc.).

Sous psql, les extensions présentes dans la base sont visibles avec `\dx` :

```
# \dx
                          Liste des extensions installées
        Nom          | Version |   Schéma   |                    Description
---------------------+---------+------------+-----------------------------------------
 ↪  ----------
 amcheck             | 1.2     | public     | functions for verifying relation integrity
 file_fdw            | 1.0     | public     | foreign-data wrapper for flat file access
 hstore              | 1.6     | public     | data type for storing sets of (key,
  ↪  value) pairs
 pageinspect         | 1.9     | public     | inspect the contents of database pages
  ↪  at...
 pg_buffercache      | 1.3     | public     | examine the shared buffer cache
 pg_prewarm          | 1.2     | public     | prewarm relation data
 pg_rational         | 0.0.1   | public     | bigint fractions
 pg_stat_statements  | 1.10    | public     | track execution statistics of all SQL
  ↪  statements...
 plpgsql             | 1.0     | pg_catalog | PL/pgSQL procedural language
 plpython3u          | 1.0     | pg_catalog | PL/Python3U untrusted procedural language
 postgres_fdw        | 1.0     | public     | foreign-data wrapper for remote
  ↪  PostgreSQL servers
 unaccent            | 1.1     | public     | text search dictionary that removes
  ↪  accents
```

5.3 CONTRIBS - FONCTIONNALITÉS

- Livrées avec le code source de PostgreSQL
- Habituellement packagées (`postgresql-*-contrib`)
- De qualité garantie car maintenues par le projet
- Optionnelles, désactivées par défaut
- Ou en cours de stabilisation
- Documentées : Chapitre F : « Modules supplémentaires fournis »[1]

Une « contrib » est habituellement une extension, sauf quelques exceptions qui ne créent pas d'objets de catalogue (`auto_explain` par exemple). Elles sont fournies directement dans l'arborescence de PostgreSQL, et suivent donc strictement son rythme de révision. Leur compatibilité est ainsi garantie. Les distributions les proposent parfois dans des paquets séparés (`postgresql-contrib-9.6`, `postgresql14-contrib`…), dont l'installation est fortement conseillée.

Il s'agit soit de fonctionnalités qui n'intéressent pas tout le monde (`hstore`, `uuid`, `pg_trgm`, `pgstattuple`…), ou en cours de stabilisation (comme l'autovacuum avant PostgreSQL 8.1), ou à l'inverse de dépréciation (`xml2`).

La documentation des contribs est dans le chapitre F des annexes[2], et est donc fréquemment oubliée par les nouveaux utilisateurs.

[2] https://docs.postgresql.fr/current/contrib.html

5.4 QUELQUES EXTENSIONS

…plus ou moins connues

5.4.1 pgcrypto

Module contrib de chiffrement :

- Nombreuses fonctions pour chiffrer et déchiffrer des données
- Gros inconvénient : oubliez les index sur les données chiffrées !
- N'oubliez pas de chiffrer la connexion (SSL)
- Permet d'avoir une seule méthode de chiffrement pour tout ce qui accède à la base

Fourni avec PostgreSQL, vous permet de chiffrer vos données[3] :

- directement ;
- avec une clé PGP (gérée par exemple avec GnuPG), ce qui est préférable ;
- selon divers algorithmes courants ;
- différemment selon chaque ligne/champ.

Voici un exemple de code:

```
CREATE EXTENSION pgcrypto;
UPDATE utilisateurs SET mdp = crypt('mon nouveau mot de passe',gen_salt('md5'));
INSERT INTO table_secrete (encrypted)
VALUES (pgp_sym_encrypt('mon secret','motdepasse'));
```

L'appel à `gen_salt` permet de rajouter une partie aléatoire à la chaîne à chiffrer, ce qui évite que la même chaîne chiffrée deux fois retourne le même résultat. Cela limite donc les attaques par dictionnaire.

La base effectuant le (dé)chiffrement, cela évite certains allers-retours. Il est préférable que la clé de déchiffrement ne soit pas *dans* l'instance, et soit connue et fournie par l'applicatif. La communication avec cet applicatif doit être sécurisée par SSL pour que les clés et données ne transitent pas en clair.

Un gros inconvénient des données chiffrées dans la table est l'impossibilité complète de les indexer, même avec un index fonctionnel : les données déchiffrées seraient en clair dans cet index ! Une recherche implique donc de parcourir et déchiffrer chaque ligne…

[3] https://docs.postgresql.fr/current/pgcrypto.html

5.4.2 hstore : stockage clé/valeur

- Contrib
- Type `hstore`
- Stockage clé-valeur
- Plus simple que JSON

```
INSERT INTO demo_hstore (meta) VALUES ('river=>t');
SELECT * FROM demo_hstore WHERE meta@>'river=>t';
```

`hstore` fournit un type très simple pour stocker des clés/valeur :

```
CREATE EXTENSION hstore ;

CREATE TABLE demo_hstore(id serial, meta hstore);
INSERT INTO demo_hstore (meta) VALUES ('river=>t');
INSERT INTO demo_hstore (meta) VALUES ('road=>t,secondary=>t');
INSERT INTO demo_hstore (meta) VALUES ('road=>t,primary=>t');
CREATE INDEX idxhstore ON demo_hstore USING gist (meta);

SELECT * FROM demo_hstore WHERE meta@>'river=>t';

 id |     meta
----+--------------
 15 | "river"=>"t"
```

Cette extension a rendu, et rend encore, bien des services. Cependant le type JSON (avec le type binaire `jsonb`) est généralement préféré.

5.4.3 PostgreSQL Anonymizer

- Extension externe (Dalibo)
- Masquage statique et dynamique
- Export anonyme (`pg_dump_anon`)
- Les règles de masquage sont écrites en SQL
- Autodétection de colonnes identifiantes
- Plus simple et plus sûr qu'un ETL

Postgresql Anonymizer[4] est une extension pour masquer ou remplacer les données personnelles[5] dans une base PostgreSQL. Elle est développée par Damien Clochard de Dalibo.

Le projet fonctionne selon une **approche déclarative**, c'est à dire que les règles de masquage[6] sont déclarées directement dans le modèle de données avec des ordres DDL.

Une fois que les règles de masquage sont définies, on peut accéder aux données masquées de 3 façons différentes :

- export anonyme[7] : extraire les données masquées dans un fichier SQL ;
- masquage statique[8] : supprimer une fois pour toutes les données personnelles ;
- masquage dynamique[9] : cacher les données personnelles seulement pour les utilisateurs masqués.

Par ailleurs, l'extension fournit toute une gamme de fonctions de masquage[10] : randomisation, génération de données factices, destruction partielle, brassage, ajout de bruit, etc. On peut également écrire ses propres fonctions de masquage !

Au-delà du masquage, il est également possible d'utiliser une autre approche appelée généralisation[11] qui est bien adaptée pour les statistiques et l'analyse de données.

Enfin, l'extension offre un panel de fonctions de détection[12] qui tentent de deviner quelles colonnes doivent être anonymisées.

Un module de formation lui est consacré[13].

Exemple :

```
=# SELECT * FROM people;

 id | firstname | lastname |   phone
----+-----------+----------+------------
 T1 | Sarah     | Conor    | 0609110911
```

Étape 1 : activer le masquage dynamique

```
=# CREATE EXTENSION IF NOT EXISTS anon CASCADE;
=# SELECT anon.start_dynamic_masking();
```

Étape 2 : déclarer un utilisateur masqué

```
=# CREATE ROLE skynet LOGIN;
=# SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';
```

Étape 3 : déclarer les règles de masquage

[4] https://postgresql-anonymizer.readthedocs.io/en/stable/
[5] https://en.wikipedia.org/wiki/Personally_identifiable_information
[6] https://postgresql-anonymizer.readthedocs.io/en/stable/declare_masking_rules/
[7] https://postgresql-anonymizer.readthedocs.io/en/stable/anonymous_dumps/
[8] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/
[9] https://postgresql-anonymizer.readthedocs.io/en/stable/dynamic_masking/
[10] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions/
[11] https://postgresql-anonymizer.readthedocs.io/en/stable/generalization/
[12] https://postgresql-anonymizer.readthedocs.io/en/stable/detection/
[13] https://dali.bo/y5_html

```
=# SECURITY LABEL FOR anon ON COLUMN people.lastname
-# IS 'MASKED WITH FUNCTION anon.fake_last_name()';

=# SECURITY LABEL FOR anon ON COLUMN people.phone
-# IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$******$$,2)';
```

Étape 4 : se connecter avec l'utilisateur masqué

```
=# \c - skynet
=# SELECT * FROM people;

 id | firstname | lastname  |   phone
----+-----------+-----------+------------
 T1 | Sarah     | Stranahan | 06******11
```

5.4.4 PostGIS

- Projet indépendant, GPL, https://postgis.net/
- Module spatial pour PostgreSQL
  - Extension pour types géométriques/géographiques & outils
  - La référence des bases de données spatiales
  - « quelles sont les routes qui coupent le Rhône ? »
  - « quelles sont les villes adjacentes à Toulouse ? »
  - « quels sont les restaurants situés à moins de 3 km de la Nationale 12 ? »

PostGIS ajoute le support d'objets géographiques à PostgreSQL. C'est un projet totalement indépendant développé par la société Refractions Research sous licence GPL, soutenu par une communauté

active, utilisée par des spécialistes du domaine géospatial (IGN, BRGM, AirBNB, Mappy, Openstreetmap, Agence de l'eau…), mais qui peut convenir pour des projets plus modestes.

Techniquement, c'est une extension transformant PostgreSQL en serveur de données spatiales, qui sera utilisé par un Système d'Information Géographique (SIG), tout comme le SDE de la société ESRI ou bien l'extension Oracle Spatial. PostGIS se conforme aux directives du consortium OpenGIS et a été certifié par cet organisme comme tel, ce qui est la garantie du respect des standards par PostGIS.

PostGIS permet d'écrire des requêtes de ce type :

```
SELECT restaurants.geom, restaurants.name FROM restaurants
  WHERE EXISTS (SELECT 1 FROM routes
                  WHERE ST_DWithin(restaurants.geom, routes.geom, 3000)
                AND route.name = 'Nationale 12')
```

PostGIS fournit les fonctions d'indexation qui permettent d'accéder rapidement aux objets géométriques, au moyen d'index GiST. La requête ci-dessus n'a évidemment pas besoin de parcourir tous les restaurants à la recherche de ceux correspondant aux critères de recherche.

La liste des fonctionnalités comprend le support des coordonnées géodésiques ; des projections et reprojections dans divers systèmes de coordonnées locaux (Lambert93 en France par exemple) ; des opérateurs d'analyse géométrique (enveloppe convexe, simplification…)

PostGIS est intégré aux principaux serveurs de carte, ETL, et outils de manipulation.

La version 3.0 apporte la gestion du parallélisme, un meilleur support de l'indexation SP-GiST et GiST, ainsi qu'un meilleur support du type GeoJSON.

5.4.5 Mais encore…

- **uuid-ossp** : gérer des UUID
- **unaccent** : supprime des accents
- **citex** : recherche insensible à la casse

5.4.6 Autres extensions connues

- Compatibilité :
  - orafce
- Extensions propriétaires évitant un *fork* :
  - Citus (*sharding*)
  - TimescaleDB (*time series*)
  - être sûr que PostgreSQL a atteint ses limites !

Les extensions permettent de diffuser des bibliothèques de fonction pour la compatibilité avec du code d'autres produits : orafce est un exemple bien connu.

Pour éviter de maintenir un *fork* complet de PostgreSQL, certains éditeurs offrent leur produit sous forme d'extension, souvent avec une version communautaire intégrant les principales fonctionnalités. Par exemple :

- Citus permet du *sharding* ;
- TimescaleDB gère les séries temporelles.

Face à des extensions extérieures, on gardera à l'esprit qu'il s'agit d'un produit supplémentaire à maîtriser et administrer, et l'on cherchera d'abord à tirer le maximum du PostgreSQL communautaire.

5.5 EXTENSIONS POUR DE NOUVEAUX LANGAGES

- PL/pgSQL par défaut
- Ajouter des langages :
  - PL/python
  - PL/perl
  - PL/lua
  - PL/sh
  - PL/R
  - PL/Java
  - etc.

SQL et PL/pgSQL ne sont pas les seuls langages utilisables au niveau d'un serveur PostgreSQL. PL/pgSQL est installé par défaut en tant qu'extension. Il est possible de rajouter les langages python, perl, R, etc. et de coder des fonctions dans ces langages. Ces langages ne sont pas fournis par l'installation standard de PostgreSQL. Une installation via les paquets du système d'exploitation est sans doute le plus simple.

5.6 ACCÈS DISTANTS

Accès à des bases distantes

- Contribs :
  - dblink (ancien)
  - les *foreign data wrappers* : postgresql_fdw, mysql_fdw…
- Sharding :
  - PL/Proxy
  - Citus

Les accès distants à d'autres bases de données sont généralement disponibles par des extensions. L'extension dblink permet d'accéder à une autre instance PostgreSQL mais elle est ancienne, et l'on préférera le *foreign data wrapper* postgresql_fdw, disponible dans les contribs. D'autres FDW sont des projets extérieurs : ora_fdw, mysql_fdw, etc.

Une solution de *sharding* n'est pas encore intégrée à PostgreSQL mais des outils existent : PL/Proxy fournit des fonctions pour répartir des accès mais implique de refondre le code. Citus est une extension plus récente et plus transparente.

5.7 CONTRIBS ORIENTÉS DBA

Accès à des informations ou des fonctions de bas niveau :

- **pg_prewarm** : sauvegarde & restauration de l'état du cache de la base
- **pg_buffercache** : état du cache
- **pgstattuple** (fragmentation des tables et index), **pg_freespacemap** (blocs libres), **pg_visibility** (*visibility map*)
- **pageinspect** : inspection du contenu d'une page
- **pgrowlocks** : informations détaillées sur les enregistrements verrouillés
- **pg_stat_statement** (requêtes normalisées), **auto_explain** (plans)
- **amcheck** : validation des index
- … et de nombreux projets externes

Tous ces modules permettent de manipuler une facette de PostgreSQL à laquelle on n'a normalement pas accès. Leur utilisation est parfois très spécialisée et pointue.

En plus des contribs listés ci-dessus, de nombreux projets externes existent : toastinfo, pg_stat_kcache, pg_qualstats, PoWa, pg_wait_sampling, hypopg…

Pour plus de détails, consulter les modules X2[14] et X3[15].

[14] https://dali.bo/x2_html
[15] https://dali.bo/x3_html

5.8 PGXN

PostgreSQL eXtension Network :

- Site web : pgxn.org[16]
  - nombreuses extensions
  - volontariat
  - aucune garantie de qualité
  - tests soigneux requis
- Et optionnellement client en python pour automatisation de déploiement
- Ancêtre : pgFoundry
- Beaucoup de projets sont aussi sur github

Le site PGXN fournit une vitrine à de nombreux projets gravitant autour de PostgreSQL.

PGXN a de nombreux avantages, dont celui de demander aux projets participants de respecter un certain cahier des charges permettant l'installation automatisée des modules hébergés. Ceci peut par exemple être réalisé avec le client `pgxn` fourni :

```
> pgxn search --dist fdw
multicdr_fdw 1.2.2
    MultiCDR *FDW* ================== Foreign Data Wrapper for representing
    CDR files stream as an external SQL table. CDR files from a directory
    can be read into a table with a specified field-to-column...

redis_fdw 1.0.0
    Redis *FDW* for PostgreSQL 9.1+ ============================== This
    PostgreSQL extension implements a Foreign Data Wrapper (*FDW*) for the
    Redis key/value database: http://redis.io/ This code is...

jdbc_fdw 1.0.0
    Also,since the JVM being used in jdbc *fdw* is created only once for the
    entire psql session,therefore,the first query issued that uses jdbc
    +fdw* shall set the value of maximum heap size of the JVM(if...

mysql_fdw 2.1.2
    ... This PostgreSQL extension implements a Foreign Data Wrapper (*FDW*)
    for [MySQL][1]. Please note that this version of mysql_fdw only works
    with PostgreSQL Version 9.3 and greater, for previous version...

www_fdw 0.1.8
    ... library contains a PostgreSQL extension, a Foreign Data Wrapper
    (*FDW*) handler of PostgreSQL which provides easy way for interacting
    with different web-services.

mongo_fdw 2.0.0
    MongoDB *FDW* for PostgreSQL 9.2 ============================== This
    PostgreSQL extension implements a Foreign Data Wrapper (*FDW*) for
```

```
    MongoDB.

firebird_fdw 0.1.0
    ...  -
    http://www.postgresql.org/docs/current/interactive/postgres-*fdw*.html *
    Other FDWs  - https://wiki.postgresql.org/wiki/*Fdw*  -
    http://pgxn.org/tag/*fdw*/

json_fdw 1.0.0
    ... This PostgreSQL extension implements a Foreign Data Wrapper (*FDW*)
    for JSON files. The extension doesn't require any data to be loaded into
    the database, and supports analytic queries against array...

postgres_fdw 1.0.0
    This port provides a read-only Postgres *FDW* to PostgreSQL servers in
    the 9.2 series. It is a port of the official postgres_fdw contrib module
    available in PostgreSQL version 9.3 and later.

osm_fdw 3.0.0
    ... "Openstreetmap pbf foreign data wrapper") (*FDW*) for reading
    [Openstreetmap PBF](http://wiki.openstreetmap.org/wiki/PBF_Format
    "Openstreetmap PBF") file format (*.osm.pbf) ## Requirements  *...

odbc_fdw 0.1.0
    ODBC *FDW* (beta) for PostgreSQL 9.1+
    ================================== This PostgreSQL extension implements
    a Foreign Data Wrapper (*FDW*) for remote databases using Open Database
    Connectivity(ODBC)...

couchdb_fdw 0.1.0
    CouchDB *FDW* (beta) for PostgreSQL 9.1+
    ===================================== This PostgreSQL extension
    implements a Foreign Data Wrapper (*FDW*) for the CouchDB document-
    oriented database...

treasuredata_fdw 1.2.14
    ## INSERT INTO statement This *FDW* supports `INSERT INTO` statement.
    With `atomic_import` is `false`, the *FDW* imports INSERTed rows as
    follows.

twitter_fdw 1.1.1
    Installation ------------  $ make && make install  $ psql -c "CREATE
    EXTENSION twitter_fdw" db The CREATE EXTENSION statement creates not
    only *FDW* handlers but also Data Wrapper, Foreign Server, User...

ldap_fdw 0.1.1
    ... is an initial working on a PostgreSQL's Foreign Data Wrapper (*FDW*)
    to query LDAP servers. By all means use it, but do so entirely at your
    own risk! You have been warned! Do you like to use it in...

git_fdw 1.0.2
    # PostgreSQL Git Foreign Data Wrapper [![Build Status](https://travis-
    ci.org/franckverrot/git_fdw.svg?branch=master)](https://travis-
    ci.org/franckverrot/git_fdw) git\_fdw is a Git Foreign Data...

oracle_fdw 2.0.0
```

```
    Foreign Data Wrapper for Oracle ==============================
    oracle_fdw is a PostgreSQL extension that provides a Foreign Data
    Wrapper for easy and efficient access to Oracle databases, including...

foreign_table_exposer 1.0.0
    # foreign_table_exposer This PostgreSQL extension exposes foreign tables
    like a normal table with rewriting Query tree. Some BI tools can't
    detect foreign tables since they don't consider them when...

cstore_fdw 1.6.0
    cstore_fdw ========== [![Build Status](https://travis-
    ci.org/citusdata/cstore_fdw.svg?branch=master)][status] [![Coverage](htt
    p://img.shields.io/coveralls/citusdata/cstore_fdw/master.svg)][coverage]
    ...

multicorn 1.3.5
    [![PGXN version](https://badge.fury.io/pg/multicorn.svg)](https://badge.
    fury.io/pg/multicorn) [![Build
    Status](https://jenkins.dalibo.info/buildStatus/public/Multicorn)]()
    Multicorn =========...

tds_fdw 1.0.7
    # TDS Foreign data wrapper * **Author:** Geoff Montee * **Name:**
    tds_fdw * **File:** tds_fdw/README.md ## About This is a [PostgreSQL
    foreign data...

pmpp 1.2.3
    ... Having foreign server definitions and user mappings makes for
    cleaner function invocations.

file_textarray_fdw 1.0.1
    ### File Text Array Foreign Data Wrapper for PostgreSQL This *FDW* is
    similar to the provided file_fdw, except that instead of the foreign
    table having named fields to match the fields in the data...

floatfile 1.3.0
    Also I'd need to compare the performance of this vs an *FDW*. If I do
    switch to an *FDW*, I'll probably use [Andrew Dunstan's
    `file_text_array_fdw`](https://github.com/adunstan/file_text_array_fdw)
    as a...

pg_pathman 1.4.13
    ... event handling;  * Non-blocking concurrent table partitioning;  *
    +FDW* support (foreign partitions);  * Various GUC toggles and
    configurable settings.
```

Pour peu que le Instant Client d'Oracle soit installé, on peut par exemple lancer :

```
> pgxn install oracle_fdw
INFO: best version: oracle_fdw 1.1.0
INFO: saving /tmp/tmpihaor2is/oracle_fdw-1.1.0.zip
INFO: unpacking: /tmp/tmpihaor2is/oracle_fdw-1.1.0.zip
INFO: building extension
gcc -O3 -O0 -Wall -Wmissing-prototypes -Wpointer-arith [...]
[...]
INFO: installing extension
```

```
/usr/bin/mkdir -p '/opt/postgres/lib'
/usr/bin/mkdir -p '/opt/postgres/share/extension'
/usr/bin/mkdir -p '/opt/postgres/share/extension'
/usr/bin/mkdir -p '/opt/postgres/share/doc/extension'
/usr/bin/install -c -m 755  oracle_fdw.so '/opt/postgres/lib/oracle_fdw.so'
/usr/bin/install -c -m 644 oracle_fdw.control '/opt/postgres/share/extension/'
/usr/bin/install -c -m 644 oracle_fdw--1.1.sql\oracle_fdw--1.0--1.1.sql
                           '/opt/postgres/share/extension/'
/usr/bin/install -c -m 644 README.oracle_fdw \
                           '/opt/postgres/share/doc/extension/'
```

Attention : le fait qu'un projet soit hébergé sur PGXN n'est absolument pas une validation de la part du projet PostgreSQL. De nombreux projets hébergés sur PGXN sont encore en phase de développement, voire abandonnés. Il faut avoir le même recul que pour n'importe quel autre brique libre.

5.9 CRÉER SON EXTENSION

- Pas si compliqué
- Peut-être juste quelques fonctions SQL
- Référence : documentation, *Empaqueter des objets dans une extension*[17]
- Exemples SQL et C : blog Dalibo[18]

Il n'est pas très compliqué de créer sa propre extension pour diffuser aisément des outils. Elle peut se limiter à des fonctions en SQL ou PL/pgSQL. Le versionnement des extensions et la facilité de mise à jour peuvent être extrêmement utiles.

Deux exemples de création de fonctions en SQL ou C sont disponibles sur le blog Dalibo[19]. Un autre billet de blog présente une extension utilisable pour l'archivage[20].

La référence reste évidemment la documentation de PostgreSQL, chapitre *Empaqueter des objets dans une extension*[21].

[19] https://blog.dalibo.com/2023/06/08/hackingpg1.html
[20] https://blog.dalibo.com/2023/07/28/hackingpg2.html
[21] https://docs.postgresql.fr/current/extend-extensions.html

5.10 CONCLUSION

- Un nombre toujours plus important d'extension pour étendre les possibilités de PostgreSQL
- Un site central pour les extensions : PGXN.org
- Rajoutez les vôtres !

Cette possibilité d'étendre les fonctionnalités de PostgreSQL est vraiment un atout majeur du projet PostgreSQL. Cela permet de tester des fonctionnalités sans avoir à toucher au moteur de PostgreSQL et risquer des états instables.

Une fois l'extension mature, elle peut être intégrée directement dans le code de PostgreSQL si elle est considérée utile au moteur.

N'hésitez pas à créer vos propres extensions et à les diffuser !

5.10.1 Questions

N'hésitez pas, c'est le moment !

5.11 TRAVAUX PRATIQUES

5.11.1 Masquage statique de données avec PostgreSQL Anonymizer

But : Découverte de l'extension PostgreSQL Anonymizer et du masquage statique

Installer l'extension PostgreSQL Anonymizer en suivant la procédure décrite sur la page Installation[a] de la documentation.

[a] https://postgresql-anonymizer.readthedocs.io/en/stable/INSTALL/

Créer une table `customer` :

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    firstname TEXT,
    lastname TEXT,
    phone TEXT,
    birth DATE,
    postcode TEXT
);
```

Ajouter des individus dans la table :

```
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;
```

Lire la documentation sur comment déclarer une règle de masquage[a] et placer une règle pour générer un faux nom de famille sur la colonne `lastname`. Déclarer une règle de masquage statique sur la colonne `lastname` et l'appliquer. Vérifier le contenu de la table.

[a] https://postgresql-anonymizer.readthedocs.io/en/latest/declare_masking_rules/

Réappliquer le masquage statique[a]. Qu'observez-vous ?

[a] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/

5.11.2 Masquage dynamique de données avec PostgreSQL Anonymizer

But : Mettre en place un masquage dynamique avec PostgreSQL Anonymizer

> Parcourir la liste des fonctions de masquage[a] et écrire une règle pour cacher partiellement le numéro de téléphone. Activer le masquage dynamique. Appliquer le masquage dynamique uniquement sur la colonne phone pour un nouvel utilisateur nommé **soustraitant**.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions/

5.11.3 Masquage statique de données avec PostgreSQL Anonymizer

> Installer l'extension PostgreSQL Anonymizer en suivant la procédure décrite sur la page Installation[a] de la documentation.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/INSTALL/

Sur Rocky Linux ou autre dérivé Red Hat, depuis les dépôts du PGDG :

```
sudo dnf install postgresql_anonymizer_14
```

Au besoin, remplacer 14 par la version de l'instance PostgreSQL.

La base de travail ici se nomme **sensible**. Se connecter à l'instance pour initialiser l'extension :

```
ALTER DATABASE sensible SET session_preload_libraries = 'anon' ;
```

Après reconnexion à la base **sensible** :

```
CREATE EXTENSION anon CASCADE;

SELECT anon.init();  -- ne pas oublier !
```

> Créer une table `customer` :

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    firstname TEXT,
    lastname TEXT,
    phone TEXT,
    birth DATE,
    postcode TEXT
);
```

> Ajouter des individus dans la table :

```
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;

SELECT * FROM customer ;
```

```
 id  | firstname | lastname  |    phone     |   birth    | postcode
-----+-----------+-----------+--------------+------------+----------
 107 | Sarah     | Conor     | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Skywalker |              | 1951-09-25 | 90120
 341 | Don       | Draper    | 347-515-3423 | 1926-06-01 | 04520
```

> Lire la documentation sur comment déclarer une règle de masquage[a] et placer une règle pour générer un faux nom de famille sur la colonne `lastname`. Déclarer une règle de masquage statique sur la colonne `lastname` et l'appliquer. Vérifier le contenu de la table.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/latest/declare_masking_rules/

```
SECURITY LABEL FOR anon ON COLUMN customer.lastname
IS 'MASKED WITH FUNCTION anon.fake_last_name()' ;
```

Si on consulte la table avec :

```
SELECT * FROM customer ;
```

les données ne sont pas encore masquées car la règle n'est pas appliquée. L'application se fait avec :

```
SELECT anon.anonymize_table('customer') ;
```

```
SELECT * FROM customer;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 107 | Sarah     | Waelchi  | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Lemke    |              | 1951-09-25 | 90120
 341 | Don       | Shanahan | 347-515-3423 | 1926-06-01 | 04520
```

NB : les données de la table ont ici bien été modifiées sur le disque.

> Réappliquer le masquage statique[a]. Qu'observez-vous ?
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/

Si l'on relance l'anonymisation plusieurs fois, les données factices vont changer car la fonction `fake_last_name()` renvoie des valeurs différentes à chaque appel.

```
SELECT anon.anonymize_table('customer');
```

```
SELECT * FROM customer;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 107 | Sarah     | Smith    | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Sanford  |              | 1951-09-25 | 90120
 341 | Don       | Goldner  | 347-515-3423 | 1926-06-01 | 04520
```

5.11.4 Masquage dynamique de données avec PostgreSQL Anonymizer

> Parcourir la liste des fonctions de masquage[a] et écrire une règle pour cacher partiellement le numéro de téléphone. Activer le masquage dynamique. Appliquer le masquage dynamique uniquement sur la colonne phone pour un nouvel utilisateur nommé **soustraitant**.
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions/

```
SELECT anon.start_dynamic_masking();

SECURITY LABEL FOR anon ON COLUMN customer.phone
IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)';

SELECT anon.anonymize_column('customer','phone');

SELECT * FROM customer ;
```

Les numéros de téléphone apparaissent encore car ils ne sont pas masqués à l'utilisateur en cours. Il faut le déclarer pour les utilisateurs concernés :

```
CREATE ROLE soustraitant LOGIN ;
\password soustraitant

GRANT SELECT ON customer TO soustraitant ;
SECURITY LABEL FOR anon ON ROLE soustraitant  IS 'MASKED';
```

Ce nouvel utilisateur verra à chaque fois des noms différents (masquage dynamique), et des numéros de téléphone partiellement masqués :

```
\c sensible soustraitant
SELECT * FROM customer ;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 107 | Sarah     | Kovacek  | 06X-XXX-XX11 | 1965-10-10 | 90016
 258 | Luke      | Effertz  | ø            | 1951-09-25 | 90120
 341 | Don       | Turcotte | 34X-XXX-XX23 | 1926-06-01 | 04520
```

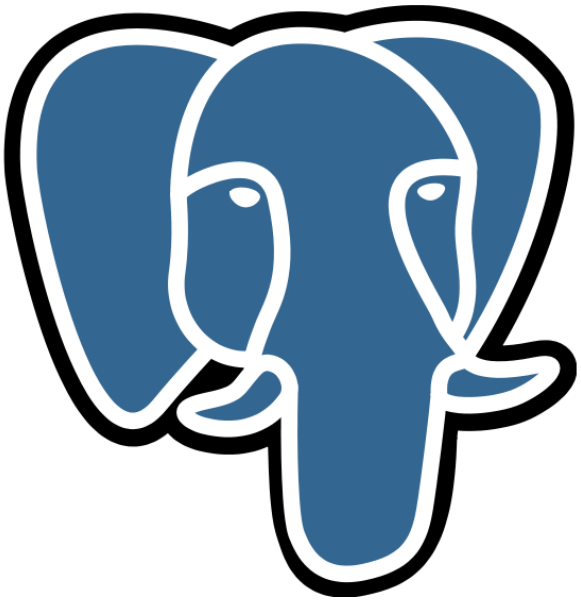
Pour consulter la configuration de masquage en place, utiliser une des vues fournies dans le schéma anon :

```
=# SELECT * FROM anon.pg_masks \gx
```

```
-[ RECORD 1 ]----+-------------------------------------------------------
attrelid         | 41853
attnum           | 3
relnamespace     | public
relname          | customer
attname          | lastname
format_type      | text
col_description  | MASKED WITH FUNCTION anon.fake_last_name()
masking_function | anon.fake_last_name()
masking_value    |
priority         | 100
masking_filter   | anon.fake_last_name()
trusted_schema   | t
-[ RECORD 2 ]----+-------------------------------------------------------
attrelid         | 41853
```

```
attnum           | 4
relnamespace     | public
relname          | customer
attname          | phone
format_type      | text
col_description  | MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)
masking_function | anon.partial(phone,2,$$X-XXX-XX$$,2)
masking_value    |
priority         | 100
masking_filter   | anon.partial(phone,2,$$X-XXX-XX$$,2)
trusted_schema   | t
```

6/ Partitionnement sous PostgreSQL



- Ses principes et intérêts
- Historique
- Les différents types

6.1 PRINCIPE & INTÉRÊTS DU PARTITIONNEMENT

- Faciliter la maintenance de gros volumes
  - VACUUM (FULL), réindexation, déplacements, sauvegarde logique…
- Performances
  - parcours complet sur de plus petites tables
  - statistiques par partition plus précises
  - purge par partitions entières
  - pg_dump parallélisable
  - tablespaces différents (données froides/chaudes)
- Attention à la maintenance sur le code

Maintenir de très grosses tables peut devenir fastidieux, voire impossible : VACUUM FULL trop long, espace disque insuffisant, autovacuum pas assez réactif, réindexation interminable… Il est aussi aberrant de conserver beaucoup de données d'archives dans des tables lourdement sollicitées pour les données récentes.

Le partitionnement consiste à séparer une même table en plusieurs sous-tables (partitions) manipulables en tant que tables à part entière.

Maintenance

La maintenance s'effectue sur les partitions plutôt que sur l'ensemble complet des données. En particulier, un VACUUM FULL ou une réindexation peuvent s'effectuer partition par partition, ce qui permet de limiter les interruptions en production, et lisser la charge. pg_dump ne sait pas paralléliser la sauvegarde d'une table volumineuse et non partitionnée, mais parallélise celle de différentes partitions d'une même table.

C'est aussi un moyen de déplacer une partie des données dans un autre *tablespace* pour des raisons de place, ou pour déporter les parties les moins utilisées de la table vers des disques plus lents et moins chers.

Parcours complet de partitions

Certaines requêtes (notamment décisionnelles) ramènent tant de lignes, ou ont des critères si complexes, qu'un parcours complet de la table est souvent privilégié par l'optimiseur.

Un partitionnement, souvent par date, permet de ne parcourir qu'une ou quelques partitions au lieu de l'ensemble des données. C'est le rôle de l'optimiseur de choisir la partition (*partition pruning*), par exemple celle de l'année en cours, ou des mois sélectionnés.

Suppression des partitions

La suppression de données parmi un gros volume peut poser des problèmes d'accès concurrents ou de performance, par exemple dans le cas de purges.

En configurant judicieusement les partitions, on peut résoudre cette problématique en supprimant une partition (`DROP TABLE nompartition ;`), ou en la *détachant* (`ALTER TABLE table_partitionnee DETACH PARTITION nompartition ;`) pour l'archiver (et la réattacher au besoin) ou la supprimer ultérieurement.

D'autres optimisations sont décrites dans ce billet de blog d'Adrien Nayrat[1] : statistiques plus précises au niveau d'une partition, réduction plus simple de la fragmentation des index, jointure par rapprochement des partitions…

La principale difficulté d'un système de partitionnement consiste à partitionner avec un impact minimal sur la maintenance du code par rapport à une table classique.

[1] https://blog.anayrat.info/2021/09/01/cas-dusages-du-partitionnement-natif-dans-postgresql/

6.2 PARTITIONNEMENT APPLICATIF

- Intégralement géré au niveau applicatif
- Complexité pour le développeur
- Intégrité des liens entre les données ?
- Réinvention de la roue

L'application peut gérer le partitionnement elle-même, par exemple en créant des tables numérotées par mois, année… Le moteur de PostgreSQL ne voit que des tables classiques et ne peut assurer l'intégrité entre ces données.

C'est au développeur de réinventer la roue : choix de la table, gestion des index… La lecture des données qui concerne plusieurs tables peut devenir délicate.

Ce modèle extrêmement fréquent est bien sûr à éviter.

6.3 MÉTHODES DE PARTITIONNEMENT INTÉGRÉES À POSTGRESQL

- Partitionnement par héritage (historique, < v10)
- Partitionnement déclaratif (>=v10, préférer v13+)

Un partitionnement entièrement géré par le moteur, n'existe réellement que depuis la version 10 de PostgreSQL. Il a été grandement amélioré en versions 11 et 12, en fonctionnalités comme en performances.

Jusqu'à PostgreSQL 9.6 n'existaient que le partitionnement dit par héritage, nettement moins flexible, et bien sûr le partitionnement géré intégralement par l'applicatif.

6.4 PARTITIONNEMENT PAR HÉRITAGE

- Syntaxe :

```
CREATE TABLE primates (debout boolean) INHERITS (mammiferes) ;
```

- Table mère :
  - définie normalement
- Tables filles :
  - héritent des propriétés de la table mère
  - mais pas des contraintes, index et droits
  - colonnes supplémentaires possibles
- Insertion applicative ou par trigger (lent !)

PostgreSQL permet de créer des tables qui héritent les unes des autres.

L'héritage d'une table mère transmet les propriétés suivantes à la table fille :

- les colonnes, avec le type et les valeurs par défaut ;
- les contraintes CHECK.

Les tables filles peuvent ajouter leurs propres colonnes. Par exemple :

```
CREATE TABLE animaux (nom text PRIMARY KEY);

INSERT INTO animaux VALUES ('Éponge');
INSERT INTO animaux VALUES ('Ver de terre');


CREATE TABLE cephalopodes (nb_tentacules integer)
INHERITS (animaux);

INSERT INTO cephalopodes VALUES ('Poulpe', 8);

CREATE TABLE vertebres (
    nb_membres integer
)
INHERITS (animaux);

CREATE TABLE tetrapodes () INHERITS (vertebres);

ALTER TABLE ONLY tetrapodes
ALTER COLUMN nb_membres
SET DEFAULT 4;
```

```
CREATE TABLE poissons (eau_douce boolean)
INHERITS (tetrapodes);

INSERT INTO poissons (nom, eau_douce) VALUES ('Requin', false);
INSERT INTO poissons (nom, nb_membres, eau_douce) VALUES ('Anguille', 0, false);
```

La table `poissons` possède les champs des tables dont elle hérite :

```
\d+ poissons
                                    Table "public.poissons"
   Column    |  Type   | Collation | Nullable | Default | Storage  | Stats target |
 ↪  Description
-------------+---------+-----------+----------+---------+----------+--------------+--
 ↪  -----------
 nom         | text    |           | not null |         | extended |              |
 nb_membres  | integer |           |          | 4       | plain    |              |
 eau_douce   | boolean |           |          |         | plain    |              |
Inherits: tetrapodes
Access method: heap
```

On peut créer toute une hiérarchie avec des branches parallèles, chacune avec ses colonnes propres :

```
CREATE TABLE reptiles (venimeux boolean)
INHERITS (tetrapodes);

INSERT INTO reptiles VALUES ('Crocodile', 4, false);
INSERT INTO reptiles VALUES ('Cobra', 0, true);

CREATE TABLE mammiferes () INHERITS (tetrapodes);

CREATE TABLE cetartiodactyles (
    cornes boolean,
    bosse boolean
)
INHERITS (mammiferes);

INSERT INTO cetartiodactyles VALUES ('Girafe', 4, true, false);
INSERT INTO cetartiodactyles VALUES ('Chameau', 4, false, true);

CREATE TABLE primates (debout boolean)
INHERITS (mammiferes);

INSERT INTO primates (nom, debout) VALUES ('Chimpanzé', false);
INSERT INTO primates (nom, debout) VALUES ('Homme', true);

\d+ primates

                                    Table "public.primates"
   Column    |  Type   | Collation | Nullable | Default | Storage  | Stats target |
    ↪  Description
-------------+---------+-----------+----------+---------+----------+--------------+--
 ↪  -----------
 nom         | text    |           | not null |         | extended |              |
 nb_membres  | integer |           |          | 4       | plain    |              |
 debout      | boolean |           |          |         | plain    |              |
```

```
Inherits: mammiferes
Access method: heap
```

On remarquera que la clé primaire manque. En effet, l'héritage ne transmet pas :

- les contraintes d'unicité et référentielles ;
- les index ;
- les droits.

Chaque table possède ses propres lignes :

```
SELECT * FROM poissons ;
```

```
   nom    | nb_membres | eau_douce
----------+------------+-----------
 Requin   |          4 | f
 Anguille |          0 | f
```

Par défaut une table affiche aussi le contenu de ses tables filles et les colonnes communes :

```
SELECT * FROM animaux ORDER BY 1 ;
```

```
   nom
--------------
 Anguille
 Chameau
 Chimpanzé
 Cobra
 Crocodile
 Éponge
 Girafe
 Homme
 Poulpe
 Requin
 Ver de terre
```

```
SELECT * FROM tetrapodes ORDER BY 1 ;
```

```
    nom    | nb_membres
-----------+------------
 Anguille  |          0
 Chameau   |          4
 Chimpanzé |          4
 Cobra     |          0
 Crocodile |          4
 Girafe    |          4
 Homme     |          4
 Requin    |          4
```

```
EXPLAIN SELECT * FROM tetrapodes ORDER BY 1 ;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Sort  (cost=420.67..433.12 rows=4982 width=36)
   Sort Key: tetrapodes.nom
   ->  Append  (cost=0.00..114.71 rows=4982 width=36)
         ->  Seq Scan on tetrapodes  (cost=0.00..0.00 rows=1 width=36)
```

```
         ->  Seq Scan on poissons  (cost=0.00..22.50 rows=1250 width=36)
         ->  Seq Scan on reptiles  (cost=0.00..22.50 rows=1250 width=36)
         ->  Seq Scan on mammiferes  (cost=0.00..0.00 rows=1 width=36)
         ->  Seq Scan on cetartiodactyles  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on primates  (cost=0.00..22.50 rows=1250 width=36)
```

Pour ne consulter que le contenu de la table sans ses filles :

```
SELECT * FROM ONLY animaux ;
```

```
     nom
--------------
 Éponge
 Ver de terre
```

En conséquence, on a bien affaire à des tables indépendantes. Rien n'empêche d'avoir des doublons entre la table mère et la table fille. Cela empêche aussi bien sûr la mise en place de clé étrangère, puisqu'une clé étrangère s'appuie sur une contrainte d'unicité de la table référencée. Lors d'une insertion, voire d'une mise à jour, le choix de la table cible se fait par l'application ou un trigger sur la table mère.

Il faut être vigilant à bien recréer les contraintes et index manquants ainsi qu'à attribuer les droits sur les objets de manière adéquate. L'une des erreurs les plus fréquentes est d'oublier de créer les contraintes, index et droits qui n'ont pas été transmis.

Ce type de partitionnement est un héritage des débuts de PostgreSQL, à l'époque de la mode des « bases de donnée objet ». Dans la pratique, dans les versions antérieures à la version 10, l'héritage était utilisé pour mettre en place le partitionnement. La maintenance des index, des contraintes et la nécessité d'un trigger pour aiguiller les insertions vers la bonne table fille, ne facilitaient pas la maintenance. Les performances en écritures étaient bien en-deçà des tables classiques ou du nouveau partitionnement déclaratif (voir comparaison plus bas).

Si le partitionnement par héritage fonctionne toujours sur les versions récentes de PostgreSQL, il est déconseillé pour les nouveaux développements.

6.5 PARTITIONNEMENT DÉCLARATIF

- Préférer version 13+
- Mise en place et administration simplifiées (intégrées au moteur)
- Gestion automatique des lectures et écritures
- Partitions
  - attacher/détacher une partition
  - contrainte implicite de partitionnement
  - expression possible pour la clé de partitionnement
  - sous-partitions possibles
  - partition par défaut

Le partitionnement déclaratif est le système à privilégier de nos jours. Apparu en version 10, il est à présent mûr.

Son but est de permettre une mise en place et une administration simples des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL, comme `CREATE TABLE` et `ALTER TABLE`, pour attacher (`ATTACH PARTITION`) et détacher des partitions (`DETACH PARTITION`).

Au niveau de la simplification de la mise en place par rapport à l'ancien partitionnement par héritage, on peut noter qu'il n'est pas nécessaire de créer une fonction *trigger* ni d'ajouter des *triggers* pour gérer les insertions et mises à jour. Le routage est géré de façon automatique en fonction de la définition des partitions, au besoin vers une partition par défaut. Contrairement au partitionnement par héritage, la table partitionnée ne contient pas elle-même de ligne, ce n'est qu'une coquille vide. Du fait de ce routage automatique, les durées d'insertions ne subissent pas de pénalité.

6.5.1 Partitionnement par liste

- Liste de valeurs par partition
- Clé de partitionnement forcément mono-colonne
- Syntaxe :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1) ;

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
…
```

Il est possible de partitionner une table par valeurs. Ce type de partitionnement fonctionne uniquement avec une clé de partitionnement mono-colonne.

Voici un exemple de création d'une table partitionnée et de ces partitions :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);

CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
```

Les noms des partitions sont à définir par l'utilisateur, il n'y a pas d'automatisme ni de convention particulière.

Et voici quelques insertions de données :

```
=# INSERT INTO t1 VALUES (1);
INSERT 0 1

=# INSERT INTO t1 VALUES (2);
INSERT 0 1

=# INSERT INTO t1 VALUES (5);
INSERT 0 1
```

Lors de l'insertion, les données sont correctement redirigées vers leur partition, comme le montre cette requête :

```
SELECT tableoid::regclass, * FROM t1;

 tableoid | c1 | c2
----------+----+----
 t1_a     |  1 |
 t1_a     |  2 |
 t1_b     |  5 |
```

Il est aussi possible d'interroger les partitions séparément :

```
SELECT * FROM t1_a ;

 c1 | c2
----+----
  1 |
  2 |
```

Si aucune partition correspondant à la clé insérée n'est trouvée et qu'aucune partition par défaut n'est déclarée (fonctionnalité disponible qu'à partir de la version 11), une erreur se produit.

```
=# INSERT INTO t1 VALUES (0);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

=# INSERT INTO t1 VALUES (6);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).
```

Si la clé de partitionnement d'une ligne est modifiée par un UPDATE, la ligne change automatiquement de partition (sauf en version 10, où ce n'est pas implémenté et provoque une erreur).

6.5.2 Partitionnement par intervalle

- Intervalle de valeurs par partition
- Clé de partitionnement mono- ou multi-colonnes
- Bornes « infinies » :
  - MINVALUE / MAXVALUE
- Syntaxe :

```
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);

CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
…
```

Voici un exemple de création de la table partitionnée et de deux partitions :

```
=# CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE

=# CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) to (100);
CREATE TABLE
```

```
=# CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
```

Le MAXVALUE indique la valeur maximale du type de données : `t2_2` acceptera donc tous les entiers supérieurs ou égaux à 100.

Noter que les bornes supérieures des partitions sont **exclues** ! La valeur 100 ira donc dans la seconde partition.

Lors de l'insertion, les données sont redirigées vers leur partition, s'il y en a une :

```
=# INSERT INTO t2 VALUES (0);
ERROR:  no PARTITION OF relation "t2" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

=# INSERT INTO t2 VALUES (10, 'dix');
INSERT 0 1

=# INSERT INTO t2 VALUES (100, 'cent');
INSERT 0 1

=# INSERT INTO t2 VALUES (10000, 'dix mille');
INSERT 0 1

=# SELECT * FROM t2 ;

  c1   |    c2
-------+-----------
    10 | dix
   100 | cent
 10000 | dix mille
(3 lignes)

=# SELECT * FROM t2_2 ;

  c1   |    c2
-------+-----------
   100 | cent
 10000 | dix mille
(2 lignes)
```

La colonne système `tableoid` permet de connaître la partition d'où provient une ligne :

```
=# SELECT ctid, tableoid::regclass, * FROM t2 ;

 ctid  | tableoid |  c1   |    c2
-------+----------+-------+-----------
 (0,1) | t2_1     |    10 | dix
 (0,1) | t2_2     |   100 | cent
 (0,2) | t2_2     | 10000 | dix mille
```

6.5.3 Partitionnement par hachage

- À partir de la version 11
- Hachage de valeurs par partition
  - indiquer un modulo et un reste
- Clé de partitionnement mono- ou multi-colonnes
- Syntaxe :

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);

CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder
 ↪  0);
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder
 ↪  1);
CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder
 ↪  2);
```

Voici comment partitionner par hachage une table en trois partitions :

```
=# CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);
CREATE TABLE

=# CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 0);
CREATE TABLE

=# CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 1);
CREATE TABLE

=# CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 2);
CREATE TABLE
```

Une grosse insertion de données répartira les données de manière équitable entre les différentes partitions :

```
=# INSERT INTO t3 SELECT generate_series(1, 1000000);
INSERT 0 1000000

=# SELECT relname, count(*)
FROM t3
JOIN pg_class ON t3.tableoid=pg_class.oid
GROUP BY 1;

 relname | count
---------+--------
 t3_1    | 333263
 t3_2    | 333497
 t3_3    | 333240
```

6.5.4 Clé de partitionnement multi-colonnes

- Clé sur plusieurs colonnes acceptée
  - si partitionnement par intervalle ou hash, pas pour liste
- Syntaxe :

```
CREATE TABLE t1(c1 integer, c2 text, c3 date)
PARTIONNED BY RANGE (c1, c3) ;

CREATE TABLE t1_a PARTITION OF t1
FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11') ;
…
```

Quand on utilise le partitionnement par intervalle, il est possible de créer les partitions en utilisant plusieurs colonnes.

On profitera de l'exemple ci-dessous pour montrer l'utilisation conjointe de tablespaces différents.

Commençons par créer les tablespaces :

```
=# CREATE TABLESPACE ts0 LOCATION '/tablespaces/ts0';
CREATE TABLESPACE

=# CREATE TABLESPACE ts1 LOCATION '/tablespaces/ts1';
CREATE TABLESPACE

=# CREATE TABLESPACE ts2 LOCATION '/tablespaces/ts2';
CREATE TABLESPACE

=# CREATE TABLESPACE ts3 LOCATION '/tablespaces/ts3';
CREATE TABLESPACE
```

Créons maintenant la table partitionnée et deux partitions :

```
=# CREATE TABLE t2(c1 integer, c2 text, c3 date not null)
       PARTITION BY RANGE (c1, c3);
CREATE TABLE

=# CREATE TABLE t2_1 PARTITION OF t2
       FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11')
       TABLESPACE ts1;
CREATE TABLE

=# CREATE TABLE t2_2 PARTITION OF t2
       FOR VALUES FROM (100,'2017-08-11') TO (200, '2017-08-12')
       TABLESPACE ts2;
CREATE TABLE
```

La borne supérieure étant exclue, la valeur `(100, '2017-08-11')` fera donc partie de la seconde partition.

Si les valeurs sont bien comprises dans les bornes :

```
=# INSERT INTO t2 VALUES (1, 'test', '2017-08-10');
INSERT 0 1

=# INSERT INTO t2 VALUES (150, 'test2', '2017-08-11');
INSERT 0 1
```

Si la valeur pour `c1` est trop petite :

```
=# INSERT INTO t2 VALUES (0, 'test', '2017-08-10');
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (0, 2017-08-10).
```

Si la valeur pour `c3` (colonne de type date) est antérieure :

```
=# INSERT INTO t2 VALUES (1, 'test', '2017-08-09');
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (1, 2017-08-09).
```

Les valeurs spéciales `MINVALUE` et `MAXVALUE` permettent de ne pas indiquer de valeur de seuil limite. Les partitions `t2_0` et `t2_3` pourront par exemple être déclarées comme suit et permettront d'insérer les lignes qui étaient ci-dessus en erreur.

```
=# CREATE TABLE t2_0 PARTITION OF t2
       FOR VALUES FROM (MINVALUE, MINVALUE) TO (1,'2017-08-10')
       TABLESPACE ts0;

=# CREATE TABLE t2_3 PARTITION OF t2
       FOR VALUES FROM (200,'2017-08-12') TO (MAXVALUE, MAXVALUE)
       TABLESPACE ts3;
```

Enfin, on peut consulter la table `pg_class` afin de vérifier la présence des différentes partitions :

```
=# ANALYZE t2;

ANALYZE

=# SELECT relname,relispartition,relkind,reltuples
           FROM pg_class WHERE relname LIKE 't2%';

 relname | relispartition | relkind | reltuples
---------+----------------+---------+-----------
 t2      | f              | p       |         0
 t2_0    | t              | r       |         2
 t2_1    | t              | r       |         1
 t2_2    | t              | r       |         1
 t2_3    | t              | r       |         0
```

6.5.5 Performances en insertion

t1 (non partitionnée) :

```
INSERT INTO t1 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 7774.443 ms (00:07.774)
```

t2 (nouveau partitionnement) :

```
INSERT INTO t2 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 8062.570 ms (00:08.063)
```

t3 (ancien partitionnement par héritage) :

```
INSERT INTO t3 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 68928.431 ms (01:08.928)
```

La table *t1* est une table non partitionnée. Elle a été créée comme suit :

```
CREATE TABLE t1 (c1 integer, c2 text);
```

La table `t2` est une table partitionnée utilisant les nouvelles fonctionnalités de la version 10 de PostgreSQL :

```
CREATE TABLE t2 (c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (      0) TO ( 1000000);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (1000000) TO ( 2000000);
CREATE TABLE t2_3 PARTITION OF t2 FOR VALUES FROM (2000000) TO ( 3000000);
CREATE TABLE t2_4 PARTITION OF t2 FOR VALUES FROM (3000000) TO ( 4000000);
CREATE TABLE t2_5 PARTITION OF t2 FOR VALUES FROM (4000000) TO ( 5000000);
CREATE TABLE t2_6 PARTITION OF t2 FOR VALUES FROM (5000000) TO ( 6000000);
CREATE TABLE t2_7 PARTITION OF t2 FOR VALUES FROM (6000000) TO ( 7000000);
CREATE TABLE t2_8 PARTITION OF t2 FOR VALUES FROM (7000000) TO ( 8000000);
CREATE TABLE t2_9 PARTITION OF t2 FOR VALUES FROM (8000000) TO ( 9000000);
CREATE TABLE t2_0 PARTITION OF t2 FOR VALUES FROM (9000000) TO (10000000);
```

Enfin, la table *t3* est une table utilisant l'ancienne méthode de partitionnement :

```
CREATE TABLE t3 (c1 integer, c2 text);
CREATE TABLE t3_1 (CHECK (c1 BETWEEN       0 AND  999999)) INHERITS (t3);
CREATE TABLE t3_2 (CHECK (c1 BETWEEN 1000000 AND 1999999)) INHERITS (t3);
CREATE TABLE t3_3 (CHECK (c1 BETWEEN 2000000 AND 2999999)) INHERITS (t3);
CREATE TABLE t3_4 (CHECK (c1 BETWEEN 3000000 AND 3999999)) INHERITS (t3);
CREATE TABLE t3_5 (CHECK (c1 BETWEEN 4000000 AND 4999999)) INHERITS (t3);
CREATE TABLE t3_6 (CHECK (c1 BETWEEN 5000000 AND 5999999)) INHERITS (t3);
CREATE TABLE t3_7 (CHECK (c1 BETWEEN 6000000 AND 6999999)) INHERITS (t3);
CREATE TABLE t3_8 (CHECK (c1 BETWEEN 7000000 AND 7999999)) INHERITS (t3);
CREATE TABLE t3_9 (CHECK (c1 BETWEEN 8000000 AND 8999999)) INHERITS (t3);
```

```
CREATE TABLE t3_0 (CHECK (c1 BETWEEN 9000000 AND 9999999)) INHERITS (t3);

CREATE OR REPLACE FUNCTION insert_into() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.c1    BETWEEN       0 AND  999999 THEN
    INSERT INTO t3_1 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 1000000 AND 1999999 THEN
    INSERT INTO t3_2 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 2000000 AND 2999999 THEN
    INSERT INTO t3_3 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 3000000 AND 3999999 THEN
    INSERT INTO t3_4 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 4000000 AND 4999999 THEN
    INSERT INTO t3_5 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 5000000 AND 5999999 THEN
    INSERT INTO t3_6 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 6000000 AND 6999999 THEN
    INSERT INTO t3_7 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 7000000 AND 7999999 THEN
    INSERT INTO t3_8 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 8000000 AND 8999999 THEN
    INSERT INTO t3_9 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 9000000 AND 9999999 THEN
    INSERT INTO t3_0 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_insert_t3 BEFORE INSERT ON t3 FOR EACH ROW EXECUTE PROCEDURE
 ↪  insert_into();
```

6.5.6 Partition par défaut

- Pour le partitionnement par liste et par intervalle
- Toutes les données n'allant pas dans les partitions définies iront dans la partition par défaut

```
CREATE TABLE t2_autres PARTITION OF t2 DEFAULT ;
```

Ajouter une partition par défaut permet de ne plus avoir d'erreur au cas où une partition n'est pas définie.

En voici un exemple à partir de la table `t1` définie ci-dessus :

```
=# INSERT INTO t1 VALUES (0);
```

```
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

=# INSERT INTO t1 VALUES (6);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).

=# CREATE TABLE t1_defaut PARTITION OF t1 DEFAULT;
CREATE TABLE

=# INSERT INTO t1 VALUES (0);
INSERT 0 1

=# INSERT INTO t1 VALUES (6);
INSERT 0 1

=# SELECT tableoid::regclass, * FROM t1;

 tableoid  | c1 | c2
-----------+----+----
 t1_a      |  1 |
 t1_a      |  2 |
 t1_b      |  5 |
 t1_defaut |  0 |
 t1_defaut |  6 |
```

Comme la partition par défaut risque d'être parcourue intégralement à chaque ajout d'une nouvelle partition, il vaut mieux la garder de petite taille.

Un partitionnement par hachage ne peut posséder de table par défaut.

6.5.7 Attacher une partition

```
ALTER TABLE … ATTACH PARTITION … FOR VALUES …;
```

- La table doit préexister
- Vérification du respect de la contrainte par les données existantes
  - parcours complet de la table
  - potentiellement lent !
    * …sauf si ajout préalable d'une contrainte CHECK identique
- Si la partition par défaut a des données qui iraient dans cette partition :
  - erreur à l'ajout de la nouvelle partition
  - détacher la partition par défaut
  - ajouter la nouvelle partition
  - déplacer les données de l'ancienne partition par défaut
  - ré-attacher la partition par défaut

Ajouter une table comme partition d'une table partitionnée est possible mais cela nécessite de vérifier que la contrainte de partitionnement est valide pour toute la table attachée, et que la partition par défaut ne contient pas de données qui devraient figurer dans cette nouvelle partition.

Cela résulte en un parcours complet de la table attachée, et de la partition par défaut si elle existe, ce qui sera d'autant plus lent qu'elles sont volumineuses.

Ce peut être très coûteux en disque, mais le plus gros problème est la durée du verrou sur la table partitionnée, pendant toute cette opération. Il est donc conseillé d'ajouter une contrainte CHECK adéquate **avant** l'ATTACH : la durée du verrou sera raccourcie d'autant.

Si des lignes pour cette nouvelle partition figurent déjà dans la partition par défaut, des opérations supplémentaires sont à réaliser pour les déplacer. Ce n'est pas automatique.

6.5.8 Détacher une partition

```
ALTER TABLE … DETACH PARTITION …
```

- Simple et rapide
- Mais nécessite un verrou exclusif
  - option CONCURRENTLY (v14+)

Détacher une partition est beaucoup plus rapide qu'en attacher une. En effet, il n'est pas nécessaire de procéder à des vérifications sur les données des partitions.

Cependant, il est nécessaire d'acquérir un verrou exclusif sur la table partitionnée, ce qui peut prendre du temps si des transactions sont en cours d'exécution. L'option CONCURRENTLY (à partir de PostgreSQL 14) mitige le problème malgré quelques restrictions[2].

La partition détachée devient alors une table tout à fait classique. Elle conserve les index, contraintes, etc. dont elle a pu hériter de la table partitionnée originale.

6.5.9 Supprimer une partition

```
DROP TABLE nom_partition ;
```

Une partition étant une table, supprimer la table revient à supprimer la partition, et bien sûr les données qu'elle contient. Il n'y a pas besoin de la détacher explicitement auparavant.

L'opération est simple et rapide, mais demande un verrou exclusif.

[2] https://docs.postgresql.fr/14/sql-altertable.html#SQL-ALTERTABLE-DETACH-PARTITION

6.5.10 Fonctions de gestion et vues système

- Disponibles à partir de la v12
  - `\dP`
  - `pg_partition_tree ('logs')` : liste entière des partitions
  - `pg_partition_root ('logs_2019')` : racine d'une partition
  - `pg_partition_ancestors ('logs_201901')` : parents d'une partition

Voici le jeu de tests pour l'exemple qui suivra. Il illustre également l'utilisation de sous-partitions (ici sur la même clé, mais cela n'a rien d'obligatoire).

```
-- Table partitionnée
CREATE TABLE logs (dreception timestamptz, contenu text) PARTITION BY
 ↪  RANGE(dreception);

-- Partition 2018, elle-même partitionnée
CREATE TABLE logs_2018 PARTITION OF logs FOR VALUES FROM ('2018-01-01') TO
 ↪  ('2019-01-01')
                       PARTITION BY range(dreception);
-- Sous-partitions 2018
CREATE TABLE logs_201801 PARTITION OF logs_2018 FOR VALUES FROM ('2018-01-01') TO
 ↪  ('2018-02-01');
CREATE TABLE logs_201802 PARTITION OF logs_2018 FOR VALUES FROM ('2018-02-01') TO
 ↪  ('2018-03-01');
…
-- Idem en 2019
CREATE TABLE logs_2019 PARTITION OF logs FOR VALUES FROM ('2019-01-01') TO
 ↪  ('2020-01-01')
                       PARTITION BY range(dreception);
CREATE TABLE logs_201901 PARTITION OF logs_2019 FOR VALUES FROM ('2019-01-01') TO
 ↪  ('2019-02-01');
…
```

Et voici le test des différentes fonctions :

```
=# SELECT pg_partition_root('logs_2019');

 pg_partition_root
-------------------
 logs

=# SELECT pg_partition_root('logs_201901');

 pg_partition_root
-------------------
 logs
```

```
=# SELECT pg_partition_ancestors('logs_2018');

 pg_partition_ancestors
------------------------
 logs_2018
 logs

=# SELECT pg_partition_ancestors('logs_201901');

 pg_partition_ancestors
------------------------
 logs_201901
 logs_2019
 logs

=# SELECT * FROM pg_partition_tree('logs');

    relid    | parentrelid | isleaf | level
-------------+-------------+--------+-------
 logs        |             | f      |     0
 logs_2018   | logs        | f      |     1
 logs_2019   | logs        | f      |     1
 logs_201801 | logs_2018   | t      |     2
 logs_201802 | logs_2018   | t      |     2
 logs_201901 | logs_2019   | t      |     2
```

Noter les propriétés de « feuille » (*leaf*) et le niveau de profondeur dans le partitionnement.

Sous psql, `\d` affichera toutes les tables, partitions comprises, ce qui peut vite encombrer l'affichage.

À partir de la version 12 du client, `\dP` affiche uniquement les tables et index partitionnés :

```
=# \dP
                Liste des relations partitionnées
 Schéma |   Nom    | Propriétaire |        Type         |  Table
--------+----------+--------------+---------------------+----------
 public | logs     | postgres     | table partitionnée |
 public | t2       | postgres     | index partitionné  | bigtable
```

La table système `pg_partitioned_table`[3] permet des requêtes plus complexes. Le champ `pg_class.relpartbound`[4] contient les définitions des clés de partitionnement.

6.5.11 Indexation

- Propagation automatique (v11+)
- Index supplémentaires par partition possibles
- Clés étrangères entre tables partitionnées (v12+)

[3] https://www.postgresql.org/docs/current/catalog-pg-partitioned-table.html
[4] https://www.postgresql.org/docs/14/catalog-pg-class.html

À partir de la version 11, les index sont propagés de la table mère aux partitions : tout index créé sur la table partitionnée sera automatiquement créé sur les partitions existantes. Toute nouvelle partition disposera des index de la table partitionnée. La suppression d'un index se fait sur la table partitionnée et concernera toutes les partitions. Il n'est pas possible de supprimer un tel index d'une seule partition.

Gérer des index manuellement sur certaines partitions est possible. Par exemple, on peut n'avoir besoin de certains index que sur les partitions de données récentes, et ne pas les créer sur des partitions de données d'archives. (Et avec PostgreSQL 10, les index doivent ainsi être gérés à la main, partition par partition : cette version ne permet donc pas de gérer des clés primaires sur toute la table.)

Toujours à partir de PostgreSQL 11, on peut créer une clé primaire ou unique sur une table partitionnée (mais elle devra contenir toutes les colonnes de la clé de partitionnement) ; ainsi qu'une clé étrangère d'une table partitionnée *vers* une table normale.

Cependant, il faut au minimum PostgreSQL 12 pour pouvoir créer une clé étrangère *vers* une table partitionnée. Par exemple, si des tables `ventes` et `lignes_ventes` sont partitionnées, poser une clé étrangère entre les deux échouera avant PostgreSQL 12 :

```
ALTER TABLE lignes_ventes
ADD CONSTRAINT lignes_ventes_ventes_fk
FOREIGN KEY (vente_id) REFERENCES ventes(vente_id) ;
```

(En version 11, la colonne `lignes_ventes.vente_id` peut tout de même être indexée, et il reste possible d'ajouter manuellement des contraintes entre les partitions une à une, si les clés de partitionnement sont compatibles.)

6.5.12 Opérations de maintenance

- Changement de tablespace
- autovacuum/analyze
  - sur les partitions comme sur toute table
- `VACUUM`, `VACUUM FULL`, `ANALYZE`
  - sur table mère : redescendent sur les partitions
- `REINDEX`
  - avant v14 : uniquement par partition
- `ANALYZE`
  - prévoir aussi sur la table mère (manuellement…)

Les opérations de maintenance profitent grandement du fait de pouvoir scinder les opérations en autant d'étapes qu'il y a de partitions.

Des données « froides » peuvent être déplacées dans un autre tablespace sur des disques moins chers, partition par partition, ce qui est impossible avec une table monolithique :

```
ALTER TABLE pgbench_accounts_8 SET TABLESPACE hdd ;
```

L'autovacuum et l'autoanalyze fonctionnent normalement et indépendamment sur chaque partition, comme sur les tables classiques. Ainsi ils peuvent se déclencher plus souvent sur les partitions actives. Par rapport à une grosse table monolithique, il y a moins souvent besoin de régler l'autovacuum.

Les ordres `ANALYZE` et `VACUUM` peuvent être effectués sur une partition, mais aussi sur la table partitionnée, auquel cas l'ordre redescendra en cascade sur les partitions (l'option `VERBOSE` permet de le vérifier). Les statistiques seront calculées par partition, donc plus précises.

Reconstruire une table partitionnée avec `VACUUM FULL` se fera généralement partition par partition. Dans certains cas, l'opération devient possible seulement grâce au partitionnement : le verrou sur une table monolithique serait trop long, ou l'espace disque total serait insuffisant.

Au-dessus des partitions, noter cependant ces spécificités sur les tables partitionnées :

REINDEX :

À partir de PostgreSQL 14, un `REINDEX` sur la table partitionnée réindexe toutes les partitions automatiquement. Dans les versions précédentes, il faut réindexer partition par partition.

ANALYZE :

L'autovacuum ne crée pas spontanément de statistiques sur les données pour la table partitionnée dans son ensemble, mais uniquement partition par partition. Pour obtenir des statistiques sur toute la table partitionnée, il faut exécuter manuellement :

```
ANALYZE table_partitionnée ;
```

6.5.13 Intérêts du partitionnement déclaratif

- Souple
- Performant
- Intégration au moteur

Par rapport à l'ancien système, le partitionnement déclaratif n'a que des avantages : rapidité d'insertion, souplesse dans le choix du partitionnement, intégration au moteur (ce qui garantit l'intégrité des données)…

6.5.14 Limitations du partitionnement déclaratif et versions

- Temps de planification ! Max ~100 partitions si transactions courtes
- Pas de création automatique des partitions
- Pas d'héritage multiple, schéma fixe
- Partitions distantes sans propagation d'index
- PostgreSQL >= 13 conseillée !
  - v10 : ni partition par défaut, ni propagation des index & contraintes
  - v10/11 : pas de clé étrangère *vers* une table partitionnée
  - v10/11/12 : pas de triggers `BEFORE UPDATE … FOR EACH ROW`
  - contournement : travailler par partition

Une table partitionnée ne peut être convertie en table classique, ni vice-versa. (Par contre, une table classique peut être attachée comme partition, ou une partition détachée).

Les partitions ont forcément le même schéma de données que leur partition mère.

Leur création n'est pas automatisée : il faut les créer par avance manuellement, et éventuellement prévoir une partition par défaut pour les cas qui ont pu être oubliés.

Les clés de partition ne doivent pas se recouvrir. Les contraintes ne peuvent s'exercer qu'au sein d'une même partition : les clés d'unicité doivent donc inclure toute la clé de partitionnement, les contraintes d'exclusion ne peuvent vérifier toutes les partitions.

Il n'y a pas de notion d'héritage multiple.

Une limitation sérieuse du partitionnement tient au temps de planification qui augmente très vite avec le nombre de partitions, même petites. En général, on considère qu'il ne faut pas dépasser 100 partitions si l'on ne veut pas pénaliser les transactions courtes.

Cela est moins un problème pour les requêtes longues (analytiques). Pour contourner cette limite, il est possible de manipuler directement les partitions, s'il est facile pour le développeur (ou le générateur de code…) de trouver leur nom.

L'ordre `CLUSTER`, pour réécrire une table dans l'ordre d'un index donné, ne fonctionne pas pour les tables partitionnées ; il doit être exécuté manuellement table par table.

Il est possible d'attacher comme partitions des tables distantes, généralement déclarée avec `postgres_fdw` ; cependant la propagation d'index ne fonctionnera pas sur ces tables. Il faudra les créer manuellement sur les instances distantes. (Restriction supplémentaire en version 10 : les

partitions distantes ne sont accessibles qu'en lecture, si accédées *via* la table mère.) Un TRUNCATE d'une table distante n'est pas possible avant PostgreSQL 14.

Les partitions par défaut n'existent pas en version 10.

Les limitations sur les index et clés primaires et étrangères avant la version 12 ont été évoquées plus haut.

Les triggers de lignes ne se propagent pas en version 10. En v11, on peut créer des triggers AFTER UPDATE … FOR EACH ROW, mais les BEFORE UPDATE … FOR EACH ROW ne peuvent toujours pas être créés sur la table mère. Il reste là encore la possibilité de les créer partition par partition au besoin. À partir de la version 13, les triggers BEFORE UPDATE … FOR EACH ROW sont possibles, mais il ne permettent pas de modifier la partition de destination.

Enfin, la version 10 ne permet pas de faire une mise à jour (UPDATE) d'une ligne où la clé de partitionnement est modifiée de telle façon que la ligne doit changer de partition. Il faut faire un DELETE et un INSERT à la place. La version 11 gère mieux ce cas en déplaçant directement la ligne dans la bonne partition.

Toujours en version 10 uniquement, un UPDATE sur une ligne ne peut encore la faire changer de changer de partition : il faut faire un DELETE et un INSERT à la place.

On constate que des limitations évoquées plus haut dépendent des versions de PostgreSQL. Si le partitionnement vous intéresse, il est conseillé d'utiliser une version la plus récente possible, au moins PostgreSQL 13.

6.6 EXTENSIONS & OUTILS

- Extension pg_partman[5]
  - automatisation
- Extensions dédiées à un domaine :
  - timescaledb
  - citus

L'extension pg_partman[6], de Crunchy Data, est un complément aux systèmes de partitionnement de PostgreSQL. Elle est apparue d'abord pour automatiser le partitionnement par héritage. Elle peut être utile avec le partitionnement déclaratif, pour simplifier la maintenance d'un partitionnement sur une échelle temporelle ou de valeurs (par *range*).

PostgresPro proposait un outil nommé pg_pathman[7], à présent déprécié en faveur du partitionnement déclaratif intégré à PostgreSQL.

timescaledb est une extension spécialisée dans les séries temporelles. Basée sur le partitionnement par héritage, elle vaut surtout pour sa technique de compression et ses utilitaires. La version communautaire sur Github[8] ne comprend pas tout ce qu'offre la version commerciale.

citus[9] est une autre extension commerciale. Le principe est de partitionner agressivement les tables sur plusieurs instances, et d'utiliser simultanément les processeurs, disques de toutes ces instances (*sharding*). Citus gère la distribution des requêtes, mais pas la maintenance des instances PostgreSQL supplémentaires. L'éditeur Citusdata a été racheté par Microsoft, qui le propose à présent dans Azure. En 2022, l'entièreté du code est passée sous licence libre[10]. Le gain de performance peut être impressionnant, mais attention : certaines requêtes se prêtent très mal au *sharding*.

[6] https://github.com/pgpartman/pg_partman
[7] https://github.com/postgrespro/pg_pathman
[8] https://github.com/timescale/timescaledb
[9] https://github.com/citusdata/citus
[10] https://www.citusdata.com/blog/2022/06/17/citus-11-goes-fully-open-source/

6.7 CONCLUSION

- Le partitionnement déclaratif est mûr
- Préférer une version récente de PostgreSQL

Le partitionnement par héritage n'a plus d'utilité pour la plupart des applications.

Le partitionnement déclaratif apparu en version 10 est mûr dans les dernières versions. Il introduit une complexité supplémentaire, mais peut rendre de grands services quand la volumétrie augmente.

6.8 QUIZ

https://dali.bo/v1_quiz

6.9 TRAVAUX PRATIQUES

6.9.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Nous travaillons sur la base **cave**. La base **cave** peut être téléchargée depuis https://dali.bo/tp_cave (dump de 2,6 Mo, pour 71 Mo sur le disque au final) et importée ainsi :

```
$ psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
$ psql -c "CREATE DATABASE cave OWNER caviste"
$ pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

Nous allons partitionner la table `stock` sur l'année.

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions) :

```
-- Création de lignes en 2001-2005
INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
-- purge des lignes prédédentes
DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

6.9.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

6.9.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

NB : Pour le TP, la base sera d'échelle 10 (environ 168 Mo). Des échelles 100 ou 1000 seraient plus réalistes.

Dans une fenêtre en arrière-plan, laisser tourner un processus pgbench avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

6.9.2.2 Partitionnement par *hash*

Pour partitionner la table pgbench_accounts par *hash* sur la colonne aid sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis pgbench_accounts ;
- la substitution de la table partitionnée à la table originale.

Tester et exécuter.

Supprimer l'ancienne table pgbench_accounts_old.

6.9.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

La table pgbench_history se remplit avec le temps. Elle doit être partitionnée par date (champ mtime). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.

- Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.

NB : Pour éviter de coder des dates en dur, il est possible, avec psql, d'utiliser une variable :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;
```

Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

6.9.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

6.9.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre `pgbench_accounts` et `pgbench_history`. Voir les contraintes créées.

Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

6.9.2.6 Index global

On veut créer un index sur `pgbench_history (aid)`.

Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

6.10 TRAVAUX PRATIQUES (SOLUTIONS)

6.10.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions).

```
INSERT INTO stock
SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock
GROUP BY vin_id, contenant_id, 2001 + annee % 5 ;

DELETE FROM stock WHERE annee<2001 ;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

```
ALTER TABLE stock RENAME TO stock_old;
CREATE TABLE stock(LIKE stock_old) PARTITION BY LIST (annee);
```

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

```
CREATE TABLE stock_2001 PARTITION of stock FOR VALUES IN (2001) ;
CREATE TABLE stock_2002 PARTITION of stock FOR VALUES IN (2002) ;
CREATE TABLE stock_2003 PARTITION of stock FOR VALUES IN (2003) ;
CREATE TABLE stock_2004 PARTITION of stock FOR VALUES IN (2004) ;
CREATE TABLE stock_2005 PARTITION of stock FOR VALUES IN (2005) ;
```

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

```
INSERT INTO stock SELECT * FROM stock_old;
```

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

```
ANALYZE;
```

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

```
SELECT count(*) FROM stock_2001;
SELECT count(*) FROM ONLY stock;
```

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE annee=2002;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Append  (cost=0.00..417.36 rows=18192 width=16) (...)
   ->  Seq Scan on stock_2002  (cost=0.00..326.40 rows=18192 width=16) (...)
         Filter: (annee = 2002)
 Planning Time: 0.912 ms
 Execution Time: 21.518 ms
```

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

```
CREATE UNIQUE INDEX ON stock (vin_id,contenant_id,annee);
```

Les autres index ne servent à rien sur les partitions : `idx_stock_annee` est évidemment inutile, mais `idx_stock_vin_annee` aussi, puisqu'il est inclus dans l'index unique que nous venons de créer.

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE vin_id=1725 AND annee=2003 ;
```

```
 Append  (cost=0.29..4.36 rows=3 width=16) (...)
   ->  Index Scan using stock_2003_vin_id_contenant_id_annee_idx on stock_2003 (...)
         Index Cond: ((vin_id = 1725) AND (annee = 2003))
 Planning Time: 1.634 ms
 Execution Time: 0.166 ms
```

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

```
UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

```
ERROR:  duplicate key value violates unique constraint
 ↪  "stock_2004_vin_id_contenant_id_annee_idx"
DETAIL:  Key (vin_id, contenant_id, annee)=(1725, 1, 2004) already exists.
```

C'est une violation de contrainte unique, qui est une erreur normale : nous avons déjà un enregistrement de stock pour ce vin pour l'année 2004.

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

```
DELETE FROM stock WHERE annee=2004 and vin_id=1725;

UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

```
DROP TABLE stock_2001 ;
```

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;

ERROR:  no partition of relation "stock" found for row
DETAIL:  Partition key of the failing row contains (annee) = (2006).
```

Il n'existe pas de partition définie pour l'année 2006, cela échoue donc.

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

```
CREATE TABLE stock_default PARTITION OF stock DEFAULT ;

INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;
```

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

```
CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

ERROR:  updated partition constraint for default partition "stock_default"
        would be violated by some row
```

Cela échoue car des enregistrements présents dans la partition par défaut répondent à cette nouvelle contrainte de partitionnement.

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

```
BEGIN ;

ALTER TABLE stock DETACH PARTITION stock_default;

CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

INSERT INTO stock SELECT * FROM stock_default WHERE annee = 2006 ;

DELETE FROM stock_default WHERE annee = 2006 ;

ALTER TABLE stock ATTACH PARTITION stock_default DEFAULT ;

COMMIT ;
```

6.10.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

6.10.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 10 pgbench
```

Dans une fenêtre en arrière-plan, laisser tourner un processus `pgbench` avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

```
$ /usr/pgsql-14/bin/pgbench -n -T3600 -c20 -j2 --debug pgbench
```

L'activité est à ajuster en fonction de la puissance de la machine. Laisser l'affichage défiler dans une fenêtre pour bien voir les blocages.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

Un verrou en attente peut bloquer les opérations d'autres transactions venant après. On peut annuler l'opération à partir d'un certain seuil pour éviter ce phénomène :

```
pgbench=# SET lock_timeout TO '10s' ;
```

Cela ne concerne cependant pas les opérations une fois que les verrous sont acquis. On peut garantir qu'un ordre donné ne durera pas plus d'une certaine durée :

```
SET statement_timeout TO '10s' ;
```

En fonction de la rapidité de la machine et des données à déplacer, cette interruption peut être tolérable ou non.

6.10.2.2 Partitionnement par *hash*

Pour partitionner la table `pgbench_accounts` par *hash* sur la colonne `aid` sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis `pgbench_accounts` ;
- la substitution de la table partitionnée à la table originale.

> Tester et exécuter.

Le champ `aid` n'a pas de signification, un partitionnement par *hash* est adéquat.

Le script peut être le suivant :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

BEGIN ;

-- Nouvelle table partitionnée

CREATE TABLE pgbench_accounts_part (LIKE pgbench_accounts INCLUDING ALL)
PARTITION BY HASH (aid) ;

CREATE TABLE pgbench_accounts_1 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 0 ) ;

CREATE TABLE pgbench_accounts_2 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 1 ) ;

CREATE TABLE pgbench_accounts_3 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 2 ) ;

-- Transfert des données

-- Bloquer les accès à la table le temps du transfert
-- (sinon risque de perte de données !)
LOCK TABLE pgbench_accounts ;

-- Copie des données
INSERT INTO pgbench_accounts_part
SELECT * FROM pgbench_accounts ;

-- Substitution par renommage
ALTER TABLE pgbench_accounts RENAME TO pgbench_accounts_old ;
ALTER TABLE pgbench_accounts_part RENAME TO pgbench_accounts ;

-- Contrôle

\d+

-- On ne validera qu'après contrôle
-- (pendant ce temps les sessions concurrentes restent bloquées !)

COMMIT ;
```

À la moindre erreur, la transaction tombe en erreur. Il faudra demander manuellement ROLLBACK.

Si la durée fixée par `statement_timeout` est dépassée, on aura cette erreur :

```
ERROR:  canceling statement due to statement timeout
Time: 10115.506 ms (00:10.116)
```

Surtout, le traitement pgbench reprend en arrière-plan. On peut alors relancer le script corrigé plus tard.

Si tout se passe bien, un `\d+` renvoie ceci :

```
                          Liste des relations
 Schéma |         Nom          |        Type        | Propriétaire | Taille  | …
--------+----------------------+--------------------+--------------+---------+--
 public | pgbench_accounts     | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_2   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_3   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_old | table              | postgres     | 130 MB  |
 public | pgbench_branches     | table              | postgres     | 136 kB  |
 public | pgbench_history      | table              | postgres     | 5168 kB |
 public | pgbench_tellers      | table              | postgres     | 216 kB  |
```

On peut vérifier rapidement que les valeurs de `aid` sont bien réparties entre les 3 partitions :

```
SELECT aid FROM pgbench_accounts_1 LIMIT 3 ;

 aid
-----
   2
   6
   8

SELECT aid FROM pgbench_accounts_2 LIMIT 3 ;

 aid
-----
   3
   7
  10

SELECT aid FROM pgbench_accounts_3 LIMIT 3 ;

 aid
-----
   1
   9
  11
```

Après la validation du script, on voit apparaître les lignes dans les nouvelles partitions :

```
SELECT relname, n_live_tup
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_accounts%' ;

       relname        | n_live_tup
----------------------+------------
 pgbench_accounts_old |    1000002
 pgbench_accounts_1   |     333263
 pgbench_accounts_2   |     333497
 pgbench_accounts_3   |     333240
```

> Supprimer l'ancienne table `pgbench_accounts_old`.

```
DROP TABLE pgbench_accounts_old ;
```

6.10.2.3 Partitionnement par valeur

pgbench doit continuer ses opérations en tâche de fond.

> La table `pgbench_history` se remplit avec le temps. Elle doit être partitionnée par date (champ `mtime`). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.
>
> - Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.
>
> NB : Pour éviter de coder des dates en dur, il est possible, avec `psql`, d'utiliser une variable :
>
> ```
> SELECT ( now()+ interval '60s') AS date_frontiere \gset
> SELECT :'date_frontiere'::timestamptz ;
> ```

La « date frontière » doit être dans le futur (proche). En effet, pgbench va modifier les tables en permanence, on ne sait pas exactement à quel moment la transition aura lieu (et de toute façon on ne maîtrise pas les valeurs de `mtime`) : il continuera donc à écrire dans l'ancienne table, devenue partition, pendant encore quelques secondes.

Cette date est arbitrairement à 1 minute dans le futur, pour dérouler le script manuellement :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
```

Et on peut réutiliser cette variable ainsi ;

```
SELECT :'date_frontiere'::timestamptz ;
```

Le script peut être celui-ci :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;

BEGIN ;

-- Nouvelle table partitionnée
CREATE TABLE pgbench_history_part (LIKE pgbench_history INCLUDING ALL)
PARTITION BY RANGE (mtime) ;

-- Des partitions pour les prochaines minutes
```

```
CREATE TABLE pgbench_history_1
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz  )
             TO (:'date_frontiere'::timestamptz + interval '1min' ) ;

CREATE TABLE pgbench_history_2
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz + interval '1min' )
             TO (:'date_frontiere'::timestamptz + interval '2min' ) ;

-- Au cas où le service perdure au-delà des partitions prévues,
-- on débordera dans cette table
CREATE TABLE pgbench_history_default
PARTITION OF pgbench_history_part DEFAULT ;

-- Jusqu'ici pgbench continue de tourner en arrière plan

-- La table devient une simple partition
-- Ce renommage pose un verrou, les sessions pgbench sont bloquées
ALTER TABLE pgbench_history RENAME TO pgbench_history_orig ;

ALTER TABLE pgbench_history_part
ATTACH PARTITION pgbench_history_orig
FOR VALUES FROM (MINVALUE) TO (:'date_frontiere'::timestamptz) ;

-- Contrôle
\dP

-- Substitution de la table partitionnée à celle d'origine.
ALTER TABLE pgbench_history_part RENAME TO pgbench_history ;

-- Contrôle
\d+ pgbench_history


COMMIT ;
```

> Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

Pour surveiller le contenu des tables jusqu'à la transition :

```
SELECT relname, n_live_tup, now()
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_history%' ;

\watch 3
```

Un \d+ doit renvoyer ceci :

```
                                  Liste des relations
 Schéma |           Nom           |        Type        | Propriétaire | Taille  | …
--------+-------------------------+--------------------+--------------+---------+--
 public | pgbench_accounts        | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1      | table              | postgres     | 44 MB   |
 public | pgbench_accounts_2      | table              | postgres     | 44 MB   |
```

```
 public | pgbench_accounts_3      | table              | postgres     | 44 MB   |
 public | pgbench_branches        | table              | postgres     | 136 kB  |
 public | pgbench_history         | table partitionnée | postgres     | 0 bytes |
 public | pgbench_history_1       | table              | postgres     | 672 kB  |
 public | pgbench_history_2       | table              | postgres     | 0 bytes |
 public | pgbench_history_default | table              | postgres     | 0 bytes |
 public | pgbench_history_orig    | table              | postgres     | 8736 kB |
 public | pgbench_tellers         | table              | postgres     | 216 kB  |
```

6.10.2.4 Purge

- Continuer de laisser tourner pgbench en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

```
ALTER TABLE pgbench_history
DETACH PARTITION pgbench_history_orig ;

-- On pourrait faire le DROP directement

DROP TABLE pgbench_history_orig ;
```

6.10.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre pgbench_accounts et pgbench_history. Voir les contraintes créées.

NB : les clés étrangères entre tables partitionnées ne sont pas disponibles avant PostgreSQL 12.

```
SET lock_timeout TO '3s' ;
SET statement_timeout TO '10s' ;

CREATE INDEX ON pgbench_history (aid) ;

ALTER TABLE pgbench_history
ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid) REFERENCES
 ↪  pgbench_accounts ;
```

On voit que chaque partition porte un index comme la table mère. La contrainte est portée par chaque partition.

```
pgbench=# \d+ pgbench_history
  Table partitionnée « public.pgbench_history »
 …
Clé de partition : RANGE (mtime)
Index :
    "pgbench_history_aid_idx" btree (aid)
Contraintes de clés étrangères :
    "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Partitions: pgbench_history_1 FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                              TO ('2020-02-14 17:42:08.298445'),
            pgbench_history_2 FOR VALUES FROM ('2020-02-14 17:42:08.298445')
                              TO ('2020-02-14 17:43:08.298445'),
            pgbench_history_default DEFAULT
```

```
pgbench=# \d+ pgbench_history_1
   Table « public.pgbench_history_1 »
…
Partition de : pgbench_history FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                                          TO ('2020-02-14 17:42:08.298445')
Contrainte de partition : ((mtime IS NOT NULL)
    AND(mtime >= '2020-02-14 17:41:08.298445'::timestamp without time zone)
    AND (mtime < '2020-02-14 17:42:08.298445'::timestamp without time zone))
Index :
    "pgbench_history_1_aid_idx" btree (aid)
Contraintes de clés étrangères :
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                            FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Méthode d'accès : heap
```

> Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

Le `statement_timeout` peut être un problème :

```
SET
pgbench=# ALTER TABLE pgbench_history
          ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid)
          REFERENCES pgbench_accounts ;
ERROR:  canceling statement due to statement timeout
```

On peut créer les contraintes séparément sur les tables. Cela permet de ne poser un verrou sur la partition active (sans doute `pgbench_history_default`) que pendant le strict minimum de temps (les autres partitions de `pgbench_history` ne sont pas utilisées).

```
SET statement_timeout to '1s' ;
ALTER TABLE pgbench_history_1 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_2 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_default ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

La contrainte au niveau global sera alors posée presque instantanément :

```
ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

6.10.2.6 Index global

> On veut créer un index sur `pgbench_history (aid)`.
>
> Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

Construire un index de manière concurrente (clause `CONCURRENTLY`) permet de ne pas bloquer la table en écriture pendant la création de l'index, qui peut être très longue. Mais il n'est pas possible de le faire sur la table partitionnée :

```
CREATE INDEX CONCURRENTLY ON pgbench_history (aid) ;

ERROR:  cannot create index on partitioned table "pgbench_history" concurrently
```

Mais on peut créer l'index sur chaque partition séparément :

```
CREATE INDEX CONCURRENTLY ON pgbench_history_1 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_2 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_default (aid) ;
```

S'il y a beaucoup de partitions, on peut générer dynamiquement ces ordres :

```
SELECT 'CREATE INDEX CONCURRENTLY ON ' ||
       c.oid::regclass::text || ' (aid) ; '
FROM pg_class c
WHERE relname like 'pgbench_history%' AND relispartition \gexec
```

Comme lors de toute création concurrente, il faut vérifier que les index sont bien valides : la requête suivante ne doit rien retourner.

```
SELECT indexrelid::regclass FROM pg_index WHERE NOT  indisvalid ;
```

Enfin on crée l'index au niveau de la table partitionnée : il réutilise les index existants et sera donc créé presque instantanément :

```
CREATE INDEX ON pgbench_history(aid) ;

pgbench=# \d+ pgbench_history
..
Partition key: RANGE (mtime)
Indexes:
    "pgbench_history_aid_idx" btree (aid)
…
```

7/ Connexions distantes

7.1 ACCÈS À DISTANCE À D'AUTRES SOURCES DE DONNÉES

- Modules historiques : dblink
- SQL/MED & *Foreign Data Wrappers*
- Sharding par fonctions : PL/Proxy
- Le sharding est *Work In Progress*

Nativement, lorsqu'un utilisateur est connecté à une base de données PostgreSQL, sa vision du monde est contenue hermétiquement dans cette base. Il n'a pas accès aux objets des autres bases de la même instance ou d'une autre instance.

Cependant, il existe principalement 3 méthodes pour accéder à des données externes à la base sous PostgreSQL.

La norme SQL/MED est la méthode recommandée pour accéder à des objets distants. Elle permet l'accès à de nombreuses sources de données différentes grâce l'utilisation de connecteurs appelés *Foreign Data Wrappers*.

Historiquement, les utilisateurs de PostgreSQL passaient par l'extension `dblink`, qui permet l'accès à des données externes. Cependant, cet accès ne concerne que des serveurs PostgreSQL. De plus, son utilisation prête facilement à accès moins performant et moins sécurisés que la norme SQL/MED.

PL/Proxy est un cas d'utilisation très différent : cette extension, au départ développée par Skype, permet de distribuer des appels de fonctions PL sur plusieurs nœuds.

Le sharding n'est pas intégré de manière simple à PostgreSQL dans sa version communautaire. Il est déjà possible d'en faire une version primitive avec des partitions basées sur des tables distantes (donc avec SQL/MED), mais nous n'en sommes qu'au début. Des éditeurs proposent des extensions, propriétaires ou expérimentales, ou des *forks* de PostgreSQL dédiés. Comme souvent, il faut se poser la question du besoin réel par rapport à une instance PostgreSQL bien optimisée avant d'utiliser des outils qui vont ajouter une couche supplémentaire de complexité dans votre infrastructure.

7.2 SQL/MED

- *Management of External Data*
- Extension de la norme SQL ISO
- Données externes présentées comme des tables
- Grand nombre de fonctionnalités disponibles
  - mais tous les connecteurs n'implémentent pas tout
- Données accessibles par l'intermédiaire de tables
  - ces tables ne contiennent pas les données localement
  - l'accès à ces tables provoque une récupération des données distantes

SQL/MED est un des tomes de la norme SQL, traitant de l'accès aux données externes (Management of External Data).

Elle fournit donc un certain nombre d'éléments conceptuels, et de syntaxe, permettant la déclaration d'accès à des données externes. Ces données externes sont bien sûr présentées comme des tables.

PostgreSQL suit cette norme et est ainsi capable de requêter des tables distantes à travers des pilotes (appelés *Foreign Data Wrapper*). Les seuls connecteurs livrés par défaut sont `file_fdw` (pour lire des fichiers plats de type CSV accessibles du serveur PostgreSQL) et `postgres_fdw` (qui permet de se connecter à un autre serveur PostgreSQL.

7.2.1 Objets proposés par SQL/MED

- Foreign Data Wrapper
  - connecteur permettant la connexion à un serveur externe et l'exécution de requête
- Foreign Server
  - serveur distant
- User Mapping
  - correspondance d'utilisateur local vers distant
- Foreign Table
  - table distante (ou table externe)

La norme SQL/MED définit quatre types d'objets.

Le *Foreign Data Wrapper* est le connecteur permettant la connexion à un serveur distant, l'exécution de requêtes sur ce serveur, et la récupération des résultats par l'intermédiaire d'une table distante.

Le *Foreign Server* est la définition d'un serveur distant. Il est lié à un *Foreign Data Wrapper* lors de sa création, des options sont disponibles pour indiquer le fichier ou l'adresse IP et le port, ainsi que d'autres informations d'importance pour le connecteur.

Un *User Mapping* permet de définir qui localement a le droit de se connecter sur un serveur distant en tant que tel utilisateur sur le serveur distant. La définition d'un *User Mapping* est optionnel.

Une *Foreign Table* contient la définition de la table distante : nom des colonnes, et type. Elle est liée à un *Foreign Server*.

7.2.2 Foreign Data Wrapper

- Pilote d'accès aux données
- Couverture variable des fonctionnalités
- Qualité variable
- Exemples de connecteurs
  - PostgreSQL, SQLite, Oracle, MySQL (lecture/écriture)
  - fichier CSV, fichier fixe (en lecture)
  - ODBC, JDBC
  - CouchDB, Redis (NoSQL)
- Disponible généralement sous la forme d'une extension
  - ajouter l'extension ajoute le Foreign Data Wrapper à une base

Les trois *Foreign Data Wrappers* les plus aboutis sont sans conteste ceux pour PostgreSQL (disponible en module contrib), Oracle et SQLite. Ces trois pilotes supportent un grand nombre de fonctionnalités (si ce n'est pas toutes) de l'implémentation SQL/MED par PostgreSQL.

De nombreux pilotes spécialisés existent, entre autres pour accéder à des bases NoSQL comme MongDB, CouchDB ou Redis, ou à des fichiers.

Il existe aussi des drivers génériques :

- ODBC : utilisation de driver ODBC
- JDBC : utilisation de driver JDBC

La liste complète des *Foreign Data Wrappers* disponibles pour PostgreSQL peut être consultée sur le wiki de postgresql.org[1]. Encore une fois, leur couverture des fonctionnalités disponibles est très variable ainsi que leur qualité. Il convient de rester prudent et de bien tester ces extensions.

Par exemple, pour ajouter le *Foreign Data Wrapper* pour PostgreSQL, on procédera ainsi :

```
CREATE EXTENSION postgres_fdw;
```

La création cette extension dans une base provoquera l'ajout du *Foreign Data Wrapper* :

```
b1=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION

b1=# \dx+ postgres_fdw
     Objects in extension "postgres_fdw"
             Object descriptiong
------------------------------------------
```

[1] https://wiki.postgresql.org/wiki/Foreign_data_wrappers

```
 foreign-data wrapper postgres_fdw
 function postgres_fdw_disconnect(text)
 function postgres_fdw_disconnect_all()
 function postgres_fdw_get_connections()
 function postgres_fdw_handler()
 function postgres_fdw_validator(text[],oid)
(6 rows)

b1=# \dew
                       List of foreign-data wrappers
     Name     |  Owner   |       Handler        |       Validatorg
--------------+----------+----------------------+------------------------
 postgres_fdw | postgres | postgres_fdw_handler | postgres_fdw_validator
(1 row)
```

7.2.3 Fonctionnalités disponibles pour un FDW (1/2)

- Support des lecture de tables (SELECT)
- Support des écriture de tables (y compris TRUNCATE)
  - directement pour INSERT
  - récupération de la ligne en local pour un UPDATE/DELETE
- Envoi sur le serveur distant
  - des prédicats
  - des jointures si les deux tables jointes font partie du même serveur distant
  - des agrégations

L'implémentation SQL/MED permet l'ajout de ces fonctionnalités dans un *Foreign Data Wrapper*. Cependant, une majorité de ces fonctionnalités est optionnelle. Seule la lecture des données est obligatoire.

Les chapitres suivant montrent des exemples de ces fonctionnalités sur deux *Foreign Data Wrappers*.

7.2.4 Fonctionnalités disponibles pour un FDW (2/2)

- Mais aussi
  - support du EXPLAIN
  - support du ANALYZE
  - support de la parallélisation
  - support des exécutions asynchrones (v14)
  - possibilité d'importer un schéma complet

7.2.5 Foreign Server

- Encapsule les informations de connexion
- Le Foreign Data Wrapper utilise ces informations pour la connexion
- Chaque Foreign Data Wrapper propose des options spécifiques
  - nom du fichier pour un FDW listant des fichiers
  - adresse IP, port, nom de base pour un serveur SQL
  - autres

Pour accéder aux données d'un autre serveur, il faut pouvoir s'y connecter. Le Foreign Server regroupe les informations permettant cette connexion : par exemple adresse IP et port.

Voici un exemple d'ajout de serveur distant :

```
CREATE SERVER serveur2
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '192.168.122.1',
           port '5432',
           dbname 'b1') ;
```

7.2.6 User Mapping

- Correspondance utilisateur local / utilisateur distant
- Mot de passe stocké chiffré
- Optionnel
  - aucun intérêt pour les FDW fichiers
  - essentiel pour les FDW de bases de données

Définir un *User Mapping* permet d'indiquer au *Foreign Data Wrapper* quel utilisateur utilisé pour la connexion au serveur distant.

Par exemple, avec cette définition :

```
CREATE USER MAPPING FOR bob SERVER serveur2 OPTIONS (user 'alice', password
↪  'secret');
```

Si l'utilisateur bob local accède à une table distante dépendant du serveur distant `serveur2`, la connexion au serveur distant passera par l'utilisateur `alice` sur le serveur distant.

7.2.7 Foreign Table

- Définit une table distante
- Doit comporter les colonnes du bon type
  - pas forcément toutes
  - pas forcément dans le même ordre
- Peut être une partition d'une table partitionnée
- Possibilité d'importer un schéma complet
  - simplifie grandement la création des tables distantes

Voici un premier exemple pour une table simple :

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    titre       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
```

```
    type        varchar(10),
    duree       interval hour to minute
)
SERVER serveur2 ;
```

Lors de l'accès (avec un SELECT par exemple) à la table `films`, PostgreSQL va chercher la définition du serveur `serveur2`, ce qui lui permettra de connaître le *Foreign Data Wrapper* responsable de la récupération des données et donnera la main à ce connecteur.

Et voici un second exemple, cette fois pour une partition :

```
CREATE FOREIGN TABLE stock202112
    PARTITION OF stock FOR VALUES FROM ('2021-12-01') TO ('2022-01-01')
    SERVER serveur2;
```

Dans ce cas, l'accès à la table partitionnée locale `stock` accédera à des données locales (les autres partitions) mais aussi à des données distantes avec au moins la partition `stock202112`.

Cette étape de création des tables distantes est fastidieuse et peut amener des problèmes si on se trompe sur le nom des colonnes ou sur leur type. C'est d'autant plus vrai que le nombre de tables à créer est important. Dans ce cas, elle peut être avantageusement remplacée par un appel à l'ordre `IMPORT FOREIGN SCHEMA`. Disponible à partir de la version 9.5, il permet l'import d'un schéma complet.

7.2.8 Exemple : file_fdw

Foreign Data Wrapper de lecture de fichiers CSV.

```
CREATE EXTENSION file_fdw;

CREATE SERVER fichier FOREIGN DATA WRAPPER file_fdw ;

CREATE FOREIGN TABLE donnees_statistiques (f1 numeric, f2 numeric)
   SERVER fichier
   OPTIONS (filename  '/tmp/fichier_donnees_statistiques.csv',
            format    'csv',
            delimiter ';') ;
```

Quel que soit le connecteur, la création d'un accès se fait en 3 étapes minimum :

- Installation du connecteur : aucun *Foreign Data Wrapper* n'est présent par défaut. Il se peut que vous ayez d'abord à l'installer sur le serveur au niveau du système d'exploitation.
- Création du serveur : permet de spécifier un certain nombre d'informations génériques à un serveur distant, qu'on n'aura pas à repréciser pour chaque objet de ce serveur.
- Création de la table distante : l'objet qu'on souhaite rendre visible.

Éventuellement, on peut vouloir créer un *User Mapping*, mais ce n'est pas nécessaire pour le FDW `file_fdw`.

En reprenant l'exemple ci-dessus et avec un fichier `/tmp/fichier_donnees_statistiques.csv` contenant les lignes suivantes :

```
1;1.2
2;2.4
3;0
4;5.6
```

Voici ce que donnerait quelques opérations sur cette table distante :

```
SELECT * FROM donnees_statistiques;
```

```
 f1 | f2g
----+-----
  1 | 1.2
  2 | 2.4
  3 |   0
  4 | 5.6
(4 rows)
```

```
SELECT * FROM donnees_statistiques WHERE f1=2;
```

```
 f1 | f2g
----+-----
  2 | 2.4
(1 row)
```

```
EXPLAIN SELECT * FROM donnees_statistiques WHERE f1=2;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Foreign Scan on donnees_statistiques  (cost=0.00..1.10 rows=1 width=64)
   Filter: (f1 = '2'::numeric)
   Foreign File: /tmp/fichier_donnees_statistiques.csv
   Foreign File Size: 25 b
(4 rows)
```

```
postgres=# insert into donnees_statistiques values (5,100.23);
ERROR:  cannot insert into foreign table "donnees_statistiques"
```

7.2.9 Exemple : postgres_fdw

- Pilote le plus abouti, et pour cause
  - il permet de tester les nouvelles fonctionnalités de SQL/MED
  - il sert d'exemple pour les autres FDW
- Propose en plus :
  - une gestion des transactions explicites
  - un pooler de connexions

Nous créons une table sur un serveur distant. Par simplicité, nous utiliserons le même serveur mais une base différente. Créons cette base et cette table :

```
dalibo=# CREATE DATABASE distante;
CREATE DATABASE

dalibo=# \c distante
You are now connected to database "distante" as user "dalibo".

distante=# CREATE TABLE personnes (id integer, nom text);
CREATE TABLE

distante=# INSERT INTO personnes (id, nom) VALUES (1, 'alice'),
                  (2, 'bertrand'), (3, 'charlotte'), (4, 'david');
INSERT 0 4

distante=# ANALYZE personnes;
ANALYZE
```

Maintenant nous pouvons revenir à notre base d'origine et mettre en place la relation avec le « serveur distant » :

```
distante=# \c dalibo
You are now connected to database "dalibo" as user "dalibo".

dalibo=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION

dalibo=# CREATE SERVER serveur_distant FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (HOST 'localhost',PORT '5432', DBNAME 'distante');
CREATE SERVER

dalibo=# CREATE USER MAPPING FOR dalibo SERVER serveur_distant
OPTIONS (user 'dalibo', password 'mon_mdp');
CREATE USER MAPPING

dalibo=# CREATE FOREIGN TABLE personnes (id integer, nom text)
SERVER serveur_distant;
CREATE FOREIGN TABLE
```

Et c'est tout ! Nous pouvons désormais utiliser la table distante `personnes` comme si elle était une table locale de notre base.

```
SELECT * FROM personnes;

 id |    nom
----+-----------
  1 | alice
  2 | bertrand
  3 | charlotte
  4 | david

EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM personnes;

                              QUERY PLAN
-----------------------------------------------------------------------
```

```
 Foreign Scan on public.personnes  (cost=100.00..150.95 rows=1365 width=36)
                                   (actual time=0.655..0.657 rows=4 loops=1)
   Output: id, nom
   Remote SQL: SELECT id, nom FROM public.personnes
 Total runtime: 1.197 ms
```

En plus, si nous filtrons notre requête, le filtre est exécuté sur le serveur distant, réduisant considérablement le trafic réseau et le traitement associé.

```
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM personnes WHERE id = 3;

                                QUERY PLAN
-------------------------------------------------------------------------
 Foreign Scan on public.personnes  (cost=100.00..127.20 rows=7 width=36)
                                   (actual time=1.778..1.779 rows=1 loops=1)
   Output: id, nom
   Remote SQL: SELECT id, nom FROM public.personnes WHERE ((id = 3))
 Total runtime: 2.240 ms
```

Noter qu'EXPLAIN exige l'option VERBOSE pour afficher le code envoyé à l'instance distante.

Il est possible d'écrire vers ces tables aussi, à condition que le connecteur FDW le permette.

En utilisant l'exemple de la section précédente, on note qu'il y a un aller-retour entre la sélection des lignes à modifier (ou supprimer) et la modification (suppression) de ces lignes :

```
EXPLAIN (ANALYZE, VERBOSE)
UPDATE personnes
SET nom = 'agathe' WHERE id = 1 ;

                               QUERY PLAN
-------------------------------------------------------------------------------
 Update on public.personnes  (cost=100.00..140.35 rows=12 width=10)
                             (actual time=2.086..2.086 rows=0 loops=1)
   Remote SQL: UPDATE public.personnes SET nom = $2 WHERE ctid = $1
   ->  Foreign Scan on public.personnes  (cost=100.00..140.35 rows=12 width=10)
                                         (actual time=1.040..1.042 rows=1 loops=1)
         Output: id, 'agathe'::text, ctid
         Remote SQL: SELECT id, ctid FROM public.personnes WHERE ((id = 1))
                     FOR UPDATE
 Total runtime: 2.660 ms

SELECT * FROM personnes;

 id |    nom
----+-----------
  2 | bertrand
  3 | charlotte
  4 | david
  1 | agathe
```

On peut aussi constater que l'écriture distante respecte les transactions :

```
dalibo=# BEGIN;
BEGIN

dalibo=# DELETE FROM personnes WHERE id=2;
```

```
DELETE 1

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  3 | charlotte
  4 | david
  1 | agathe
(3 rows)

dalibo=# ROLLBACK;
ROLLBACK

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  2 | bertrand
  3 | charlotte
  4 | david
  1 | agathe
(4 rows)
```

Attention à ne pas perdre de vue qu'une table distante n'est pas une table locale. L'accès à ses données est plus lent, surtout quand on souhaite récupérer de manière répétitive peu d'enregistrements : on a systématiquement une latence réseau, éventuellement une analyse de la requête envoyée au serveur distant, etc.

Les jointures ne sont pas « poussées » au serveur distant avant PostgreSQL 9.6 et pour des bases PostgreSQL. Un accès par *Nested Loop* (boucle imbriquée entre les deux tables) est habituellement inenvisageable entre deux tables distantes : la boucle interne (celle qui en local serait un accès à une table par index) entraînerait une requête individuelle par itération, ce qui serait horriblement peu performant.

Comme avec tout FDW, il existe des restrictions. Par exemple, avec `postgres_fdw`, un `TRUNCATE` d'une table distante n'est pas possible avant PostgreSQL 14.

Les tables distantes sont donc à réserver à des accès intermittents. Il ne faut pas les utiliser pour développer une application transactionnelle par exemple. Noter qu'entre serveurs PostgreSQL, chaque version améliore les performances (notamment pour « pousser » le maximum d'informations et de critères au serveur distant).

7.2.10 SQL/MED : Performances

- Tous les FDW : vues matérialisées et indexations
- postgres_fdw : `fetch_size`

Pour améliorer les performances lors de l'utilisation de *Foreign Data Wrapper*, une pratique courante est de faire une vue matérialisée de l'objet distant. Les données sont récupérées en bloc et cette vue matérialisée peut être indexée. C'est une sorte de mise en cache. Évidemment cela ne convient pas à toutes les applications.

La documentation de postgres_fdw[2] mentionne plusieurs paramètres, et le plus intéressant pour des requêtes de gros volume est `fetch_size` : la valeur par défaut n'est que de 100, et l'augmenter permet de réduire les aller-retours à travers le réseau.

7.2.11 SQL/MED : héritage

- Une table locale peut hériter d'une table distante et inversement
- Permet le partitionnement sur plusieurs serveurs
- Pour rappel, l'héritage ne permet pas de conserver
  - les contraintes d'unicité et référentielles
  - les index
  - les droits

Cette fonctionnalité utilise le mécanisme d'héritage de PostgreSQL.

Exemple d'une table locale qui hérite d'une table distante

La table parent (ici une table distante) sera la table `fgn_stock_londre` et la table enfant sera la table `local_stock` (locale). Ainsi la lecture de la table `fgn_stock_londre` retournera les enregistrements de la table `fgn_stock_londre` et de la table `local_stock`.

Sur l'instance distante :

Créer une table `stock_londre` sur l'instance distante dans la base nommée « cave » et insérer des valeurs :

[2] https://docs.postgresql.fr/current/postgres-fdw.html

```
CREATE TABLE stock_londre (c1 int);
INSERT INTO stock_londre VALUES (1),(2),(4),(5);
```

Sur l'instance locale :

Créer le serveur et la correspondance des droits :

```
CREATE EXTENSION postgres_fdw ;

CREATE SERVER pgdistant
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '192.168.0.42', port '5432', dbname 'cave');

CREATE USER MAPPING FOR mon_utilisateur
SERVER pgdistant
OPTIONS (user 'utilisateur_distant', password 'mdp_utilisateur_distant');
```

Créer une table distante `fgn_stock_londre` correspondant à la table `stock_londre` de l'autre instance :

```
CREATE FOREIGN TABLE fgn_stock_londre (c1 int) SERVER pgdistant
OPTIONS (schema_name 'public' , table_name 'stock_londre');
```

On peut bien lire les données :

```
SELECT tableoid::regclass,* FROM fgn_stock_londre;
```

```
     tableoid     | c1
------------------+----
 fgn_stock_londre |  1
 fgn_stock_londre |  2
 fgn_stock_londre |  4
 fgn_stock_londre |  5
(4 lignes)
```

Voici le plan d'exécution associé :

```
EXPLAIN ANALYZE SELECT * FROM fgn_stock_londre;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Foreign Scan on fgn_stock_londre  (cost=100.00..197.75 rows=2925 width=4)
                                   (actual time=0.388..0.389 rows=4 loops=1)
```

Créer une table `local_stock` sur l'instance locale qui va hériter de la table mère :

```
CREATE TABLE local_stock () INHERITS (fgn_stock_londre);
```

On insère des valeurs dans la table `local_stock` :

```
INSERT INTO local_stock VALUES  (10),(15);
INSERT 0 2
```

La table `local_stock` ne contient bien que 2 valeurs :

```
SELECT * FROM local_stock ;
```

```
 c1
----
 10
 15
(2 lignes)
```

En revanche, la table `fgn_stock_londre` ne contient plus 4 valeurs mais 6 valeurs :

```
SELECT tableoid::regclass,* FROM fgn_stock_londre;
```

```
     tableoid     | c1
------------------+----
 fgn_stock_londre |  1
 fgn_stock_londre |  2
 fgn_stock_londre |  4
 fgn_stock_londre |  5
 local_stock      | 10
 local_stock      | 15
(6 lignes)
```

Dans le plan d'exécution on remarque bien la lecture des deux tables :

```
EXPLAIN ANALYZE SELECT * FROM fgn_stock_londre;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Append  (cost=100.00..233.25 rows=5475 width=4)
         (actual time=0.438..0.444 rows=6 loops=1)
   ->  Foreign Scan on fgn_stock_londre
           (cost=100.00..197.75 rows=2925 width=4)
           (actual time=0.438..0.438 rows=4 loops=1)
   ->  Seq Scan on local_stock  (cost=0.00..35.50 rows=2550 width=4)
                                (actual time=0.004..0.005 rows=2 loops=1)
 Planning time: 0.066 ms
 Execution time: 0.821 ms
(5 lignes)
```

Note : Les données de la table `stock_londre` sur l'instance distante n'ont pas été modifiées.

Exemple d'une table distante qui hérite d'une table locale

La table parent sera la table `master_stock` et la table fille (ici distante) sera la table `fgn_stock_londre`. Ainsi une lecture de la table `master_stock` retournera les valeurs de la table `master_stock` et de la table `fgn_stock_londre`, sachant qu'une lecture de la table `fgn_stock_londre` retourne les valeurs de la table `fgn_stock_londre` et `local_stock`. Une lecture de la table `master_stock` retournera les valeurs des 3 tables : `master_stock`, `fgn_stock_londre`, `local_stock`.

Créer une table `master_stock`, insérer des valeurs dedans :

```
CREATE TABLE master_stock (LIKE fgn_stock_londre);
INSERT INTO master_stock VALUES (100),(200);

SELECT tableoid::regclass,* FROM master_stock;
```

```
   tableoid   | c1
--------------+-----
```

```
 master_stock | 100
 master_stock | 200
(2 rows)
```

Modifier la table `fgn_stock_londre` pour qu'elle hérite de la table `master_stock` :

```
ALTER TABLE fgn_stock_londre INHERIT master_stock ;
```

La lecture de la table `master_stock` nous montre bien les valeurs des 3 tables :

```
SELECT tableoid::regclass,* FROM master_stock ;
```

```
     tableoid     | c1
------------------+-----
 master_stock     | 100
 master_stock     | 200
 fgn_stock_londre |   1
 fgn_stock_londre |   2
 fgn_stock_londre |   4
 fgn_stock_londre |   5
 local_stock      |  10
 local_stock      |  15
(8 lignes)
```

Le plan d'exécution confirme bien la lecture des 3 tables :

```
EXPLAIN ANALYSE SELECT * FROM master_stock ;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=0.00..236.80 rows=5730 width=4)
         (actual time=0.004..0.440 rows=8 loops=1)
   ->  Seq Scan on master_stock  (cost=0.00..3.55 rows=255 width=4)
                                 (actual time=0.003..0.003 rows=2 loops=1)
   ->  Foreign Scan on fgn_stock_londre
           (cost=100.00..197.75 rows=2925 width=4)
           (actual time=0.430..0.430 rows=4 loops=1)
   ->  Seq Scan on local_stock  (cost=0.00..35.50 rows=2550 width=4)
                                (actual time=0.003..0.004 rows=2 loops=1)
 Planning time: 0.073 ms
 Execution time: 0.865 ms
(6 lignes)
```

Dans cet exemple, on a un héritage « imbriqué » :

- la table `master_stock` est parent de la table distante `fgn_stock_londre`
- la table distante `fgn_stock_londre` est enfant de la table `master_stock` et parent de la table `local_stock`
- ma table `local_stock` est enfant de la table distante `fgn_stock_londre`

```
master_stock
  ├─fgn_stock_londre => stock_londre
      ├─local_stock
```

Créons un index sur `master_stock` et ajoutons des données dans la table `master_stock` :

```
CREATE INDEX fgn_idx ON master_stock(c1);
INSERT INTO master_stock (SELECT generate_series(1,10000));
```

Maintenant effectuons une simple requête de sélection :

```
SELECT tableoid::regclass,* FROM master_stock WHERE c1=10;
```

```
   tableoid   | c1
--------------+----
 master_stock | 10
 local_stock  | 10
(2 lignes)
```

Étudions le plan d'exécution associé :

```
EXPLAIN ANALYZE SELECT tableoid::regclass,* FROM master_stock WHERE c1=10;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.29..192.44 rows=27 width=8)
         (actual time=0.010..0.485 rows=2 loops=1)
   ->  Append  (cost=0.29..192.44 rows=27 width=8)
               (actual time=0.009..0.483 rows=2 loops=1)
         ->  Index Scan using fgn_idx on master_stock
                     (cost=0.29..8.30 rows=1 width=8)
                     (actual time=0.009..0.010 rows=1 loops=1)
               Index Cond: (c1 = 10)
         ->  Foreign Scan on fgn_stock_londre
                     (cost=100.00..142.26 rows=13 width=8)
                     (actual time=0.466..0.466 rows=0 loops=1)
         ->  Seq Scan on local_stock  (cost=0.00..41.88 rows=13 width=8)
                                      (actual time=0.007..0.007 rows=1 loops=1)
               Filter: (c1 = 10)
               Rows Removed by Filter: 1
```

L'index ne se fait que sur `master_stock`.

En ajoutant l'option `ONLY` après la clause `FROM`, on demande au moteur de n'afficher que la table `master_stock` et pas les tables filles :

```
SELECT tableoid::regclass,* FROM ONLY master_stock WHERE c1=10;
```

```
   tableoid   | c1
--------------+----
 master_stock | 10
(1 ligne)
```

Attention, si on supprime les données sur la table parent, la suppression se fait aussi sur les tables filles :

```
BEGIN;
DELETE FROM master_stock;
-- [DELETE 10008]
SELECT * FROM master_stock ;

 c1
----
```

```
(0 ligne)

ROLLBACK;
```

En revanche avec l'option ONLY, on ne supprime que les données de la table parent :

```
BEGIN;
DELETE FROM ONLY master_stock;
-- [DELETE 10002]
ROLLBACK;
```

Enfin, si nous ajoutons une contrainte CHECK sur la table distante, l'exclusion de partition basées sur ces contraintes s'appliquent naturellement :

```
ALTER TABLE fgn_stock_londre ADD CHECK (c1 < 100);
ALTER TABLE local_stock ADD CHECK (c1 < 100);
    --local_stock hérite de fgn_stock_londre !

EXPLAIN (ANALYZE,verbose) SELECT tableoid::regclass,*g
FROM master_stock WHERE c1=200;

                          QUERY PLAN
---------------------------------------------------------------
 Result  (cost=0.29..8.32 rows=2 width=8)
         (actual time=0.009..0.011 rows=2 loops=1)
   Output: (master_stock.tableoid)::regclass, master_stock.c1
   ->  Append  (cost=0.29..8.32 rows=2 width=8)
               (actual time=0.008..0.009 rows=2 loops=1)
         ->  Index Scan using fgn_idx on public.master_stock
                     (cost=0.29..8.32 rows=2 width=8)
                     (actual time=0.008..0.008 rows=2 loops=1)
               Output: master_stock.tableoid, master_stock.c1
               Index Cond: (master_stock.c1 = 200)
 Planning time: 0.157 ms
 Execution time: 0.025 ms
(8 rows)
```

Attention : La contrainte CHECK sur `fgn_stock_londre` est **locale** seulement. Si cette contrainte n'existe pas sur la table distants, le résultat de la requête pourra alors être faux !

Sur le serveur distant :

```
INSERT INTO stock_londre VALUES (200);
```

Sur le serveur local :

```
SELECT tableoid::regclass,* FROM master_stock WHERE c1=200;

   tableoid   | c1
--------------+-----
 master_stock | 200
 master_stock | 200

ALTER TABLE fgn_stock_londre DROP CONSTRAINT fgn_stock_londre_c1_check;

SELECT tableoid::regclass,* FROM master_stock WHERE c1=200;
```

```
     tableoid     | c1
------------------+-----
 master_stock     | 200
 master_stock     | 200
 fgn_stock_londre | 200
```

7.3 DBLINK

- Permet le requêtage inter-bases PostgreSQL
- Simple et bien documenté
- En lecture seule sauf à écrire des triggers sur vue
- Ne transmet pas les prédicats
  - tout l'objet est systématiquement récupéré
- Préférer `postgres_fdw`

Documentation officielle[3].

Le module `dblink` de PostgreSQL a une logique différente de SQL/MED : ce dernier crée des tables virtuelles qui masquent des accès distants, alors qu'avec `dblink`, une requête est fournie à une fonction, qui l'exécute à distance puis renvoie le résultat.

Voici un exemple d'utilisation :

```
SELECT *
FROM dblink('host=serveur port=5432 user=postgres dbname=b1',
            'SELECT proname, prosrc FROM pg_proc')
     AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

L'appel à la fonction `dblink()` va réaliser une connexion à la base `b1` et l'exécution de la requête indiquée dans le deuxième argument. Le résultat de cette requête est renvoyé comme résultat de la fonction. Noter qu'il faut nommer les champs obtenus.

Généralement, on encapsule l'appel à `dblink()` dans une vue, ce qui donnerait par exemple :

```
CREATE VIEW pgproc_b1 AS
  SELECT *
  FROM dblink('host=serveur port=5432 user=postgres dbname=b1',
              'SELECT proname, prosrc FROM pg_proc')
       AS t1(proname name, prosrc text);

SELECT *
FROM pgprocb1
WHERE proname LIKE 'bytea%';
```

Un problème est que, rapidement, on ne se rappelle plus que c'est une table externe et que, même si le résultat contient peu de lignes, tout le contenu de la table distante est récupérés avant que le filtre ne soit exécuté. Donc même s'il y a un index qui aurait pu être utilisé pour ce prédicat, il ne pourra pas être utilisé. Il est rapidement difficile d'obtenir de bonnes performances avec cette extension.

Noter que `dblink` n'est pas aussi riche que son homonyme dans d'autres SGBD concurrents.

[3] https://docs.postgresql.fr/current/contrib-dblink-function.html

De plus, cette extension est un peu ancienne et ne bénéficie pas de nouvelles fonctionnalités sur les dernières versions de PostgreSQL. On préférera utiliser à la place l'implémentation de SQL/MED de PostgreSQL et le *Foreign Data Wrapper* `postgres_fdw` qui évoluent de concert à chaque version majeure et deviennent de plus en plus puissants au fil des versions. Cependant, `dblink` a encore l'intérêt d'émuler des transactions autonomes ou d'appeler des fonctions sur le serveur distant, ce qui est impossible directement avec `postgres_fdw`.

`dblink` fournit quelques fonctions plus évoluées que l'exemple ci-dessus, décrites dans la documentation[4].

[4] https://docs.postgresql.fr/current/dblink.html

7.4 PL/PROXY

- Langage de procédures
  - développée à la base par Skype
- Fonctionnalités
  - connexion à un serveur ou à un ensemble de serveurs
  - exécution de fonctions, pas de requêtes
- Possibilité de distribuer les requêtes
- Utile pour le « partionnement horizontal »
- Uniquement si votre application n'utilise que des appels de fonction
  - dans le cas contraire, il faut revoir l'application

PL/Proxy propose d'exécuter une fonction suivant un mode parmi trois :

- ANY : la fonction est exécutée sur un seul nœud au hasard
- ALL : la fonction est exécutée sur tous les nœuds
- EXACT : la fonction est exécutée sur un nœud précis, défini dans le corps de la fonction

On peut mettre en place un ensemble de fonctions PL/Proxy pour « découper » une table volumineuse et la répartir sur plusieurs instances PostgreSQL.

Le langage PL/Proxy offre alors la possibilité de développer une couche d'abstraction transparente pour l'utilisateur final qui peut alors consulter et manipuler les données comme si elles se trouvaient dans une seule table sur une seule instance PostgreSQL.

On peut néanmoins se demander l'avenir de ce projet. La dernière version date de septembre 2020, et il n'y a eu aucune modification des sources depuis cette version. La société qui a développé ce langage au départ a été rachetée par Microsoft. Le développement du langage dépend donc d'un très petit nombre de contributeurs.

7.5 CONCLUSION

- Privilégier SQL/MED
- `dblink` et PL/Proxy en perte de vitesse
  - à n'utiliser que s'ils résolvent un problème non gérable avec SQL/MED

7.6 TRAVAUX PRATIQUES

7.6.1 Foreign Data Wrapper sur un fichier

But : Lire un fichier extérieur depuis PostgreSQL par un FDW

Avec le *foreign data wrapper* `file_fdw`, créer une table distante qui présente les champs du fichier `/etc/passwd` sous forme de table.

Vérifier son bon fonctionnement avec un simple `SELECT`.

7.6.2 Foreign Data Wrapper sur une autre base

But : Accéder à une autre base par un FDW

Accéder à une table de votre choix d'une autre machine, par exemple `stock` dans la base `cave`, à travers une table distante (`postgres_fdw`) : configuration du `pg_hba.conf`, installation de l'extension dans une base locale, création du serveur, de la table, du mapping pour les droits.

Visualiser l'accès par un `EXPLAIN (ANALYZE VERBOSE) SELECT ...`.

7.7 TRAVAUX PRATIQUES (SOLUTIONS)

7.7.1 Foreign Data Wrapper sur un fichier

> Avec le *foreign data wrapper* `file_fdw`, créer une table distante qui présente les champs du fichier `/etc/passwd` sous forme de table.

> Vérifier son bon fonctionnement avec un simple `SELECT`.

```
CREATE EXTENSION file_fdw;

CREATE SERVER files FOREIGN DATA WRAPPER file_fdw;

CREATE FOREIGN TABLE passwd (
  login text,
  passwd text,
  uid int,
  gid int,
  username text,
  homedir text,
  shell text)
SERVER files
OPTIONS (filename '/etc/passwd', format 'csv', delimiter ':');
```

7.7.2 Foreign Data Wrapper sur une autre base

> Accéder à une table de votre choix d'une autre machine, par exemple `stock` dans la base `cave`, à travers une table distante (`postgres_fdw`) : configuration du `pg_hba.conf`, installation de l'extension dans une base locale, création du serveur, de la table, du mapping pour les droits.

> Visualiser l'accès par un `EXPLAIN (ANALYZE VERBOSE) SELECT ...`.

Tout d'abord, vérifier que la connexion se fait sans mot de passe à la cible depuis le compte **postgres** de l'instance locale vers la base distante où se trouve la table cible.

Si cela ne fonctionne pas, vérifier le `listen_addresses`, le fichier `pg_hba.conf` et le *firewall* de la base distance, et éventuellement le `~postgres/.pgpass` sur le serveur local.

Une fois la connexion en place, dans la base locale voulue, installer le *foreign data wrapper* :

```
CREATE EXTENSION postgres_fdw ;
```

Créer le *foreign server* vers le serveur cible (ajuster les options) :

```
CREATE SERVER serveur_voisin
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '192.168.0.18', port '5432', dbname 'cave');
```

Créer un *user mapping*, c'est-à-dire une correspondance entre l'utilisateur local et l'utilisateur distant :

```
CREATE USER MAPPING FOR mon_utilisateur
SERVER serveur_voisin
OPTIONS (user 'utilisateur_distant', password 'mdp_utilisateur_distant');
```

Puis créer la *foreign table* :

```
CREATE FOREIGN TABLE stock_voisin (
vin_id integer, contenant_id integer, annee integer, nombre integer)
SERVER serveur_voisin
OPTIONS (schema_name 'public', table_name 'stock_old');
```

Vérifier le bon fonctionnement :

```
SELECT * FROM stock_voisin WHERE vin_id=12;
```

Vérifier le plan :

```
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM stock_voisin WHERE vin_id=12 ;
```

Il faut l'option VERBOSE pour voir la requête envoyée au serveur distant. Vous constatez que le prédicat sur `vin_id` a été transmis, ce qui est le principal avantage de cette implémentation sur les DBLinks.

8/ Fonctionnalités avancées pour la performance

8.1 PRÉAMBULE

Comme tous les SGBD, PostgreSQL fournit des fonctionnalités avancées.
Ce module présente des fonctionnalités internes au moteur généralement liées aux performances.

8.1.1 Au menu

- Tables temporaires
- Tables non journalisées
- JIT
- Recherche Full Text

8.2 TABLES TEMPORAIRES

```
CREATE TEMP TABLE travail (…) ;
```

- N'existent que pendant la session
- Non journalisées
- Ne pas en abuser !
- Ignorées par autovacuum : ANALYZE et VACUUM manuels !
- Paramétrage :
  - `temp_buffers` : cache disque pour les objets temporaires, par session, à augmenter ?

Principe :

Sous PostgreSQL, les tables temporaires sont créées dans une session, et disparaissent à la déconnexion. Elles ne sont pas visibles par les autres sessions. Elles ne sont pas journalisées, ce qui est très intéressant pour les performances. Elles s'utilisent comme les autres tables, y compris pour l'indexation, les triggers, etc.

Les tables temporaires semblent donc idéales pour des tables de travail temporaires et « jetables ».

Cependant, il est déconseillé d'abuser des tables temporaires. En effet, leur création/destruction permanente entraîne une fragmentation importante des tables systèmes (en premier lieu `pg_catalog.pg_class`, `pg_catalog.pg_attribute`…), qui peuvent devenir énormes. Ce n'est jamais bon pour les performances, et peut nécessiter un `VACUUM FULL` des tables système !

Le démon autovacuum ne voit pas les tables temporaires ! Les statistiques devront donc être mises à jour manuellement avec `ANALYZE`, et il faudra penser à lancer `VACUUM` explicitement après de grosses modifications.

Aspect technique :

Les tables temporaires sont créées dans un schéma temporaire `pg_temp_…`, ce qui explique qu'elles ne sont pas visibles dans le schéma `public`.

Physiquement, par défaut, elles sont stockées sur le disque avec les autres données de la base, et non dans `base/pgsql_tmp` comme les fichiers temporaires. Il est possible de définir des tablespaces dédiés aux objets temporaires (fichiers temporaires et données des tables temporaires) à l'aide du paramètre `temp_tablespaces`, à condition de donner des droits CREATE dessus aux utilisateurs. Le nom du fichier d'une table temporaire est reconnaissable car il commence par `t`. Les éventuels index de la table suivent les même règles.

Exemple :

```
CREATE TEMP TABLE travail (x int PRIMARY KEY) ;

EXPLAIN (COSTS OFF, ANALYZE, BUFFERS, WAL)
INSERT INTO travail SELECT i FROM generate_series (1,1000000) i ;

                              QUERY PLAN
-----------------------------------------------------------------------------
 Insert on travail (actual time=1025.752..1025.753 rows=0 loops=1)
   Buffers: shared hit=13, local hit=2172174 read=4 dirtied=7170 written=10246
   I/O Timings: read=0.012
   ->  Function Scan on generate_series i (actual time=77.112..135.624 rows=1000000
↪  loops=1)
 Planning Time: 0.028 ms
 Execution Time: 1034.984 ms

SELECT pg_relation_filepath ('travail') ;
 pg_relation_filepath
----------------------
 base/13746/t7_5148873

\d pg_temp_7.travail

                Table « pg_temp_7.travail »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 x       | integer |                 | not null  |
Index :
    "travail_pkey" PRIMARY KEY, btree (x)
```

Cache :

Dans les plans d'exécution avec BUFFERS, l'habituelle mention `shared` est remplacée par `local` pour les tables temporaires. En effet, leur cache disque dédié est au niveau de la session, non des *shared buffers*. Ce cache est défini par le paramètre `temp_buffers` (exprimé par session, et à 8 Mo par défaut). Ce paramètre peut être augmenté, avant la création de la table. Bien sûr, on risque de saturer la RAM en cas d'abus ou s'il y a trop de sessions, comme avec `work_mem`. Ce cache n'empêche pas l'écriture des petites tables temporaires sur le disque.

Pour éviter de recréer perpétuellement la même table temporaire, une table *unlogged* (voir plus bas) sera sans doute plus indiquée. Le contenu de cette dernière sera aussi visible des autres sessions, ce qui est pratique pour suivre la progression d'un traitement, faciliter le travail de l'autovacuum, ou déboguer. Sinon, il est fréquent de pouvoir remplacer une table temporaire par une CTE (clause `WITH`) ou un tableau en mémoire.

L'extension pgtt[1] émule un autre type de table temporaire dite « globale » pour la compatibilité avec d'autres SGBD.

[1] https://github.com/darold/pgtt

8.3 TABLES NON JOURNALISÉES (UNLOGGED)

- La durabilité est parfois accessoire :
  - tables temporaires et de travail
  - caches…
- Tables non journalisées
  - non répliquées, non restaurées
  - **remises à zéro en cas de crash**
- Respecter les contraintes

Une table *unlogged* est une table non journalisée. Comme la journalisation est responsable de la durabilité, une table non journalisée n'a pas cette garantie.

La table est systématiquement remise à zéro au redémarrage après un arrêt brutal. En effet, tout arrêt d'urgence peut entraîner une corruption des fichiers de la table ; et sans journalisation, il ne serait pas possible de la corriger au redémarrage et de garantir l'intégrité.

La non-journalisation de la table implique aussi que ses données ne sont pas répliquées vers des serveurs secondaires, et que les tables ne peuvent figurer dans une publication (réplication logique). En effet, les modes de réplication natifs de PostgreSQL utilisent les journaux de transactions. Pour la même raison, une restauration de sauvegarde PITR ne restaurera pas le contenu de la table. Le bon côté est qu'on allège la charge sur la sauvegarde et la réplication.

Les contraintes doivent être respectées même si la table *unlogged* est vidée : une table normale ne peut donc avoir de clé étrangère pointant vers une table *unlogged*. La contrainte inverse est possible, tout comme une contrainte entre deux tables *unlogged*.

À part ces limitations, les tables *unlogged* se comportent exactement comme les autres. Leur intérêt principal est d'être en moyenne 5 fois plus rapides à la mise à jour. Elles sont donc à réserver à des cas d'utilisation particuliers, comme :

- table de *spooling*/*staging* ;
- table de cache/session applicative ;
- table de travail partagée entre sessions ;
- table de travail systématiquement reconstruite avant utilisation dans le flux applicatif ;
- et de manière générale toute table contenant des données dont on peut accepter la perte sans impact opérationnel ou dont on peut regénérer aisément les données.

Les tables *unlogged* ne doivent pas être confondues avec les tables temporaires (non journalisées et visibles uniquement dans la session qui les a créées). Les tables *unlogged* ne sont pas ignorées par l'autovacuum (les tables temporaires le sont). Abuser des tables temporaires a tendance à générer de la fragmentation dans les tables système, alors que les tables *unlogged* sont en général créées une fois pour toutes.

8.3.1 Tables non journalisées : mise en place

```
CREATE UNLOGGED TABLE ma_table (col1 int …) ;
```

Une table *unlogged* se crée exactement comme une table journalisée classique, excepté qu'on rajoute le mot UNLOGGED dans la création.

8.3.2 Bascule d'une table en/depuis unlogged

```
ALTER TABLE table_normale SET UNLOGGED ;
```

- réécriture

```
ALTER TABLE table_unlogged SET LOGGED ;
```

- passage du contenu dans les WAL !

Il est possible de basculer une table à volonté de normale à *unlogged* et vice-versa.

Quand une table devient *unlogged*, on pourrait imaginer que PostgreSQL n'a rien besoin d'écrire. Malheureusement, pour des raisons techniques, la table doit tout de même être réécrite. Elle est défragmentée au passage, comme lors d'un VACUUM FULL. Ce peut être long pour une grosse table, et il faudra voir si le gain par la suite le justifie.

Les écritures dans les journaux à ce moment sont théoriquement inutiles, mais là encore des optimisations manquent et il se peut que de nombreux journaux soient écrits si les sommes de contrôles ou `wal_log_hints` sont activés. Par contre il n'y aura plus d'écritures dans les journaux lors des modifications de cette table, ce qui reste l'intérêt majeur.

Quand une table *unlogged* devient *logged* (journalisée), la réécriture a aussi lieu, et tout le contenu de la table est journalisé (c'est indispensable pour la sauvegarde PITR et pour la réplication notamment), ce qui génère énormément de journaux et peut prendre du temps.

Par exemple, une table modifiée de manière répétée pendant un batch, peut être définie *unlogged* pour des raisons de performance, puis basculée en *logged* en fin de traitement pour pérenniser son contenu.

8.4 JIT : LA COMPILATION À LA VOLÉE

- Compilation *Just In Time* des requêtes
- Utilise le compilateur LLVM
- Vérifier que l'installation est fonctionnelle
- Activé par défaut
    - sauf en v11 ; et absent auparavant

Une des nouveautés les plus visibles et techniquement pointues de la v11 est la « compilation à la volée » (*Just In Time compilation*, ou JIT) de certaines expressions dans les requêtes SQL. Le JIT n'est activé par défaut qu'à partir de la version 12.

Dans certaines requêtes, l'essentiel du temps est passé à décoder des enregistrements (*tuple deforming*), à analyser des clauses `WHERE`, à effectuer des calculs. En conséquence, l'idée du JIT est de transformer tout ou partie de la requête en un programme natif directement exécuté par le processeur.

Cette compilation est une opération lourde qui ne sera effectuée que pour des requêtes qui en valent le coup, donc qui dépassent un certain coût. Au contraire de la parallélisation, ce coût n'est pas pris en compte par le planificateur. La décision d'utiliser le JIT ou pas se fait une fois le plan décidé, si le coût calculé de la requête dépasse un certain seuil.

Le JIT de PostgreSQL s'appuie actuellement sur la chaîne de compilation LLVM, choisie pour sa flexibilité. L'utilisation nécessite un PostgreSQL compilé avec l'option `--with-llvm` et l'installation des bibliothèques de LLVM.

Sur Debian, avec les paquets du PGDG, les dépendances sont en place dès l'installation.

Sur Rocky Linux/Red Hat 8, l'installation du paquet dédié suffit :

```
# dnf install postgresql14-llvmjit
```

Sur CentOS/Red Hat 7, ce paquet supplémentaire nécessite lui-même des paquets du dépôt EPEL :

```
# yum install epel-release
# yum install postgresql14-llvmjit
```

Les systèmes CentOS/Red Hat 6 ne permettent pas d'utiliser le JIT.

Si PostgreSQL ne trouve pas les bibliothèques nécessaires, il ne renvoie pas d'erreur et continue sans tenter de JIT. Pour tester si le JIT est fonctionnel sur votre machine, il faut le chercher dans un plan quand on force son utilisation ainsi :

```
SET jit=on;
SET jit_above_cost TO 0 ;
EXPLAIN (ANALYZE) SELECT 1;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=4) (… rows=1 loops=1)
 Planning Time: 0.069 ms
 JIT:
   Functions: 1
   Options: Inlining false, Optimization false, Expressions true,
            Deforming true
   Timing:  Generation 0.123 ms, Inlining 0.000 ms, Optimization 0.187 ms,
            Emission 2.778 ms, Total 3.088 ms
 Execution Time: 3.952 ms
```

La documentation officielle est assez accessible : https://doc.postgresql.fr/current/jit.html

8.4.1 JIT : qu'est-ce qui est compilé ?

- *Tuple deforming*
- Évaluation d'expressions :
  - WHERE
  - agrégats, GROUP BY
- Appels de fonctions (*inlining*)
- Mais pas les jointures

Le JIT ne peut pas encore compiler toute une requête. La version actuelle se concentre sur des goulots d'étranglement classiques :

- le décodage des enregistrements (*tuple deforming*) pour en extraire les champs intéressants ;
- les évaluations d'expressions, notamment dans les clauses WHERE pour filtrer les lignes ;
- les agrégats, les GROUP BY…

Les jointures ne sont pas (encore ?) concernées par le JIT.

Le code résultant est utilisable plus efficacement avec les processeurs actuels qui utilisent les pipelines et les prédictions de branchement.

Pour les détails, on peut consulter notamment cette conférence très technique au FOSDEM 2018[2] par l'auteur principal du JIT, Andres Freund.

[2] https://archive.fosdem.org/2018/schedule/event/jiting_postgresql_using_llvm/

8.4.2 JIT : algorithme « naïf »

- `jit` (défaut : on)
- `jit_above_cost` (défaut : 100 000)
- `jit_inline_above_cost` (défaut : 500 000)
- `jit_optimize_above_cost` (défaut : 500 000)
- À comparer au coût de la requête… I/O comprises
- Seuils arbitraires !

De l'avis même de son auteur, l'algorithme de déclenchement du JIT est « naïf ». Quatre paramètres existent (hors débogage).

`jit = on` (défaut à partir de la v12) active le JIT **si** l'environnement technique évoqué plus haut le permet.

La compilation n'a cependant lieu que pour un coût de requête calculé d'au moins `jit_above_cost` (par défaut 100 000, une valeur élevée). Puis, si le coût atteint `jit_inline_above_cost` (500 000), certaines fonctions utilisées par la requête et supportées par le JIT sont intégrées dans la compilation. Si `jit_optimize_above_cost` (500 000) est atteint, une optimisation du code compilé est également effectuée. Ces deux dernières opérations étant longues, elles ne le sont que pour des coûts assez importants.

Ces seuils sont à comparer avec les coûts des requêtes, qui incluent les entrées-sorties, donc pas seulement le coût CPU. Ces seuils sont un peu arbitraires et nécessiteront sans doute un certain tuning en fonction de vos requêtes et de vos processeurs.

Des contre-performances dues au JIT ont déjà été observées, menant à monter les seuils. Le JIT est trop jeune pour que les développeurs de PostgreSQL eux-mêmes aient des règles d'ajustement des valeurs des différents paramètres. Il est fréquent de le désactiver ou de monter radicalement les seuils de déclenchement.

Un exemple de plan d'exécution sur une grosse table donne :

```
# EXPLAIN (ANALYZE) SELECT sum(x), count(id)
  FROM bigtable WHERE id + 2 > 500000  ;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=3403866.94..3403866.95 rows=1 width=16) (…)
   ->  Gather        (cost=3403866.19..3403866.90 rows=7 width=16)
                (actual time=11778.983..11784.235 rows=8 loops=1)
         Workers Planned: 7
         Workers Launched: 7
         ->  Partial Aggregate  (cost=3402866.19..3402866.20 rows=1 width=16)(…)
               ->  Parallel Seq Scan on bigtable  (…)
                     Filter: ((id + 2) > 500000)
                     Rows Removed by Filter: 62500
```

```
Planning Time: 0.047 ms
JIT:
  Functions: 42
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing:  Generation 5.611 ms, Inlining 422.019 ms, Optimization 229.956 ms,
           Emission 125.768 ms, Total 783.354 ms
Execution Time: 11785.276 ms
```

Le plan d'exécution est complété, à la fin, des informations suivantes :

- le nombre de fonctions concernées ;
- les temps de génération, d'inclusion des fonctions, d'optimisation du code compilé…

Dans l'exemple ci-dessus, on peut constater que ces coûts ne sont pas négligeables par rapport au temps total. Il reste à voir si ce temps perdu est récupéré sur le temps d'exécution de la requête… ce qui en pratique n'a rien d'évident.

Sans JIT, la durée de cette requête était d'environ 17 s. Ici le JIT est rentable.

8.4.3 Quand le JIT est-il utile ?

- Goulot d'étranglement au niveau CPU (pas I/O)
- Requêtes complexes (calculs, agrégats, appels de fonctions…)
- Beaucoup de lignes, filtres
- Assez longues pour « rentabiliser » le JIT
- Analytiques, pas ERP

Vu son coût élevé, le JIT n'a d'intérêt que pour les requêtes utilisant beaucoup le CPU et où il est le facteur limitant.

Ce seront donc surtout des requêtes analytiques agrégeant beaucoup de lignes, comprenant beaucoup de calculs et filtres, et non les petites requêtes d'un ERP.

Il n'y a pas non plus de mise en cache du code compilé.

Si gain il y a, il est relativement modeste en deçà de quelques millions de lignes, et devient de plus en plus important au fur et à mesure que la volumétrie augmente, à condition bien sûr que d'autres limites n'apparaissent pas (bande passante…).

Documentation officielle : https://docs.postgresql.fr/current/jit-decision.html

8.5 RECHERCHE PLEIN TEXTE

Full Text Search : Recherche Plein Texte

- Recherche « à la Google » ; fonctions dédiées
- On n'indexe plus une chaîne de caractère mais
  - les mots (« lexèmes ») qui la composent
  - on peut rechercher sur chaque lexème indépendamment
- Les lexèmes sont soumis à des règles spécifiques à chaque langue
  - notamment termes courants
  - permettent une normalisation, des synonymes…

L'indexation FTS est un des cas les plus fréquents d'utilisation non-relationnelle d'une base de données : les utilisateurs ont souvent besoin de pouvoir rechercher une information qu'ils ne connaissent pas parfaitement, d'une façon floue :

- recherche d'un produit/article par rapport à sa description ;
- recherche dans le contenu de livres/documents…

PostgreSQL doit donc permettre de rechercher de façon efficace dans un champ texte. L'avantage de cette solution est d'être intégrée au SGBD. Le moteur de recherche est donc toujours parfaitement à jour avec le contenu de la base, puisqu'il est intégré avec le reste des transactions.

Le principe est de décomposer le texte en « lexèmes » propres à chaque langue. Cela implique donc une certaine forme de normalisation, et permettent aussi de tenir compte de dictionnaires de synonymes. Le dictionnaire inclue aussi les termes courants inutiles à indexer (*stop words*) propres à la langue (le, la, et, the, and, der, daß…).

Décomposition et recherche en plein texte utilisent des fonctions et opérateurs dédiés, ce qui nécessite donc une adaptation du code. Ce qui suit n'est qu'un premier aperçu. La recherche plein texte est un chapitre entier de la documentation officielle[3].

Adrien Nayrat a donné une excellente conférence sur le sujet au PGDay France 2017 à Toulouse[4] (slides[5]).

[3] https://docs.postgresql.fr/current/textsearch.html
[4] https://www.youtube.com/embed/9S5dBqMbw8A
[5] https://2017.pgday.fr/slides/nayrat_Le_Full_Text_Search_dans_PostgreSQL.pdf

8.5.1 Full Text Search : exemple

- Décomposition :

```
SELECT to_tsvector ('french','Longtemps je me suis couché de bonne
↪  heure');

              to_tsvector
----------------------------------------
 'bon':7 'couch':5 'heur':8 'longtemp':1
```

- Recherche sur 2 mots :

```
SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('Valjean & Cosette');
```

- Recherche sur une phrase : `phrase_totsquery`

`to_tsvector` analyse un texte et le décompose en lexèmes, et non en mots. Les chiffres indiquent ici les positions et ouvrent la possibilité à des scores de proximité. Mais des indications de poids sont possibles.

Autre exemple de décomposition d'une phrase :

```
SHOW default_text_search_config ;

 default_text_search_config
----------------------------
 pg_catalog.french

SELECT to_tsvector ('La documentation de PostgreSQL est sur
↪  https://www.postgresql.org/') ;

                    to_tsvector
---------------------------------------------------
 'document':2 'postgresql':4 'www.postgresql.org':7
```

Les mots courts et le verbe « être » sont repérés comme termes trop courants, la casse est ignorée, même l'URL est décomposée en protocole et hôte. On peut voir en détail comment la FTS a procédé :

```
SELECT description, token, dictionary, lexemes
FROM ts_debug('La documentation de PostgreSQL est sur https://www.postgresql.org/') ;

   description    |       token        | dictionary  |       lexemes
------------------+--------------------+-------------+----------------------
 Word, all ASCII | La                 | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | documentation      | french_stem | {document}
 Space symbols   |                    | ¤           | ¤
```

```
 Word, all ASCII | de                 | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | PostgreSQL         | french_stem | {postgresql}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | est                | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Word, all ASCII | sur                | french_stem | {}
 Space symbols   |                    | ¤           | ¤
 Protocol head   | https://           | ¤           | ¤
 Host            | www.postgresql.org | simple      | {www.postgresql.org}
 Space symbols   | /                  | ¤           | ¤
```

Si l'on se trompe de langue, les termes courants sont mal repérés (et la recherche sera inefficace) :

```
SELECT to_tsvector ('english',
'La documentation de PostgreSQL est sur https://www.postgresql.org/');
```

```
                                  to_tsvector
-------------------------------------------------------------------------------
 'de':3 'document':2 'est':5 'la':1 'postgresql':4 'sur':6 'www.postgresql.org':7
```

Pour construire un critère de recherche, `to_tsquery` est nécessaire :

```
SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('Valjean & Cosette');
```

Les termes à chercher peuvent être combinés par &, | (ou), ! (négation), <-> (mots successifs), <N> (séparés par N lexèmes). @@ est l'opérateur de correspondance. Il y en a d'autres[6].

Il existe une fonction `phraseto_tsquery` pour donner une phrase entière comme critère, laquelle sera décomposée en lexèmes :

```
SELECT livre, contenu FROM textes
WHERE
      livre ILIKE 'Les Misérables Tome V%'
AND ( to_tsvector ('french',contenu)
        @@ phraseto_tsquery('c''est la fautes de Voltaire')
  OR  to_tsvector ('french',contenu)
        @@ phraseto_tsquery('nous sommes tombés à terre')
    );
```

```
                     livre                        |           contenu
--------------------------------------------------+-----------------------------
…
Les misérables Tome V Jean Valjean, Hugo, Victor | Je suis tombé par terre,
Les misérables Tome V Jean Valjean, Hugo, Victor | C'est la faute à Voltaire,
```

[6] https://docs.postgresql.fr/current/functions-textsearch.html

8.5.2 Full Text Search : dictionnaires

- Configurations liées à la langue
  - basées sur des dictionnaires (parfois fournis)
  - dictionnaires filtrants (`unaccent`)
  - synonymes
- Extensible grâce à des sources extérieures
- Configuration par défaut : `default_text_search_config`

Les lexèmes, les termes courants, la manière de décomposer un terme… sont fortement liés à la langue.

Des configurations toutes prêtes sont fournies par PostgreSQL pour certaines langues :

```
# \dF
        Liste des configurations de la recherche de texte
   Schéma   |    Nom     |              Description
------------+------------+---------------------------------------
 pg_catalog | arabic     | configuration for arabic language
 pg_catalog | danish     | configuration for danish language
 pg_catalog | dutch      | configuration for dutch language
 pg_catalog | english    | configuration for english language
 pg_catalog | finnish    | configuration for finnish language
 pg_catalog | french     | configuration for french language
 pg_catalog | german     | configuration for german language
 pg_catalog | hungarian  | configuration for hungarian language
 pg_catalog | indonesian | configuration for indonesian language
 pg_catalog | irish      | configuration for irish language
 pg_catalog | italian    | configuration for italian language
 pg_catalog | lithuanian | configuration for lithuanian language
 pg_catalog | nepali     | configuration for nepali language
 pg_catalog | norwegian  | configuration for norwegian language
 pg_catalog | portuguese | configuration for portuguese language
 pg_catalog | romanian   | configuration for romanian language
 pg_catalog | russian    | configuration for russian language
 pg_catalog | simple     | simple configuration
 pg_catalog | spanish    | configuration for spanish language
 pg_catalog | swedish    | configuration for swedish language
 pg_catalog | tamil      | configuration for tamil language
 pg_catalog | turkish    | configuration for turkish language
```

La recherche plein texte est donc directement utilisable pour le français ou l'anglais et beaucoup d'autres langues européennes. La configuration par défaut dépend du paramètre `default_text_search_config`, même s'il est conseillé de toujours passer explicitement la configuration aux fonctions. Ce paramètre peut être modifié globalement, par session ou par un `ALTER DATABASE SET`.

En demandant le détail de la configuration `french`, on peut voir qu'elle se base sur des « dictionnaires » pour chaque type d'élément qui peut être rencontré : mots, phrases mais aussi URL, entiers…

```
# \dF+ french
Configuration « pg_catalog.french » de la recherche de texte
Analyseur : « pg_catalog.default »
      Jeton      | Dictionnaires
-----------------+---------------
 asciihword      | french_stem
 asciiword       | french_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | french_stem
 hword_asciipart | french_stem
 hword_numpart   | simple
 hword_part      | french_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | french_stem
```

On peut lister ces dictionnaires :

```
# \dFd
            Liste des dictionnaires de la recherche de texte
   Schéma   |       Nom       |               Description
------------+-----------------+--------------------------------------------
…
 pg_catalog | english_stem    | snowball stemmer for english language
…
 pg_catalog | french_stem     | snowball stemmer for french language
…
 pg_catalog | simple          | simple dictionary: just lower case
                              | and check for stopword
…
```

Ces dictionnaires sont de type « Snowball[7] », incluant notamment des algorithmes différents pour chaque langue. Le dictionnaire `simple` n'est pas lié à une langue et correspond à une simple décomposition après passage en minuscule et recherche de termes courants anglais : c'est suffisant pour des éléments comme les URL.

D'autres dictionnaires peuvent être combinés aux existants pour créer une nouvelle configuration. Le principe est que les dictionnaires reconnaissent certains éléments, et transmettent aux suivants ce qu'ils n'ont pas reconnu. Les dictionnaires précédents, de type Snowball, reconnaissent tout et doivent donc être placés en fin de liste.

[7] https://snowballstem.org/

Par exemple, la contrib `unaccent` permet de faire une configuration négligeant les accents[8]. La contrib `dict_int` fournit un dictionnaire qui réduit la précision des nombres[9] pour réduire la taille de l'index. La contrib `dict_xsyn` permet de créer un dictionnaire pour gérer une liste de synonymes[10]. Mais les dictionnaires de synonymes peuvent être gérés manuellement[11]. Les fichiers adéquats sont déjà présents ou à ajouter dans `$SHAREDIR/tsearch_data/` (par exemple `/usr/pgsql-14/share/tsearch_data` sur Red Hat/CentOS ou `/usr/share/postgresql/14/tsearch` sur Debian).

Par exemple, en utilisant le fichier d'exemple `$SHAREDIR/tsearch_data/synonym_sample.syn`, dont le contenu est :

```
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

on peut définir un dictionnaire de synonymes, créer une nouvelle configuration reprenant `french`, et y insérer le nouveau dictionnaire en premier élément :

```
CREATE TEXT SEARCH DICTIONARY messynonymes (template=synonym,
 ↪  synonyms='synonym_sample');

CREATE TEXT SEARCH CONFIGURATION french2 (copy=french);

ALTER TEXT SEARCH CONFIGURATION french2
ALTER MAPPING FOR asciiword,hword,asciihword,word
WITH messynonymes, french_stem ;
```

À l'usage :

```
SELECT to_tsvector ('french2', 'PostgreSQL s''abrège en pgsql ou Postgres') ;

         to_tsvector
-------------------------
 'abreg':3 'pgsql':1,5,7
```

Les trois versions de « PostgreSQL » ont été reconnues.

Pour une analyse plus fine, on peut ajouter d'autres dictionnaires linguistiques depuis des sources extérieures (Ispell, OpenOffice…). Ce n'est pas intégré par défaut à PostgreSQL mais la procédure est dans la documentation[12].

Des « thesaurus » peuvent être même être créés pour remplacer des expressions par des synonymes (et identifier par exemple « le meilleur SGBD » et « PostgreSQL »).

[8] https://docs.postgresql.fr/current/unaccent.html
[9] https://docs.postgresql.fr/current/dict-int.html
[10] https://docs.postgresql.fr/current/dict-xsyn.html
[11] https://docs.postgresql.fr/current/textsearch-dictionaries.html#textsearch-synonym-dictionary
[12] https://docs.postgresql.fr/current/textsearch-dictionaries.html

8.5.3 Full Text Search : stockage & indexation

- Stocker `to_tsvector (champtexte)`
  - colonne mise à jour par trigger
  - ou colonne générée (v12)
- Indexation GIN ou GiST

Principe :

Sans indexation, une recherche FTS fonctionne, mais parcourra entièrement la table. L'indexation est possible, avec GIN ou GiST. On peut stocker le vecteur résultat de `to_tsvector` dans une autre colonne de la table, et c'est elle qui sera indexée. Jusque PostgreSQL 11, il est nécessaire de le faire manuellement, ou d'écrire un trigger pour cela. À partir de PostgreSQL 12, on peut utiliser une colonne générée (il est nécessaire de préciser la configuration FTS) :

```
ALTER TABLE textes
ADD COLUMN vecteur tsvector
GENERATED ALWAYS AS (to_tsvector ('french', contenu)) STORED ;
```

Les critères de recherche porteront sur la colonne `vecteur` :

```
SELECT * FROM textes
WHERE vecteur @@ to_tsquery ('french','Roméo <2> Juliette');
```

Cette colonne sera ensuite indexée par GIN pour avoir des temps d'accès corrects :

```
CREATE INDEX on textes USING gin (vecteur) ;
```

Alternative : index fonctionnel

Plus simplement, il peut suffire de créer juste un index fonctionnel sur `to_tsvector ('french', contenu)`. On épargne ainsi l'espace du champ calculé dans la table.

Par contre, l'index devra porter sur le critère de recherche exact, sinon il ne sera pas utilisable. Cela n'est donc pertinent que si la majorité des recherches porte sur un nombre très restreint de critères, et il faudra un index par critère.

```
CREATE INDEX idx_fts ON public.textes
USING gin (to_tsvector('french'::regconfig, contenu))

SELECT * FROM textes
WHERE to_tsvector ('french', contenu) @@ to_tsquery ('french','Roméo <2> Juliette');
```

Exemple complet de mise en place de FTS :

- Création d'une configuration de dictionnaire dédiée avec dictionnaire français, sans accent, dans une table de dépêches :

```
CREATE TEXT SEARCH CONFIGURATION depeches (COPY= french);

CREATE EXTENSION unaccent ;

ALTER TEXT SEARCH CONFIGURATION depeches ALTER MAPPING FOR
hword, hword_part, word WITH unaccent,french_stem;
```

- Ajout d'une colonne vectorisée à la table depeche, avec des poids différents pour le titre et le texte, ici gérée manuellement avec un trigger.

```
CREATE TABLE depeche (id int, titre text, texte text) ;

ALTER TABLE depeche ADD vect_depeche tsvector;

UPDATE depeche
SET vect_depeche =
(setweight(to_tsvector('depeches',coalesce(titre,'')), 'A') ||
setweight(to_tsvector('depeches',coalesce(texte,'')), 'C'));

CREATE FUNCTION to_vectdepeche( )
 RETURNS trigger
 LANGUAGE plpgsql
 -- common options:  IMMUTABLE  STABLE  STRICT  SECURITY DEFINER
AS $function$
BEGIN
  NEW.vect_depeche :=
    setweight(to_tsvector('depeches',coalesce(NEW.titre,'')), 'A') ||
    setweight(to_tsvector('depeches',coalesce(NEW.texte,'')), 'C');
  return NEW;
END
$function$;

CREATE TRIGGER trg_depeche before INSERT OR update ON depeche
FOR EACH ROW execute procedure to_vectdepeche();
```

- Création de l'index associé au vecteur :

```
 CREATE INDEX idx_gin_texte ON depeche USING gin(vect_depeche);
```

- Collecte des statistiques sur la table :

```
ANALYZE depeche ;
```

- Utilisation basique :

```
SELECT titre,texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle');
SELECT titre,texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle & médecin');
```

- Tri par pertinenence :

```
SELECT titre,texte
FROM depeche
WHERE vect_depeche @@ to_tsquery('depeches','varicelle & médecin')
ORDER BY ts_rank_cd(vect_depeche, to_tsquery('depeches','varicelle & médecin'));
```

- Cette requête peut s'écrire aussi ainsi :

```
SELECT titre,ts_rank_cd(vect_depeche,query) AS rank
FROM depeche, to_tsquery('depeches','varicelle & médecin') query
WHERE query@@vect_depeche
ORDER BY rank DESC ;
```

8.5.4 Full Text Search sur du JSON

- Vectorisation possible des JSON

```
SELECT info FROM commandes c
WHERE to_tsvector ('french', c.info) @@ to_tsquery('papier') ;

                                                info
-------------------------------------------------------------------------------
 ↪  ----------
 {"items": {"qté": 5, "produit": "Rame papier normal A4"}, "client":
 ↪  "Benoît Delaporte"}
 {"items": {"qté": 5, "produit": "Pochette Papier dessin A3"}, "client":
 ↪  "Lucie Dumoulin"}
```

Depuis la version 10 de PostgreSQL, une recherche FTS est directement possible sur des champs JSON. Voici un exemple :

```
CREATE TABLE commandes (info jsonb);

INSERT INTO commandes (info)
VALUES
   (
      '{ "client": "Jean Dupont",
         "articles": {"produit": "Enveloppes A4","qté": 24}}'
   ),
   (
      '{ "client": "Jeanne Durand",
         "articles": {"produit": "Imprimante","qté": 1}}'
   ),
   (
      '{ "client": "Benoît Delaporte",
         "items": {"produit": "Rame papier normal A4","qté": 5}}'
   ),
   (
      '{ "client": "Lucie Dumoulin",
         "items": {"produit": "Pochette Papier dessin A3","qté": 5}}'
   );
```

La décomposition par FTS donne :

```
SELECT to_tsvector('french', info) FROM commandes ;
```

```
                  to_tsvector
-------------------------------------------------
 'a4':5 'dupont':2 'envelopp':4 'jean':1
 'durand':2 'imprim':4 'jeann':1
 'a4':4 'benoît':6 'delaport':7 'normal':3 'papi':2 'ram':1
 'a3':4 'dessin':3 'dumoulin':7 'luc':6 'papi':2 'pochet':1
```

Une recherche sur « papier » donne :

```
SELECT info FROM commandes c
WHERE to_tsvector ('french', c.info) @@ to_tsquery('papier') ;
```

```
                                        info
-------------------------------------------------------------------------------
 {"items": {"qté": 5, "produit": "Rame papier normal A4"}, "client": "Benoît
 ↪  Delaporte"}
 {"items": {"qté": 5, "produit": "Pochette Papier dessin A3"}, "client": "Lucie
 ↪  Dumoulin"}
```

Plus d'information chez Depesz : Full Text Search support for json and jsonb[13].

[13] https://www.depesz.com/2017/04/04/waiting-for-postgresql-10-full-text-search-support-for-json-and-jsonb/

8.6 QUIZ

https://dali.bo/t1_quiz

8.7 TRAVAUX PRATIQUES

8.7.1 Tables non journalisées

But : Tester les tables non journalisées

Afficher le nom du journal de transaction courant.

Créer une base **pgbench** vierge, de taille 80 (environ 1,2 Go). Les tables doivent être en mode **unlogged**.

Afficher la liste des objets **unlogged** dans la base **pgbench**.

Afficher le nom du journal de transaction courant. Que s'est-il passé ?

Passer l'ensemble des tables de la base **pgbench** en mode **logged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé ?

Repasser toutes les tables de la base **pgbench** en mode **unlogged**.

Afficher le nom du journal de transaction courant. Que s'est-il passé ?

Réinitialiser la base **pgbench** toujours avec une taille 80 mais avec les tables en mode **logged**. Que constate-t-on ?

Réinitialiser la base **pgbench** mais avec une taille de 10. Les tables doivent être en mode **unlogged**.

Compter le nombre de lignes dans la table `pgbench_accounts`.

Simuler un crash de l'instance PostgreSQL.

Redémarrer l'instance PostgreSQL.

Compter le nombre de lignes dans la table `pgbench_accounts`. Que constate-t-on ?

8.7.2 Indexation Full Text

But : Tester l'indexation *Full Text*

Vous aurez besoin de la base **textes**. La base est disponible en deux versions : complète sur https://dali.bo/tp_gutenberg (dump de 0,5 Go, table de 21 millions de lignes dans 3 Go) ou https://dali.bo/tp_gutenberg10 pour un extrait d'un dizième. Le dump peut se charger dans une base préexistante avec `pg_restore` et créera juste une table nommée `textes`.

Ce TP utilise la version complète de la base **textes** basée sur le projet Gutenberg. Un index GIN va permettre d'utiliser la *Full Text Search* sur la table **textes**.

Créer un index GIN sur le vecteur du champ `contenu` (fonction `to_tsvector`).

Quelle est la taille de cet index ?

Quelle performance pour trouver « Fantine » (personnage des *Misérables* de Victor Hugo) dans la table ? Le résultat contient-il bien « Fantine » ?

Trouver les lignes qui contiennent à la fois les mots « affaire » et « couteau » et voir le plan.

8.8 TRAVAUX PRATIQUES (SOLUTIONS)

8.8.1 Tables non journalisées

> Afficher le nom du journal de transaction courant.

```
SELECT pg_walfile_name(pg_current_wal_lsn()) ;
```

```
     pg_walfile_name
--------------------------
 000000010000000100000024
```

> Créer une base **pgbench** vierge, de taille 80 (environ 1,2 Go). Les tables doivent être en mode **unlogged**.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 80 --unlogged-tables pgbench
```

```
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
8000000 of 8000000 tuples (100%) done (elapsed 4.93 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 8.84 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 5.02 s,
vacuum 1.79 s, primary keys 2.02 s).
```

> Afficher la liste des objets **unlogged** dans la base **pgbench**.

```
SELECT relname FROM pg_class
WHERE relpersistence = 'u' ;
```

```
        relname
-----------------------
 pgbench_accounts
 pgbench_branches
 pgbench_history
 pgbench_tellers
 pgbench_branches_pkey
 pgbench_tellers_pkey
 pgbench_accounts_pkey
```

Les 3 objets avec le suffixe **pkey** correspondent aux clés primaires des tables créées par **pgbench**. Comme elles dépendent des tables, elles sont également en mode **unlogged**.

> Afficher le nom du journal de transaction courant. Que s'est-il passé ?

```
SELECT pg_walfile_name(pg_current_wal_lsn()) ;
```

```
     pg_walfile_name
--------------------------
 000000010000000100000024
```

Comme l'initialisation de **pgbench** a été réalisée en mode **unlogged**, aucune information concernant les tables et les données qu'elles contiennent n'a été inscrite dans les journaux de transaction. Donc le journal de transaction est toujours le même.

> Passer l'ensemble des tables de la base **pgbench** en mode **logged**.

```
ALTER TABLE pgbench_accounts SET LOGGED;
ALTER TABLE pgbench_branches SET LOGGED;
ALTER TABLE pgbench_history SET LOGGED;
ALTER TABLE pgbench_tellers SET LOGGED;
```

> Afficher le nom du journal de transaction courant. Que s'est-il passé ?

```
SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
     pg_walfile_name
--------------------------
 000000010000000100000077
```

Comme toutes les tables de la base **pgbench** ont été passées en mode **logged**, une réécriture de celles-ci a eu lieu (comme pour un `VACUUM FULL`). Cette réécriture additionnée au mode **logged** a entraîné une forte écriture dans les journaux de transaction. Dans notre cas, 83 journaux de transaction ont été consommés, soit approximativement 1,3 Go d'utilisé sur disque.

Il faut donc faire particulièrement attention à la quantité de journaux de transaction qui peut être générée lors du passage d'une table du mode **unlogged** à **logged**.

> Repasser toutes les tables de la base **pgbench** en mode **unlogged**.

```
ALTER TABLE pgbench_accounts SET UNLOGGED;
ALTER TABLE pgbench_branches SET UNLOGGED;
ALTER TABLE pgbench_history SET UNLOGGED;
ALTER TABLE pgbench_tellers SET UNLOGGED;
```

> Afficher le nom du journal de transaction courant. Que s'est-il passé ?

```
SELECT pg_walfile_name(pg_current_wal_lsn());
```

```
     pg_walfile_name
--------------------------
 000000010000000100000077
```

Le processus est le même que précedemment, mais, lors de la réécriture des tables, aucune information n'est stockée dans les journaux de transaction.

> Réinitialiser la base **pgbench** toujours avec une taille 80 mais avec les tables en mode **logged**. Que constate-t-on ?

```
$ /usr/pgsql-14/bin/pgbench -i -s 80 -d pgbench
```

```
dropping old tables...
creating tables...
generating data (client-side)...
8000000 of 8000000 tuples (100%) done (elapsed 9.96 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 16.60 s (drop tables 0.11 s, create tables 0.00 s, client-side generate 10.12
↪  s,
vacuum 2.87 s, primary keys 3.49 s).
```

On constate que le temps mis par **pgbench** pour initialiser sa base est beaucoup plus long en mode **logged** que **unlogged**. On passe de 8,84 secondes en **unlogged** à 16,60 secondes en mode **logged**. Cette augmentation du temps de traitement est due à l'écriture dans les journaux de transaction.

> Réinitialiser la base **pgbench** mais avec une taille de 10. Les tables doivent être en mode **unlogged**.

```
$ /usr/pgsql-14/bin/pgbench -i -s 10 -d pgbench --unlogged-tables
```

```
dropping old tables...
creating tables...
generating data (client-side)...
1000000 of 1000000 tuples (100%) done (elapsed 0.60 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 1.24 s (drop tables 0.02 s, create tables 0.02 s, client-side generate 0.62 s,
vacuum 0.27 s, primary keys 0.31 s).
```

> Compter le nombre de lignes dans la table `pgbench_accounts`.

```
SELECT count(*) FROM pgbench_accounts ;
```

```
  count
---------
 1000000
```

> Simuler un crash de l'instance PostgreSQL.

```
$ ps -ef | grep postmaster
```

```
postgres   697  1  0 14:32 ?   00:00:00 /usr/pgsql-14/bin/postmaster -D ...
```

```
$ kill -9 697
```

Ne faites jamais un `kill -9` sur un processus de l'instance PostgreSQL en production, bien sûr !

> Redémarrer l'instance PostgreSQL.

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data start
```

> Compter le nombre de lignes dans la table `pgbench_accounts`. Que constate-t-on ?

```
SELECT count(*) FROM pgbench_accounts ;

 count
-------
     0
```

Lors d'un crash, PostgreSQL remet tous les objets **unlogged** à zéro.

8.8.2 Indexation Full Text

> Créer un index GIN sur le vecteur du champ `contenu` (fonction `to_tsvector`).

```
textes=# CREATE INDEX idx_fts ON textes
USING gin (to_tsvector('french',contenu));
CREATE INDEX
```

> Quelle est la taille de cet index ?

La table « pèse » 3 Go (même si on pourrait la stocker de manière beaucoup plus efficace). L'index GIN est lui-même assez lourd dans la configuration par défaut :

```
textes=# SELECT pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 593 MB
(1 ligne)
```

> Quelle performance pour trouver « Fantine » (personnage des *Misérables* de Victor Hugo) dans la table ? Le résultat contient-il bien « Fantine » ?

```
textes=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');

                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=107.94..36936.16 rows=9799 width=123)
                          (actual time=0.423..1.149 rows=326 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu)
                 @@ '''fantin'''::tsquery)
   Heap Blocks: exact=155
   Buffers: shared hit=159
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..105.49 rows=9799 width=0)
                             (actual time=0.210..0.211 rows=326 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu)
                      @@ '''fantin'''::tsquery)
         Buffers: shared hit=4
 Planning Time: 1.248 ms
 Execution Time: 1.298 ms
```

On constate donc que le *Full Text Search* est très efficace du moins pour le *Full Text Search* + GIN : trouver 1 mot parmi plus de 100 millions avec 300 enregistrements correspondants dure 1,5 ms (cache chaud).

Si l'on compare avec une recherche par trigramme (extension `pg_trgm` et index GIN), c'est bien meilleur. À l'inverse, les trigrammes permettent des recherches floues (orthographe approximative), des recherches sur autre chose que des mots, et ne nécessitent pas de modification de code.

Par contre, la recherche n'est pas exacte, « Fantin » est fréquemment trouvé. En fait, le plan montre que c'est le vrai critère retourné par `to_tsquery('french','fantine')` et transformé en `'fantin'::tsquery`. Si l'on tient à ce critère précis il faudra ajouter une clause plus classique `contenu LIKE '%Fantine%'` pour filtrer le résultat après que le FTS ait « dégrossi » la recherche.

> Trouver les lignes qui contiennent à la fois les mots « affaire » et « couteau » et voir le plan.

10 lignes sont ramenées en quelques millisecondes :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','affaire  & couteau')
;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=36.22..154.87 rows=28 width=123)
                      (actual time=6.642..6.672 rows=10 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu)
                 @@ '''affair'' & ''couteau'''::tsquery)
   Heap Blocks: exact=10
   Buffers: shared hit=53
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..36.21 rows=28 width=0)
                            (actual time=6.624..6.624 rows=10 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu)
                     @@ '''affair'' & ''couteau'''::tsquery)
         Buffers: shared hit=43
 Planning Time: 0.519 ms
 Execution Time: 6.761 ms
```

Noter que les pluriels « couteaux » et « affaires » figurent parmi les résultats puisque la recherche porte sur les lexèmes `'affair'' & ''couteau'`.

9/ Masquage de données & postgresql_anonymizer

9.1 CAS D'USAGE

- Paul : le propriétaire
- Pierre : *data scientist*
- Jack : employé chargé des fournisseurs

La boutique de Paul a beaucoup de clients. Paul demande à son ami Pierre, *data scientist*, des statistiques sur ses clients (âge moyen, etc.).

Pierre demande un accès direct à la base de données pour écrire ses requêtes SQL.

Jack est un employé de Paul, chargé des relations avec les divers fournisseurs de la boutique.

Paul respecte la vie privée de ses fournisseurs. Il doit masquer les informations personnelles à Pierre, mais Jack doit pouvoir lire les vraies données, et y écrire.

Crédits

Cet exemple pratique est un travail collectif de Damien Clochard, Be Hai Tran, Florent Jardin et Frédéric Yhuel.

La version originale en anglais est diffusée sous licence PostgreSQL[1].

Le présent document en est l'adaptation en français.

Paul's Boutique est le second album studio du groupe de hip-hop américain les *Beastie Boys*, sorti le 25 juillet 1989 chez Capitol Records.

La photo ci-dessus est d'Erwin Bernal[2], sous licence CC BY 2.0[3].

9.1.1 Objectifs

Nous allons découvrir :

- comment écrire des règles de masquage
- la différence entre masquage dynamique et masquage statique
- comment implémenter un masquage avancé

[1] https://gitlab.com/dalibo/postgresql_anonymizer/-/tree/master/docs/how-to

[2] https://www.flickr.com/photos/edogisgod/16858046971/in/photolist-rFFU3g-4NZreN-xKEkv-h9ZxgT-aMD5mr-8dAvwU-cru1CN-xFfJgh-8QhtH6-6E81fG-zUN3Rg-7dCVPA-5VbEct-ewX2Lc-hA4JqP-psCh1y-dmZpzf-pjkwX-cu5NNQ-fTyVqj-MjtjRc-cLdmvW-3fd5BM-9m6ChY-dwLhRK-9d2A9s-6WsfHq-abVSJd-dWYBD4-gmJyHe-bVyJNn-SHTV2b-BoMvci-abVPJW-5pgugb-r4oJpD-6YeAE3-6kKVpZ-e4LeQN-BLUvZG-do1bDr-o5YACZ-9karFj-dPuSLW-btYwsQ-e4L9h9-abT31z-3eWVAZ-abVN43-btYwvJ

[3] https://creativecommons.org/licenses/by/2.0/deed.fr

10/ PostgreSQL Anonymizer

10.0.1 Principe

Principe :

- Extension
- Déclaratif (DDL)

`postgresql_anonymizer` est une extension pour masquer ou remplacer des données personnelles[1] (ou PII pour *personnally identifiable information*) ou toute donnée sensible dans une base de données PostgreSQL.

Le projet a une **approche déclarative** de l'anonymisation. Vous pouvez déclarer les règles de masquage[2] dans PostgreSQL avec du DDL (*Data Definition Language*, ou langage de définition des données) et spécifier votre stratégie d'anonymisation dans la définition de la table.

10.0.2 Masquages

Principe :

- statique
- dynamique
- sauvegardes anonymisées
- « généralisation »

[1] https://en.wikipedia.org/wiki/Personally_identifiable_information
[2] https://postgresql-anonymizer.readthedocs.io/en/stable/declare_masking_rules/

Une fois les règles de masquage définies, vous pouvez accéder aux données anonymisées de quatre manières différentes :

- le masquage statique[3] : supprime les données personnelles en fonction des règles ;
- le masquage dynamique[4] : cache les données personnelles uniquement des utilisateurs masqués ;
- les sauvegardes anonymisées[5] : export des données masquées vers un fichier SQL (sauvegarde logique) ;
- la généralisation[6] : crée des « vues brouillées » des données originales.

Cette présentation n'entrera pas dans le détail du RGPD et des principes généraux d'anonymisation. Pour plus d'informations, référez-vous à la présentation de Damien Clochard ci-dessous :

- Anonymisation, Au-delà du RGPD (vidéo)[7] (PGSession 12, Paris 2019)
- Anonymisation, Au-delà du RGPD (PDF)[8]
- Anonymization, Beyond GDPR (PDF en anglais)[9]

10.0.3 Pré-requis

Cet exemple nécessite :

- une instance PostgreSQL ;
- l'extension **PostgreSQL Anonymizer** (anon)
  - installée, initialisée par un super-utilisateur
- une base **boutique**
  - dont le propriétaire est **paul**, super-utilisateur
- les rôles **pierre** et **jack**
  - avec droits de connexion à **boutique**

Voir section « Installation »[10] dans la documentation[11] pour savoir comment installer l'extension dans votre instance PostgreSQL.

[3] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/
[4] https://postgresql-anonymizer.readthedocs.io/en/stable/dynamic_masking/
[5] https://postgresql-anonymizer.readthedocs.io/en/stable/anonymous_dumps/
[6] https://postgresql-anonymizer.readthedocs.io/en/stable/generalization/
[7] https://www.youtube.com/watch?v=KGSlp4UygdU
[8] https://public.dalibo.com/exports/conferences/20191210_poss_anonymisation/anonymisation.pdf
[9] https://public.dalibo.com/exports/conferences/20191016_anonymisation_beyond_GDPR/anonymisation_beyond_gdpr.pdf
[10] https://postgresql-anonymizer.readthedocs.io/en/stable/INSTALL
[11] https://postgresql-anonymizer.readthedocs.io/en/stable/

Par exemple :

10.0.3.1 Sous Rocky Linux 8

- Les dépôts du PGDG[12] doivent être installés.
- Lancer :

```
sudo yum install postgresql_anonymizer_14
```

10.0.3.2 Par le PGXN

Sur Debian/Ubuntu, les paquets ne sont pas disponibles au moment où ceci est écrit. Le PGXN permet d'installer l'extension (ici en PostgreSQL 14) :

```
sudo apt install pgxnclient postgresql-server-dev-14
sudo pgxn install postgresql_anonymizer
```

S'il y a plusieurs versions de PostgreSQL installées, indiquer le répertoire des binaires de la bonne version ainsi :

```
sudo PATH=/usr/lib/postgresql/14/bin:$PATH   pgxn install postgresql_anonymizer
```

L'extension sera compilée et installée.

10.0.4 Base d'exemple

pierre, **paul**, **jack** et la base **boutique** :

```
CREATE ROLE paul LOGIN SUPERUSER;
CREATE ROLE pierre LOGIN;
CREATE ROLE jack LOGIN;

-- Define a password for each user with:
-- \password paul  or  ALTER ROLE paul PASSWORD 'change-me';

CREATE DATABASE boutique OWNER paul;

ALTER DATABASE boutique
  SET session_preload_libraries = 'anon';
```

Sauf précision contraire, toutes les commandes sont à exécuter en tant que **paul**.

[12] https://yum.postgresql.org/

11/ Masquage statique avec postgresql_anonymizer

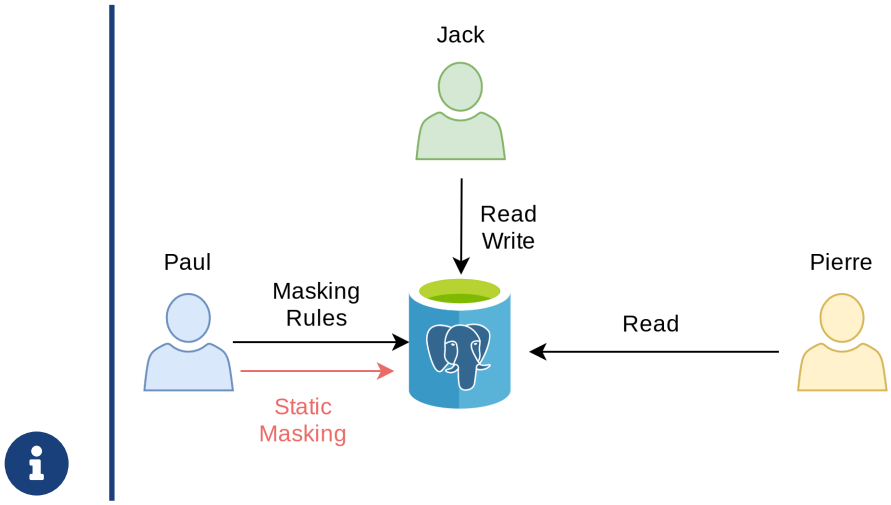
- Le plus simple
- Destructif

Le masquage statique est la manière la plus simple de cacher des données personnelles. L'idée est simplement de détruire les données originales et de les remplacer par des données artificielles.

11.1 L'HISTOIRE

- Au fil des années, Paul a accumulé des données sur ses clients et leurs achats dans une base de données très simple.
- Il a récemment installé un nouveau logiciel de ventes, et l'ancienne base est obsolète.
- Avant de l'archiver, il voudrait en supprimer toutes les données personnelles.

11.2 COMMENT ÇA MARCHE



11.3 OBJECTIFS

Nous allons voir :

- comment écrire des règles de masquage simples
- les intérêts et limitations du masquage statique
- le concept de « singularisation » d'une personne (*singling out*)

11.4 TABLE « CUSTOMER »

```
\c boutique paul
DROP TABLE IF EXISTS customer CASCADE;
DROP TABLE IF EXISTS payout CASCADE;

CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    firstname TEXT,
    lastname TEXT,
    phone TEXT,
    birth DATE,
    postcode TEXT
);
```

11.4.1 Quelques clients

Insertion de quelques personnes :

```
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;
```

```
SELECT * FROM customer;
```

```
 id  | firstname | lastname  |    phone     |   birth    | postcode
-----+-----------+-----------+--------------+------------+----------
 107 | Sarah     | Conor     | 060-911-0911 | 1965-10-10 | 90016
 258 | Luke      | Skywalker |              | 1951-09-25 | 90120
 341 | Don       | Draper    | 347-515-3423 | 1926-06-01 | 04520
(3 lignes)
```

11.5 TABLE « PAYOUT »

Les ventes sont suivies dans cette simple table :

```
CREATE TABLE payout (
    id SERIAL PRIMARY KEY,
    fk_customer_id INT REFERENCES customer(id),
    order_date DATE,
    payment_date DATE,
    amount INT
);
```

11.5.1 Quelques données

Quelques commandes :

```
INSERT INTO payout
VALUES
(1,107,'2021-10-01','2021-10-01', '7'),
(2,258,'2021-10-02','2021-10-03', '20'),
(3,341,'2021-10-02','2021-10-02', '543'),
(4,258,'2021-10-05','2021-10-05', '12'),
(5,258,'2021-10-06','2021-10-06', '92')
;
```

11.5.2 Activer l'extension

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE ;
SELECT anon.init() ;
SELECT setseed(0) ;
```

NB : l'extension `pgcrypto` sera installée automatiquement. Ses binaires sont généralement livrés avec PostgreSQL.

11.6 DÉCLARER LES RÈGLES DE MASQUAGE

```
SECURITY LABEL FOR anon ON COLUMN customer.lastname
IS 'MASKED WITH FUNCTION anon.fake_last_name()';

SECURITY LABEL FOR anon ON COLUMN customer.phone
IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)';
```

Paul veut masquer le nom de famille et le numéro de téléphone de ses clients.

Pour cela, il utilise les fonctions `fake_last_name()` et `partial()`.

11.7 APPLIQUER LES RÈGLES DE MANIÈRE PERMANENTE

```
SELECT anon.anonymize_table('customer');
```

Cette fonction ne fait qu'appliquer la règle et doit renvoyer True. Ensuite :

```
SELECT id, firstname, lastname, phone
FROM customer;
```

```
 id  | firstname | lastname |    phone
-----+-----------+----------+--------------
 107 | Sarah     | Okuneva  | 06X-XXX-XX11
 258 | Luke      | Okuneva  |
 341 | Don       | Boyle    | 34X-XXX-XX23
```

Cette technique est nommée « masquage statique » car la donnée réelle a été détruite de manière définitive. L'anonymisation dynamique et les exports seront vus plus loin.

11.8 EXERCICES

11.8.1 E101 - Masquer les prénoms des clients

> Déclarer une nouvelle règle de masquage et relancer l'anonymisation statique.

11.8.2 E102 - Masquer les 3 derniers chiffres du code postal

Paul réalise que le code postal est un bon indice sur l'endroit où vivent ses clients. Cependant, il voudrait pouvoir faire des statistiques par département.

> Créer une règle de masquage pour remplacer les 3 derniers chiffres du code postal par 'x'.

11.8.3 E103 - Compter le nombre de clients dans chaque département.

> Agréger les clients selon le code postal anonymisé.

11.8.4 E104 - Ne garder que l'année dans les dates de naissance

Paul veut des statistiques selon l'âge. Mais il veut aussi masquer les vraies dates de naissance de ses clients.

> Remplacer toutes les dates de naissance par le 1er janvier, en conservant l'année réelle. Utiliser la fonction make_date[a].
>
> [a] https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME-TABLE

11.8.5 E105 - Identifier un client particulier

Même si un client est correctement anonymisé, il est possible d'isoler un individu grâce à des données d'autres tables. Par exemple, il est possible d'identifier le meilleur client de Paul avec une telle requête :

```
WITH best_client AS (
    SELECT SUM(amount), fk_customer_id
    FROM payout
    GROUP BY fk_customer_id
    ORDER BY 1 DESC
    LIMIT 1
)
SELECT c.*
FROM customer c
JOIN best_client b ON (c.id = b.fk_customer_id) ;
```

```
 id  | firstname | lastname |    phone     |   birth    | postcode
-----+-----------+----------+--------------+------------+----------
 341 | Don       | Boyle    | 34X-XXX-XX23 | 1926-06-01 | 04520
```

Ce processus est appelé « singularisation » (*singling out*[1]) d'une personne.

Il faut donc aller plus loin dans l'anonymisation, en supprimant le lien entre une personne et sa société. Dans la table des commandes `order`, ce lien est matérialisé par une clé étrangère sur le champ `fk_company_id`. Mais nous ne pouvons supprimer des valeurs de cette colonne ou y insérer de faux identifiants, car cela briserait la contrainte de clé étrangère.

Comment séparer les clients de leurs paiements tout en respectant l'intégrité des données ?

> Trouver une fonction qui mélange les valeurs de `fk_company_id` dans la table `payout`. Consulter la section *shuffling*[a] de la documentation[b].
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/#shuffling
> [b] https://postgresql-anonymizer.readthedocs.io/en/stable/

[1] https://www.pnas.org/content/117/15/8344

11.9 SOLUTIONS

11.9.1 S101

```
SECURITY LABEL FOR anon ON COLUMN customer.firstname
IS 'MASKED WITH FUNCTION anon.fake_first_name()' ;

SELECT anon.anonymize_table('customer') ;
```

La table anonymisée devient :

```
SELECT id, firstname, lastname
FROM customer ;

 id  | firstname | lastname
-----+-----------+----------
 107 | Hans      | Barton
 258 | Jacqueline | Dare
 341 | Sibyl     | Runte
```

11.9.2 S102

```
SECURITY LABEL FOR anon ON COLUMN customer.postcode
IS 'MASKED WITH FUNCTION anon.partial(postcode,2,$$xxx$$,0)' ;

SELECT anon.anonymize_table('customer') ;
```

Le code postal anonymisé devient :

```
SELECT id, firstname, lastname, postcode
FROM customer ;

 id  | firstname | lastname | postcode
-----+-----------+----------+----------
 107 | Curt      | O'Hara   | 90xxx
 258 | Agusta    | Towne    | 90xxx
 341 | Sid       | Hane     | 04xxx
```

Noter que les noms ont encore changé après application de `anon.anonymize_table()`.

11.9.3 S103

```
SELECT postcode, COUNT(id)
FROM customer
GROUP BY postcode;

 postcode | count
----------+-------
 90xxx    |     2
 04xxx    |     1
```

11.9.4 S104

```
SECURITY LABEL FOR anon ON COLUMN customer.birth
IS 'MASKED WITH FUNCTION make_date(EXTRACT(YEAR FROM birth)::INT,1,1)';

SELECT anon.anonymize_table('customer');
```

Les dates de naissance anonymisées deviennent :

```
SELECT id, firstname, lastname, birth
FROM customer ;
```

```
 id  | firstname | lastname |   birth
-----+-----------+----------+------------
 107 | Pinkie    | Sporer   | 1965-01-01
 258 | Zebulon   | Gerlach  | 1951-01-01
 341 | Erna      | Emmerich | 1926-01-01
```

11.9.5 S105

Pour mélanger les valeurs de `fk_customer_id` :

```
SELECT anon.shuffle_column('payout','fk_customer_id','id') ;
```

Si l'on essaie à nouveau d'identifier le meilleur client :

```
WITH best_client AS (
    SELECT SUM(amount), fk_customer_id
    FROM payout
    GROUP BY fk_customer_id
    ORDER BY 1 DESC
    LIMIT 1
)
SELECT c.*
FROM customer c
JOIN best_client b ON (c.id = b.fk_customer_id) ;
```

```
 id  | firstname | lastname | phone |   birth    | postcode
-----+-----------+----------+-------+------------+----------
 258 | Zebulon   | Gerlach  |       | 1951-01-01 | 90xxx
```

Noter que le lien entre un client (`customer`) et ses paiements (`payout`) est à présent complètement faux !

Par exemple, si un client A a deux paiements, l'un se retrouvera associé à un client B, et l'autre à un client C. En d'autres termes, cette méthode de mélange respectera la contrainte d'intégrité technique, mais brisera l'intégrité des données. Pour certaines utilisations, ce peut être problématique.

Ici, Pierre ne pourra pas produire de rapport avec les données mélangées.

12/ Masquage dynamique avec postgresql_anonymizer

12.1 PRINCIPE DU MASQUAGE DYNAMIQUE

- Masquer les données personnelles à certains utilisateurs
  - mais pas tous

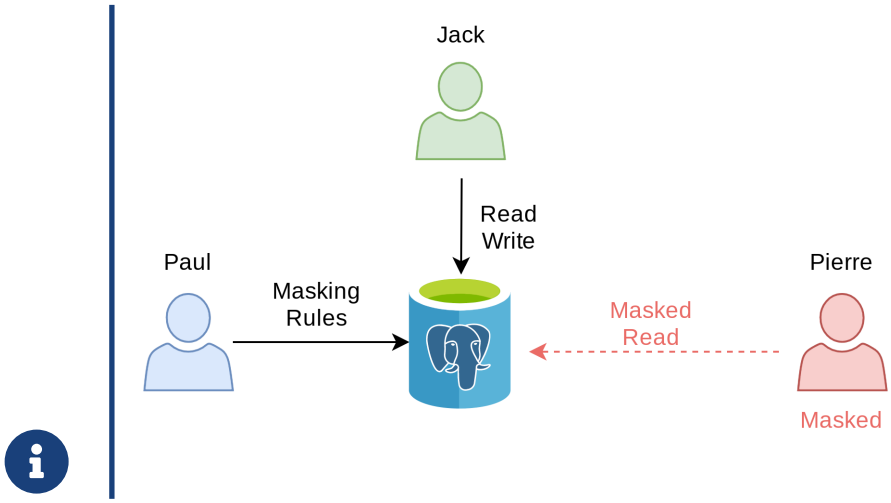
Avec le masquage dynamique, le propriétaire de la base peut masquer les données personnelles à certains utilisateurs, tout en laissant aux autres les droits de lire et modifier les données réelles.

12.2 L'HISTOIRE

Paul a 2 employés :

- **Jack** s'occupe du nouveau logiciel de ventes.
  - il a besoin d'accéder aux vraies données
  - pour le RGPD c'est un « processeur de données »
- **Pierre** est un analyste qui exécute des requêtes statistiques
  - il ne doit pas avoir accès aux données personnelles

12.3 COMMENT ÇA MARCHE



12.4 OBJECTIFS DE LA SECTION

Nous allons voir :

- comment écrire des règles de masquage simple
- les avantages et limitations du masquage dynamique
- le concept de « recoupement » d'une personne (*linkability*)

12.5 TABLE « COMPANY »

```
DROP TABLE IF EXISTS supplier CASCADE;
DROP TABLE IF EXISTS company CASCADE;

CREATE TABLE company (
    id SERIAL PRIMARY KEY,
    name TEXT,
    vat_id TEXT UNIQUE
);
```

12.5.1 Quelques données

```
INSERT INTO company
VALUES
(952,'Shadrach', 'FR62684255667'),
(194,E'Johnny\'s Shoe Store','CHE670945644'),
(346,'Capitol Records','GB663829617823')
;
```

```
SELECT * FROM company ;
```

```
 id  |        name         |     vat_id
-----+---------------------+----------------
 952 | Shadrach            | FR62684255667
 194 | Johnny's Shoe Store | CHE670945644
 346 | Capitol Records     | GB663829617823
```

12.6 TABLE « SUPPLIER »

```
CREATE TABLE supplier (
    id SERIAL PRIMARY KEY,
    fk_company_id INT REFERENCES company(id),
    contact TEXT,
    phone TEXT,
    job_title TEXT
) ;
```

12.6.1 Quelques données

```
INSERT INTO supplier
VALUES
(299,194,'Johnny Ryall','597-500-569','CEO'),
(157,346,'George Clinton', '131-002-530','Sales manager')
;
```

```
SELECT * FROM supplier;
```

```
 id  | fk_company_id |    contact     |    phone    |   job_title
-----+---------------+----------------+-------------+---------------
 299 |           194 | Johnny Ryall   | 597-500-569 | CEO
 157 |           346 | George Clinton | 131-002-530 | Sales manager
```

12.7 ACTIVER L'EXTENSION

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE ;
SELECT anon.init() ;
SELECT setseed(0) ;
```

12.8 ACTIVER LE MASQUAGE DYNAMIQUE

```
SELECT anon.start_dynamic_masking();
```

12.9 RÔLE MASQUÉ

```
SECURITY LABEL FOR anon ON ROLE pierre IS 'MASKED' ;

GRANT ALL ON SCHEMA public TO jack ;
GRANT ALL ON ALL TABLES IN SCHEMA public TO jack ;
GRANT SELECT ON supplier TO pierre ;
```

Le rôle **pierre** devient « masqué », dans le sens où le super-utilisateur va pouvoir lui imposer un masque qui va changer sa vision des données.

En tant que Pierre, on essaie de lire la table des fournisseurs :

```
\c boutique pierre
SELECT * FROM supplier;
```

```
 id  | fk_company_id |    contact     |    phone    |   job_title
-----+---------------+----------------+-------------+---------------
 299 |           194 | Johnny Ryall   | 597-500-569 | CEO
 157 |           346 | George Clinton | 131-002-530 | Sales manager
```

Pour le moment, il n'y a pas de règle de masquage : Pierre peut voir les données originales dans chaque table.

12.10 MASQUER LE NOM DES FOURNISSEURS

En tant que Paul, une règle de masquage se définit ainsi :

```
\c boutique paul
SECURITY LABEL FOR anon ON COLUMN supplier.contact
IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';
```

Pierre essaie de lire la table des fournisseurs :

```
\c boutique pierre
SELECT * FROM supplier ;
```

```
 id  | fk_company_id |   contact    |    phone    |   job_title
-----+---------------+--------------+-------------+---------------
 299 |           194 | CONFIDENTIAL | 597-500-569 | CEO
 157 |           346 | CONFIDENTIAL | 131-002-530 | Sales manager
```

Si Jack essaie de lire les vraies données, ce sont encore les bonnes :

```
\c boutique jack
SELECT * FROM supplier;
```

```
 id  | fk_company_id |    contact     |    phone    |   job_title
-----+---------------+----------------+-------------+---------------
 299 |           194 | Johnny Ryall   | 597-500-569 | CEO
 157 |           346 | George Clinton | 131-002-530 | Sales manager
```

12.11 EXERCICES

12.11.1 E201 - Deviner qui est le PDG de « Johnny's Shoe Store »

Masquer le nom du fournisseur n'est pas suffisant pour anonymiser les données.

> Se connecter en tant que Pierre. Écrire une requête simple permettant de recouper certains fournisseurs en se basant sur leur poste et leur société.

Les noms des sociétés et les postes de travail sont disponibles dans de nombreux jeux de données publics. Une simple recherche sur LinkedIn ou Google révèle les noms des principaux dirigeants de la plupart des sociétés…

On nomme « recoupement » la possibilité de rapprocher plusieurs données concernant la même personne.

12.11.2 E202 - Anonymiser les sociétés

Nous devons donc anonymiser aussi la table `company`. Même si elle ne contient pas d'informations personnelles, certains champs peuvent être utilisés pour identifier certains de leurs employés…

> Écrire deux règles de masquage pour la table `company`. La première doit remplacer le champ `nom` avec un faux nom. La seconde remplacer `vat_id` avec une suite aléatoire de dix caractères. NB : dans la documentation[a], consulter les générateurs de données factices[b] et fonctions aléatoires[c] (*faking functions*).
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/
> [b] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions#faking
> [c] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions#randomization

> Vérifier que Pierre ne peut pas voir les vraies données sur la société.

12.11.3 E203 - Pseudonymiser le nom des sociétés

À cause du masquage dynamique, les valeurs artificielles sont différentes **à chaque fois que Pierre lit la table**. Ce n'est pas toujours très pratique.

Pierre préfère appliquer tout le temps les mêmes valeurs artificielles pour une même société. Cela correspond à la « pseudonymisation ».

La **pseudonymisation** consiste à générer systématiquement les mêmes données artificielles pour un individu donné à la place de ses données réelles.

> Écrire une nouvelle règle de masquage à partir du champ `name`, grâce à une fonction de pseudonymisation[a].
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions#pseudonymization

12.12 SOLUTIONS

12.12.1 S201

```
\c boutique pierre
SELECT s.id, s.contact, s.job_title, c.name
FROM supplier s
JOIN company c ON s.fk_company_id = c.id ;
```

```
 id  |   contact    |   job_title   |        name
-----+--------------+---------------+---------------------
 299 | CONFIDENTIAL | CEO           | Johnny's Shoe Store
 157 | CONFIDENTIAL | Sales manager | Capitol Records
```

12.12.2 S202

```
\c boutique paul
SECURITY LABEL FOR anon ON COLUMN company.name
IS 'MASKED WITH FUNCTION anon.fake_company()';

SECURITY LABEL FOR anon ON COLUMN company.vat_id
IS 'MASKED WITH FUNCTION anon.random_string(10)';
```

En tant Pierre, relire la table :

```
\c boutique pierre
SELECT * FROM company;
```

```
 id  |            name              |   vat_id
-----+------------------------------+------------
 952 | Graham, Davis and Bauer      | LYFVSI3WT5
 194 | Martinez-Smith               | 9N62K8M6JD
 346 | James, Rodriguez and Nelson  | OHB20OZ4Q3
```

À chaque lecture de la table, Pierre voit des données différentes :

```
SELECT * FROM company;
```

```
 id  |            name              |   vat_id
-----+------------------------------+------------
 952 | Holt, Moreno and Richardson  | KPAJP2Q4PK
 194 | Castillo Group               | NVGHZ1K50Z
 346 | Mccarthy-Davis               | GS3AHXBQTK
```

12.12.3 S203

```
\c boutique paul
ALTER FUNCTION anon.pseudo_company SECURITY DEFINER;

SECURITY LABEL FOR  anon ON COLUMN company.name
IS 'MASKED WITH FUNCTION anon.pseudo_company(id)';
```

Pour Pierre, les valeurs pseudonymisées resteront identiques entre deux appels (mais pas le code TVA) :

```
\c boutique pierre
SELECT * FROM company;

 id  |      name       |   vat_id
-----+-----------------+------------
 952 | Wilkinson LLC   | IKL88GJVT4
 194 | Johnson PLC     | VOOJ6UKR6H
 346 | Young-Carpenter | DUR78F15VD

SELECT * FROM company;

 id  |      name       |   vat_id
-----+-----------------+------------
 952 | Wilkinson LLC   | DIUAMTI653
 194 | Johnson PLC     | UND2DQGL4S
 346 | Young-Carpenter | X6EOT023AK
```

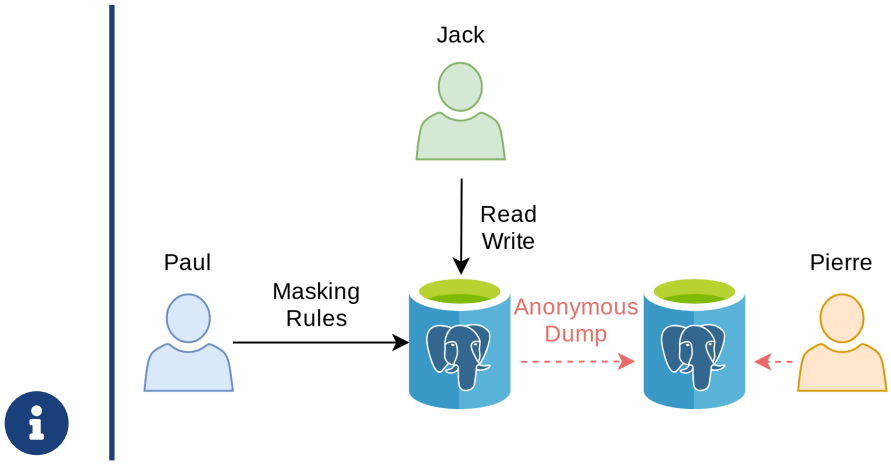
13/ Sauvegardes anonymes avec postgresql_anonymizer

Dans beaucoup de situations, le besoin est simplement d'exporter les données anonymisées pour les importer dans une autre base de données, pour mener des tests ou produire des statistiques. C'est ce que permet de faire l'outil **pg_dump_anon**.

13.1 L'HISTOIRE

- Paul a un site web qui dispose d'une section commentaires où les utilisateurs peuvent partager leurs points de vue.
- Paul a engagé un prestataire pour développer le nouveau design de son site web.
- Le prestataire lui demande un export de la base de données.
- Paul veut « nettoyer » le dump et y retirer toute information personnelle qui pourrait figurer dans la section commentaire.

13.2 COMMENT ÇA MARCHE ?



13.3 OBJECTIFS

- Extraire les données anonymisées de la base de données
- Écrire une fonction de masquage personnalisée pour gérer une colonne de type JSON

13.4 TABLE « WEBSITE_COMMENT »

```
\c boutique paul
DROP TABLE IF EXISTS website_comment CASCADE ;

CREATE TABLE website_comment (
  id SERIAL PRIMARY KEY,
  message jsonb
) ;
```

13.4.1 Quelques données

```
curl -Ls https://dali.bo/website_comment -o /tmp/website_comment.tsv
head /tmp/website_comment.tsv
```

```
1   {"meta": {"name": "Lee Perry", "ip_addr": "40.87.29.113"},
 ↪  "content": "Hello Nasty!"}
2   {"meta": {"name": "", "email": "biz@bizmarkie.com"}, "content":
 ↪  "Great Shop"}
3   {"meta": {"name": "Jimmy"}, "content": "Hi! This is me, Jimmy
 ↪  James "}
```

```
\c boutique paul
\copy website_comment from '/tmp/website_comment.tsv'

SELECT jsonb_pretty(message)
FROM website_comment
ORDER BY id ASC
LIMIT 1 ;
```

```
           jsonb_pretty
-----------------------------------
 {                                 +
     "meta": {                     +
         "name": "Lee Perry",      +
         "ip_addr": "40.87.29.113"+
     },                            +
     "content": "Hello Nasty!"     +
 }
```

13.5 ACTIVER L'EXTENSION

```
\c boutique paul

CREATE EXTENSION IF NOT EXISTS anon CASCADE;

SELECT anon.init();

SELECT setseed(0);
```

13.6 MASQUER UNE COLONNE DE TYPE JSON

Généralement, les données non structurées sont difficiles à masquer…

```
SELECT message - ARRAY['content']
FROM website_comment
WHERE id=1 ;
```

La colonne `comment` contient beaucoup d'informations personnelles. Le fait de ne pas utiliser un schéma standard pour les commentaires rend ici la tâche plus complexe.

Comme on peut le voir, les visiteurs du site peuvent écrire toutes sortes d'informations dans la section « commentaire ». La meilleure option serait donc de supprimer entièrement la clé JSON car il est impossible d'y exclure les données sensibles.

Il est possible de nettoyer la colonne `comment` en supprimant la clé `content` :

```
SELECT message - ARRAY['content']
FROM website_comment
WHERE id=1 ;
```

13.6.1 Fonctions de masquage personnalisées

```
\c boutique paul

CREATE SCHEMA IF NOT EXISTS my_masks;

SECURITY LABEL FOR anon ON SCHEMA my_masks IS 'TRUSTED';

CREATE OR REPLACE FUNCTION my_masks.remove_content(j jsonb)
RETURNS jsonb
AS $func$
  SELECT j - ARRAY['content']
$func$
LANGUAGE sql ;
```

- Super-utilisateurs seulement !

Créer en premier lieu un schéma dédié, ici **my_masks**, et le déclarer en `trusted` (« de confiance »). Cela signifie que l'extension anon va considérer les fonctions de ce schéma comme des fonctions de masquage valides.

Seul un super-utilisateur devrait être capable d'ajouter des fonctions dans ce schéma !

Cette fonction de masquage se contente de supprimer du JSON le champ avec le message :

```
CREATE OR REPLACE FUNCTION my_masks.remove_content(j jsonb)
RETURNS jsonb
AS $func$
  SELECT j - ARRAY['content']
$func$
LANGUAGE sql ;
```

Exécuter la fonction :

```
SELECT my_masks.remove_content(message)
FROM website_comment ;
```

```
 {"meta": {"name": "Lee Perry", "ip_addr": "40.87.29.113"}}
 {"meta": {"name": "", "email": "biz@bizmarkie.com"}}
 {"meta": {"name": "Jimmy"}}
```

La fonction va pouvoir ensuite être utilisée dans une règle de masquage.

13.6.2 Utilisation de la fonction de masquage personnalisée

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.remove_content(message)';
```

13.6.3 Sauvegarde anonymisée

- Export

```
pg_dump_anon.sh -U paul -d boutique --table=website_comment >
↪  /tmp/dump.sql
```

- Limitations : risque d'inconsistence, format plain

Enfin, une **sauvegarde logique anonymisée** de la table peut être exportée avec l'utilitaire pg_dump_anon. Celui-ci est un script, livré avec l'extension, disponible dans le répertoire des binaires :

L'outil utilise pg_dump, il vaut mieux qu'il n'y ait pas d'ambiguïté sur le chemin :

```
export PATH=$PATH:$(pg_config --bindir)
pg_dump_anon.sh --help
```

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
export PGUSER=paul
pg_dump_anon.sh boutique --table=website_comment > /tmp/dump.sql
```

En ne demandant que les données (option -a), le résultat contient notamment :

```
COPY public.website_comment (id, message) FROM stdin;
1       {"meta": {"name": "Lee Perry", "ip_addr": "40.87.29.113"}, "content": "Hello
 ↪  Nasty!"}
2       {"meta": {"name": "", "email": "biz@bizmarkie.com"}, "content": "Great Shop"}
3       {"meta": {"name": "Jimmy"}, "content": "Hi ! This is me, Jimmy James "}
\.
```

pg_dump_anon ne vise pas à réimplémenter toutes les fonctionnalités de pg_dump. Il n'en supporte qu'une partie, notamment l'extraction d'objets précis, mais pas la compression par exemple. De plus, en raison de son fonctionnement interne, il y a un risque que la restauration des données soit incohérente, notamment en cas de DML ou DDL pendant la sauvegarde. Une sauvegarde totalement cohérente impose de passer un masquage statique et un export classique par pg_dump.

Le produit est en développement actif et la situation peut avoir changé depuis que ces lignes ont été écrites. Il est notamment prévu que le script bash soit remplacé par un script en go.

13.7 EXERCICES

13.7.1 E301 - Exporter les données anonymisées dans une nouvelle base de données

> Créer une base de données nommée **boutique_anon**. y insérer les données anonymisées provenant de la base de données **boutique**.

13.7.2 E302 - Pseudonymiser les métadonnées du commentaire

Pierre compte extraire des informations générales depuis les métadonnées. Par exemple, il souhaite calculer le nombre de visiteurs uniques sur la base des adresses IP des visiteurs, mais une adresse IP est un **identifiant indirect**.

Paul doit donc anonymiser cette colonne tout en conservant la possibilité de faire le lien entre les enregistrements provenant de la même adresse.

> Remplacer la fonction `remove_content` par la fonction `clean_comment` (ci-dessous), qui : - supprime la clé JSON `content` ; - remplace la valeur dans la colonne `name` par un faux nom ; - remplace l'adresse IP dans la colonne `ip_address` par sa somme de contrôle `md5` ; - met à `NULL` la clé `email`.

```
CREATE OR REPLACE FUNCTION my_masks.clean_comment(message jsonb)
RETURNS jsonb
VOLATILE
LANGUAGE SQL
AS $func$
SELECT
  jsonb_set(
    message,
    ARRAY['meta'],
    jsonb_build_object(
        'name',anon.fake_last_name(),
        'ip_address', md5((message->'meta'->'ip_addr')::TEXT),
        'email', NULL
    )
  ) - ARRAY['content'];
$func$;
```

13.8 SOLUTIONS

13.8.1 S301

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
export PGUSER=paul
dropdb --if-exists boutique_anon
createdb boutique_anon --owner paul
pg_dump_anon.sh boutique | psql --quiet boutique_anon
```

```
export PGHOST=localhost
export PGUSER=paul
psql boutique_anon -c 'SELECT COUNT(*) FROM company'
```

```
 count
-------
     3
```

13.8.2 S302

Suite à utilisation de la fonction personnalisée `clean_comment`, les données n'ont plus rien à voir :

```
SELECT my_masks.clean_comment(message)
FROM website_comment;
```

```
                                          clean_comment
-------------------------------------------------------------------------------------
 ↪  -----------
 {"meta": {"name": "Heller", "email": null, "ip_address":
  ↪  "1d8cbcdef988d55982af1536922ddcd1"}}
 {"meta": {"name": "Christiansen", "email": null, "ip_address": null}}
 {"meta": {"name": "Frami", "email": null, "ip_address": null}}
(3 lignes)
```

On applique le masquage comme à l'habitude :

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.clean_comment(message)';
```

14/ Généralisation avec postgresql_anonymizer

14.1 PRINCIPE

- Flouter les données
- Par ex : 25 juillet 1989 ⇒ années 1980

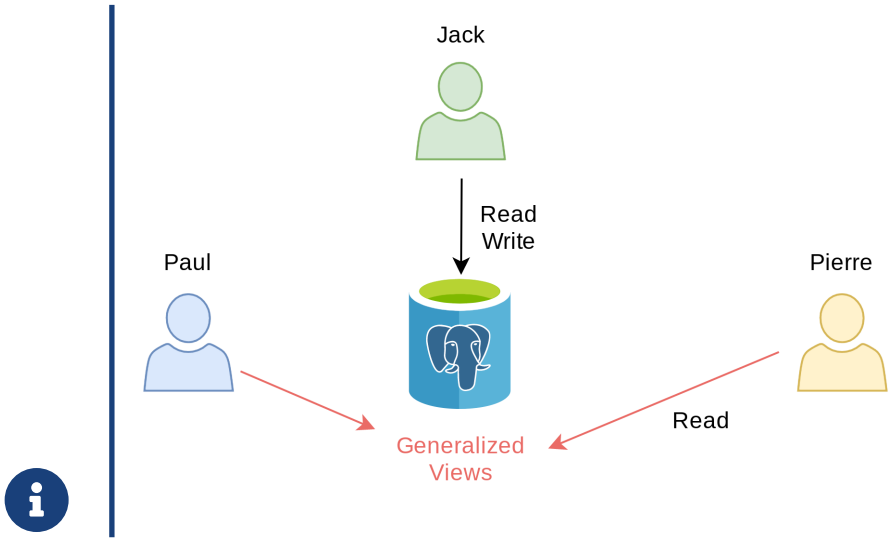
L'idée derrière la **généralisation** est de pouvoir flouter une donnée originale.

Par exemple, au lieu de dire « Monsieur X est né le 25 juillet 1989 », on peut dire « Monsieur X est né dans les années 1980 ». L'information reste vraie, bien que moins précise, et elle rend plus difficile l'identification de la personne.

14.2 L'HISTOIRE

- Paul a embauché des dizaines de salariés au fil du temps.
- Il conserve une trace sur la couleur de leurs cheveux, leurs tailles, et leurs conditions médicales.
- Paul souhaite extraire des statistiques depuis ces détails.
- Il fournit des vues **généralisées** à Pierre.

14.3 COMMENT ÇA MARCHE ?



14.4 OBJECTIFS

Nous allons voir :

- la différence entre le **masquage** et la **généralisation**
- le concept de « *k*-anonymat »

14.5 TABLE « EMPLOYEE »

```
DROP TABLE IF EXISTS employee CASCADE;
CREATE TABLE employee (
  id INT PRIMARY KEY,
  full_name  TEXT,
  first_day DATE, last_day DATE,
  height INT,
  hair TEXT, eyes TEXT, size TEXT,
  asthma BOOLEAN,
  CHECK(hair = ANY(ARRAY['bald','blond','dark','red'])),
  CHECK(eyes = ANY(ARRAY['blue','green','brown'])) ,
  CHECK(size = ANY(ARRAY['S','M','L','XL','XXL']))
);
```

- Légal ?

Bien sûr, stocker les caractéristiques physiques d'employés est généralement illégal. Quoi qu'il en soit, il sera impératif de les masquer.

14.6 QUELQUES DONNÉES

```
curl -Ls https://dali.bo/employee -o /tmp/employee.tsv

\c boutique paul
\COPY employee FROM '/tmp/employee.tsv'
```

Ce fichier charge 16 lignes, dont :

```
SELECT full_name,first_day, hair, size, asthma
FROM employee
LIMIT 3 ;
```

```
  full_name   | first_day  | hair  | size | asthma
--------------+------------+-------+------+--------
 Luna Dickens | 2018-07-22 | blond | L    | t
 Paul Wolf    | 2020-01-15 | bald  | M    | f
 Rowan Hoeger | 2018-12-01 | dark  | XXL  | t
```

14.7 SUPPRESSION DE DONNÉES

Pierre peut trouver un lien entre asthme et yeux verts :

```
\c boutique paul
DROP MATERIALIZED VIEW IF EXISTS v_asthma_eyes ;

CREATE MATERIALIZED VIEW v_asthma_eyes AS
SELECT eyes, asthma
FROM employee ;
```

Paul souhaite savoir s'il y a une corrélation entre l'asthme et la couleur des yeux.

Il fournit à Pierre la vue ci-dessus, qui peut désormais écrire des requêtes sur cette vue :

```
SELECT *
FROM v_asthma_eyes
LIMIT 3;
```

```
 eyes  | asthma
-------+--------
 blue  | t
 brown | f
 blue  | t
```

```
SELECT
  eyes,
  100*COUNT(1) FILTER (WHERE asthma) / COUNT(1) AS asthma_rate
FROM v_asthma_eyes
GROUP BY eyes ;
```

```
 eyes  | asthma_rate
-------+-------------
 green |         100
 brown |          37
 blue  |          33
```

Paul vient de prouver que l'asthme est favorisé par les yeux verts, et surtout de trouver une corrélation entre deux champs.

14.8 CALCULER LE K-ANONYMAT

- Les colonnes `asthma` et `eyes` sont considérés comme des identifiants indirects.

```
\c boutique paul
SECURITY LABEL FOR anon ON COLUMN v_asthma_eyes.eyes
IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR anon ON COLUMN v_asthma_eyes.asthma
IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_asthma_eyes');
```

```
SELECT anon.k_anonymity('v_asthma_eyes');
```

```
 k_anonymity
-------------
           2
```

La vue `v_asthma_eyes` a le niveau « 2-anonymity ». Cela signifie que chaque combinaison de quasi-identifiants (`eyes-asthma`) apparaît au moins 2 fois dans le jeu de données.

En d'autres termes, cela veut dire qu'un individu ne peut pas être distingué d'au moins un autre individu (k-1) dans cette vue.

Pour les détails sur le K-anonymat, voir cet article sur Wikipédia[1].

[1] https://en.wikipedia.org/wiki/K-anonymity

14.9 FONCTIONS D'INTERVALLE ET DE GÉNÉRALISATION

```
\c boutique paul
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_month ;

CREATE MATERIALIZED VIEW v_staff_per_month AS
SELECT
    anon.generalize_daterange(first_day,'month') AS first_day,
    anon.generalize_daterange(last_day, 'month') AS last_day
FROM employee ;

GRANT SELECT ON v_staff_per_month TO pierre ;
```

```
\c boutique pierre
SELECT *
FROM v_staff_per_month
LIMIT 3;
```

```
        first_day         |         last_day
--------------------------+--------------------------
 [2018-07-01,2018-08-01) | [2018-12-01,2019-01-01)
 [2020-01-01,2020-02-01) | (,)
 [2018-12-01,2019-01-01) | [2018-12-01,2019-01-01)
```

Pierre peut écrire une requête pour trouver le nombre d'employés embauchés en novembre 2021 :

```
SELECT COUNT(1)
       FILTER (
            WHERE make_date(2019,11,1)
            BETWEEN lower(first_day)
            AND COALESCE(upper(last_day),now())
       )
FROM v_staff_per_month ;
```

```
 count
-------
     4
```

14.9.1 Déclarer les identifiants indirects

Calculer le facteur de *k*-anonymat de cette vue :

```
\c boutique paul
SECURITY LABEL FOR anon ON COLUMN v_staff_per_month.first_day
IS 'INDIRECT IDENTIFIER';
SECURITY LABEL FOR anon ON COLUMN v_staff_per_month.last_day
IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_month');
```

Dans ce cas, le résultat est 1, ce qui veut dire qu'au moins une personne peut être directement identifiée par les dates de ses premier et dernier jour en poste.

Dans ce cas, la généralisation est insuffisante.

14.10 EXERCICES

14.10.1 E401 - Simplifier la vue v_staff_per_month pour en réduire la granularité.

Généraliser les dates en mois n'est pas suffisant. > Écrire une autre vue > v_staff_per_year qui va généraliser les dates en années. > Simplifier également la vue en utilisant un intervalle de `int` pour stocker > l'année, plutôt qu'un intervalle de `date`.

14.10.2 E402 - Progression du personnel au fil des années

> Combien de personnes ont travaillé pour Paul chaque année entre 2018 et 2021 ?

14.10.3 E403 - Atteindre le facteur *2-anonymity* sur la vue v_staff_per_year

> Quel est le facteur *k*-anonymat de la vue v_staff_per_year ?

14.11 SOLUTIONS

14.11.1 S401

Cette vue généralise les dates en années :

```
\c boutique paul
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_year;
CREATE MATERIALIZED VIEW v_staff_per_year AS
SELECT
  int4range(
    extract(year from first_day)::INT,
    extract(year from last_day)::INT,
    '[]' -- include upper bound
  ) AS period
FROM employee;

SELECT *
FROM v_staff_per_year
LIMIT 3;

   period
-------------
 [2018,2019)
 [2020,)
 [2018,2019)
```

14.11.2 S402

Les personnes ayant travaillé pour Paul entre 2018 et 2021 sont :

```
SELECT
  year,
  COUNT(1) FILTER (
      WHERE year <@ period
  )
FROM
    generate_series(2018,2021) year,
    v_staff_per_year
GROUP BY year
ORDER BY year ASC;

 year | count
------+-------
 2018 |     4
 2019 |     6
 2020 |     9
 2021 |    10
```

14.11.3 S403

Le k-anonymat de cette vue est meilleur :

```
SECURITY LABEL FOR anon ON COLUMN v_staff_per_year.period
IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_year');

 k_anonymity
-------------
           2
```

15/ Conclusion sur postgresql_anonymizer

15.1 BEAUCOUP DE STRATÉGIES DE MASQUAGE

- Masquage statique[1] : parfait pour une anonymisation « une fois pour toute »
- Masquage dynamique[2] : utile pour masquer des informations à certains utilisateurs
- Sauvegardes anonymisées[3] : peuvent être utilisées dans des traitement CI/CD
- Généralisation[4] : adaptée aux statistiques et à l'analyse de données

15.2 BEAUCOUP DE FONCTIONS DE MASQUAGE

- Destruction partielle ou totale
- Ajout de bruit
- Randomisation
- Falsification et falsification avancée
- Pseudonymisation
- Hachage générique
- Masquage personnalisé

RTFM -> Fonctions de masquage[5]

15.3 AVANTAGES

- Règles de masquage en SQL
- Règles de masquage stockées dans le schéma de la base
- Pas besoin d'un ETL
- Fonctionne avec toutes les versions actuelles de PostgreSQL
- Multiples stratégies, multiples fonctions.

15.4 INCONVÉNIENTS

- Ne fonctionne pas avec d'autres systèmes de gestion de bases de données (comme le nom l'indique)
- Peu de retour d'expérience sur de gros volumes (> 10 To)

15.5 POUR ALLER PLUS LOIN

D'autres projets qui pourraient vous plaire :

- pg_sample[6] : Extraire un petit jeu de données d'une base de données volumineuse
- PostgreSQL Faker[7] : Une extension de falsification avancée basée sur la bibliothèque python `Faker`.

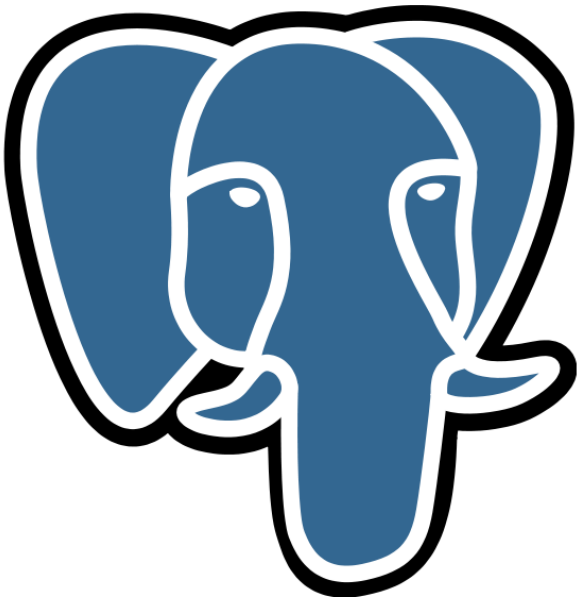
15.6 CONTRIBUEZ !

C'est un projet libre !
labs.dalibo.com/postgresql_anonymizer[8]
Merci de vos retours sur la manière dont vous l'utilisez, comment il répond ou non à vos attentes, etc.

15.6.1 Questions

N'hésitez pas, c'est le moment !

16/ Pooling



16.1 AU MENU

- Concepts
- Pool de connexion avec PgBouncer

Ce module permet d'aborder le *pooling*.

Ce qui suit ne portera que sur un unique serveur, et n'aborde pas le sujet de la répartition de charge.

Nous étudierons principalement un logiciel : PgBouncer.

16.1.1 Objectifs

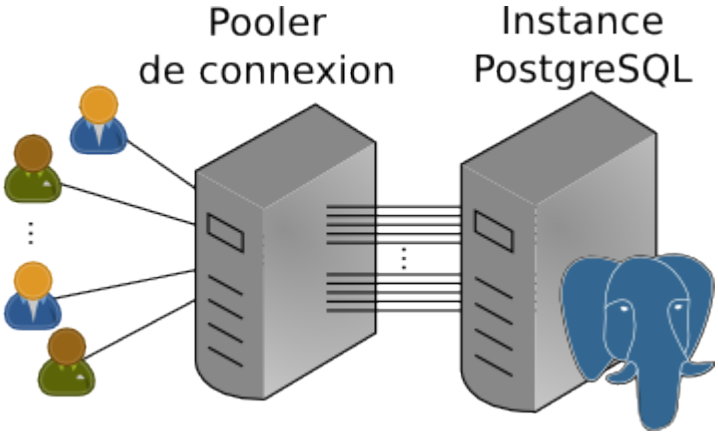
- Savoir ce qu'est un pool de connexion ?
- Avantage, inconvénients & limites
- Savoir mettre en place un pooler de connexion avec PgBouncer

16.2 POOL DE CONNEXION

- Qu'est ce qu'un pool de connexion ?
- Présentation
- Avantages et inconvénients
- Mise en œuvre avec PgBouncer

Dans cette partie, nous allons étudier la théorie des poolers de connexion. La partie suivante sera la mise en pratique avec l'outil PgBouncer.

16.2.1 Serveur de pool de connexions



16.2.2 Serveur de pool de connexions

- S'intercale entre le SGBD et les clients
- Maintient des connexions ouvertes avec le SGBD
- Distribue aux clients ses connexions au SGBD
- Attribue une connexion existante au SGBD dans ces conditions
  - même rôle
  - même base de donnée
- Différents poolers :
  - intégrés aux applicatifs
  - service séparé (où ?)

Un serveur de pool de connexions s'intercale entre les clients et le système de gestion de bases de données. Les clients ne se connectent plus directement sur le SGBD pour accéder aux bases. Ils passent par le pooler qui se fait passer pour le serveur de bases de données. Le pooler maintient alors des connexions vers le SGBD et en gère lui-même l'attribution aux utilisateurs.

Chaque connexion au SGBD est définie par deux paramètres : le rôle de connexion et la base de donnée. Ainsi, une connexion maintenue par le pooler ne sera attribuée à un utilisateur que si ce couple rôle/base de donnée est le même.

Les conditions de création de connexions au SGBD sont donc définies dans la configuration du pooler.

Un pooler peut se présenter sous différentes formes :

- comme **brique logicielle** incorporée dans le code applicatif sur les serveurs d'applications (fourni par Hibernate ou Apache Tomcat, par exemple) ;
- comme **service** séparé, démarré sur un serveur et écoutant sur un port donné, où les clients se connecteront pour accéder à la base de donnée voulue (exemples : PgBouncer, pgPool)

Nous nous consacrons dans ce module aux pools de connexions accessibles à travers un service.

Noter qu'il ne faut pas confondre un pooler avec un outil de répartition de charge (même si un pooler peut également permettre la répartition de charge, comme PgPool).

L'emplacement d'un pooler se décide au cas par cas selon l'architecture. Il peut se trouver intégré à l'application, et lui être dédié, ce qui garantit une latence faible entre pooler et application. Il peut être centralisé sur le serveur de bases de données et servir plusieurs applications, voire se trouver sur une troisième machine. Il faut aussi réfléchir à ce qui se passera en cas de bascule entre deux instances.

16.2.3 Intérêts du pool de connexions

- Évite le coût de connexion
  - …et de déconnexion
- Optimise l'utilisation des ressources du SGBD
- Contrôle les connexions, peut les rediriger
- Évite des déconnexions
  - redémarrage (mise à jour, bascule)
  - saturation temporaires des connexions sur l'instance

Le maintien des connexions entre le pooler et le SGBD apporte un gain non négligeable lors de l'établissement des connexions. Effectivement, pour chaque nouvelle connexion à PostgreSQL, nous avons :

- la création d'un nouveau processus ;
- l'allocation des ressources mémoires utiles à la session ;
- le positionnement des paramètres de session de l'utilisateur.

Tout ceci engendre une consommation processeur.

Ce travail peut durer plusieurs dizaines, voire centaines de millisecondes. Cette latence induite peut alors devenir un réel goulot d'étranglement dans certains contextes. Or, une connexion déjà active maintenue dans un pool peut être attribuée à une nouvelle session immédiatement : cette latence est donc *de facto* fortement limitée par le pooler.

En fonction du mode de fonctionnement, de la configuration et du type de pooler choisi, sa transparence vis-à-vis de l'application et son impact sur les performances seront différents.

De plus, cette position privilégiée entre les utilisateurs et le SGBD permet au pooler de contrôler et centraliser les connexions vers le ou les SGBD. Effectivement, les applications pointant sur le serveur de pool de connexions, le SGBD peut être situé n'importe où, voire sur plusieurs serveurs différents. Le pooler peut aiguiller les connexions vers un serveur différent en fonction de la base de données demandée. Certains poolers peuvent détecter une panne d'un serveur et aiguiller vers un autre. En cas de *switchover*, *failover*, évolution ou déplacement du SGBD, il peut suffire de reconfigurer le pooler.

Enfin, les sessions entrantes peuvent être mises en attente si plus aucune connexion n'est disponible et qu'elles ne peuvent pas en créer de nouvelle. On évite donc de lever immédiatement une erreur, ce qui est le comportement par défaut de PostgreSQL.

Pour la base de données, le pooler est une application comme une autre.

Si la configuration le permet (`pg_hba.conf`), il est possible de se connecter à une instance aussi bien via le pooler que directement selon l'utilisation (application, batch, administration…)

16.2.4 Inconvénients du pool de connexions

- Transparence suivant le mode :
  - par sessions
  - par transactions
  - par requêtes
- Performances, si mal configuré (latence)
- Point délicat : l'authentification !
- Complexité
- SPOF potentiel
- Impact sur les fonctionnalités, selon le mode

Les fonctionnalités de PostgreSQL utilisables au travers d'un pooler varient suivant son mode de fonctionnement du pooler (par requêtes, transactions ou sessions). Nous verrons que plus la mutualisation est importante, plus les restrictions apparaissent.

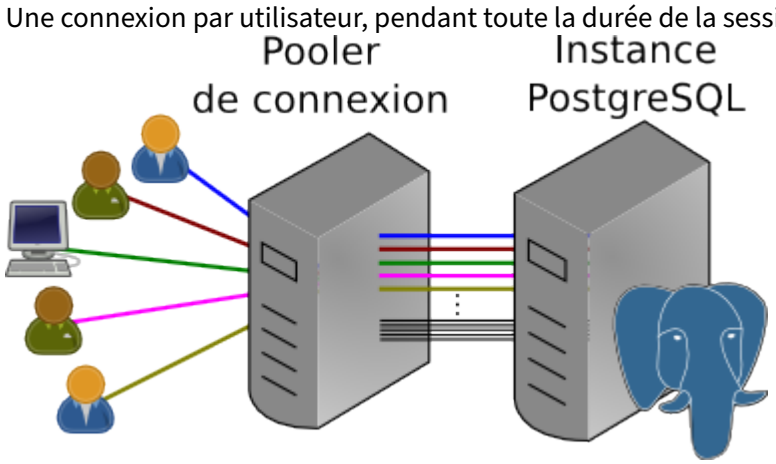
Un pooler est un élément en plus entre l'application et vos données, donc il aura un coût en performances. Il ajoute notamment une certaine latence. On n'introduit donc pas un pooler sans avoir identifié un problème. Si la configuration est bien faite, cet impact est normalement négligeable, ou en tout cas sera compensé par des gains au niveau de la base de données, ou en administration.

Comme dans tout système de proxy, un des points délicats de la configuration est l'authentification, avec certaines restrictions.

Un pooler est un élément en plus dans votre architecture. Il la rend donc plus complexe et y ajoute ses propres besoins en administration, supervision et ses propres modes de défaillance. Si vous faites passer toutes vos connexions par un pooler, celui-ci devient un nouveau point de défaillance possible (SPOF). Une redondance est bien sûr possible mais complique à nouveau les choses.

16.3 POOLING DE SESSIONS

Une connexion par utilisateur, pendant toute la durée de la session.



Un pool de connexion par session attribue une connexion au SGBD à un unique utilisateur pendant toute la durée de sa session. Si aucune connexion à PostgreSQL n'est disponible, une nouvelle connexion est alors créée, dans la limite exprimée dans la configuration du pooler. Si cette limite est atteinte, la session est mise en attente ou une erreur est levée.

16.3.1 Intérêts du pooling de sessions

- Avantages :
  - limite le temps d'établissement des connexions
  - mise en attente si trop de sessions
  - simple
  - transparent pour les applications
- Inconvénients :
  - périodes de non activité des sessions conservées
  - nombre de sessions active au pooler égal au nombre de connexions actives au SGBD

L'intérêt d'un pool de connexion en mode session est principalement de conserver les connexions ouvertes vers le SGBD. On économise ainsi le temps d'établissement de la connexion pour les nou-

velles sessions entrantes si une connexion est déjà disponible. Dans ce cas, le pooler permet d'avoir un comportement de type *pre-fork* côté SGBD.

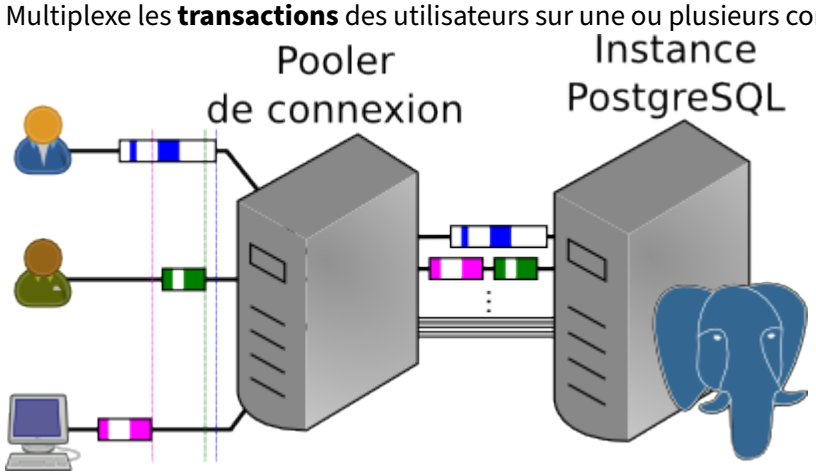
L'autre intérêt est de ne pas rejeter une connexion, même s'il n'y a plus de connexions possibles au SGBD. Contrairement au comportement de PostgreSQL, les connexions sont placées en attente si elles ne peuvent pas être satisfaites immédiatement.

Ce mode de fonctionnement est très simple et robuste, c'est le plus transparent vis-à-vis des sessions clientes, avec un impact quasi nul sur le code applicatif.

Aucune optimisation du temps de travail côté SGBD n'est donc possible. S'il peut être intéressant de limiter le nombre de sessions ouvertes sur le pooler, il sera en revanche impossible d'avoir plus de sessions ouvertes sur le pooler que de connexions disponibles sur le SGDB.

16.4 POOLING DE TRANSACTIONS

Multiplexe les **transactions** des utilisateurs sur une ou plusieurs connexions.



Dans le schéma présenté ici, chaque bloc représente une transaction délimitée par une instruction `BEGIN`, suivie plus tard d'un `COMMIT` ou d'un `ROLLBACK`. Chaque zone colorée représente une requête au sein de la transaction.

Un pool de connexions par transactions multiplexe les transactions des utilisateurs entre une ou plusieurs connexions au SGBD. Une transaction est débutée sur la première connexion à la base qui soit inactive (`idle`). Toutes les requêtes d'une transaction sont envoyées sur la même connexion.

Ce schéma suppose que le pool accorde la première connexion disponible en partant du haut dans l'ordre où les transactions se présentent.

16.4.1 Avantages & inconvénients du pooling de transactions

- Avantages
  - mêmes avantages que le pooling de sessions
  - meilleure utilisation du temps de travail des connexions
    - les connexions sont utilisées par une ou plusieurs sessions
  - plus de sessions possibles côté pooler pour moins de connexions au SGBD
- Inconvénients
  - interdit les requêtes préparées
  - période de non activité des sessions toujours possible

Les intérêts d'un pool de connexion en mode transaction sont multiples en plus de cumuler ceux d'un pool de connexion par session.

Il est désormais possible de partager une même connexion au SGBD entre plusieurs sessions utilisateurs. En effet, il existe de nombreux contextes où une session a un taux d'occupation relativement faible : requêtes très simples et exécutées très rapidement, génération des requêtes globalement plus lente que la base de données, couche applicative avec des temps de traitement des données reçues plus importants que l'exécution côté SGBD, etc.

Avoir la capacité de multiplexer les transactions de plusieurs sessions entre plusieurs connexions permet ainsi de limiter le nombre de connexions à la base en optimisant leur taux d'occupation. Cet économie de connexions côté SGBD a plusieurs avantages :

- moins de connexions à gérer par le serveur, qui est donc plus disponible pour les connexions actives ;
- moins de connexions, donc économie de mémoire, devenue disponible pour les requêtes ;
- possibilité d'avoir un plus grand nombre de clients connectés côté pooler sans pour autant atteindre un nombre critique de connexions côté SGBD.

En revanche, avec ce mode de fonctionnement, le pool de connexions n'assure pas aux client connectés que leurs requêtes et transactions iront toujours vers la même connexion, bien au contraire ! Ainsi, si l'application utilise des requêtes préparées (c'est-à-dire en trois phases `PREPARE`, `BIND`, `EXECUTE`), la commande `PREPARE` pourrait être envoyée sur une connexion alors que les commandes `EXECUTE` pourraient être dirigées vers d'autres connexions, menant leur exécution tout droit à une erreur.

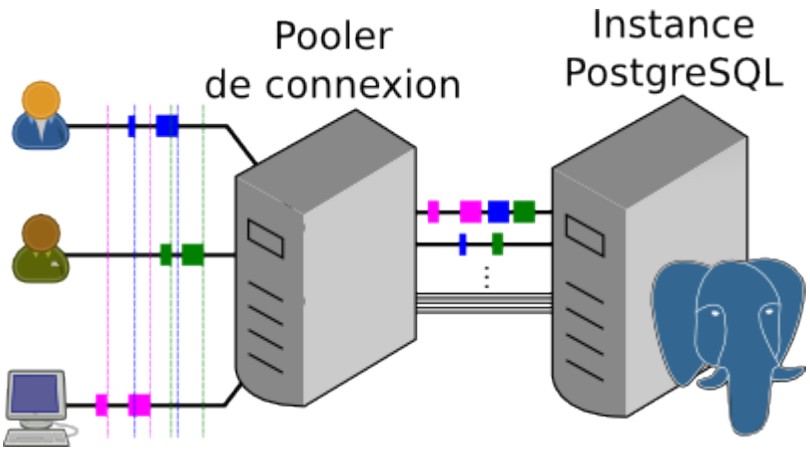
Seules les requêtes au sein d'une même transaction sont assurées d'être exécutées sur la même connexion. Ainsi, au début de cette transaction, la connexion est alors réservée exclusivement à l'utilisateur propriétaire de la transaction. Donc si le client prend son temps entre les différentes

étapes d'une transaction (statut `idle in transaction` pour PostgreSQL), il monopolisera la connexion sans que les autres clients puissent en profiter.

Ce type de pool de connexion a donc un impact non négligeable à prendre en compte lors du développement.

16.5 POOLING DE REQUÊTES

- Un pool de connexions en mode requêtes multiplexe toutes les **requêtes** sur une ou plusieurs connexions



Un pool de connexions par requêtes multiplexe les requêtes des utilisateurs entre une ou plusieurs connexions au SGBD.

Dans le schéma présenté ici, chaque bloc coloré représente une requête. Elles sont placées exactement aux mêmes instants que dans le schéma présentant le pool de connexion en mode transactions.

16.5.1 Avantages & inconvénients du pooling de requêtes

- Avantages
  - les mêmes que pour le pooling de sessions et de transactions.
  - utilisation optimale du temps de travail des connexions
  - encore plus de sessions possibles côté pooler pour moins de connexions au SGBD
- Inconvénients
  - les mêmes que pour le pooling de transactions
  - interdiction des transactions !

Les intérêts d'un pool de connexions en mode requêtes sont les mêmes que pour un pool de connexion en mode de transactions. Cependant, dans ce mode, toutes les requêtes des clients sont multiplexées à travers les différentes connexions disponibles et inactives.

Ainsi, il est désormais possible d'optimiser encore plus le temps de travail des connexions au SGBD, supprimant la possibilité de bloquer une connexion dans un état `idle in transaction`. Nous sommes alors capables de partager une même connexion avec encore plus de clients, augmentant ainsi le nombre de sessions disponibles sur le pool de connexions tout en conservant un nombre limité de connexions côté SGBD.

En revanche, si les avantages sont les mêmes que ceux d'un pooler de connexion en mode transactions, les limitations sont elles aussi plus importantes. Il n'est effectivement plus possible d'utiliser des transactions, en plus des requêtes préparées !

En pratique, le pooling par requête sert à interdire totalement les transactions. En effet, un pooling par transaction n'utilisant que des transactions implicites (d'un seul ordre) parviendra au même résultat.

16.6 POOLING AVEC PGBOUNCER

- Deux projets existent : PgBouncer et PgPool-II
- Les deux sont sous licence BSD
- PgBouncer
  - le plus évolué et éprouvé pour le *pooling*

Deux projets sous licence BSD coexistent dans l'écosystème de PostgreSQL pour mettre en œuvre un pool de connexion : PgBouncer et PgPool-II.

PgPool-II[1] est le projet le plus ancien, développé et maintenu principalement par SRA OSS[2]. Ce projet est un véritable couteau suisse capable d'effectuer bien plus que du pooling (répartition de charge, bascules…). Malheureusement, cette polyvalence a un coût important en terme de fonctionnalités et complexités. PgPool n'est effectivement capable de travailler qu'en tant que pool de connexion par session.

PgBouncer[3] est un projet créé par Skype. Il a pour objectifs :

- de n'agir qu'en tant que pool de connexion ;
- d'être le plus léger possible ;
- d'avoir les meilleures performances possibles ;
- d'avoir le plus de fonctionnalités possibles sur son cœur de métier.

PgBouncer étant le plus évolué des deux, nous allons le mettre en œuvre dans les pages suivantes.

[1] https://www.pgpool.net/
[2] https://www.sraoss.co.jp/index_en.php
[3] https://www.pgbouncer.org/

16.6.1 PgBouncer : Fonctionnalités

- Techniquement : un démon
- Disponible sous Unix & Windows
- Modes sessions / transactions / requêtes
- Redirection vers des serveurs et/ou bases différents
- Mise en attente si plus de connexions disponibles
- Mise en pause des connexions
- Paramétrage avancé des sessions clientes et des connexions aux bases
- Mise à jour sans couper les sessions existantes
- Supervision depuis une base virtuelle de maintenance
- Pas de répartition de charge

PgBouncer est techniquement assez simple : il s'agit d'un simple démon, auxquelles les applicatifs se connectent (en croyant avoir affaire à PostgreSQL), et qui retransmet requêtes et données.

PgBouncer dispose de nombreuses fonctionnalités, toutes liées au pooling de connexions. La majorité de ces fonctionnalités ne sont pas disponibles avec PgPool.

À l'inverse de ce dernier, PgBouncer n'offre pas de répartition de charge. Ses créateurs renvoient vers des outils au niveau TCP comme HAProxy. De même, pour les bascules d'un serveur à l'autre, ils conseillent plutôt de s'appuyer sur le niveau DNS.

Ce qui suit n'est qu'un extrait de la documentation de référence, assez courte : https://www.pgbouncer.org/config.html. La FAQ[4] est également à lire.

16.6.2 PgBouncer : Installation

- Par les paquets fournis par le PGDG :
  - `yum|dnf install pgbouncer`
  - `apt install pgbouncer`
- Installation par les sources
  - Dépôt pgbouncer[5]

[4] https://www.pgbouncer.org/faq.html

PgBouncer est disponible sous la forme d'un paquet binaire sur les principales distributions Linux et les dépôts du PGDG.

Il y a quelques différences mineures d'empaquetage : sous Red Hat/CentOS/Rocky Linux, le processus tourne avec un utilisateur système **pgbouncer** dédié, alors que sur Debian et dérivées, il fonctionne sous l'utilisateur **postgres**.

Il est bien sûr possible de recompiler depuis les sources.

Sous Windows, le projet fournit une archive[6] à décompresser.

16.6.3 PgBouncer : Fichier de configuration

- Format `ini`
- Un paramètre par ligne
- Aucune unité dans les valeurs
- Tous les temps sont exprimés en seconde
- Sections : `[databases]`, `[users]`, `[pgbouncer]`

Les paquets binaires créent un fichier de configuration `/etc/pgbouncer/pgbouncer.ini`.

Une ligne de configuration concerne un seul paramètre, avec le format suivant :

```
parametre = valeur
```

PgBouncer n'accepte pas que l'utilisateur spécifie une unité pour les valeurs. L'unité prise en compte par défaut est la seconde.

Il y a plusieurs sections :

- les bases de données (`[databases]`), où on spécifie pour chaque base la chaîne de connexion à utiliser ;
- les utilisateurs (`[users]`), pour des propriétés liées aux utilisateurs ;
- le moteur (`[pgbouncer]`), où se fait tout le reste de la configuration de PgBouncer.

[6] https://github.com/pgbouncer/pgbouncer/releases/

16.6.4 PgBouncer : Connexions

- TCP/IP
  - `listen_addr` : adresses
  - `listen_port` (6432)
- Socket Unix (`unix_socket_dir`, `unix_socket_mode`, `unix_socket_group`)
- Chiffrement TLS

PgBouncer accepte les connexions en mode socket Unix et via TCP/IP. Les paramètres disponibles ressemblent beaucoup à ce que PostgreSQL propose.

`listen_addr` correspond aux interfaces réseaux sur lesquels PgBouncer va écouter. Il est par défaut configuré à la boucle locale, mais vous pouvez ajouter les autres interfaces disponibles, ou tout simplement une étoile pour écouter sur toutes les interfaces. `listen_port` précise le port de connexion : traditionnellement, c'est 6432, mais on peut le changer, par exemple à 5432 pour que la configuration de connexion des clients reste identique.

Si PostgreSQL se trouve sur le même serveur et que vous voulez utiliser le port 5432 pour PgBouncer, il faudra bien sûr changer le port de connexion de PostgreSQL.

Pour une connexion uniquement en local par la socket Unix, il est possible d'indiquer où le fichier socket doit être créé (paramètre `unix_socket_dir` : `/tmp` sur Red Hat/CentOS, `/var/run/postgresql` sur Debian et dérivés), quel groupe doit lui être affecté (`unix_socket_group`) et les droits du fichier (`unix_socket_mode`). Si un groupe est indiqué, il est nécessaire que l'utilisateur détenteur du processus `pgbouncer` soit membre de ce groupe.

Cela est pris en compte par les paquets binaires d'installation.

PgBouncer supporte également le chiffrement TLS.

16.6.5 PgBouncer : Définition des accès aux bases

- Section `[databases]`
- Une ligne par base sous la forme libpq :

```
data1 = host=localhost port=5433 dbname=data1 pool_size=50
```

- Paramètres de connexion :
  - `host`, `port`, `dbname` ; `user`, `password`
  - `pool_size`, `pool_mode`, `connect_query`
  - `client_encoding`, `datestyle`, `timezone`…
- Base par défaut :

```
 + = host=ip1 port=5432 dbname=data0
```

- `auth_hba_file` : équivalent à `pg_hba.conf`

Lorsque l'utilisateur cherche à se connecter à PostgreSQL, il va indiquer l'adresse IP du serveur où est installé PgBouncer et le numéro de port où écoute PgBouncer. Il va aussi indiquer d'autres informations comme la base qu'il veut utiliser, le nom d'utilisateur pour la connexion, son mot de passe, etc.

Lorsque PgBouncer reçoit cette requête de connexion, il extrait le nom de la base et va chercher dans la section `[databases]` si cette base de données est indiquée. Si oui, il remplacera tous les paramètres de connexion qu'il trouve dans son fichier de configuration et établira la connexion entre ce client et cette base. Si jamais la base n'est pas indiquée, il cherchera s'il existe une base de connexion par défaut (nom indiqué par une étoile) et l'utilisera dans ce cas.

Exemples de chaîne de connexion :

```
prod = host=p1 port=5432 dbname=erp pool_size=40 pool_mode=transaction
prod = host=p1 port=5432 dbname=erp pool_size=10 pool_mode=session
```

Il est donc possible de faire beaucoup de chose :

- n'accéder qu'à un serveur dont les bases sont décrites ;
- accéder à différents serveurs PostgreSQL depuis un même serveur de pooling, suivant le nom de la base ou de l'utilisateur ;
- remplacer l'utilisateur de connexion par celui défini par `user` ;
- etc.

Néanmoins, les variables `user` et `password` sont très peu utilisées.

La chaîne de connexion est du type libpq mais tout ce qu'accepte la libpq n'est pas forcément accepté par PgBouncer (notamment pas de variable service, pas de possibilité d'utiliser directement le fichier standard `.pgpass`).

Le paramètre `auth_hba_file` peut pointer vers un fichier de même format que `pg_hba.conf` pour filtrer les accès au niveau du pooler (en plus des bases).

16.6.6 PgBouncer : Authentification par fichier de mots de passe

- Liste des utilisateurs contenue dans `userlist.txt`
- Contenu de ce fichier
  - `"utilisateur" "mot de passe"`
- Paramètres dans le fichier de configuration
  - `auth_type` : type d'authentification (`trust`, `md5`,`scram-sha-256`…)
  - `auth_file` : emplacement de la liste des utilisateurs et mots de passe
  - `admin_users` : liste des administrateurs
  - `stats_users` : liste des utilisateurs de supervision

PgBouncer n'a pas accès à l'authentification de PostgreSQL. De plus, son rôle est de donner accès à des connexions déjà ouvertes à des clients. PgBouncer doit donc s'authentifier auprès de PostgreSQL à la place des clients, et vérifier lui-même les mots de passe de ces clients. (Ce mécanisme ne dispense évidemment pas les clients de fournir les mots de passe.)

La première méthode, et la plus simple, est de déclarer les utilisateurs dans le fichier pointé par le paramètre `auth_file`, par défaut `userlist.txt`. Les utilisateurs et mots de passe y sont stockés comme ci-dessous selon le type d'authentification, obligatoirement encadrés avec des guillemets doubles.

```
"guillaume" "supersecret"
"marc" "md59fa7827a30a483125ca3b7218bad6fee"
"pgbench" "SCRAM-SHA-256$4096:Rqk+MWaDN9rKXOLuoj8eCw==$ry5DD2Ptk…+6do76FN/ys="
```

Le type d'authentification est plus limité que ce que PostgreSQL propose. Le type `trust` indique que l'utilisateur sera accepté par PgBouncer quel que soit le mot de passe qu'il fournit ; il faut que le serveur PostgreSQL soit configuré de la même façon. Cela est bien sûr déconseillé. `auth_type` peut prendre les valeurs `md5` ou `scram-sha-256` pour autoriser des mots de passe chiffrés. Pour des raisons de compatibilité descendante, `md5` permet aussi d'utiliser `scram-sha-256`.

Les paramètres de configuration `admin_users` et `stats_users` permettent d'indiquer la liste d'utilisateurs pouvant se connecter à PgBouncer directement pour obtenir des commandes de contrôle sur PgBouncer ainsi que des statistiques d'activité. Ils peuvent être déclarés dans le fichier des mots de passe avec un mot de passe arbitraire en clair.

`userlist.txt` est évidemment un fichier dont les accès doivent être les plus restreints possibles.

16.6.7 PgBouncer : Authentification par délégation

- Créer un rôle dédié
- Copier son hash de mot de passe (MD5 !) dans `userlist.txt`
- Déclaration dans le pool avec `auth_user` :

```
prod = host=p1 port=5432 dbname=erp auth_user=frontend
```

- `auth_query` : requête pour vérifier le mot de passe via ce rôle
- => Plus la peine de déclarer les autres rôles

La maintenance du fichier de mots de passe peut vite devenir fastidieuse. Il est possible de déléguer un rôle à la recherche des mots de passe avec le paramètre `auth_user` (à poser globalement ou au niveau de la base).

```
prod = host=p1 port=5432 dbname=erp pool_mode=transaction auth_user=frontend
```

Ce rôle se connectera et ira valider dans l'instance le hash du mot de passe du client. Il sera donc inutile de déclarer d'autres rôles dans `userlist.txt`.

Il n'y aura pas de problème avec l'authentification MD5. Par contre, le principe même de SCRAM-SHA-256 interdit de passer par un proxy. Le mot de passe de l'utilisateur dédié devra donc forcément être encodé en MD5.

Exemple de configuration :

```
SET password_encryption = 'md5' ;
CREATE ROLE frontend PASSWORD 'pass' LOGIN ;
SELECT rolpassword FROM pg_authid WHERE rolname = 'frontend' \gx
```

Le hachage obtenu (ici en MD5) est recopié dans `userlist.txt` :

```
"frontend" "md5b935ea59a93354a09864a11ff102b548"
```

Le paramètre `auth_query` définit la requête à exécuter pour ensuite comparer les résultats avec les identifiants de connexion. Par défaut, il s'agit simplement de requêter la vue `pg_shadow` :

```
auth_query = SELECT usename, passwd FROM pg_shadow WHERE usename=$1
```

D'autres variantes sont possibles, comme une requête plus élaborée sur `pg_authid`, ou une fonction avec les bons droits de consultation avec une clause `SECURITY DEFINER` (la documentation donne un exemple[7]). Il faut évidemment que l'utilisateur choisi ait les droits nécessaires, et cela dans toutes les bases impliquées. La mise en place de cette configuration est facilement source d'erreur, il faut bien surveiller les traces de PostgreSQL *et* PgBouncer.

16.6.8 PgBouncer : Nombre de connexions

- Côté client :
  - `max_client_conn` (100)
  - attention à `ulimit` !
  - `max_db_connections`
- Par utilisateur/base :
  - `default_pool_size` (20)
  - `min_pool_size` (0)
  - `reserve_pool_size` (0)

PostgreSQL dispose d'un nombre de connexions maximum (`max_connections` dans `postgresql.conf`, 100 par défaut). Il est un compromis entre le nombre de requêtes simultanément actives, leur complexité, le nombre de CPU, le nombre de processus gérables par l'OS… L'utilisation d'un pooler en multiplexage se justifie notamment quand des centaines, voire milliers, de connexions simultanées sont nécessaires, celles-ci étant inactives la plus grande partie du temps. Même avec un nombre modeste de connexions, une application se connectant et se déconnectant très souvent peut profiter d'un pooler.

Les paramètres suivants de `pgbouncer.ini` permettent de paramétrer tout cela et de poser différentes limites. Les valeurs dépendent beaucoup de l'utilisation : *pooler* unique pour une seule base, *poolers* multiples pour plusieurs bases, utilisateur applicatif unique ou pas…

Nombre de connexions côté client :

Le paramètre de configuration `max_client_conn` permet d'indiquer le nombre total maximum de connexions clientes à PgBouncer. Sa valeur par défaut est de seulement 100, comme l'équivalent sous PostgreSQL.

Un `max_client_conn` élevé permet d'accepter plus de connexions depuis les applications que n'en offrirait PostgreSQL. Si ce nombre de clients est dépassé, les applications se verront refuser les connexions. En-dessous, PgBouncer accepte les connexions, et, au pire, les met en attente. Cela peut arriver si la base PostgreSQL, saturée en connexions, refuse la connexion ; ou si PgBouncer ne peut

[7] http://www.pgbouncer.org/config.html#example

ouvrir plus de connexions à la base à cause d'une des autres limites ci-dessous. L'application subira donc une latence supplémentaire, mais évitera un refus de connexion qu'elle ne saura pas forcément bien gérer.

`max_db_connections` représente le maximum de connexions, tous utilisateurs confondus, à une base donnée, déclarée dans PgBouncer, donc du point de vue d'un client. Cela peut être modifié dans les chaînes de connexions pour arbitrer entre les différentes bases.

S'il n'y a qu'une base utile, côté serveur comme côté PgBouncer, et que tout l'applicatif passe par ce dernier, `max_db_connections` peut être proche du `max_connections`. Mais il faut laisser un peu de place aux connexions administratives, de supervision, etc.

Connexions côté serveur :

`default_pool_size` est le nombre maximum de connexions PgBouncer/PostgreSQL d'un *pool*. Un *pool* est un couple utilisateur/base de données côté PgBouncer. Il est possible de personnaliser cette valeur base par base, en ajoutant `pool_size=…` dans la chaîne de connexion. Si dans cette même chaîne il y a un paramètre `user` qui impose le nom, il n'y a plus qu'un *pool*.

S'il y a trop de demandes de connexion pour le pool, les transactions sont mises en attente. Cela peut être nécessaire pour équilibrer les ressources entre les différents utilisateurs, ou pour ne pas trop charger le serveur ; mais l'attente peut devenir intolérable pour l'application. Une « réserve » de connexions peut alors être définie avec `reserve_pool_size` : ces connexions sont utilisables dans une situation grave, c'est-à-dire si des connexions se retrouvent à attendre plus d'un certain délai, défini par `reserve_pool_timeout` secondes.

À l'inverse, pour faciliter les montées en charge rapides, `min_pool_size` définit un nombre de connexions qui seront immédiatement ouvertes dès que le pool voit sa première connexion, puis maintenues ouvertes.

Ces deux derniers paramètres peuvent aussi être globaux ou personnalisés dans les chaînes de connexion.

Descripteurs de fichiers :

PgBouncer utilise des descripteurs de fichiers pour les connexions. Le nombre de descripteurs peut être bien plus important que ce que n'autorise par défaut le système d'exploitation. Le maximum théorique est de :

```
max_client_conn + (max_pool_size * nombre de bases * nombre d'utilisateurs)
```

Le cas échéant (en pratique, au-delà de 1000 connexions au pooler), il faudra augmenter le nombre de descripteurs disponibles, sous peine d'erreurs de connexion :

```
ERROR accept() failed: Too many open files
```

Sur Debian et dérivés, un moyen simple est de rajouter cette commande dans `/etc/default/pgbouncer` :

```
ulimit -n 8192
```

Mais plus généralement, il est possible de modifier le service systemd ainsi :

```
sudo systemctl edit pgbouncer
```

ce qui revient à créer un fichier `/etc/systemd/system/pgbouncer.service.d/override.conf` contenant la nouvelle valeur :

```
[Service]
LimitNOFILE=8192
```

Puis il faut redémarrer le pooler :

```
sudo systemctl restart pgbouncer
```

et vérifier la prise en compte dans le fichier de traces de PgBouncer, nommé `pgbouncer.log` (dans `/var/log/postgresql/` sous Debian, `/var/log/pgbouncer/` sur CentOS/Red Hat) :

```
LOG kernel file descriptor limit: 8192 (hard: 8192);
    max_client_conn: 4000, max expected fd use: 6712
```

16.6.9 PgBouncer : types de connexions

- Mode de multiplexage
  - `pool_mode` (session)
- À la connexion
  - `ignore_startup_parameter = options`
  - attention à `PGOPTIONS` !
- À la déconnexion
  - `server_reset_query`
  - défaut : `DISCARD ALL`

Grâce au paramètre `pool_mode` (dans la chaîne de connexion à la base par exemple), PgBouncer accepte les différents modes de pooling :

- par **session**, pour économiser les temps de (dé)connexion : c'est le défaut ;
- par **transaction**, pour optimiser les connexions en place ;
- par **requête**, notamment si l'on peut se passer des transactions explicites (courant sur plusieurs ordres).

Les restrictions de chaque mode sont listées sur le site[8].

Lorsqu'un client se connecte, il peut utiliser des paramètres de connexion que PgBouncer ne connaît pas ou ne sait pas gérer. Si PgBouncer détecte un paramètre de connexion qu'il ne connaît pas, il rejette purement et simplement la connexion. Le paramètre `ignore_startup_parameters`

[8] https://www.pgbouncer.org/features.html

permet de changer ce comportement, d'ignorer le paramètre et de procéder à la connexion. Par exemple, une variable d'environnement PGOPTIONS interdit la connexion depuis psql, il faudra donc définir :

```
ignore_startup_parameters = options
```

ce qui malheureusement réduit à néant l'intérêt de cette variable pour modifier le comportement de PostgreSQL.

À la déconnexion du client, comme la connexion côté PostgreSQL peut être réutilisée par un autre client, il est nécessaire de réinitialiser la session : enlever la configuration de session, supprimer les tables temporaires, supprimer les curseurs, etc. Pour cela, PgBouncer exécute une liste de requêtes configurables ainsi :

```
server_reset_query = DISCARD ALL
```

Ce défaut suffira généralement. Il n'est en principe utile qu'en pooling de session, mais peut être forcé en pooling par transaction ou par requête :

```
server_reset_query_always = 1
```

16.6.10 PgBouncer : Durée de vie

- D'une tentative de connexion
  - `client_login_timeout`
  - `server_connect_timeout`
- D'une connexion
  - `server_lifetime`
  - `server_idle_timeout`
  - `client_idle_timeout`
- Pour recommencer une demande de connexion
  - `server_login_retry`
- D'une requête
  - `query_timeout = 0`

PgBouncer dispose d'un grand nombre de paramètres de durée de vie. Ils permettent d'éviter de conserver des connexions trop longues, notamment si elles sont inactives. C'est un avantage sur PostgreSQL qui ne dispose pas de ce type de paramétrage.

Les paramètres en `client_*` concernent les connexions entre le client et PgBouncer, ceux en `server_*` concernent les connexions entre PgBouncer et PostgreSQL.

Il est ainsi possible de libérer plus ou moins rapidement des connexions inutilisées, notamment s'il y a plusieurs *pools* concurrents, ou plusieurs sources de connexions à la base, ou si les pics de connexions sont irréguliers.

Il faut cependant faire attention. Par exemple, interrompre les connexions inactives avec `client_idle_timeout` peut couper brutalement la connexion à une application cliente qui ne s'y attend pas.

16.6.11 PgBouncer : Traces

- Fichier
  - `logfile`
- Événements tracés
  - `log_connections`
  - `log_disconnections`
  - `log_pooler_errors`
- Statistiques
  - `log_stats` (tous les `stats_period` s)

PgBouncer dispose de quelques options de configuration pour les traces.

Le paramètre `logfile` indique l'emplacement (par défaut `/var/log/pgbouncer` sur Red Hat/CentOS, `/var/log/postgres` sur Debian et dérivés). On peut rediriger vers `syslog`.

Ensuite, il est possible de configurer les événements tracés, notamment les connexions (avec `log_connections`) et les déconnexions (avec `log_disconnections`).

Par défaut, `log_stats` est activé : PgBouncer trace alors les statistiques sur les dernières 60 secondes (paramètres`stats_period`).

```
2020-11-30 19:10:07.839 CET [290804] LOG stats: 54 xacts/s, 380 queries/s,
   in 23993 B/s, out 10128 B/s, xact 304456 us, query 43274 us, wait 14685821 us
```

16.6.12 PgBouncer : Administration

- Pseudo-base pgbouncer :

```
sudo -iu postgres psql -h /var/run/postgresql -p 6432 -d pgbouncer
```

- Administration
  - RELOAD, PAUSE, SUSPEND, RESUME, SHUTDOWN
- Supervision
  - SHOW CONFIG|DATABASES|POOLS|CLIENTS|…
  - …|SERVERS|STATS|FDS|SOCKETS|…
  - …|ACTIVE_SOCKETS|LISTS|MEM

PgBouncer possède une pseudo-base nommée pgbouncer. Il est possible de s'y connecter avec psql ou un autre outil. Il faut pour cela se connecter avec un utilisateur autorisé (déclaration par les paramètres admin_users et stats_users). Elle permet de répondre à quelques ordres d'administration et de consulter quelques vues.

Les utilisateurs « administrateurs » ont le droit d'exécuter des instructions de contrôle, comme recharger la configuration (RELOAD), mettre le système en pause (PAUSE), supprimer la pause (RESUME), forcer une déconnexion/reconnexion dès que possible (RECONNECT, le plus propre en cas de modification de configuration), tuer toutes les sessions d'une base (KILL), arrêter PgBouncer (SHUTDOWN), etc.

Les utilisateurs statistiques peuvent récupérer des informations sur l'activité de PgBouncer : statistiques sur les bases, les pools de connexions, les clients, les serveurs, etc. avec SHOW STATS, SHOW STATS_AVERAGE, SHOW TOTALS, SHOW MEM, etc.

```
# sudo -iu postgres psql -h /var/run/postgresql -p 6432 pgbouncer
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1), serveur 1.14.0/bouncer)

pgbouncer=# SHOW help ;
NOTICE:  Console usage
DÉTAIL :
        SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|USERS|VERSION
        SHOW FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
        SHOW DNS_HOSTS|DNS_ZONES
        SHOW STATS|STATS_TOTALS|STATS_AVERAGES|TOTALS
        SET key = arg
        RELOAD
        PAUSE [<db>]
        RESUME [<db>]
        DISABLE <db>
        ENABLE <db>
```

```
        RECONNECT [<db>]
        KILL <db>
        SUSPEND
        SHUTDOWN

pgbouncer=# SHOW DATABASES \gx
-[ RECORD 1 ]-------+--------------------------------------
name                | pgbench_1000_sur_server3
host                | 192.168.74.5
port                | 13002
database            | pgbench_1000
force_user          |
pool_size           | 10
reserve_pool        | 7
pool_mode           | session
max_connections     | 0
current_connections | 17
paused              | 0
disabled            | 0
-[ RECORD 2 ]-------+--------------------------------------
…

pgbouncer=# SHOW POOLS \gx
-[ RECORD 1 ]------------------------------------
database   | pgbench_1000_sur_server3
user       | pgbench
cl_active  | 10
cl_waiting | 80
sv_active  | 10
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 835428
pool_mode  | session
-[ RECORD 2 ]------------------------------------
database   | pgbouncer
user       | pgbouncer
cl_active  | 1
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | statement

pgbouncer=# SHOW STATS \gx
-[ RECORD 1 ]-----+-------------------------
database          | pgbench_1000_sur_server3
total_xact_count  | 16444
total_query_count | 109711
total_received    | 6862181
```

```
total_sent         | 3041536
total_xact_time    | 8885633095
total_query_time   | 8873756132
total_wait_time    | 14123238083
avg_xact_count     | 103
avg_query_count    | 667
avg_recv           | 41542
avg_sent           | 17673
avg_xact_time      | 97189
avg_query_time     | 14894
avg_wait_time      | 64038262
-[ RECORD 2 ]------+-------------------------
database           | pgbouncer
total_xact_count   | 1
total_query_count  | 1
total_received     | 0
total_sent         | 0
total_xact_time    | 0
total_query_time   | 0
total_wait_time    | 0
avg_xact_count     | 0
avg_query_count    | 0
avg_recv           | 0
avg_sent           | 0
avg_xact_time      | 0
avg_query_time     | 0
avg_wait_time      | 0
```

```
pgbouncer=# SHOW MEM ;
     name      | size | used | free | memtotal
---------------+------+------+------+----------
 user_cache    |  360 |   11 |   39 |    18000
 db_cache      |  208 |    5 |   73 |    16224
 pool_cache    |  480 |    2 |   48 |    24000
 server_cache  |  560 |   17 |   33 |    28000
 client_cache  |  560 |   91 | 1509 |   896000
 iobuf_cache   | 4112 |   74 | 1526 |  6579200
```

Toutes ces informations sont utilisées notamment par la sonde Nagios check_postgres[9] pour permettre une supervision de cet outil.

L'outil d'audit pgCluu[10] peut intégrer cette base à ses rapports. Il faudra penser à ajouter la chaîne de connexion à PgBouncer, souvent `--pgbouncer-args='-p 6432'`, aux paramètres de `pgcluu_collectd`.

[9] https://github.com/bucardo/check_postgres
[10] https://pgcluu.darold.net

16.7 CONCLUSION

- Un outil pratique :
  - pour parer à certaines limites de PostgreSQL
  - pour faciliter l'administration
- Limitations généralement tolérables
- Ne jamais installer un pooler sans être certain de son apport :
  - SPOF
  - complexité

16.7.1 Questions

```
SELECT * FROM questions ;
```

16.8 TRAVAUX PRATIQUES

Créer un rôle PostgreSQL nommé **pooler** avec un mot de passe.

Pour mieux suivre les traces, activer `log_connections` et `log_disconnections`, et passer `log_min_duration_statement` à 0.

Installer PgBouncer. Configurer `/etc/pgbouncer/pgbouncer.ini` pour pouvoir se connecter à n'importe quelle base du serveur via PgBouncer (port 6432). Ajouter **pooler** dans `/etc/pgbouncer/userlist.txt`. L'authentification doit être `md5`. Ne pas oublier `pg_hba.conf`. Suivre le contenu de `/var/log/pgbouncer/pgbouncer.log`. Se connecter par l'intermédiaire du pooler sur une base locale.

Activer l'accès à la pseudo-base `pgbouncer` pour les utilisateurs **postgres** et **pooler**. Laisser la session ouverte pour suivre les connexions en cours.

16.8.1 Pooling par session

Ouvrir deux connexions sur le pooler. Combien de connexions sont-elles ouvertes côté serveur ?

16.8.2 Pooling par transaction

Passer PgBouncer en pooling par transaction. Bien vérifier qu'il n'y a plus de connexions ouvertes.

Rouvrir deux connexions via PgBouncer. Cette fois, combien de connexions sont ouvertes côté serveur ?

Successivement et à chaque fois dans une transaction, créer une table dans une des sessions ouvertes, puis dans l'autre insérer des données. Suivre le nombre de connexions ouvertes. Recommencer avec des transactions simultanées.

16.8.3 Pooling par requête

Passer le pooler en mode pooling par requête et tenter d'ouvrir une transaction.

Repasser PgBouncer en pooling par session.

16.8.4 pgbench

Créer une base nommée `bench` appartenant à **pooler**. Avec pgbench, l'initialiser avec un *scale factor* de 100.

Lancer des tests (lectures uniquement, avec `--select`) de 60 secondes avec 80 connexions : une fois sur le pooler, et une fois directement sur le serveur. Comparer les performances.

Refaire ce test en demandant d'ouvrir et fermer les connexions (`-C`), sur le serveur puis sur le pooler. Effectuer un `SHOW POOLS` pendant ce dernier test.

16.9 TRAVAUX PRATIQUES (SOLUTIONS)

> Créer un rôle PostgreSQL nommé **pooler** avec un mot de passe.

Les connexions se feront avec l'utilisateur **pooler** que nous allons créer avec le (trop évident) mot de passe « pooler » :

```
$ createuser --login --pwprompt --echo pooler
Saisir le mot de passe pour le nouveau rôle :
Le saisir de nouveau :
…
CREATE ROLE pooler PASSWORD 'md52a1394e4bcb2e9370746790c13ac33ac'
NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

(NB : le hash sera beaucoup plus complexe si le chiffrement SCRAM-SHA-256 est activé, mais cela ne change rien au principe.)

> Pour mieux suivre les traces, activer `log_connections` et `log_disconnections`, et passer `log_min_duration_statement` à 0.

PostgreSQL trace les rejets de connexion, mais, dans notre cas, il est intéressant de suivre aussi les connexions abouties.

Dans `postgresql.conf` :

```
log_connections = on
log_disconnections = on
log_min_duration_statement = 0
```

Puis on recharge la configuration :

```
sudo systemctl reload postgresql-14
```

En cas de problème, le suivi des connexions dans `/var/lib/pgsql/14/data/log` peut être très pratique.

> Installer PgBouncer. Configurer `/etc/pgbouncer/pgbouncer.ini` pour pouvoir se connecter à n'importe quelle base du serveur via PgBouncer (port 6432). Ajouter **pooler** dans `/etc/pgbouncer/userlist.txt`. L'authentification doit être `md5`. Ne pas oublier `pg_hba.conf`. Suivre le contenu de `/var/log/pgbouncer/pgbouncer.log`. Se connecter par l'intermédiaire du pooler sur une base locale.

L'installation est simple :

```
sudo dnf install pgbouncer
```

La configuration se fait dans `/etc/pgbouncer/pgbouncer.ini`.

Dans la section `[databases]` on spécifie la chaîne de connexion à l'instance, pour toute base :

```
* = host=127.0.0.1 port=5432
```

Il faut ajouter l'utilisateur au fichier `/etc/pgbouncer/userlist.txt`. La syntaxe est de la forme `"user" "hachage du mot de passe"`. La commande `createuser` l'a renvoyé ci-dessus, mais généralement il faudra aller interroger la vue `pg_shadow` ou la table `pg_authid` de l'instance PostgreSQL :

```
SELECT usename,passwd FROM pg_shadow WHERE usename = 'pooler';
```

```
 usename |              passwd
---------+-------------------------------------
 pooler  | md52a1394e4bcb2e9370746790c13ac33ac
```

Le fichier `/etc/pgbouncer/userlist.txt` contiendra donc :

```
"pooler" "md52a1394e4bcb2e9370746790c13ac33ac"
```

Il vaut mieux que seul l'utilisateur système dédié (**pgbouncer** sur Red Hat/CentOS/Rocky Linux) voit ce fichier :

```
sudo chown pgbouncer: userlist.txt
```

De plus il faut préciser dans `pgbouncer.ini` que nous fournissons des mots de passe hachés :

```
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
```

Si ce n'est pas déjà possible, il faut autoriser l'accès de **pooler** en local à l'instance PostgreSQL. Du point de vue de PostgreSQL, les connexions se feront depuis 127.0.0.1 (IP du pooler). Ajouter cette ligne dans le fichier `pg_hba.conf` et recharger la configuration de l'instance :

```
host    all             pooler          127.0.0.1/32            md5
```

```
sudo systemctl reload postgresql-14
```

Enfin, on peut démarrer le pooler :

```
sudo systemctl restart pgbouncer
```

Dans une autre session, on peut suivre les tentatives de connexion :

```
sudo tail -f /var/log/pgbouncer/pgbouncer.log
```

La connexion directement au pooler doit fonctionner :

```
psql -h 127.0.0.1 -p 6432 -U pooler -d postgres
Mot de passe pour l'utilisateur pooler :
psql (14.1)
Saisissez « help » pour l'aide.

postgres=>
```

Dans `pgbouncer.log` :

```
2020-12-02 08:42:35.917 UTC [2208] LOG C-0x152a490: postgres/pooler@127.0.0.1:55096
                                       login attempt: db=postgres user=pooler tls=no
```

Noter qu'en cas d'erreur de mot de passe, l'échec apparaîtra dans ce dernier fichier, et pas dans `postgresql.log`.

> Activer l'accès à la pseudo-base pgbouncer pour les utilisateurs **postgres** et **pooler**. Laisser la session ouverte pour suivre les connexions en cours.

```
; comma-separated list of users, who are allowed to change settings
admin_users = postgres,pooler

; comma-separated list of users who are just allowed to use SHOW command
stats_users = stats, postgres,pooler
```

```
sudo systemctl reload pgbouncer
```

```
$ psql -h 127.0.0.1 -p6432 -U pooler -d pgbouncer
Mot de passe pour l'utilisateur pooler :
psql (14.1, serveur 1.15.0/bouncer)
Saisissez « help » pour l'aide.

pgbouncer=# SHOW HELP ;
NOTICE:  Console usage
DÉTAIL :
        SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|USERS|VERSION
…
```

Si une connexion via PgBouncer est ouverte par ailleurs, on la retrouve ici :

```
pgbouncer=# SHOW POOLS \gx
-[ RECORD 1 ]---------
database   | pgbouncer
user       | pgbouncer
cl_active  | 1
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | statement
-[ RECORD 2 ]---------
database   | postgres
user       | pooler
cl_active  | 1
cl_waiting | 0
sv_active  | 1
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | session
```

16.9.1 Pooling par session

> Ouvrir deux connexions sur le pooler. Combien de connexions sont-elles ouvertes côté serveur ?

Le pooling par session est le mode par défaut de PgBouncer.

On se connecte dans 2 sessions différentes :

```
$ psql -h 127.0.0.1 -p6432 -U pooler -d postgres
psql (14.1)

postgres=>
```

```
$ psql -h 127.0.0.1 -p6432 -U pooler -d postgres
…
SELECT COUNT(*) FROM pg_stat_activity
WHERE backend_type='client backend' AND usename='pooler' ;
 count
-------
     2
```

Ici, PgBouncer a donc bien ouvert autant de connexions côté serveur que côté pooler.

16.9.2 Pooling par transaction

> Passer PgBouncer en pooling par transaction. Bien vérifier qu'il n'y a plus de connexions ouvertes.

Il faut changer le `pool_mode` dans `pgbouncer.ini`, soit globalement :

```
; When server connection is released back to pool:
;   session      - after client disconnects
;   transaction  - after transaction finishes
;   statement    - after statement finishes
pool_mode = transaction
```

soit dans la définition des connexions :

```
* = host=127.0.0.1 port=5432 pool_mode=transaction
```

En toute rigueur, il n'y a besoin que de recharger la configuration de PgBouncer, mais il y a le problème des connexions ouvertes. Dans notre cas, nous pouvons forcer une déconnexion brutale :

```
sudo systemct restart pgbouncer
```

> Rouvrir deux connexions via PgBouncer. Cette fois, combien de connexions sont ouvertes côté serveur ?

Après reconnexion de 2 sessions, la pseudo-base indique 2 connexions clientes, 1 serveur :

```
pgbouncer=# SHOW POOLS \gx
…
-[ RECORD 2 ]-----------
database   | postgres
```

```
user       | pooler
cl_active  | 2
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 1
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | transaction
```

Ce que l'on retrouve en demandant directement au serveur :

```
postgres=> SELECT COUNT(*) FROM pg_stat_activity
           WHERE backend_type='client backend' AND usename='pooler' ;
 count
-------
     1
```

> **Successivement** et à chaque fois dans une transaction, créer une table dans une des sessions ouvertes, puis dans l'autre insérer des données. Suivre le nombre de connexions ouvertes. Recommencer avec des transactions simultanées.

Dans la première connexion ouvertes :

```
BEGIN ;
CREATE TABLE log (i timestamptz) ;
COMMIT ;
```

Dans la deuxième :

```
BEGIN ;
INSERT INTO log SELECT now() ;
END ;
```

On a bien toujours une seule connexion :

```
pgbouncer=# SHOW POOLS \gx
…
-[ RECORD 2 ]-----------
database   | postgres
user       | pooler
cl_active  | 2
cl_waiting | 0
sv_active  | 0
sv_idle    | 0
sv_used    | 1
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | transaction
```

Du point de vue du serveur PostgreSQL, tout s'est passé dans la même session (même PID) :

```
… 10:01:45.448 UTC [2841] LOG:  duration: 0.025 ms  statement: BEGIN ;
… 10:01:45.450 UTC [2841] LOG:  duration: 0.631 ms  statement: CREATE TABLE log (i
 ↪  timestamptz) ;
… 10:01:45.454 UTC [2841] LOG:  duration: 4.037 ms  statement: COMMIT ;
… 10:01:49.128 UTC [2841] LOG:  duration: 0.053 ms  statement: BEGIN ;
… 10:01:49.129 UTC [2841] LOG:  duration: 0.338 ms  statement: INSERT INTO log SELECT
 ↪  now() ;
… 10:01:49.763 UTC [2841] LOG:  duration: 4.393 ms  statement: END ;
```

À présent, commençons la seconde transaction avant la fin de la première.

Session 1 :

```
BEGIN ; INSERT INTO log SELECT now() ;
```

Session 2 :

```
BEGIN ; INSERT INTO log SELECT now() ;
```

De manière transparente, une deuxième connexion au serveur a été créée :

```
pgbouncer=# show pools \gx
…
-[ RECORD 2 ]-----------
database   | postgres
user       | pooler
cl_active  | 2
cl_waiting | 0
sv_active  | 2
sv_idle    | 0
sv_used    | 0
sv_tested  | 0
sv_login   | 0
maxwait    | 0
maxwait_us | 0
pool_mode  | transaction
```

Ce que l'on voit dans les traces de PostgreSQL :

```
… 10:05:49.695 UTC [2841] LOG:  duration: 0.144 ms  statement: select 1
… 10:05:49.695 UTC [2841] LOG:  duration: 0.014 ms  statement: BEGIN ;
… 10:05:49.695 UTC [2841] LOG:  duration: 0.110 ms  statement: INSERT INTO log SELECT
 ↪  now() ;
… 10:05:52.320 UTC [2943] LOG:  connection received: host=127.0.0.1 port=50554
… 10:05:52.321 UTC [2943] LOG:  connection authorized: user=pooler database=postgres
… 10:05:52.323 UTC [2943] LOG:  duration: 0.171 ms  statement: SET
 ↪  application_name='psql';
… 10:05:52.323 UTC [2943] LOG:  duration: 0.015 ms  statement: BEGIN ;
… 10:05:52.324 UTC [2943] LOG:  duration: 0.829 ms  statement: INSERT INTO log SELECT
 ↪  now() ;
```

Du point de l'application, cela a été transparent.

Cette deuxième connexion va rester ouverte, mais elle n'est pas forcément associée à la deuxième session. Cela peut se voir simplement ainsi en demandant le PID du *backend* sur le serveur, qui sera le même dans les deux sessions :

```
postgres=> SELECT pg_backend_pid() ;

 pg_backend_pid
----------------
           2841
```

16.9.3 Pooling par requête

> Passer le pooler en mode pooling par requête et tenter d'ouvrir une transaction.

De la même manière que ci-dessus, soit :

```
pool_mode = statement
```

soit :

```
* = host=127.0.0.1 port=5432 pool_mode=statement
```

Redémarrage du pooler :

```
 # systemctl restart pgbouncer
```

Si on essaie de démarrer une transaction :

```
BEGIN;
ERROR:  transaction blocks not allowed in statement pooling mode
la connexion au serveur a été coupée de façon inattendue
        Le serveur s'est peut-être arrêté anormalement avant ou durant le
        traitement de la requête.
La connexion au serveur a été perdue. Tentative de réinitialisation : Succès.
```

Le pooling par requête empêche l'utilisation de transactions.

> Repasser PgBouncer en pooling par session.

Cela revient à revenir au mode par défaut (`pool_mode=session`).

16.9.4 Pgbench

> Créer une base nommée `bench` appartenant à **pooler**. Avec pgbench, l'initialiser avec un *scale factor* de 100.

Le pooler n'est pas configuré pour que **postgres** puisse s'y connecter, il faut donc se connecter directement à l'instance pour créer la base :

```
postgres$ createdb -h /var/run/postgresql -p 5432 --owner pooler  bench
```

La suite peut passer par le pooler :

```
$ /usr/pgsql-14/bin/pgbench -i -s 100 -U pooler -h 127.0.0.1 -p 6432 bench
Password:
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
10000000 of 10000000 tuples (100%) done (elapsed 25.08 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 196.24 s (drop tables 0.00 s, create tables 0.06 s, client-side generate
↪  28.00 s,
                  vacuum 154.35 s, primary keys 13.83 s).
```

> Lancer des tests (lectures uniquement, avec `--select`) de 60 secondes avec 80 connexions : une fois sur le pooler, et une fois directement sur le serveur. Comparer les performances.

NB : Pour des résultats rigoureux, pgbench doit être utilisé sur une plus longue durée.

Sur le pooler, on lance :

```
$ /usr/pgsql-14/bin/pgbench \
  --select -T 60 -c 80 -p 6432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 209465
latency average = 22.961 ms
tps = 3484.222638 (including connections establishing)
tps = 3484.278500 (excluding connections establishing)
```

(Ces chiffres ont été obtenus sur un portable avec SSD.)

On recommence directement sur l'instance. (Si l'ordre échoue par saturation des connexions, il faudra attendre que PgBouncer relâche les 20 connexions qu'il a gardées ouvertes.)

```
$ /usr/pgsql-14/bin/pgbench \
  --select -T 60 -c 80 -p 5432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 241482
latency average = 19.884 ms
tps = 4023.255058 (including connections establishing)
tps = 4023.573501 (excluding connections establishing)
```

Le test n'est pas assez rigoureux (surtout sur une petite machine de test) pour dire plus que : les résultats sont voisins.

> Refaire ce test en demandant d'ouvrir et fermer les connexions (-C), sur le serveur puis sur le pooler. Effectuer un SHOW POOLS pendant ce dernier test.

Sur le serveur :

```
$ /usr/pgsql-14/bin/pgbench \
  -C --select -T 60 -c 80 -p 5432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
Password:
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 9067
latency average = 529.654 ms
tps = 151.041956 (including connections establishing)
tps = 152.922609 (excluding connections establishing)
```

On constate une division par 26 du débit de transactions : le coût des connexions/déconnexions est énorme.

Si on passe par le pooler :

```
$ /usr/pgsql-14/bin/pgbench \
  -C --select -T 60 -c 80 -p 6432 -U pooler -h 127.0.0.1 -d bench 2>/dev/null
Password:
transaction type: <builtin: select only>
scaling factor: 100
query mode: simple
number of clients: 80
number of threads: 1
duration: 60 s
number of transactions actually processed: 49926
latency average = 96.183 ms
tps = 831.745556 (including connections establishing)
tps = 841.461561 (excluding connections establishing)
```

On ne retrouve pas les performances originales, mais le gain est tout de même d'un facteur 5, puisque les connexions existantes sur le serveur PostgreSQL sont réutilisées et n'ont pas à être recréées.

Pendant ce dernier test, on peut consulter les connexions ouvertes : il n'y en que 20, pas 80. Noter le grand nombre de celles en attente.

```
pgbouncer=# SHOW POOLS \gx
-[ RECORD 1 ]---------
database   | bench
user       | pooler
cl_active  | 20
cl_waiting | 54
sv_active  | 20
sv_idle    | 0
```

```
sv_used     | 0
sv_tested   | 0
sv_login    | 0
maxwait     | 0
maxwait_us | 73982
pool_mode   | session
…
```

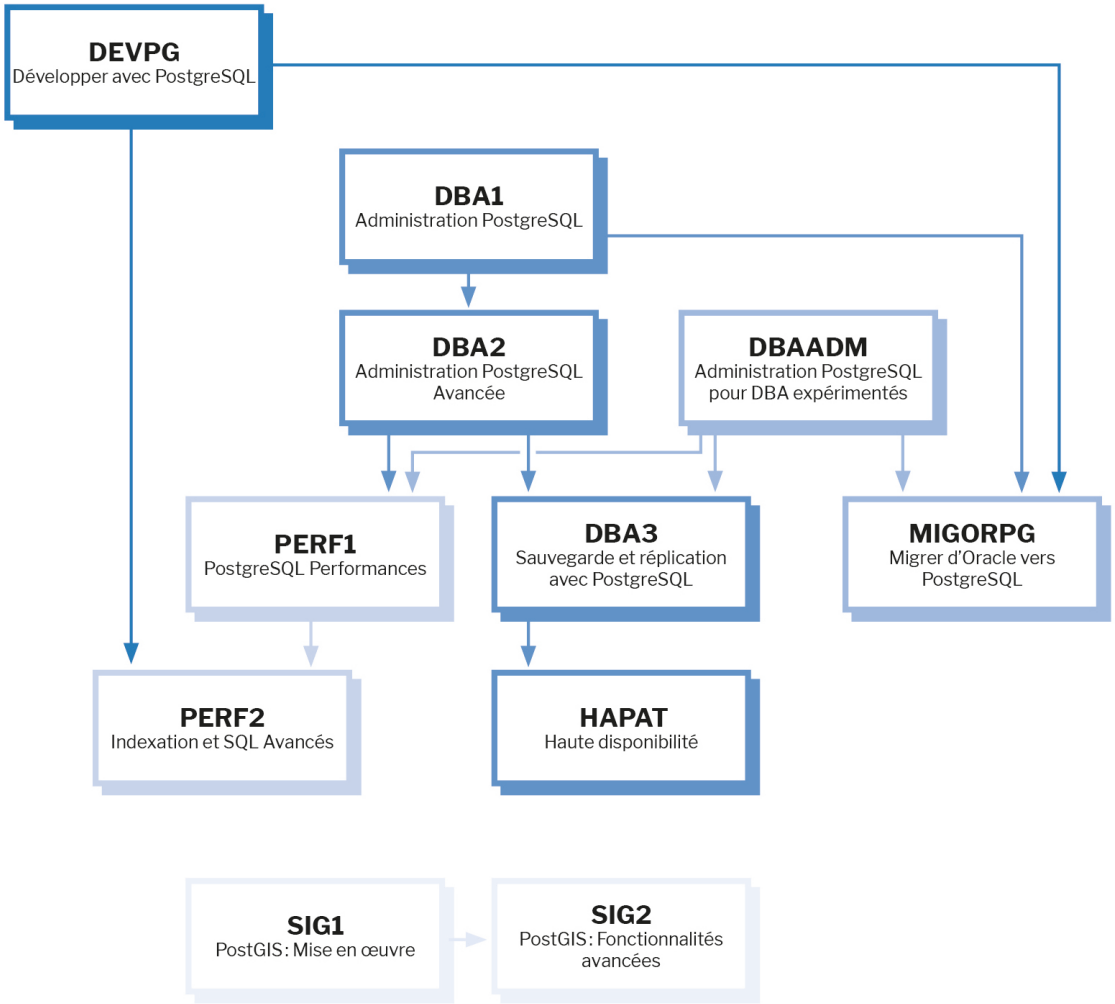
Ces tests n'ont pas pour objectif d'être représentatif mais juste de mettre en évidence le coût d'ouverture/fermeture de connexion. Dans ce cas, le pooler peut apporter un gain très significatif sur les performances.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.