Formation DBAADM

PostgreSQL pour DBA expérimentés

25.03

Table des matières

Sur ce document . . . 1
Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 2
Forme de ce manuel . . . 2
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 3
Versions de PostgreSQL couvertes . . . 3

1/ PostgreSQL : historique & communauté **5**
1.1 Préambule . . . 6
1.1.1 Au menu . . . 6
1.2 Un peu d'histoire… . . . 8
1.2.1 Licence . . . 8
1.2.2 PostgreSQL ?!?! . . . 9
1.2.3 Principes fondateurs . . . 9
1.2.4 Origines . . . 11
1.2.5 Apparition de la communauté internationale . . . 12
1.2.6 Progression du code . . . 13
1.3 Les versions de PostgreSQL . . . 15
1.3.1 Historique . . . 15
1.3.2 Versions & fonctionnalités . . . 16
1.3.3 Numérotation . . . 17
1.3.4 Mises à jour mineure . . . 18
1.3.5 Versions courantes . . . 18
1.3.6 Versions 9.4 à 11 . . . 19
1.3.7 Version 12 . . . 21
1.3.8 Version 13 . . . 22
1.3.9 Version 14 . . . 23
1.3.10 Version 15 . . . 24
1.3.11 Version 16 . . . 25
1.3.12 Version 17 . . . 26
1.3.13 Petit résumé . . . 27
1.3.14 Quelle version utiliser en production ? . . . 28
1.3.15 Versions dérivées / Forks . . . 29
1.4 Quelques projets satellites . . . 32
1.4.1 Administration, Développement, Modélisation . . . 32
1.4.2 Sauvegardes . . . 33
1.4.3 Supervision . . . 34
1.4.4 Audit . . . 34
1.4.5 Migration . . . 35
1.4.6 PostGIS . . . 36

1.5 Sponsors & Références . . . 38
1.5.1 Sponsors principaux . . . 38
1.5.2 Autres sponsors . . . 39
1.5.3 Références . . . 40
1.5.4 Le Bon Coin . . . 41
1.6 À la rencontre de la communauté . . . 43
1.6.1 PostgreSQL, un projet mondial . . . 43
1.6.2 PostgreSQL Core Team . . . 44
1.6.3 Contributeurs . . . 45
1.6.4 Qui contribue du code ? . . . 46
1.6.5 Répartition des développeurs . . . 47
1.6.6 Utilisateurs . . . 47
1.6.7 Pourquoi participer . . . 48
1.6.8 Ressources web de la communauté . . . 48
1.6.9 Documentation officielle . . . 49
1.6.10 Serveurs francophones . . . 49
1.6.11 Listes de discussions / Listes d'annonces . . . 50
1.6.12 IRC . . . 50
1.6.13 Wiki . . . 51
1.6.14 L'avenir de PostgreSQL . . . 51
1.7 Conclusion . . . 53
1.7.1 Bibliographie . . . 53
1.7.2 Questions . . . 54
1.8 Quiz . . . 55

2/ Découverte des fonctionnalités 57
2.1 Au menu . . . 58
2.2 Fonctionnalités du moteur . . . 59
2.2.1 Respect du standard SQL . . . 59
2.2.2 ACID . . . 60
2.2.3 MVCC . . . 61
2.2.4 Transactions . . . 62
2.2.5 Niveaux d'isolation . . . 64
2.2.6 Fiabilité : journaux de transactions (1) . . . 65
2.2.7 Fiabilité : journaux de transactions (2) . . . 65
2.2.8 Sauvegardes . . . 67
2.2.9 Réplication . . . 68
2.2.10 Extensibilité . . . 69
2.2.11 Sécurité . . . 70
2.3 Objets SQL . . . 71
2.3.1 Organisation logique . . . 72
2.3.2 Instances . . . 73
2.3.3 Rôles . . . 73
2.3.4 Tablespaces . . . 74
2.3.5 Bases . . . 75

2.3.6 Schémas . . . 75
2.3.7 Tables . . . 79
2.3.8 Vues . . . 80
2.3.9 Index . . . 83
2.3.10 Types de données . . . 84
2.3.11 Contraintes . . . 87
2.3.12 Colonnes à valeur générée . . . 89
2.3.13 Langages . . . 91
2.3.14 Fonctions & procédures . . . 92
2.3.15 Opérateurs . . . 93
2.3.16 Triggers . . . 94
2.3.17 Questions . . . 95
2.4 Quiz . . . 96

3/ Installation de PostgreSQL 97
3.1 Introduction . . . 98
3.2 Pré-requis minimaux pour une instance PostgreSQL . . . 99
3.2.1 Pré-requis minimaux . . . 99
3.3 Installation à partir des sources . . . 101
3.3.1 Étapes d'une installation à partir des sources . . . 101
3.3.2 Téléchargement . . . 101
3.3.3 Phases de compilation/installation . . . 102
3.3.4 Utilisateur dédié de PostgreSQL . . . 105
3.3.5 Création du répertoire de données de l'instance . . . 106
3.3.6 Lancement et arrêt . . . 109
3.4 Installation à partir des paquets Linux . . . 111
3.4.1 Paquets . . . 111
3.4.2 Paquets Debian officiels . . . 111
3.4.3 Paquets Debian : spécificités . . . 113
3.4.4 Paquets Debian communautaires . . . 114
3.4.5 Paquets Red Hat communautaires : yum.postgresql.org . . . 115
3.4.6 Paquets Red Hat communautaires : installation . . . 115
3.4.7 Paquets Red Hat communautaires : spécificités . . . 116
3.5 Industrialisation avec pglift . . . 118
3.5.1 pglift : présentation rapide . . . 118
3.5.2 pglift : fichier de configuration . . . 119
3.5.3 pglift : exemples de commandes . . . 120
3.6 Utiliser PostgreSQL dans un conteneur . . . 125
3.6.1 Docker . . . 125
3.6.2 Kubernetes . . . 128
3.7 Installation sous Windows . . . 131
3.7.1 Comment installer sous Windows ? . . . 131
3.7.2 Installeur graphique . . . 132
3.8 Premiers réglages . . . 134
3.8.1 Au menu . . . 134

3.8.2 Sécurité . . . 134
3.8.3 Configuration minimale . . . 135
3.8.4 Précédence des paramètres . . . 136
3.8.5 Configuration des connexions : accès au serveur . . . 136
3.8.6 Configuration du nombre de connexions . . . 138
3.8.7 Configuration de la mémoire partagée . . . 139
3.8.8 Configuration : mémoire des processus . . . 141
3.8.9 Configuration des journaux de transactions 1/2 . . . 143
3.8.10 Configuration des journaux de transactions 2/2 . . . 145
3.8.11 Configuration des traces . . . 146
3.8.12 Configuration des tâches de fond . . . 147
3.8.13 Se faciliter la vie . . . 147
3.9 Mise à jour . . . 149
3.9.1 Mise à jour . . . 149
3.9.2 Recommandations . . . 149
3.9.3 Mise à jour mineure . . . 150
3.9.4 Mise à jour majeure . . . 151
3.9.5 Mise à jour majeure par dump/restore . . . 152
3.9.6 Mise à jour majeure par réplication logique . . . 153
3.9.7 Mise à jour majeure par pg_upgrade . . . 154
3.9.8 Mise à jour de l'OS . . . 155
3.10 Conclusion . . . 156
3.10.1 Pour aller plus loin . . . 156
3.10.2 Questions . . . 156
3.11 Quiz . . . 157
3.12 Travaux pratiques . . . 158
3.12.1 Installation depuis les paquets binaires du PGDG (Rocky Linux) . . . 158
3.12.2 Installation à partir des sources (optionnel) . . . 160
3.13 Travaux pratiques (solutions) . . . 162
3.13.1 Installation depuis les paquets binaires du PGDG (Rocky Linux) . . . 162
3.13.2 Installation à partir des sources (optionnel) . . . 170
3.14 Installation de PostgreSQL depuis les paquets communautaires . . . 177
3.14.1 Sur Rocky Linux 8 ou 9 . . . 177
3.14.2 Sur Debian / Ubuntu . . . 180
3.14.3 Accès à l'instance depuis le serveur même (toutes distributions) . . . 182

4/ Outils graphiques et console **185**
4.1 Préambule . . . 186
4.1.1 Plan . . . 186
4.2 Outils console de PostgreSQL . . . 187
4.2.1 Outils : Gestion des bases . . . 187
4.2.2 Outils : Sauvegarde / Restauration . . . 188
4.2.3 Outils : Maintenance . . . 189
4.2.4 Outils : Maintenance de l'instance . . . 189
4.2.5 Autres outils en ligne de commande . . . 190

4.3 Chaînes de connexion . . . 191
4.3.1 Paramètres . . . 191
4.3.2 Autres variables d'environnement . . . 194
4.3.3 Chaînes libpq clés/valeur . . . 194
4.3.4 Chaînes URI . . . 195
4.3.5 Connexion avec choix automatique du serveur . . . 196
4.3.6 Authentification d'un client (outils console) . . . 197
4.4 La console psql . . . 199
4.4.1 Obtenir de l'aide et quitter . . . 199
4.4.2 Gestion de la connexion . . . 200
4.4.3 Catalogue système : objets utilisateurs . . . 202
4.4.4 Catalogue système : rôles et accès . . . 205
4.4.5 Visualiser le code des objets . . . 207
4.4.6 Configuration . . . 208
4.4.7 Exécuter des requêtes . . . 209
4.4.8 Afficher le résultat d'une requête . . . 211
4.4.9 Afficher les détails d'une requête . . . 213
4.4.10 Exécuter le résultat d'une requête . . . 213
4.4.11 Manipuler le tampon de requêtes . . . 215
4.4.12 Entrées/sorties . . . 216
4.4.13 Gestion de l'environnement système . . . 217
4.4.14 Variables internes psql . . . 218
4.4.15 Variables utilisateur psql . . . 220
4.4.16 Tests conditionnels . . . 221
4.4.17 Personnaliser psql . . . 222
4.5 Écriture de scripts shell . . . 225
4.5.1 Exécuter un script SQL avec psql . . . 225
4.5.2 Gestion des transactions . . . 226
4.5.3 Écrire un script SQL . . . 227
4.5.4 Les blocs anonymes . . . 228
4.5.5 Utiliser des variables . . . 229
4.5.6 Gestion des erreurs . . . 230
4.5.7 Formatage des résultats . . . 231
4.5.8 Résultats en pivot (tableau croisé) . . . 232
4.5.9 Formatage dans les scripts SQL . . . 233
4.5.10 Scripts & Crontab . . . 234
4.5.11 Exemple de script de sauvegarde . . . 234
4.6 Outils graphiques . . . 236
4.6.1 temBoard . . . 236
4.6.2 temBoard - PostgreSQL Remote Control . . . 236
4.6.3 temBoard - Vue parc . . . 237
4.6.4 temBoard - Tableau de bord . . . 238
4.6.5 temBoard - Activity . . . 239
4.6.6 temBoard - Supervision . . . 240
4.6.7 temBoard - Configuration . . . 241

4.6.8 temBoard - Maintenance . . . 242
4.6.9 pgAdmin 4 . . . 243
4.6.10 pgAdmin 4 : tableau de bord . . . 244
4.6.11 DBeaver . . . 245
4.6.12 DBeaver : fenêtre principale . . . 246
4.6.13 phpPgAdmin . . . 247
4.6.14 phpPgAdmin : fonctionnalités . . . 247
4.6.15 AdminerEvo . . . 248
4.6.16 AdminerEvo : fonctionnalités . . . 248
4.6.17 pgModeler . . . 249
4.6.18 pgModeler . . . 250
4.7 Conclusion . . . 251
4.7.1 Questions . . . 251
4.8 Quiz . . . 252
4.9 Introduction à pgbench . . . 253
4.9.1 Installation . . . 253
4.9.2 Générer de l'activité . . . 254
4.10 Travaux pratiques . . . 256
4.11 Travaux pratiques (solutions) . . . 259

5/ Tâches courantes 267
5.1 Introduction . . . 268
5.2 Bases . . . 269
5.2.1 Liste des bases . . . 269
5.2.2 Modèle (template) . . . 272
5.2.3 Création d'une base . . . 273
5.2.4 Suppression d'une base . . . 276
5.2.5 Modification / configuration . . . 277
5.3 Rôles . . . 280
5.3.1 Utilisateurs et groupes . . . 280
5.3.2 Liste des rôles . . . 281
5.3.3 Création d'un rôle . . . 284
5.3.4 Suppression d'un rôle . . . 287
5.3.5 Modification d'un rôle . . . 288
5.3.6 Mot de passe . . . 291
5.3.7 Sécurité des mots de passe . . . 293
5.4 Droits sur les objets . . . 297
5.4.1 Droits sur les objets . . . 297
5.4.2 Afficher les droits . . . 302
5.4.3 Droits sur les métadonnées . . . 304
5.4.4 Droits plus globaux 1/2 . . . 306
5.4.5 Droits plus globaux 2/2 . . . 309
5.4.6 Héritage des droits . . . 310
5.4.7 Changement de rôle . . . 311

5.5 Droits de connexion . . . 313
5.5.1 Informations de connexion . . . 313
5.5.2 Configuration de l'authentification : pg_hba.conf . . . 314
5.5.3 Exemple de pg_hba.conf . . . 316
5.5.4 pg_hba.conf : colonne type . . . 317
5.5.5 pg_hba.conf : colonne database . . . 317
5.5.6 pg_hba.conf : colonne user . . . 318
5.5.7 pg_hba.conf : colonne adresse IP . . . 319
5.5.8 pg_hba.conf : colonne méthode . . . 320
5.5.9 pg_hba.conf : colonne options . . . 321
5.5.10 pg_hba.conf : méthodes internes . . . 321
5.5.11 pg_hba.conf : méthodes externes . . . 322
5.5.12 Un (mauvais) exemple de pg_hba.conf . . . 323
5.5.13 Mapping : pg_ident.conf . . . 324
5.6 Synchronisation des rôles . . . 326
5.7 Tâches de maintenance . . . 327
5.7.1 Maintenance : VACUUM . . . 328
5.7.2 Maintenance : VACUUM FULL . . . 331
5.7.3 VACUUM vs VACUUM FULL . . . 332
5.7.4 Maintenance : ANALYZE . . . 333
5.7.5 Maintenance : REINDEX . . . 335
5.7.6 Maintenance : CLUSTER . . . 337
5.7.7 Maintenance : automatisation . . . 338
5.7.8 Maintenance : autovacuum . . . 339
5.7.9 Maintenance : Script de REINDEX . . . 339
5.8 Sécurité . . . 344
5.8.1 Droits par défaut . . . 344
5.8.2 Droits par défaut (suite) . . . 346
5.8.3 Droits par défaut (suite) . . . 348
5.8.4 Restreindre les droits . . . 349
5.8.5 Arrêter une requête ou une session . . . 354
5.8.6 Chiffrements . . . 356
5.8.7 En cas de crash . . . 357
5.8.8 Corruption de données . . . 357
5.9 Conclusion . . . 360
5.9.1 Questions . . . 360
5.10 Quiz . . . 361
5.11 Travaux pratiques . . . 362
5.11.1 Traces maximales . . . 362
5.11.2 Méthode d'authentification . . . 362
5.11.3 Création des bases . . . 363
5.11.4 Mots de passe . . . 363
5.11.5 Rôles et permissions . . . 364
5.11.6 Autorisation d'accès distant . . . 366
5.11.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE . . . 366

5.11.8 Statistiques . . . 368
5.11.9 Réindexation . . . 369
5.11.10 Traces . . . 369
5.12 Travaux pratiques (solutions) . . . 371
5.12.1 Traces maximales . . . 371
5.12.2 Méthode d'authentification . . . 372
5.12.3 Création des bases . . . 373
5.12.4 Mots de passe . . . 374
5.12.5 Rôles et permissions . . . 379
5.12.6 Autorisation d'accès distant . . . 385
5.12.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE . . . 386
5.12.8 Réactivation de l'autovacuum . . . 395
5.12.9 Statistiques des données . . . 397
5.12.10 Réindexation . . . 398
5.12.11 Traces . . . 400

6/ PostgreSQL : Politique de sauvegarde 403
6.1 Introduction . . . 404
6.1.1 Au menu . . . 404
6.2 Définir une politique de sauvegarde . . . 405
6.2.1 Objectifs . . . 406
6.2.2 Différentes approches . . . 406
6.2.3 RTO/RPO . . . 407
6.2.4 Industrialisation . . . 409
6.2.5 Documentation . . . 410
6.2.6 Règle 3-2-1 . . . 410
6.2.7 Fichiers de configuration . . . 412
6.2.8 Tester la restauration . . . 412
6.3 Conclusion . . . 414
6.4 Quiz . . . 415

7/ PostgreSQL : Sauvegarde et restauration 417
7.1 Introduction . . . 418
7.1.1 Au menu . . . 418
7.2 Sauvegardes logiques . . . 419
7.2.1 Sauvegardes logiques : caractéristiques . . . 419
7.2.2 pg_dump . . . 421
7.2.3 pg_dump : formats de sauvegarde . . . 421
7.2.4 pg_dump : choisir le format de sauvegarde . . . 423
7.2.5 pg_dump : compression . . . 425
7.2.6 pg_dump : fichier ou sortie standard . . . 426
7.2.7 pg_dump : structure ou données . . . 426
7.2.8 pg_dump : sélection d'objets . . . 427
7.2.9 pg_dump : parallélisation . . . 429
7.2.10 pg_dump : objets binaires . . . 430
7.2.11 pg_dump : extensions . . . 431

7.2.12 pg_dump : options diverses . . . 432
7.2.13 pg_dumpall . . . 434
7.2.14 pg_dumpall : sélection des objets . . . 435
7.2.15 pg_dump/pg_dumpall : options de connexions . . . 437
7.2.16 pg_dump/pg_dumpall : privilèges nécessaires . . . 438
7.2.17 pg_dump/pg_dumpall : compresser la sortie . . . 439
7.3 Restauration d'une sauvegarde logique . . . 440
7.3.1 Quel outil pour restaurer une sauvegarde logique ? . . . 440
7.3.2 psql . . . 440
7.3.3 psql : options pour une restauration . . . 441
7.3.4 pg_restore . . . 443
7.3.5 pg_restore : indiquer la base de données . . . 444
7.3.6 pg_restore : fichiers en entrée/sortie . . . 445
7.3.7 pg_restore : structure ou données . . . 446
7.3.8 pg_restore : sélection d'objets . . . 448
7.3.9 pg_restore : sélection avancée . . . 449
7.3.10 pg_restore : parallélisation . . . 451
7.3.11 pg_restore : options diverses . . . 452
7.4 Autres considérations sur la sauvegarde logique . . . 453
7.4.1 Versions des outils clients et version de l'instance . . . 453
7.4.2 Script de sauvegarde idéal . . . 454
7.4.3 pg_back - Présentation . . . 458
7.4.4 Sauvegarde et restauration sans fichier intermédiaire . . . 459
7.4.5 Statistiques et maintenance après import . . . 460
7.4.6 Durée d'exécution . . . 461
7.4.7 Taille d'une sauvegarde logique . . . 462
7.4.8 Avantages de la sauvegarde logique . . . 462
7.4.9 Inconvénients de la sauvegarde logique . . . 463
7.5 Sauvegarde physique à froid des fichiers . . . 465
7.5.1 Sauvegarde physique à froid : principe . . . 465
7.5.2 Avantages des sauvegardes à froid . . . 466
7.5.3 Inconvénients des sauvegardes à froid . . . 467
7.5.4 Diminuer l'immobilisation . . . 467
7.6 Sauvegarde à chaud . . . 469
7.6.1 Sauvegarde à chaud des fichiers par snapshot de partition : principe . . . 469
7.6.2 Sauvegarde à chaud des fichiers avec PostgreSQL . . . 470
7.7 Recommandations générales . . . 471
7.7.1 Matrice des options de sauvegarde . . . 471
7.8 Conclusion . . . 472
7.8.1 Questions . . . 472
7.9 Quiz . . . 473
7.10 Travaux pratiques . . . 474
7.10.1 Sauvegardes logiques . . . 474
7.10.2 Restaurations logiques . . . 475
7.10.3 Sauvegarde et restauration partielle . . . 476

7.10.4 (Optionnel) Sauvegarde et restauration par parties avec modification 476
7.10.5 (Optionnel) Sauvegardes d'objets isolés 477
7.11 Travaux pratiques (solutions) 478
7.11.1 Sauvegardes logiques 478
7.11.2 Restaurations logiques 482
7.11.3 Sauvegarde et restauration partielle 486
7.11.4 (Optionnel) Sauvegarde et restauration par parties avec modification 486
7.11.5 (Optionnel) Sauvegardes d'objets isolés 488

8/ Architecture & fichiers de PostgreSQL 491
8.1 Au menu 492
8.2 Rappels sur l'installation 493
8.2.1 Paquets précompilés 493
8.2.2 Installons PostgreSQL 494
8.3 Processus de PostgreSQL 495
8.3.1 Introduction 495
8.3.2 Processus d'arrière-plan 496
8.3.3 Processus d'arrière-plan (suite) 497
8.4 Processus par client (client backend) 499
8.5 Gestion de la mémoire 501
8.6 Fichiers 502
8.6.1 Répertoire de données 503
8.6.2 Fichiers de configuration 504
8.6.3 Autres fichiers dans PGDATA 504
8.6.4 Fichiers de données 506
8.6.5 Fichiers liés aux transactions 508
8.6.6 Fichiers liés à la réplication 510
8.6.7 Répertoire des tablespaces 511
8.6.8 Fichiers des statistiques d'activité 512
8.6.9 Autres répertoires 513
8.6.10 Les fichiers de traces (journaux) 513
8.7 Résumé 514
8.8 Conclusion 515
8.8.1 Questions 515
8.9 Quiz 516
8.10 Travaux pratiques 517
8.10.1 Processus 517
8.10.2 Fichiers 518
8.11 Travaux pratiques (solutions) 520
8.11.1 Processus 520
8.11.2 Fichiers 523

9/ Configuration de PostgreSQL 529
9.1 Au menu 530
9.2 Paramètres en lecture seule 531
9.3 Fichiers de configuration 533

9.4 postgresql.conf . . . 534
9.4.1 Surcharge des paramètres de postgresql.conf . . . 535
9.4.2 Précédence des paramètres . . . 538
9.4.3 Survol de postgresql.conf . . . 538
9.5 pg_hba.conf et pg_ident.conf . . . 541
9.6 Tablespaces . . . 542
9.6.1 Tablespaces : mise en place . . . 544
9.6.2 Tablespaces : configuration . . . 546
9.6.3 Tablespaces : performance . . . 548
9.7 Gestion des connexions . . . 550
9.7.1 TCP . . . 550
9.7.2 SSL . . . 551
9.8 Statistiques sur l'activité . . . 552
9.8.1 Statistiques d'activité collectées . . . 553
9.8.2 Vues système . . . 554
9.9 Statistiques sur les données . . . 562
9.10 Optimiseur . . . 565
9.10.1 Optimisation par les coûts . . . 566
9.10.2 Nombre de tables considérées par le planificateur . . . 568
9.10.3 Paramètres supplémentaires de l'optimiseur . . . 571
9.10.4 Débogage de l'optimiseur . . . 572
9.11 Conclusion . . . 575
9.11.1 Questions . . . 575
9.12 Quiz . . . 576
9.13 Travaux pratiques . . . 577
9.13.1 Tablespace . . . 577
9.13.2 Statistiques d'activités, tables et vues système . . . 577
9.13.3 Statistiques sur les données . . . 578
9.14 Travaux pratiques (solutions) . . . 580
9.14.1 Tablespace . . . 580
9.14.2 Statistiques d'activités, tables et vues système . . . 581
9.14.3 Statistiques sur les données . . . 584

10/ Mémoire et journalisation dans PostgreSQL 587
10.1 Au menu . . . 588
10.2 Rappel de l'architecture de PostgreSQL . . . 589
10.3 Mémoire partagée . . . 590
10.3.1 Zones de la mémoire partagée . . . 590
10.3.2 Taille de la mémoire partagée . . . 592
10.4 Mémoire par processus . . . 593
10.4.1 work_mem, maintenance_work_mem . . . 593
10.5 Shared buffers . . . 597
10.5.1 Utilité des shared buffers . . . 597
10.5.2 Dimensionnement des shared buffers . . . 598
10.5.3 Notions essentielles de gestion du cache . . . 600

10.5.4 Ring buffer . . . 601
10.5.5 Contenu du cache . . . 602
10.5.6 Synchronisation en arrière-plan . . . 603
10.6 Journalisation . . . 604
10.6.1 Principe de la journalisation . . . 604
10.6.2 Intégrité & durabilité . . . 604
10.6.3 Journaux de transaction (rappels) . . . 605
10.6.4 Que contiennent les journaux de transactions ? . . . 606
10.6.5 Checkpoint . . . 609
10.6.6 Déclenchement & comportement des checkpoints - 1 . . . 611
10.6.7 Déclenchement & comportement des checkpoints - 2 . . . 613
10.6.8 Paramètres du background writer . . . 614
10.6.9 WAL buffers : journalisation en mémoire . . . 615
10.6.10 Compression des journaux . . . 616
10.6.11 Limiter le coût de la journalisation . . . 617
10.7 Au-delà de la journalisation . . . 618
10.7.1 L'archivage des journaux . . . 618
10.7.2 Réplication . . . 619
10.8 Conclusion . . . 621
10.8.1 Questions . . . 621
10.9 Quiz . . . 622
10.10 Travaux pratiques . . . 623
10.10.1 Mémoire partagée . . . 623
10.10.2 Mémoire de tri . . . 624
10.10.3 Cache disque de PostgreSQL . . . 624
10.10.4 Journaux . . . 625
10.11 Travaux pratiques (solutions) . . . 627
10.11.1 Mémoire partagée . . . 627
10.11.2 Mémoire de tri . . . 629
10.11.3 Cache disque de PostgreSQL . . . 630
10.11.4 Journaux . . . 634

11/ Mécanique du moteur transactionnel & MVCC 637
11.1 Introduction . . . 638
11.2 Au menu . . . 639
11.3 Présentation de MVCC . . . 640
11.3.1 Définitions . . . 640
11.3.2 Alternative à MVCC : un seul enregistrement en base . . . 640
11.3.3 Implémentation de MVCC par *undo* . . . 641
11.3.4 L'implémentation MVCC de PostgreSQL . . . 642
11.4 Niveaux d'isolation . . . 644
11.4.1 Principe des niveaux d'isolation . . . 644
11.4.2 Niveau READ UNCOMMITTED . . . 644
11.4.3 Niveau READ COMMITTED . . . 645
11.4.4 Niveau REPEATABLE READ . . . 645

11.4.5 Niveau SERIALIZABLE . . . 646
11.5 Blocs & lignes . . . 648
11.5.1 Structure d'un bloc . . . 648
11.5.2 xmin & xmax . . . 649
11.5.3 xmin & xmax (suite) . . . 650
11.5.4 xmin & xmax (suite) . . . 650
11.5.5 xmin & xmax (suite) . . . 651
11.5.6 CLOG . . . 652
11.6 Avantages & inconvénients du MVCC de PostgreSQL . . . 653
11.6.1 Avantages du MVCC de PostgreSQL . . . 653
11.6.2 Inconvénients du MVCC de PostgreSQL . . . 654
11.7 Le wraparound . . . 656
11.7.1 Le wraparound (1) . . . 656
11.7.2 Le wraparound (2) . . . 658
11.7.3 Le wraparound (3) . . . 659
11.8 Optimisations de MVCC . . . 661
11.8.1 Mise à jour jour HOT . . . 661
11.8.2 Free Space Map . . . 662
11.8.3 Visibility Map . . . 662
11.9 Verrous . . . 664
11.9.1 Verrouillage et MVCC . . . 664
11.9.2 Le gestionnaire de verrous . . . 664
11.9.3 Verrous sur enregistrement . . . 665
11.9.4 La vue pg_locks . . . 666
11.9.5 Verrous - Paramètres . . . 667
11.10 Durées de sessions, transactions & ordres . . . 670
11.10.1 Quelle durée pour les sessions & transactions ? . . . 670
11.10.2 Quelle durée pour une session ? . . . 671
11.10.3 Quelle durée pour une transaction ? . . . 672
11.11 TOAST . . . 673
11.11.1 Mécanisme TOAST . . . 673
11.11.2 TOAST & table de débordement . . . 675
11.11.3 TOAST & compression . . . 679
11.12 Conclusion . . . 681
11.12.1 Questions . . . 681
11.13 Quiz . . . 682
11.14 Travaux pratiques . . . 683
11.14.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ . . . 683
11.14.2 Niveau d'isolation SERIALIZABLE (Optionnel) . . . 684
11.14.3 Effets de MVCC . . . 685
11.14.4 Verrous . . . 686
11.15 Travaux pratiques (solutions) . . . 688
11.15.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ . . . 688
11.15.2 Niveau d'isolation SERIALIZABLE (Optionnel) . . . 690
11.15.3 Effets de MVCC . . . 694

11.15.4 Verrous 699

12/ VACUUM et autovacuum **705**
12.1 Au menu 706
12.2 VACUUM et autovacuum 707
12.3 Fonctionnement de VACUUM 708
12.3.1 Fonctionnement de VACUUM (suite) 709
12.3.2 Fonctionnement de VACUUM (suite) 710
12.4 Les options de VACUUM 711
12.4.1 Tâches d'un VACUUM 711
12.4.2 Options de performance de VACUUM 713
12.4.3 Options pour un VACUUM en urgence 715
12.4.4 Autres options de VACUUM 716
12.5 Suivi du VACUUM 718
12.5.1 Progression du VACUUM 719
12.5.2 Droit de lancer un VACUUM 721
12.6 Autovacuum 722
12.6.1 Paramétrage du déclenchement de l'autovacuum 722
12.6.2 Déclenchement de l'autovacuum 723
12.6.3 Déclenchement de l'autovacuum (suite) 724
12.7 Paramétrage de VACUUM & autovacuum 726
12.7.1 VACUUM vs autovacuum 726
12.7.2 Mémoire 727
12.7.3 Bridage du VACUUM et de l'autovacuum 729
12.7.4 Paramétrage du FREEZE (1) 731
12.7.5 Paramétrage du FREEZE (2) 732
12.8 Autres problèmes courants 736
12.8.1 L'autovacuum dure trop longtemps 736
12.8.2 Arrêter un VACUUM ? 737
12.8.3 Ce qui peut bloquer le VACUUM FREEZE 738
12.9 Résumé des conseils sur l'autovacuum 740
12.9.1 Résumé des conseils sur l'autovacuum (1/2) 740
12.9.2 Résumé des conseils sur l'autovacuum (2/2) 741
12.10 Conclusion 742
12.10.1 Questions 742
12.11 Quiz 743
12.12 Travaux pratiques 744
12.12.1 Traiter la fragmentation 744
12.12.2 Détecter la fragmentation 745
12.12.3 Gestion de l'autovacuum 746
12.13 Travaux pratiques (solutions) 748
12.13.1 Traiter la fragmentation 748
12.13.2 Détecter la fragmentation 751
12.13.3 Gestion de l'autovacuum 753

13/ Partitionnement déclaratif (introduction) 761
13.1 Principe & intérêts du partitionnement . . . 762
13.2 Partitionnement déclaratif . . . 764
13.2.1 Partitionnement par liste . . . 765
13.2.2 Partitionnement par liste : implémentation . . . 766
13.2.3 Partitionnement par intervalle . . . 766
13.2.4 Partitionnement par intervalle : implémentation . . . 767
13.2.5 Partitionnement par hachage . . . 768
13.2.6 Partitionnement par hachage : implémentation . . . 768
13.3 Performances & partitionnement . . . 770
13.3.1 Attacher/détacher une partition . . . 772
13.3.2 Supprimer une partition . . . 772
13.3.3 Limitations principales du partitionnement déclaratif . . . 772
13.4 Conclusion . . . 774
13.5 Quiz . . . 775

14/ Sauvegarde physique à chaud et PITR 777
14.1 Introduction . . . 778
14.1.1 Au menu . . . 778
14.2 Rappel sur la journalisation . . . 780
14.2.1 Journaux de transaction . . . 780
14.3 PITR . . . 782
14.3.1 Principes du PITR . . . 782
14.3.2 Avantages du PITR . . . 783
14.3.3 Inconvénients du PITR . . . 784
14.4 Copie physique à chaud ponctuelle avec pg_basebackup . . . 786
14.4.1 pg_basebackup . . . 786
14.5 Sauvegarde PITR . . . 789
14.5.1 Étapes d'une sauvegarde PITR . . . 789
14.5.2 Méthodes d'archivage . . . 789
14.5.3 Choix du répertoire d'archivage . . . 790
14.5.4 Processus archiver : configuration (1/4) . . . 790
14.5.5 Processus archiver : configuration (2/4) . . . 791
14.5.6 Processus archiver : configuration (3/4) . . . 792
14.5.7 Processus archiver : configuration (4/4) . . . 794
14.5.8 Processus archiver : lancement . . . 794
14.5.9 Processus archiver : supervision . . . 795
14.5.10 pg_receivewal . . . 797
14.5.11 pg_receivewal - configuration serveur . . . 799
14.5.12 pg_receivewal - redémarrage du serveur . . . 799
14.5.13 pg_receivewal - lancement de l'outil . . . 800
14.5.14 Avantages et inconvénients . . . 801
14.6 Sauvegarde PITR manuelle . . . 802
14.7 Étapes d'une sauvegarde PITR manuelle . . . 803
14.7.1 Sauvegarde manuelle - 1/3 : pg_backup_start . . . 804

14.7.2 Sauvegarde manuelle - 2/3 : copie des fichiers . . . 805
14.7.3 Sauvegarde manuelle - 3/3 : pg_backup_stop . . . 807
14.7.4 Sauvegarde de base à chaud : pg_basebackup . . . 808
14.7.5 Fréquence de la sauvegarde de base . . . 809
14.7.6 Suivi de la sauvegarde de base . . . 810
14.8 Restaurer une sauvegarde PITR . . . 811
14.8.1 Exemple de scénario : sauvegarde . . . 811
14.8.2 Exemple de scénario : restauration . . . 812
14.8.3 Restaurer une sauvegarde PITR (1/5) . . . 813
14.8.4 Restaurer une sauvegarde PITR (2/5) . . . 813
14.8.5 Restaurer une sauvegarde PITR (3/5) . . . 814
14.8.6 Restaurer une sauvegarde PITR (4/5) . . . 815
14.8.7 Restaurer une sauvegarde PITR (5/5) . . . 817
14.8.8 Restauration PITR : durée . . . 819
14.8.9 Restauration PITR : différentes timelines . . . 819
14.8.10 Restauration PITR : illustration des timelines . . . 822
14.8.11 Après la restauration . . . 824
14.9 Pour aller plus loin . . . 826
14.9.1 Réduire le nombre de journaux sauvegardés . . . 826
14.9.2 Compresser les journaux de transactions . . . 827
14.9.3 Outils de sauvegarde PITR dédiés . . . 828
14.9.4 pgBackRest . . . 829
14.9.5 barman . . . 830
14.10 Conclusion . . . 831
14.10.1 Questions . . . 831
14.11 Quiz . . . 832
14.12 Travaux pratiques . . . 833
14.12.1 pg_basebackup : sauvegarde ponctuelle & restauration . . . 833
14.12.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants 834
14.13 Travaux pratiques (solutions) . . . 836
14.13.1 pg_basebackup : sauvegarde ponctuelle & restauration . . . 836
14.13.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants 840

15/ Supervision 847
15.1 Introduction . . . 848
15.1.1 Menu . . . 848
15.2 Politique de supervision . . . 849
15.2.1 Objectifs de la supervision . . . 849
15.2.2 Acteurs concernés . . . 850
15.2.3 Exemples d'indicateurs - système d'exploitation . . . 851
15.2.4 Exemples d'indicateurs - base de données . . . 852
15.3 Supervision de PostgreSQL . . . 853
15.3.1 Informations internes . . . 853
15.3.2 Outils externes . . . 854
15.3.3 check_pgactivity . . . 854

15.3.4 check_postgres . . . 857
15.4 Traces . . . 858
15.4.1 Configuration des traces : principes . . . 858
15.4.2 Événements exceptionnels tracés . . . 859
15.4.3 Où tracer ? . . . 860
15.4.4 Configuration de la destination des traces . . . 861
15.4.5 Niveau des traces . . . 863
15.4.6 Tracer les requêtes et leur durée . . . 864
15.4.7 Configuration : tracer certains comportements . . . 866
15.4.8 Repérer les fichiers temporaires . . . 867
15.4.9 Configuration : divers . . . 868
15.5 Outils d'analyse des traces . . . 870
15.5.1 pgBadger . . . 871
15.5.2 pgBadger : exemple de rapport . . . 872
15.5.3 Utiliser pgBadger . . . 873
15.5.4 Configurer PostgreSQL pour pgBadger . . . 873
15.5.5 Options de pgBadger . . . 875
15.5.6 pgBadger : exemple 1 . . . 876
15.5.7 pgBadger : exemple 2 . . . 877
15.5.8 pgBadger : exemple 3 . . . 877
15.5.9 pgBadger : exemple 4 . . . 878
15.5.10 pgBadger : exemple 5 . . . 878
15.5.11 logwatch . . . 879
15.5.12 tail_n_mail . . . 881
15.5.13 Configurer tail_n_mail . . . 881
15.5.14 tail_n_mail : exemple . . . 882
15.6 Statistiques d'activité . . . 883
15.6.1 Statistiques d'activité - configuration 1 . . . 883
15.6.2 Statistiques d'activité - configuration 2 . . . 888
15.6.3 Statistiques d'activité - configuration 3 . . . 889
15.6.4 Statistiques d'activité : perte . . . 889
15.6.5 Informations intéressantes à récupérer . . . 890
15.6.6 Nombre de connexions par base . . . 890
15.6.7 Taille des bases . . . 891
15.6.8 Nombre de verrous . . . 892
15.6.9 Et un grand nombre d'autres informations . . . 893
15.6.10 Outils . . . 893
15.6.11 munin . . . 894
15.6.12 Nagios . . . 895
15.6.13 Outils - Zabbix . . . 895
15.6.14 Outils - pg_stat_statements . . . 896
15.6.15 Outils - PoWA . . . 896
15.7 Conclusion . . . 898
15.7.1 Questions . . . 898
15.8 Quiz . . . 899

15.9 Travaux pratiques . . . 900
15.10 Travaux pratiques (solutions) . . . 902

16/ PostgreSQL : Gestion d'un sinistre **907**
16.1 Introduction . . . 908
16.1.1 Au menu . . . 908
16.2 Anticiper les désastres . . . 909
16.2.1 Documentation . . . 909
16.2.2 Procédures et scripts . . . 910
16.2.3 Supervision et historisation . . . 911
16.2.4 Automatisation . . . 912
16.3 Réagir aux désastres . . . 913
16.3.1 Symptômes d'un désastre . . . 913
16.3.2 Bons réflexes 1 . . . 914
16.3.3 Bons réflexes 2 . . . 915
16.3.4 Bons réflexes 3 . . . 916
16.3.5 Bons réflexes 4 . . . 917
16.3.6 Bons réflexes 5 . . . 917
16.3.7 Bons réflexes 6 . . . 918
16.3.8 Bons réflexes 7 . . . 919
16.3.9 Bons réflexes 8 . . . 920
16.3.10 Mauvais réflexes 1 . . . 922
16.3.11 Mauvais réflexes 2 . . . 923
16.3.12 Mauvais réflexes 3 . . . 923
16.4 Rechercher l'origine du problème . . . 925
16.4.1 Prérequis . . . 925
16.4.2 Recherche d'historique . . . 925
16.4.3 Matériel . . . 926
16.4.4 Virtualisation . . . 927
16.4.5 Système d'exploitation 1 . . . 928
16.4.6 Système d'exploitation 2 . . . 928
16.4.7 Système d'exploitation 3 . . . 929
16.4.8 PostgreSQL . . . 930
16.4.9 Paramétrage de PostgreSQL : écriture des fichiers . . . 931
16.4.10 Paramétrage de PostgreSQL : les sommes de contrôle . . . 932
16.4.11 Erreur de manipulation . . . 934
16.5 Outils . . . 935
16.5.1 Outils : pg_controldata . . . 935
16.5.2 Outils pour l'export/import de données . . . 937
16.5.3 Outils : pageinspect . . . 939
16.5.4 Outils : pg_resetwal . . . 942
16.5.5 Outils : pg_surgery . . . 943
16.5.6 Outils pour vérifier les sommes de contrôle . . . 944
16.5.7 Outils : amcheck & pg_amcheck . . . 946
16.6 Cas type de désastres . . . 949

16.7 Cas type de désastres . . . 950
16.7.1 Avertissement . . . 950
16.7.2 Corruption de blocs dans des index . . . 951
16.7.3 Corruption de blocs dans des tables 1 . . . 951
16.7.4 Corruption de blocs dans des tables 2 . . . 952
16.7.5 Corruption de blocs dans des tables 3 . . . 953
16.7.6 Corruption des WAL 1 . . . 954
16.7.7 Corruption des WAL 2 . . . 955
16.7.8 Corruption du fichier de contrôle . . . 956
16.7.9 Corruption du CLOG . . . 956
16.7.10 Corruption du catalogue système . . . 957
16.8 Conclusion . . . 958
16.9 Quiz . . . 959
16.10 Travaux pratiques . . . 960
16.10.1 Corruption d'un bloc de données . . . 960
16.10.2 Corruption d'un bloc de données et incohérences . . . 961
16.11 Travaux pratiques (solutions) . . . 963
16.11.1 Corruption d'un bloc de données . . . 963
16.11.2 Corruption d'un bloc de données et incohérences . . . 966

Les formations Dalibo **971**
Cursus des formations . . . 971
Les livres blancs . . . 972
Téléchargement gratuit . . . 972

Sur ce document

Formation	Formation DBAADM
Titre	PostgreSQL pour DBA expérimentés
Révision	25.03
ISBN	N/A
PDF	https://dali.bo/dbaadm_pdf
EPUB	https://dali.bo/dbaadm_epub
HTML	https://dali.bo/dbaadm_html
Slides	https://dali.bo/dbaadm_slides

Vous trouverez en ligne les différentes versions complètes de ce document. Les solutions de TP ne figurent pas forcément dans la version imprimée, mais sont dans les versions numériques (PDF ou HTML).

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

[1] mailto:formation@dalibo.com

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 :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, 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, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cette formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

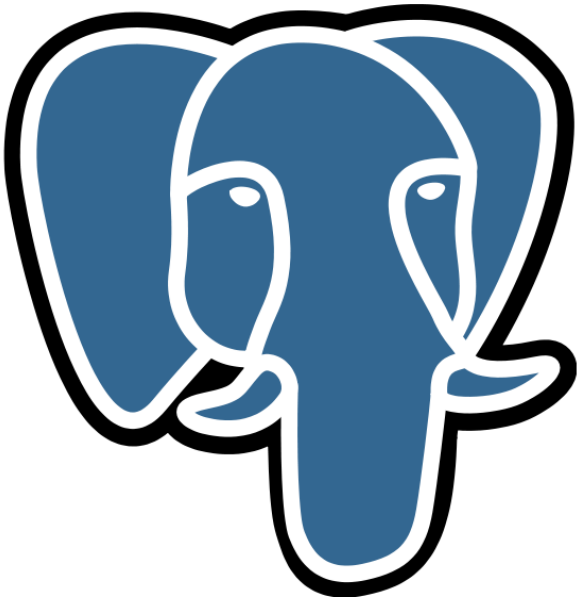
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ PostgreSQL : historique & communauté



1.1 PRÉAMBULE

- Quelle histoire !
  - parmi les plus vieux logiciels libres
  - et les plus sophistiqués
- Souvent cité comme exemple
  - qualité du code
  - indépendance des développeurs
  - réactivité de la communauté

L'histoire de PostgreSQL est longue, riche et passionnante. Au côté des projets libres Apache et Linux, PostgreSQL est l'un des plus vieux logiciels libres en activité et fait partie des SGBD les plus sophistiqués à l'heure actuelle.

Au sein des différentes communautés libres, PostgreSQL est souvent cité comme exemple à différents niveaux :

- qualité du code ;
- indépendance des développeurs et gouvernance du projet ;
- réactivité de la communauté ;
- stabilité et puissance du logiciel.

Tous ces atouts font que PostgreSQL est désormais reconnu et adopté par des milliers de grandes sociétés de par le monde.

1.1.1 Au menu

- Origines et historique du projet
- Versions et feuille de route
- Projets satellites
- Sponsors et références
- La communauté

Cette première partie est un tour d'horizon pour découvrir les multiples facettes du système de gestion de base de données libre PostgreSQL.

Les deux premières parties expliquent la genèse du projet et détaillent les différences entre les versions successives du logiciel. PostgreSQL est un des plus vieux logiciels libres ! Comprendre son histoire permet de mieux réaliser le chemin parcouru et les raisons de son succès.

Nous verrons ensuite certains projets satellites et nous listerons plusieurs utilisateurs renommés et cas d'utilisations remarquables.

Enfin, nous terminerons par une découverte de la communauté.

1.2 UN PEU D'HISTOIRE...

- La licence
- L'origine du nom
- Les origines du projet
- Les principes

1.2.1 Licence

- Licence PostgreSQL
  - libre (BSD/MIT)
  - https://www.postgresql.org/about/licence/
- Droit, sans coûts de licence, de :
  - utiliser, copier, modifier, distribuer (et même revendre)
- Reconnue par l'Open Source Initiative
- Utilisée par un grand nombre de projets de l'écosystème

PostgreSQL est distribué sous une licence spécifique, la licence PostgreSQL[1], combinant la licence BSD et la licence MIT. Elle est reconnue comme une licence libre par l'Open Source Initiative[2].

Cette licence vous donne le droit de distribuer PostgreSQL, de l'installer, de le modifier… et même de le vendre. Certaines sociétés, comme EnterpriseDB et PostgresPro, produisent leur version propriétaire de PostgreSQL de cette façon.

PostgreSQL n'est pas pour autant complètement gratuit : il peut y avoir des frais et du temps de formation, des projets de migration depuis d'autres bases, ou d'intégration des différents outils périphériques indispensables en production.

Cette licence a ensuite été reprise par de nombreux projets de la communauté : pgAdmin, pgCluu, pgstat, etc.

[1] https://www.postgresql.org/about/licence/
[2] https://opensource.org/licenses/PostgreSQL

1.2.2 PostgreSQL ?!?!

- 1985 : Michael Stonebraker recode Ingres
- post « ingres » ⇒ post~~in~~gres ⇒ postgres
- postgres ⇒ PostgreSQL

PostgreSQL a une origine universitaire.

L'origine du nom PostgreSQL remonte au système de gestion de base de données Ingres, développé à l'université de Berkeley par Michael Stonebraker. En 1985, il prend la décision de reprendre le développement à partir de zéro et nomme ce nouveau logiciel **Postgres**, comme raccourci de post-Ingres.

En 1995, avec l'ajout du support du langage SQL, Postgres fut renommé **Postgres95** puis **PostgreSQL**.

Aujourd'hui, le nom officiel est « PostgreSQL » (prononcé « post - gresse - Q - L »). Cependant, le nom « Postgres » reste accepté.

Pour aller plus loin :

- Fil de discussion sur les listes de discussion[3] ;
- Article sur le wiki officiel[4].

1.2.3 Principes fondateurs

- Sécurité des données (ACID)
- Respect des normes (ISO SQL)
- Portabilité
- Fonctionnalités intéressant le plus grand nombre
- Performances
  - si pas de péril pour les données
- Simplicité du code
- Documentation
- Tests fonctionnels

[3] https://archives.postgresql.org/pgsql-advocacy/2007-11/msg00109.php
[4] https://wiki.postgresql.org/wiki/Postgres

Depuis son origine, PostgreSQL a toujours privilégié la stabilité et le respect des standards plutôt que les performances.

La sécurité des données est un point essentiel. En premier lieu, un utilisateur doit être certain qu'à partir du moment où il a exécuté l'ordre `COMMIT` d'une transaction, les données modifiées relatives à cette transaction se trouvent bien sur disque et que même un crash ne pourra pas les faire disparaître. PostgreSQL est très attaché à ce concept et fait son possible pour forcer le système d'exploitation à ne pas conserver les données en cache, mais à les écrire sur disque dès l'arrivée d'un `COMMIT`.

L'intégrité des données, et le respect des contraintes fonctionnelles et techniques qui leur sont imposées, doivent également être garanties par le moteur à tout moment, quoi que fasse l'utilisateur. Par exemple, insérer 1000 caractères dans un champ contraint à 200 caractères maximum doit mener à une erreur explicite et non à l'insertion des 200 premiers caractères en oubliant les autres, comme cela s'est vu ailleurs. De même, un champ avec le type `date` ne contiendra jamais un 31 février, et un champ `NOT NULL` ne sera jamais vide. Tout ceci est formalisé par les propriétés (ACID[5]) que possèdent toute bonne base de données relationnelle.

Le respect des normes est un autre principe au cœur du projet. Les développeurs de PostgreSQL cherchent à coller à la norme SQL[6] le plus possible. PostgreSQL n'est pas compatible à cette norme à 100 %, aucun moteur ne l'est, mais il cherche à s'en approcher. Tout nouvel ajout d'une syntaxe ne sera accepté que si la syntaxe de la norme est ajoutée. Des extensions sont acceptées pour différentes raisons (performances, fonctionnalités en avance sur le comité de la norme, facilité de transition d'un moteur de bases de données à un autre) mais si une fonctionnalité existe dans la norme, une syntaxe différente ne peut être acceptée que si la syntaxe de la norme est elle-aussi présente.

La portabilité est importante : PostgreSQL tourne sur l'essentiel des systèmes d'exploitation : Linux (plate-forme à privilégier), macOS, les Unix propriétaires, Windows… Tout est fait pour que cela soit encore le cas dans le futur.

Ajouter des fonctionnalités est évidemment l'un des buts des développeurs de PostgreSQL. Cependant, comme il s'agit d'un projet libre, rien n'empêche un développeur de proposer une fonctionnalité, de la faire intégrer, puis de disparaître laissant aux autres la responsabilité de la corriger le cas échéant. Comme le nombre de développeurs de PostgreSQL est restreint, il est important que les fonctionnalités ajoutées soient vraiment utiles au plus grand nombre pour justifier le coût potentiel du débogage. Donc ne sont ajoutées dans PostgreSQL que ce qui est vraiment le cœur du moteur de bases de données et que ce qui sera utilisé vraiment par le plus grand nombre. Une fonctionnalité qui ne sert que une à deux personnes aura très peu de chances d'être intégrée. (Le système des extensions offre une élégante solution aux problèmes très spécifiques.)

Les performances ne viennent qu'après tout ça. En effet, rien ne sert d'avoir une modification du code qui permet de gagner énormément en performances si cela met en péril le stockage des données. Cependant, les performances de PostgreSQL sont excellentes et le moteur permet d'opérer des centaines de tables, des milliards de lignes pour plusieurs téraoctets de données, sur une seule instance, pour peu que la configuration matérielle soit correctement dimensionnée.

La simplicité du code est un point important. Le code est relu scrupuleusement par différents contributeurs pour s'assurer qu'il est facile à lire et à comprendre. En effet, cela facilitera le débogage plus

[5] https://dali.bo/a2_html#ACID
[6] https://fr.wikipedia.org/wiki/Structured_Query_Language

tard si cela devient nécessaire.

La documentation est là aussi un point essentiel dans l'admission d'une nouvelle fonctionnalité. En effet, sans documentation, peu de personnes pourront connaître cette fonctionnalité. Très peu sauront exactement ce qu'elle est supposée faire, et il serait donc très difficile de déduire si un problème particulier est un manque actuel de cette fonctionnalité ou un bug.

Enfin, les tests fonctionnels suite à la compilation sont un prérequis depuis quelques années. Ils permettent de tester les fonctionnalités proposées et de ne pas retomber dans des bugs déjà corrigés.

Tous ces points sont vérifiés à chaque relecture d'un patch (nouvelle fonctionnalité ou correction).

1.2.4 Origines

- Années 1970 : Michael Stonebraker développe **Ingres** à Berkeley
- 1985 : **Postgres** succède à Ingres
- 1995 : Ajout du langage SQL
- 1996 : Libération du code : Postgres devient **PostgreSQL**
- 1996 : Création du PostgreSQL Global Development Group

L'histoire de PostgreSQL remonte au système de gestion de base de données Ingres[7], développé dès 1973 à l'Université de Berkeley (Californie) par Michael Stonebraker[8].

Lorsque ce dernier décide en 1985 de recommencer le développement de zéro, il nomme le logiciel Postgres, comme raccourci de post-Ingres. Des versions commencent à être diffusées en 1989, puis commercialisées.

Postgres utilise alors un langage dérivé de QUEL[9], hérité d'Ingres, nommé POSTQUEL[10]. En 1995, lors du remplacement par le langage SQL par Andrew Yu and Jolly Chen, deux étudiants de Berkeley, Postgres est renommé Postgres95.

En 1996, Bruce Momijan et Marc Fournier convainquent l'Université de Berkeley de libérer complètement le code source. Est alors fondé le PGDG (*PostgreSQL Development Group*), entité informelle — encore aujourd'hui — regroupant l'ensemble des contributeurs. Le développement continue donc hors tutelle académique (et sans son fondateur historique Michael Stonebraker) : PostgreSQL 6.0 est publié début 1997.

Plus d'informations :

- Page associée sur le site officiel[11].

[7] https://en.wikipedia.org/wiki/Ingres_(database)
[8] https://en.wikipedia.org/wiki/Michael_Stonebraker
[9] https://en.wikipedia.org/wiki/QUEL_query_languages
[10] La trace se retrouve encore dans le nom de la librairie C pour les clients, la **libpq**.
[11] https://www.postgresql.org/docs/current/static/history.html

1.2.5 Apparition de la communauté internationale

- ~ 2000: Communauté japonaise (JPUG)
- 2004 : PostgreSQLFr
- 2006 : SPI
- 2007 : Communauté italienne
- 2008 : PostgreSQL Europe et US
- 2009 : Boom des PGDay
- 2011 : Postgres Community Association of Canada
- 2017 : Community Guidelines
- …et ça continue

Les années 2000 voient l'apparition de communautés locales organisées autour d'association ou de manière informelle. Chaque communauté organise la promotion, la diffusion d'information et l'entraide à son propre niveau.

En 2000 apparaît la communauté japonaise (JPUG). Elle dispose déjà d'un grand groupe, capable de réaliser des conférences chaque année, d'éditer des livres et des magazines. Elle compte, au dernier recensement connu, plus de 3000 membres.

En 2004 naît l'association française (loi 1901) appelée PostgreSQL Fr. Cette association a pour but de fournir un cadre légal pour pouvoir participer à certains événements comme Solutions Linux, les RMLL ou d'en organiser comme le pgDay.fr (qui a déjà eu lieu à Toulouse, Nantes, Lyon, Toulon, Marseille). Elle permet aussi de récolter des fonds pour aider à la promotion de PostgreSQL.

En 2006, le PGDG intègre Software in the Public Interest, Inc.(SPI)[12], une organisation à but non lucratif chargée de collecter et redistribuer des financements. Elle a été créée à l'initiative de Debian et dispose aussi de membres comme LibreOffice.org.

Jusque là, les événements liés à PostgreSQL apparaissaient plutôt en marge de manifestations, congrès, réunions… plus généralistes. En 2008, douze ans après la création du projet, des associations d'utilisateurs apparaissent pour soutenir, promouvoir et développer PostgreSQL à l'échelle internationale. PostgreSQL UK organise une journée de conférences à Londres, PostgreSQL Fr en organise une à Toulouse. Des « sur-groupes » apparaissent aussi pour aider les groupes locaux : PGUS rassemble les différents groupes américains, plutôt organisés géographiquement, par État ou grande ville. De même, en Europe, est fondée PostgreSQL Europe, association chargée d'aider les utilisateurs de PostgreSQL souhaitant mettre en place des événements. Son principal travail est l'organisation d'un événement majeur en Europe tous les ans : pgconf.eu[13], d'abord à Paris en 2009, puis dans divers pays d'Europe jusque Milan en 2019. Cependant, elle aide aussi les communautés allemande, française et suédoise à monter leur propre événement (respectivement PGConf.DE[14], pgDay Paris[15]

[12] https://fr.wikipedia.org/wiki/Software_in_the_Public_Interest
[13] https://pgconf.eu/
[14] https://pgconf.de/
[15] https://pgday.paris/

et Nordic PGday[16]).

Dès 2010, nous dénombrons plus d'une conférence par mois consacrée uniquement à PostgreSQL dans le monde. Ce mouvement n'est pas prêt de s'arrêter :

- communauté japonaise[17] ;
- communauté francophone[18] ;
- communauté italienne[19] ;
- communauté européenne[20] ;
- communauté américaine (États-Unis)[21].

En 2011, l'association Postgres Community Association of Canada voit le jour[22]. Elle est créée par quelques membres de la *Core Team* pour gérer le nom déposé PostgreSQL, le logo, le nom de domaine sur Internet, etc.

Vu l'émergence de nombreuses communautés internationales, la communauté a décidé d'écrire quelques règles pour ces communautés. Il s'agit des *Community Guidelines*, apparues en 2017, et disponibles sur le site officiel[23].

1.2.6 Progression du code

- 1,9 millions de lignes
  - ¼ de commentaires
  - le reste surtout en C
- Nombres de commit par mois :

[16] https://nordicpgday.org/
[17] https://www.postgresql.jp/
[18] https://www.postgresql.fr/
[19] https://www.itpug.org/
[20] https://www.postgresql.eu/
[21] https://www.postgresql.us/
[22] https://www.postgresql.org/message-id/4DC440BE.5040104%40agliodbs.com%3E
[23] https://www.postgresql.org/community/recognition/

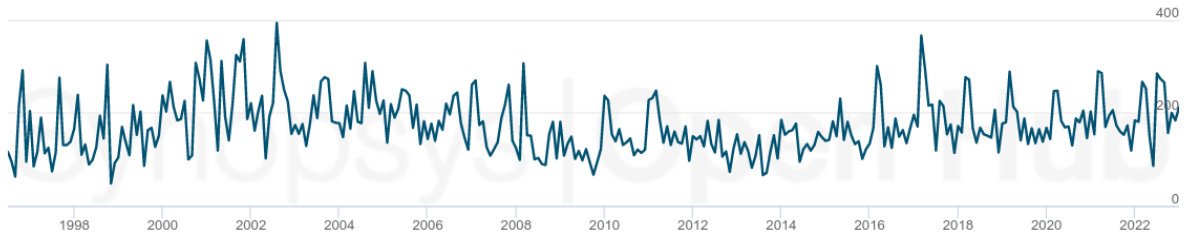


Figure 1/ .1: Évolution du nombre de commit dans le dépôt PostgreSQL

Le dépôt principal de PostgreSQL a été un dépôt CVS, passé depuis à git[24]. Il est en accès public en lecture.

Le graphe ci-dessus (source[25]) représente l'évolution du nombre de commit dans les sources de PostgreSQL. L'activité ne se dément pas. Le plus intéressant est certainement de noter que l'évolution est constante. Il n'y a pas de gros pic, ni dans un sens, ni dans l'autre.

Fin 2024, le code de PostgreSQL contient 1,9 millions de lignes,dont un quart de commentaires[26]. Ce ratio montre que le code est très commenté, très documenté, facile à lire, et donc pratique à déboguer. Et le ratio ne change pas au fil des ans. Le code est essentiellement en C, avec un peu de SQL et de Perl. pour environ 200 développeurs actifs, à environ 200 commits par mois ces dernières années. Et ce, sans compter les contributions aux outils périphériques.

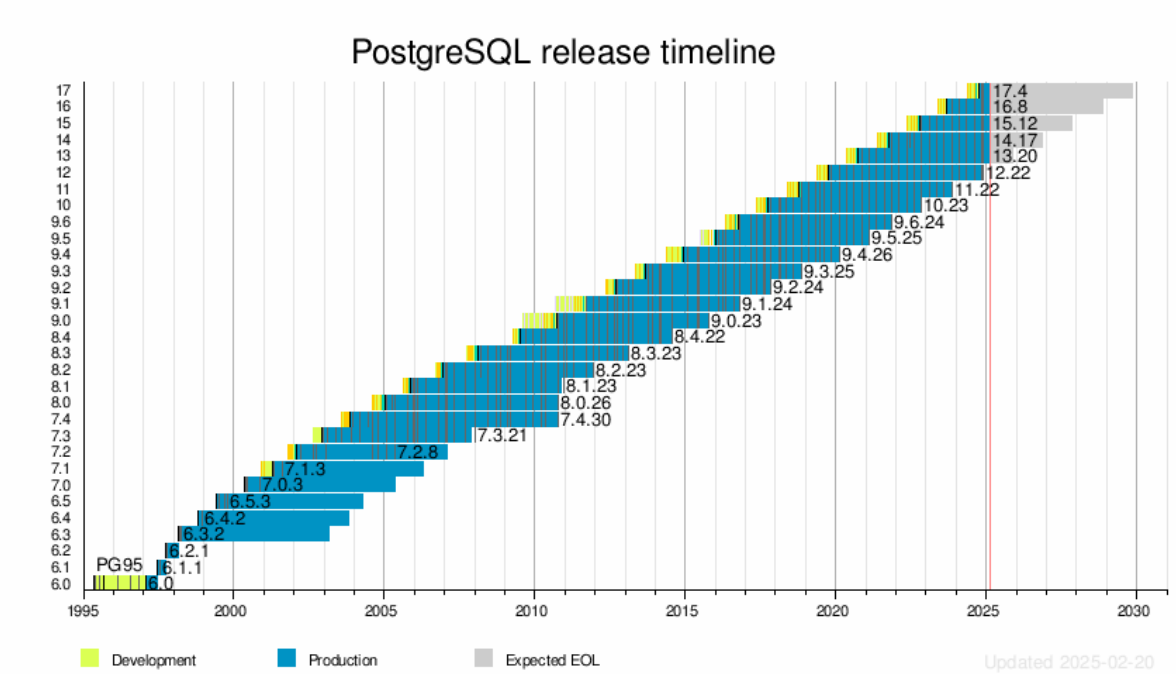
[24] https://git.postgresql.org/
[25] https://www.openhub.net/p/postgres/analyses/latest/
[26] https://www.openhub.net/p/postgres/analyses/latest/languages_summary

1.3 LES VERSIONS DE POSTGRESQL

Quelle version utiliser ?

- Historique
- Numérotation
- Mises à jour mineures et majeures
- Les versions courantes
- Quelle version en production ?
- Les forks & dérivés

1.3.1 Historique



Sources : page Wikipédia de PostgreSQL[27] et PostgreSQL Versioning Policy[28]

[27] https://en.wikipedia.org/wiki/PostgreSQL
[28] https://www.postgresql.org/support/versioning/

1.3.2 Versions & fonctionnalités

- 1996 : v6.0 -> première version publiée
- 2003 : v7.4 -> première version *réellement* stable
- 2005 : v8.0 -> arrivée sur Windows
- 2008 : v8.3 -> performances et fonctionnalités, organisation (commitfests)
- 2010 : v9.0 -> réplication physique
- 2016 : v9.6 -> parallélisation
- 2017 : v10 -> réplication logique, partitionnement déclaratif
- 2024 : v17 -> performances, fonctionnalités, administration…

La version 7.4 est la première version réellement stable. La gestion des journaux de transactions a été nettement améliorée, et de nombreuses optimisations ont été apportées au moteur.

La version 8.0 marque l'entrée tant attendue de PostgreSQL dans le marché des SGDB de haut niveau, en apportant des fonctionnalités telles que les tablespaces, les routines stockées en Java, le *Point In Time Recovery*, ainsi qu'une version native pour Windows.

La version 8.3 se focalise sur les performances et les nouvelles fonctionnalités. C'est aussi la version qui a causé un changement important dans l'organisation du développement pour encourager les contributions : gestion des commitfests, création de l'outil web associé, etc.

Les versions 9.x sont axées réplication physique. La 9.0 intègre un système de réplication asynchrone asymétrique. La version 9.1 ajoute une réplication synchrone et améliore de nombreux points sur la réplication (notamment pour la partie administration et supervision). La version 9.2 apporte la réplication en cascade. La 9.3 et la 9.4 ajoutent quelques améliorations supplémentaires. La version 9.4 intègre surtout les premières briques pour l'intégration de la réplication logique dans PostgreSQL. La version 9.6 apporte la parallélisation, ce qui était attendu par de nombreux utilisateurs.

La version 10 propose beaucoup de nouveautés, comme une amélioration nette de la parallélisation et du partitionnement (le partitionnement déclaratif complète l'ancien partitionnement par héritage), mais aussi l'ajout de la réplication logique.

Les améliorations des versions 11 à 17 sont plus incrémentales, et portent sur tous les plans. Le partitionnement déclaratif et la réplication logique sont progressivement améliorés, en performances comme en facilité de développement et maintenance. Les performances s'améliorent encore grâce à la compilation *Just In Time*, la parallélisation de plus en plus d'opérations, les index couvrants, l'affinement des statistiques. La facilité d'administration s'améliore : nouvelles vues système, rôles supplémentaires pour réduire l'utilisation du superutilisateur, outillage pour la réplication logique et la sauvegarde physique, activation des sommes de contrôle sur une instance existante.

Il est toujours possible de télécharger les sources depuis la version 1.0 jusqu'à la version courante sur postgresql.org[29].

[29] https://www.postgresql.org/ftp/source/

1.3.3 Numérotation

- Version récentes (10+)
  - X : version majeure (10, 11, … 17)
  - X.Y : version mineure (14.8, 17.4)
- Avant la version 10 (toutes périmées !)
  - X.Y : version majeure (9.4, 9.6)
  - X.Y.Z : version mineure (9.6.24)

Une version majeure apporte de nouvelles fonctionnalités, des changements de comportement, etc. Une version majeure sort généralement tous les ans à l'automne. Une migration majeure peut se faire directement depuis n'importe quelle version précédente. Le numéro est incrémenté chaque année (version 12 en 2019, version 16 en 2023).

Une version mineure ne comporte que des corrections de bugs ou de failles de sécurité. Les publications de versions mineures sont plus fréquentes que celles de versions majeures, avec un rythme de sortie trimestriel, sauf bug majeur ou faille de sécurité. Chaque bug est corrigé dans toutes les versions stables alors maintenues par le projet. Le numéro d'une version mineure porte deux nombres. Par exemple, en février 2025 sont sorties les versions 17.4, 16.8, 15.12, 14.17, et 13.20. La version 12 n'étant plus supportée, il n'y aura pas d'autre version mineure sur cette branche après la 12.22.

Avant la version 10, les versions majeures annuelles portaient deux chiffres : 9.0 en 2010, 9.6 en 2016. Les mineures avaient un numéro de plus (par exemple 9.6.24). Cela a entraîné quelques confusions, d'où le changement de numérotation. Il va sans dire que ces versions sont totalement périmées et ne sont plus supportées, mais beaucoup continuent de fonctionner.

1.3.4 Mises à jour mineure

De M.m à M.m+n :

- En général chaque trimestre
- Et sans souci
  - *Release notes*
  - tests
  - mise à jour des binaires
  - redémarrage

Une mise à jour mineure consiste à mettre à jour vers une nouvelle version de la même branche majeure, par exemple de 14.8 à 14.9, ou de 17.3 à 17.4 (mais pas d'une version 14.x à une version 17.x). Les mises à jour des versions mineures sont cumulatives : vous pouvez mettre à jour une instance 17.0 en version 17.4 sans passer par les versions 17.1 à 17.3 intermédiaires.

En général, les mises à jour mineures se font sans souci et ne nécessitent que le remplacement des binaires et un redémarrage (et donc une courte interruption). Les fichiers de données conservent le même format. Des opérations supplémentaires sont possibles mais rarissimes. Mais comme pour toute mise à jour, il convient d'être prudent sur d'éventuels effets de bord. En particulier, il faudra lire les *Release Notes* et, si possible, effectuer les tests ailleurs qu'en production.

1.3.5 Versions courantes

- 1 version majeure par an
  - maintenue 5 ans
- Dernières mises à jour mineures
  - https://www.postgresql.org/support/versioning/ (au 20 février 2025) :
  - version 13.20
  - version 14.17
  - version 15.12
  - version 16.8
  - version 17.4
- Prochaine sortie de versions mineures prévue : 8 mai 2025

La philosophie générale des développeurs de PostgreSQL peut se résumer ainsi :

> « Notre politique se base sur la qualité, pas sur les dates de sortie. »

Toutefois, même si cette philosophie reste très présente parmi les développeurs, en pratique une version stable majeure paraît tous les ans, habituellement à l'automne. Pour ne pas sacrifier la qualité des versions, toute fonctionnalité supposée insuffisamment stable est repoussée à la version suivante si des correctifs satisfaisants ne peuvent être trouvés à temps. Il est aussi arrivé que la sortie de la version majeure soit repoussée de quelques semaines à cause de bugs inacceptables mais corrigeables rapidement.

La tendance actuelle est de garantir un support pour chaque version majeure pendant une durée minimale de 5 ans. Ainsi ne sont plus supportées les versions 11 depuis novembre 2023 et 12 depuis novembre 2024. Il n'y aura pour elles plus aucune mise à jour mineure, donc plus de correction de bug ou de faille de sécurité. Le support de la version 13 s'arrêtera en novembre 2025. Le support de la dernière version majeure, la 17, devrait durer jusqu'en 2029.

Pour plus de détails :

- Politique de versionnement[30] ;
- Dates prévues des futures versions[31].

1.3.6 Versions 9.4 à 11

- `jsonb`
- Row Level Security
- Index BRIN, bloom
- Fonctions OLAP
- Parallélisation
- SQL/MED : accès distants
- Réplication logique
- Partitionnement déclaratif
- Réduction des inconvénients de MVCC
- JIT
- Index couvrants

[30] https://www.postgresql.org/support/versioning/
[31] https://www.postgresql.org/developer/roadmap/

Ces versions ne sont plus supportées !

La version 9.4 (décembre 2014) a apporté le type `jsonb`, binaire, facilitant la manipulation des objets en JSON.

La 9.5 parue en janvier 2016 apportait notamment les index BRIN et des possibilités OLAP plus avancées que `GROUP BY`. Pour plus de détails :

- Page officielle des nouveautés de la version 9.5[32] ;
- Workshop Dalibo sur la version 9.5[33].

En 9.6, la nouvelle fonctionnalité majeure est certainement la parallélisation de certaines parties de l'exécution d'une requête. Le `VACUUM FREEZE` devient beaucoup moins gênant.

- Page officielle des nouveautés de la version 9.6[34] ;
- Workshop Dalibo sur la version 9.6[35]

En version 10, les fonctionnalités majeures sont l'intégration de la réplication logique et le partitionnement déclaratif, longtemps attendus, améliorés dans les versions suivantes. Sont notables aussi les tables de transition ou les améliorations sur la parallélisation.

La version 10 a aussi été l'occasion de renommer plusieurs répertoires et fonctions système, et même des outils. Attention donc si vous rencontrez des requêtes ou des scripts adaptés aux versions précédentes. Entre autres :

- le répertoire `pg_xlog` est devenu `pg_wal` ;
- le répertoire `pg_clog` est devenu `pg_xact` ;
- dans les noms de fonctions, `xlog` a été remplacé par `wal` (par exemple `pg_switch_xlog` est devenue `pg_switch_wal`) ;
- toujours dans les fonctions, `location` a été remplacé par `lsn`.

Pour plus de détails :

- Page officielle des nouveautés de la version 10[36] ;
- Workshop Dalibo sur la version 10[37]

La version 11 (octobre 2018) améliore le partitionnement de la version 10, le parallélisme, la réplication logique… et de nombreux autres points. Elle comprend aussi une première version du JIT (*Just In Time compilation*) pour accélérer les requêtes les plus lourdes en CPU, ou encore les index couvrants.

Pour plus de détails :

[32] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.5
[33] https://kb.dalibo.com/conferences/nouveautes_de_postgresql_9.5
[34] https://wiki.postgresql.org/wiki/NewIn96
[35] https://dali.bo/workshop96_pdf
[36] https://wiki.postgresql.org/wiki/New_in_postgres_10
[37] https://dali.bo/workshop10_pdf

- Page officielle des nouveautés de la version 11[38] ;
- Workshop Dalibo sur la version 11[39]

1.3.7 Version 12

- Octobre 2019 - Novembre 2024 (n'est plus supportée !)
- Amélioration du partitionnement déclaratif
- Amélioration des performances
  - sur la gestion des index
  - sur les CTE (option MATERIALIZED)
- Colonnes générées
- Nouvelles vues de visualisation de la progression des commandes
- Refonte de la configuration de la réplication

La version 12 sortie en 2019 n'est plus supportée depuis novembre 2024. Elle améliore de nouveau le partitionnement et elle fait surtout un grand pas au niveau des performances et de la supervision.

Le fichier `recovery.conf` (pour la réplication et les restaurations physiques) est intégré au fichier `postgresql.conf`. Une source fréquente de ralentissement disparaît, avec l'intégration des CTE (clauses `WITH`) dans la requête principale. Des colonnes d'une table peuvent être automatiquement générées à partir d'autres colonnes.

Pour plus de détails :

- PostgreSQL 12 Press Kit[40]
- Workshop Dalibo sur la version 12[41]

[38] https://www.postgresql.org/docs/11/release-11.html
[39] https://dali.bo/workshop11_pdf
[40] https://www.postgresql.org/about/press/presskit12/fr/
[41] https://dali.bo/workshop12_pdf

1.3.8 Version 13

- Septembre 2020 - Septembre 2025
- Améliorations :
  - partitionnement déclaratif
  - réplication logique
- Amélioration des performances :
  - index B-tree, objet statistique, tri et agrégat
- Amélioration de l'autovacuum et du VACUUM :
  - gestion complète des tables en insertion seule
  - traitement parallélisé des index lors d'un VACUUM
- Amélioration des sauvegardes :
  - génération d'un fichier manifeste, outil pg_verifybackup
- Nouvelles vues de progression de commandes :
  - `pg_stat_progress_basebackup`, `pg_stat_progress_analyze`

La version 13 sortie en 2020 ne sera plus supportée dès novembre 2025. Elle est remplie de nombreuses petites améliorations sur différents domaines : partitionnement déclaratif, autovacuum, sauvegarde, etc. Les performances sont aussi améliorées grâce à un gros travail sur l'optimiseur, ou la réduction notable de la taille de certains index.

Pour plus de détails :

- PostgreSQL 13 Press Kit[42]
- Workshop Dalibo sur la version 13[43]

[42] https://www.postgresql.org/about/press/presskit13/fr/
[43] https://dali.bo/workshop13_pdf

1.3.9 Version 14

- Septembre 2021 - Novembre 2026
- Nouvelles vues système & améliorations
  - `pg_stat_progress_copy`, `pg_stat_wal`, `pg_lock.waitstart`, `query_id`...
- Lecture asynchrone des tables distantes
- Paramétrage par défaut adapté aux machines plus récentes
- Améliorations diverses :
  - réplications physique et logique
  - quelques facilités de syntaxe (triggers, tableaux en PL/pgSQL)
- Performances :
  - connexions en lecture seule plus nombreuses
  - index…

La version 14 est remplie de nombreuses petites améliorations sur différents domaines listés ci-dessus.

Pour plus de détails :

- PostgreSQL 14 Press Kit[44]
- Workshop Dalibo sur la version 14[45]

[44] https://www.postgresql.org/about/press/presskit14/fr/
[45] https://dali.bo/workshop14_pdf

1.3.10 Version 15

- Octobre 2022 - Novembre 2027
- Nombreuses améliorations incrémentales
  - dont en réplication logique
- Commande `MERGE`
- Performances :
  - `DISTINCT` parallélisable
  - pg_dump & sauvegardes, recovery, partitionnement
- Changements notables :
  - `public` n'est plus accessible en écriture à tous
  - sauvegarde PITR exclusive disparaît

La version 15 est également une mise à jour sans grande nouveauté fracassante, mais contenant de très nombreuses améliorations et optimisations sur de nombreux plans, comme par exemple la commande `MERGE` ou l'accélération du *recovery* sur une reprise de restauration.

Signalons deux changements de comportement importants : pour renforcer la sécurité, le schéma `public` n'est plus accessible en écriture par défaut à tous les utilisateurs ; et la sauvegarde physique en mode exclusif n'est plus disponible.

Pour plus de détails :

- PostgreSQL 15 Press Kit[46]
- Workshop Dalibo sur la version 15[47]

[46] https://www.postgresql.org/about/press/presskit15/fr/
[47] https://dali.bo/workshop15_pdf

1.3.11 Version 16

- Septembre 2023 - Novembre 2028
- Plus de tris incrémentaux (`DISTINCT`…)
- Réplication logique depuis un secondaire
- Expressions régulières dans `pg_hba.conf`
- Vues systèmes améliorées : `pg_stat_io`…
- Compression lz4 ou zstd pour pg_dump
- Optimisation et améliorations diverses (parallélisation…)

La version 16 est parue le 14 septembre 2023. Là encore, les améliorations sont incrémentales.

On notera la possibilité de rajouter des expressions régulières dans `pg_hba.conf` pour faciliter la gestion des accès. En réplication logique, un abonnement peut se faire auprès d'un serveur secondaire. La réplication logique peut devenir parallélisable. `pg_dump` acquiert des algorithmes de compression plus modernes. Le travail de parallélisation de nouveaux nœuds se poursuit. Une nouvelle vue de suivi des entrées-sorties apparaît : `pg_stat_io`. Le `VACUUM` peut être accéléré en lui permettant d'utiliser plus de mémoire dans les *shared buffers*.

Pour plus de détails :

- PostgreSQL 16 Press Kit[48]
- Workshop Dalibo sur la version 16[49]
- blog Dalibo sur la sortie de la version 16[50] ;
- présentation de Magnus Hagander au PGDay UK 2023[51]

[48] https://www.postgresql.org/about/press/presskit16/fr/
[49] https://dali.bo/workshop16_pdf
[50] https://blog.dalibo.com/2023/09/15/release_postgresql_16.html
[51] https://www.hagander.net/talks/PostgreSQL%2016.pdf

1.3.12 Version 17

- Septembre 2024 - Novembre 2029
- Améliorations du `VACUUM`
- Planificateur : `IN` et CTE
- BRIN : création parallélisée
- `JSON_TABLE`…
- Sauvegarde incrémentale (`pg_basebackup` + `pg_combinebackup`)
- Améliorations en réplication logique (`pg_createsubscriber`)
- `pg_dump --filter`
- Imports (`COPY` peut rejeter des lignes, `MERGE`)
- Améliorations diverses

La version 17, parue le 26 septembre 2024, contient quelques nouveautés intéressantes parmi ses 2635 commits.

Le `VACUUM` est encore amélioré en terme de sélectivité, de suivi, de mémoire maximum utilisable, tout en réduisant celle consommée. `GRANT MAINTAIN` et `pg_maintain` autorisent son lancement sur une table sans en être propriétaire. Le planificateur comprend quelques améliorations (pour les `IN` et les CTE entre autres). La création des index BRIN peut être parallélisée. Le chargement en masse et la transformation de données via `MERGE` et `COPY` ont également évolué en performances et fonctionnalités (`MERGE … RETURNING`). `COPY` peut enfin ignorer quelques lignes en erreur. Le travail de fond sur le partitionnement continue, avec le support natif des contraintes d'exclusions et des colonnes d'identité.

PostgreSQL inclut désormais un nouveau fournisseur de collation immutable (`builtin`), en plus de la collation de l'OS et de ICU. JSON et JSONPath voient apparaître de nouvelles fonctions (notamment `JSON_TABLE`).

La réplication logique permet enfin la gestion du *failover* du primaire, et la possibilité de créer un réplica logique via la réplication physique, avec l'outil `pg_createsubscriber`. `pg_basebackup` permet enfin de créer des sauvegardes incrémentales à recombiner avec `pg_combinebackup`. `pg_dump` a une clause `--filter` plus flexible.

Quelques nouveaux paramètres apparaissent, comme `transaction_timeout`, `allow_alter_system`, ou ceux permettant de gérer quelques caches spécifiques.

En supervision, entre autres, des éléments de `pg_stat_bgwriter` sont remplacés par d'autres dans la nouvelle vue `pg_stat_checkpointer`.

Pour plus de détails :

- workshop sur la version 17[52] ;
- blog Dalibo sur la sortie de la version 17[53] ;
- présentation de Magnus Hagander au PGDay UK 2024[54]

1.3.13 Petit résumé

- Versions 7.x :
  - fondations
  - durabilité
- Versions 8.x :
  - fonctionnalités
  - performances
- Versions 9.x :
  - réplication physique
  - extensibilité
- Versions 10 à 17 :
  - réplication logique
  - parallélisation
  - partitionnement
  - maintenabilité

Si nous essayons de voir cela avec de grosses mailles, les développements des versions 7 ciblaient les fondations d'un moteur de bases de données stable et durable. Ceux des versions 8 avaient pour but de rattraper les gros acteurs du marché en fonctionnalités et en performances. Enfin, pour les versions 9, on est plutôt sur la réplication et l'extensibilité.

La version 10 est une version mémorable grâce à la réplication logique, la parallélisation et le partitionnement. Les versions 11 à 17 améliorent ces deux points, entre mille autres améliorations en différents points du moteur, notamment les performances et la facilité d'administration.

[52] https://dali.bo/workshop17_pdf
[53] https://blog.dalibo.com/2024/10/11/release_postgresql_17.html
[54] https://www.hagander.net/talks/PostgreSQL%2017.pdf

1.3.14 Quelle version utiliser en production ?

Au premier semestre 2025 :

- 12 et inférieures
  - **Danger !**
  - planifier une migration urgemment !
- 13, 14, 15, 16
  - OK pour la production
  - ne pas oublier les mises à jour mineures
- Nouvelles installations
  - 17
- Nouveaux développements
  - 17
- https://www.postgresql.org/about/featurematrix

La version 12 n'est plus supportée depuis novembre 2024.

Si vous avez une version 12 ou inférieure, planifiez le plus rapidement possible une migration vers une version plus récente, comme la 17.

Les versions non supportées fonctionneront toujours aussi bien, mais il n'y aura plus de correction de bug, y compris pour les failles de sécurité ! Si vous utilisez ces versions, il est impératif d'étudier une migration de version dès que possible.

Les versions 13 à 16 restent recommandables pour une production. Le plus important est d'appliquer les mises à jour correctives.

La version 17 est conseillée pour les nouvelles installations en production.

Par expérience, quand une version x.0 paraît à l'automne, elle est généralement stable. Nombre de DBA préfèrent prudemment attendre les premières mises à jour mineures (en novembre généralement) pour la mise en production. Cette prudence est à mettre en balance avec l'intérêt pour les nouvelles fonctionnalités.

Le développement de la version 18 a déjà débuté. Les paquets du PGDG contiennent déjà une version 18 utilisable à titre de test (mais attention ! il n'est pas garanti que toutes les fonctionnalités ajoutées soient finalement conservées). La version 18 bêta est prévue pour mai 2025[55], et il est probable que la 18.0 paraîtra à l'automne 2025.

Pour les nouveaux développements, démarrez en version 17.

Pour plus de détails sur les fonctionnalités de chaque version, voir le tableau comparatif des versions[56].

1.3.15 Versions dérivées / Forks

Entre de nombreux autres :

- Compatibilité Oracle :
  - EnterpriseDB
- Data warehouse :
  - Greenplum, Netezza
- Forks :
  - Amazon RedShift, Aurora, Neon…
  - attention : support, extensions…
- Extensions :
  - Citus
  - timescaledb
- Packages avec des outils & support
- Bases compatibles

Il existe de nombreuses versions dérivées de PostgreSQL. Elles sont en général destinées à des cas d'utilisation très spécifiques et offrent des fonctionnalités non proposées par la version communautaire. Leur code est souvent fermé et nécessite l'acquisition d'une licence payante. La licence de PostgreSQL permet cela, et le phénomène existait déjà dès les années 1990 avec divers produits commerciaux comme Illustra.

Modifier le code de PostgreSQL a plusieurs conséquences négatives. Certaines fonctionnalités de PostgreSQL peuvent être désactivées. Il est donc difficile de savoir ce qui est réellement utilisable. De plus, chaque nouvelle version mineure demande une adaptation de leur ajout de code. Chaque nouvelle

[55] https://www.postgresql.org/developer/beta/
[56] https://www.postgresql.org/about/featurematrix

version majeure demande une adaptation encore plus importante de leur code. C'est un énorme travail, qui n'apporte généralement pas suffisamment de plus-value à la société éditrice pour qu'elle le réalise. La seule société qui le fait de façon complète est EnterpriseDB, qui arrive à proposer des mises à jour régulièrement. Par contre, si on revient sur l'exemple de Greenplum, ils sont restés bloqués pendant un bon moment sur la version 8.0. Ils ont cherché à corriger cela. Fin 2021, Greenplum 6.8 est au niveau de la version 9.4[57], version considérée alors comme obsolète par la communauté depuis plus de deux ans. En janvier 2023, Greenplum 7.0 bêta n'est toujours parvenu qu'au niveau de PostgreSQL 12.12. Depuis, le dépôt n'est plus public…

Rien ne dit non plus que la société ne va pas abandonner son fork. Par exemple, il a existé quelques forks créés lorsque PostgreSQL n'était pas disponible en natif sous Windows : ces forks ont majoritairement disparu lors de l'arrivée de la version 8.0, qui supportait officiellement Windows dans la version communautaire.

Il y a eu aussi quelques forks créés pour gérer la réplication. Là aussi, la plupart de ces forks ont été abandonnés (et leurs clients avec) quand PostgreSQL a commencé à proposer de la réplication en version 9.0.

Il faut donc bien comprendre qu'à partir du moment où un utilisateur choisit une version dérivée, il dépend fortement (voire uniquement) de la bonne volonté de la société éditrice pour maintenir son produit, le mettre à jour avec les dernières corrections et les dernières nouveautés de la version communautaire, et le rendre compatible avec la myriade d'extensions existantes. Pour éviter ce problème, certaines sociétés ont décidé de transformer leur fork en une extension. C'est beaucoup plus simple à maintenir et n'enferme pas leurs utilisateurs. C'est le cas par exemple de citusdata (racheté par Microsoft) pour son extension de *sharding* ; ou encore de TimescaleDB, avec leur extension spécialisée dans les séries temporelles.

Dans les exemples de forks dédiés aux entrepôts de données, les plus connus historiquement sont Greenplum, de Pivotal (racheté par WMWare), et Netezza (racheté par IBM). Autant Greenplum tente de se raccrocher au PostgreSQL communautaire toutes les quelques années, autant ce n'est pas le cas de Netezza, optimisé pour du matériel dédié, et qui a forké de PostgreSQL 7.2.

Amazon, avec notamment les versions Redshift[58] ou Aurora, a la particularité de modifier profondément PostgreSQL pour l'adapter à son infrastructure, mais ne diffuse pas ses modifications. Même si certaines incompatibilités sont listées, il est très difficile de savoir où ils en sont et l'impact qu'a leurs modifications.

Neon.tech[59] est un autre *fork* ayant réécrit la couche de stockage et permettant de dupliquer des bases rapidement, notamment à l'usage des développeurs.

EDB Postgres Advanced Server est une distribution PostgreSQL d'EnterpriseDB. Elle permet de faciliter la migration depuis Oracle. Son code est propriétaire et soumis à une licence payante. Certaines fonctionnalités finissent par atterrir dans le code communautaire (une fois qu'EnterpriseDB le souhaite et que la communauté a validé l'intérêt de cette fonctionnalité et sa possible intégration).

Supabase est un exemple de société intégrant PostgreSQL dans une plateforme plus vaste pour du développement web.

[57] https://web.archive.org/web/20211018012643/https://docs.greenplum.org/6-8/security-guide/topics/preface.html
[58] https://www.stitchdata.com/blog/how-redshift-differs-from-postgresql/
[59] https://neon.tech/

BDR, anciennement de 2nd Quadrant, maintenant EnterpriseDB, est un *fork* visant à fournir une version multimaître de PostgreSQL, mais le code a été refermé dans les dernières versions. Il est très difficile de savoir où ils en sont. Son utilisation implique de prendre le support chez eux.

La société russe Postgres Pro, tout comme EnterpriseDB, propose diverses fonctionnalités dans sa version propre, tout en proposant souvent leur inclusion dans la version communautaire — ce qui n'est pas automatique.

Face au leadership de PostgreSQL, une tendance récente pour certaines bases de données est de se revendiquer « compatibles PostgreSQL », par exemple YugabyteDB. Certains éditeurs de solutions de bases de données distribuées propriétaires disent que leur produit peut remplacer PostgreSQL sans modification de code côté application. Il convient de rester critique et prudent face à cette affirmation, car ces produits n'ont parfois rien à voir avec PosgreSQL. Leurs évolutions n'intégreront sans doute pas la version communautaire.

(Cet historique provient en partie de la liste exhaustive des « forks »[60], ainsi de que cette conférence de Josh Berkus[61] de 2009 et des références en bibliographie.)

Sauf cas très précis, il est recommandé d'utiliser la version officielle, libre et gratuite. Vous savez exactement ce qu'elle propose et vous choisissez librement vos partenaires (pour les formations, pour le support, pour les audits, etc). Vous profitez aussi de tous les outils de l'écosystème.

[60] https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases
[61] https://www.slideshare.net/pgconf/elephant-roads-a-tour-of-postgres-forks

1.4 QUELQUES PROJETS SATELLITES

PostgreSQL n'est que le moteur ! Besoin d'outils pour :

- Administration
- Sauvegarde
- Supervision
- Migration
- SIG

PostgreSQL n'est qu'un moteur de bases de données. Quand vous l'installez, vous n'avez que ce moteur. Vous disposez de quelques outils en ligne de commande (détaillés dans nos modules « Outils graphiques et consoles » et « Tâches courantes ») mais aucun outil graphique n'est fourni.

Du fait de ce manque, certaines personnes ont décidé de développer ces outils graphiques. Ceci a abouti à une grande richesse grâce à la grande variété de projets « satellites » qui gravitent autour du projet principal.

Par choix, nous ne présenterons ici que des logiciels libres et gratuits. Pour chaque problématique, il existe aussi des solutions propriétaires. Ces solutions peuvent parfois apporter des fonctionnalités inédites. Il faut néanmoins considérer que l'offre de la communauté Open-Source répond à la plupart des besoins des utilisateurs de PostgreSQL.

1.4.1 Administration, Développement, Modélisation

Entre autres, dédiés ou pas :

- Administration :
  - pgAdmin4
  - temBoard
- Développement :
  - DBeaver
- Modélisation :
  - pgModeler

Il existe différents outils graphiques pour l'administration, le développement et la modélisation. Une

liste plus exhaustive est disponible sur le wiki PostgreSQL[62].

pgAdmin4[63] est un outil d'administration dédié à PostgreSQL, qui permet aussi de requêter. (La version 3 est considérée comme périmée.)

temBoard[64] est une console d'administration plus complète. temBoard intègre de la supervision, des tableaux de bord, la gestion des sessions en temps réel, du bloat, de la configuration et l'analyse des performances.

DBeaver[65] est un outil de requêtage courant, utilisable avec de nombreuses bases de données différentes, et adapté à PostgreSQL.

Pour la modélisation, pgModeler[66] est dédié à PostgreSQL. Il permet la modélisation, la rétro-ingénierie d'un schéma existant, la génération de scripts de migration.

1.4.2 Sauvegardes

- Export logique :
  - pg_back : https://github.com/orgrim/pg_back/
- Sauvegarde physique (PITR) :
  - pgBackRest : https://pgbackrest.org/
  - barman : https://www.pgbarman.org/

Les outils listés ci-dessus sont les outils principaux et que nous recommandons pour la réalisation des sauvegardes et la gestion de leur rétention.

Ils se basent sur les outils standards de PostgreSQL de sauvegarde physique ou logique.

Liens :

- pg_back[67]
- pgBackRest[68]
- barman[69]

Il existe plusieurs outils propriétaires, notamment pour une sauvegarde par snapshot au niveau de la baie.

[62] https://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools
[63] https://www.pgadmin.org/
[64] https://labs.dalibo.com/temboard
[65] https://dbeaver.io/
[66] https://pgmodeler.io/
[67] https://github.com/orgrim/pg_back/releases
[68] https://pgbackrest.org/
[69] https://www.pgbarman.org

1.4.3 Supervision

- Nagios/Icinga2 :
  - check_pgactivity
  - check_postgres
- Prometheus : postgres_exporter
- PoWA

Pour ne citer que quelques projets libres et matures :

check_pgactivity[70] est une sonde Nagios pouvant récupérer un grand nombre de statistiques d'activités renseignées par PostgreSQL. Il faut de ce fait un serveur Nagios (ou un de ses nombreux forks ou surcharges) pour gérer les alertes et les graphes. Il existe aussi check_postgres[71].

postgres_exporter[72] est l'exporteur de métriques pour Prometheus.

PoWA[73] est composé d'une extension qui historise les statistiques récupérées par l'extension `pg_stat_statements` et d'une application web qui permet de récupérer les requêtes et leur statistiques facilement.

1.4.4 Audit

- pgBadger
  - https://pgbadger.darold.net/
- pgCluu
  - https://pgcluu.darold.net/

pgBadger[74] est l'outil de base pour les analyses (à posteriori) des traces de PostgreSQL, dont notamment les requêtes.

[70] https://github.com/OPMDG/check_pgactivity
[71] https://bucardo.org/check_postgres/
[72] https://github.com/prometheus-community/postgres_exporter
[73] https://powa.readthedocs.io/en/latest/
[74] https://pgbadger.darold.net/

pgCluu[75] permet une analyse du système et de PostgreSQL.

1.4.5 Migration

- Oracle, MySQL, SQL Server : Ora2Pg
  - https://ora2pg.darold.net/
- MySQL, SQL Server : pgloader
  - https://pgloader.io/

Il existe de nombreux outils pour migrer vers PostgreSQL une base de données utilisant un autre moteur. Ce qui pose le plus problème en pratique est le code applicatif (procédures stockées).

Ces outils ont été développé au fil du temps par des sociétés proposant de la prestation de service autour de la migration, ou par de grosses entreprises pour leurs besoins internes. Il sont plus ou moins avancés, efficaces, finis, maintenus, et généralement propriétaires.

Les fonctionnalités incluent généralement la migration du schéma, parfois le transfert des données de manière plus ou moins optimisée, l'estimation de la charge de migration, ou encore la migration du code applicatif (PL/SQL d'Oracle notamment). Ce dernier point est généralement le plus complexe et doit généralement être repris ou traité par des humains.

Les plus importants parmi les outils libres actifs sont :

- Ora2Pg[76], de Gilles Darold, convertit le schéma de données, migre les données, et tente même de convertir le code PL/SQL en PL/pgSQL. Il convertit aussi des bases MySQL ou SQL Server.

- pgloader[77], de Dimitri Fontaine, permet de migrer depuis MySQL, SQLite ou MS SQL Server, et importe les fichiers CSV, DBF (dBase) ou IXF (fichiers d'échange indépendants de la base).

[75] https://pgcluu.darold.net/
[76] https://ora2pg.darold.net/
[77] https://pgloader.io/

1.4.6 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.

1.5 SPONSORS & RÉFÉRENCES

- https://www.postgresql.org/about/sponsors
- Références :
  - françaises
  - et internationales

Au-delà de ses qualités, PostgreSQL suscite toujours les mêmes questions récurrentes :

- qui finance les développements ? (et pourquoi ?)
- qui utilise PostgreSQL ?

1.5.1 Sponsors principaux

- Sociétés se consacrant à PostgreSQL :
  - Crunchy Data (USA) : Tom Lane, Stephen Frost, Joe Conway…
  - EnterpriseDB (USA) : Bruce Momjian, Robert Haas, Peter Eisentraut…
  - PostgresPro (Russie) : Oleg Bartunov, Teodor Sigaev
  - Cybertec (Autriche), Dalibo (France), Redpill Linpro (Suède), Credativ (Allemagne)…
- Sociétés vendant un fork ou une extension :
  - Citusdata (Microsoft), Pivotal (VMWare), TimescaleDB

La liste des sponsors de PostgreSQL contribuant activement au développement figure sur la liste officielle des sponsors[78]. Ce qui suit n'est qu'un aperçu.

EnterpriseDB est une société américaine qui a décidé de fournir une version de PostgreSQL propriétaire fournissant une couche de compatibilité avec Oracle. Ils emploient plusieurs développeurs importants du projet PostgreSQL (dont trois font partie de la *Core Team*), et reversent un certain nombre de leurs travaux au sein du moteur communautaire. Ils ont aussi un poids financier qui leur permet de sponsoriser la majorité des grands événements autour de PostgreSQL : PGEast et PGWest aux États-Unis, PGDay en Europe.

[78] https://www.postgresql.org/about/sponsors/

En 2020, EnterpriseDB rachète 2nd Quadrant, une société anglaise fondée par le regretté Simon Riggs, développeur PostgreSQL de longue date. 2nd Quadrant développe de nombreux outils autour de PostgreSQL comme pglogical, des versions dérivées comme Postgres-XL ou BDR, ou des outils annexes comme barman ou repmgr.

Crunchy Data offre sa propre version certifiée et finance de nombreux développements.

De nombreuses autres sociétés dédiées à PostgreSQL existent dans de nombreux pays. Parmi les sponsors officiels, nous pouvons compter Cybertec en Autriche ou Redpill Linpro en Suède. En Russie, PostgresPro maintient une version locale et reverse aussi de nombreuses contributions à la communauté.

En Europe francophone, Dalibo participe pleinement à la communauté. La société est Major Sponsor du projet PostgreSQL[79], ce qui indique un support de longue date. Elle développe et maintient plusieurs outils plébiscités par la communauté, comme autrefois Open PostgreSQL Monitoring (OPM) ou la sonde check_pgactivity[80], plus récemment la console d'administration temBoard[81], avec de nombreux autres projets en cours[82], et une participation active au développement de patchs pour PostgreSQL. Dalibo sponsorise également des événements comme les PGDay français et européens, ainsi que la communauté francophone.

Des sociétés comme Citusdata (racheté par Microsoft), Pivotal (VMWare) ou TimescaleDB proposent ou ont proposé leur version dérivée sous une forme ou une autre, mais « jouent le jeu » et participent au développement de la version communautaire, notamment en cherchant à ce que leur produit n'en diverge pas.

1.5.2 Autres sponsors

- Autres sociétés :
  - VMWare, Rackspace, Heroku, Conova, Red Hat, Microsoft
  - NTT (*streaming replication*), Fujitsu, NEC
- Cloud
  - nombreuses

Contribuent également à PostgreSQL nombre de sociétés non centrées autour des bases de données.

NTT a financé de nombreux patchs pour PostgreSQL.

[79] https://www.postgresql.org/about/sponsors/
[80] https://github.com/OPMDG/check_pgactivity
[81] https://labs.dalibo.com/temboard
[82] https://labs.dalibo.com/about

Fujitsu a participé à de nombreux développements aux débuts de PostgreSQL, et emploie Amit Kapila.

VMWare a longtemps employé le développeur finlandais Heikki Linnakangas, parti ensuite un temps chez Pivotal. VMWare emploie aussi Michael Paquier ou Julien Rouhaud.

Red Hat a longtemps employé Tom Lane à plein temps pour travailler sur PostgreSQL. Il a pu dédier une très grande partie de son temps de travail à ce projet, bien qu'il ait eu d'autres affectations au sein de Red Hat. Tom Lane a travaillé également chez SalesForce, ensuite il a rejoint Crunchy Data Solutions fin 2015.

Il y a déjà plus longtemps, Skype a offert un certain nombre d'outils très intéressants : PgBouncer (pooler de connexion), Londiste (réplication par trigger), etc. Ce sont des outils utilisés en interne et publiés sous licence BSD comme retour à la communauté. Malgré le rachat par Microsoft, certains sont encore utiles et maintenus.

Zalando est connu pour l'outil de haute disponibilité patroni.

De nombreuses sociétés liées au cloud figurent aussi parmi les sponsors, comme Conova (Autriche), Heroku ou Rackspace (États-Unis), ou les mastodontes Google, Amazon Web Services et, à nouveau, Microsoft.

1.5.3 Références

- Météo France
- IGN
- RATP, SNCF
- CNAF
- MAIF, MSA
- Le Bon Coin
- Air France-KLM
- Société Générale
- Carrefour, Leclerc, Leroy Merlin
- Instagram, Zalando, TripAdvisor
- Yandex
- CNES
- …et plein d'autres

Météo France utilise PostgreSQL depuis plus d'une décennie pour l'essentiel de ses bases, dont des instances critiques de plusieurs téraoctets (témoignage sur postgresql.fr[83]).

L'IGN utilise PostGIS et PostgreSQL depuis 2006[84].

[83] https://www.postgresql.fr/temoignages/meteo_france
[84] https://www.postgresql.fr/temoignages/ign

La RATP a fait ce choix depuis 2007 également[85].

La Caisse Nationale d'Allocations Familiales a remplacé ses mainframes par des instances PostgreSQL[86] dès 2010 (4 To et 1 milliard de requêtes par jour).

Instagram utilise PostgreSQL depuis le début[87].

Zalando a décrit plusieurs fois son infrastructure PostgreSQL[88] et annonçait en 2018[89] utiliser pas moins de 300 bases de données en interne et 650 instances dans un cloud AWS. Zalando contribue à la communauté, notamment par son outil de haute disponibilité patroni[90].

Le DBA de TripAdvisor témoigne de leur utilisation de PostgreSQL dans l'interview suivante[91].

Dès 2009, Leroy Merlin migrait vers PostgreSQL des milliers de logiciels de caisse[92].

Yandex, équivalent russe de Google a décrit en 2016 la migration des 300 To de données de Yandex.Mail depuis Oracle vers PostgreSQL[93].

La Société Générale a publié son outil de migration d'Oracle à PostgreSQL[94].

Autolib à Paris utilisait PostgreSQL. Le logiciel est encore utilisé dans les autres villes où le service continue. Ils ont décrit leur infrastructure au PG Day 2018 à Marseille[95].

De nombreuses autres sociétés participent au Groupe de Travail Inter-Entreprises de PostgreSQLFr[96] : Air France, Carrefour, Leclerc, le CNES, la MSA, la MAIF, PeopleDoc, EDF…

Cette liste ne comprend pas les innombrables sociétés qui n'ont pas communiqué sur le sujet. PostgreSQL étant un logiciel libre, il n'existe nulle part de dénombrement des instances actives.

1.5.4 Le Bon Coin

- Site de petites annonces
- 4è site le plus consulté en France (2017)
- 27 millions d'annonces en ligne, 800 000 nouvelles chaque jour
- Instance PostgreSQL principale : 3 To de volume, 3 To de RAM
- 20 serveurs secondaires

[85] https://www.journaldunet.com/solutions/dsi/1013631-la-ratp-integre-postgresql-a-son-systeme-d-information/
[86] https://www.silicon.fr/cnaf-debarrasse-mainframes-149897.html?inf_by=5bc488a1671db858728b4c35
[87] https://media.postgresql.org/sfpug/instagram_sfpug.pdf
[88] http://gotocon.com/dl/goto-berlin-2013/slides/HenningJacobs_and_ValentineGogichashvili_WhyZalandoTrustsInPostgreSQL.pdf
[89] https://www.postgresql.eu/events/pgconfeu2018/schedule/session/2135-highway-to-hell-or-stairway-to-cloud/
[90] https://jobs.zalando.com/tech/blog/zalandos-patroni-a-template-for-high-availability-postgresql/
[91] https://www.citusdata.com/blog/25-terry/285-matthew-kelly-tripadvisor-talks-about-pgconf-silicon-valley
[92] https://wiki.postgresql.org/images/6/63/Adeo_PGDay.pdf
[93] https://www.pgcon.org/2016/schedule/attachments/426_2016.05.19%20Yandex.Mail%20success%20story.pdf
[94] https://github.com/societe-generale/code2pg
[95] https://www.youtube.com/watch?v=vd8B7B-Zca8
[96] https://www.postgresql.fr/entreprises/accueil

PostgreSQL tient la charge sur de grosses bases de données et des serveurs de grande taille.

Le Bon Coin privilégie des serveurs physiques dans ses propres datacenters.

Pour plus de détails et l'évolution de la configuration, voir les témoignages de ses directeurs technique[97] (témoignage de juin 2012) et infrastructure[98] (juin 2017), ou la conférence de son DBA Flavio Gurgel au pgDay Paris 2019[99].

Ce dernier s'appuie sur les outils classiques fournis par la communauté : pg_dump (pour archivage, car ses exports peuvent être facilement restaurés), barman, pg_upgrade.

[97] https://www.postgresql.fr/temoignages:le_bon_coin
[98] https://web.archive.org/web/20171222173630/https://www.kissmyfrogs.com/jean-louis-bergamo-leboncoin-ce-qui-a-ete-fait-maison-est-ultra-performant/
[99] https://www.postgresql.eu/events/pgdayparis2019/schedule/session/2376-large-databases-lots-of-servers

1.6 À LA RENCONTRE DE LA COMMUNAUTÉ

- Cartographie du projet
- Pourquoi participer
- Comment participer

1.6.1 PostgreSQL, un projet mondial

Figure 1/ .2: Carte des hackers

On le voit, PostgreSQL compte des contributeurs sur tous les continents.

Le projet est principalement anglophone. Les *core hackers* sont surtout répartis en Amérique, Europe, Asie (Japon surtout).

Il existe une très grande communauté au Japon, et de nombreux développeurs en Russie.

La communauté francophone est très dynamique, s'occupe beaucoup des outils, mais il n'y a que quelques développeurs réguliers du *core* francophones : Michael Paquier, Julien Rouhaud, Fabien Coelho…

La communauté hispanophone est naissante.

1.6.2 PostgreSQL Core Team

Figure 1/ .3: Core team

Le terme *Core Hackers* désigne les personnes qui sont dans la communauté depuis longtemps. Ces personnes désignent directement les nouveaux membres.

NB : Le terme *hacker* peut porter à confusion, il s'agit bien ici de la définition « universitaire » : https://fr.wikipedia.org/wiki/Hacker_(programmation)

La *Core Team* est un ensemble de personnes doté d'un pouvoir assez limité. Ils ne doivent pas appartenir en majorité à la même société. Ils peuvent décider de la date de sortie d'une version. Ce sont les personnes qui sont immédiatement au courant des failles de sécurité du serveur PostgreSQL. Exceptionnellement, elles tranchent certains débats si un consensus ne peut être atteint dans la communauté. Tout le reste des décisions est pris par la communauté dans son ensemble après discussion, généralement sur la liste pgsql-hackers.

Les membres actuels de la *Core Team* sont[100] :

- **Tom Lane** (Crunchy Data, Pittsburgh, États-Unis) : certainement le développeur le plus aguerri avec la vision la plus globale, notamment sur l'optimiseur ;
- **Bruce Momjian** (EDB, Pennsylvanie, États-Unis) : a lancé le projet en 1995, écrit du code (pg_upgrade notamment) et s'est beaucoup occupé de la promotion ;
- **Magnus Hagander** (Redpill Linpro, Stockholm, Suède) : développeur, a participé notamment au portage Windows, à l'outil pg_basebackup, à l'authentification, à l'administration des serveurs, président de PostgreSQL Europe ;

[100] https://www.postgresql.org/community/contributors/

- **Andres Freund** (Microsoft, San Francisco, États-Unis) : contributeur depuis des années de nombreuses fonctionnalités (JIT, réplication logique, performances…) ;
- **Dave Page** (pgEdge, Oxfordshire, Royaume-Uni) : leader du projet pgAdmin, version Windows, administration des serveurs, secrétaire de PostgreSQL Europe ;
- **Peter Eisentraut** (EDB, Dresde, Allemagne) : développement du moteur (internationalisation, SQL/Med…), participe à la définition du standard SQL, etc. ;
- **Jonathan Katz** (Amazon Web Services, New York, États-Unis) : promotion du projet, modération, revues de patchs.

1.6.3 Contributeurs

Figure 1/ .4: Contributeurs

Actuellement, les « contributeurs » se répartissent quotidiennement les tâches suivantes :

- développement des projets satellites (pgAdmin…) ;
- promotion du logiciel ;
- administration de l'infrastructure des serveurs ;
- rédaction de documentation ;
- conférences ;
- traductions ;
- organisation de groupes locaux.

Le *PGDG* a fêté son 10e anniversaire à Toronto en juillet 2006. Ce « PostgreSQL Anniversary Summit » a réuni pas moins de 80 membres actifs du projet. La photo ci-dessus a été prise à l'occasion.

PGCon2009 a réuni 180 membres actifs à Ottawa, et environ 220 en 2018 et 2019.

Voir la liste des contributeurs officiels[101].

1.6.4 Qui contribue du code ?

- Principalement des personnes payées par leur société
- 30 committers
  - Tom Lane
  - Andres Freund
  - Peter Eisentraut
  - Nikita Glukhov
  - Álvaro Herrera
  - Michael Paquier
  - Robert Haas
  - …et beaucoup d'autres
- Commitfests[102] : tous les 2 mois
  - https://commitfest.postgresql.org/

À l'automne 2024, on compte 30 committers[103], c'est-à-dire personnes pouvant écrire dans tout ou partie du dépôt de PostgreSQL. Il ne s'agit pas que de leur travail, mais pour une bonne partie de patchs d'autres contributeurs après discussion et validation des fonctionnalités mais aussi des standards propres à PostgreSQL, de la documentation, de la portabilité, de la simplicité, de la sécurité, etc. Ces autres contributeurs peuvent être potentiellement n'importe qui. En général, un patch est relu par plusieurs personnes avant d'être transmis à un *committer*.

Les discussions quant au développement ont lieu principalement (mais pas uniquement) sur la liste pgsql-hackers[104]. Les éventuels bugs sont transmis à la liste pgsql-bugs[105]. Puis les patchs en cours sont revus au moins tous les deux mois lors des Commitfests[106]. Il n'y a pas de *bug tracker* car le fonctionnement actuel est jugé satisfaisant.

Robert Haas publie chaque année une analyse sur les contributeurs de code et les participants aux discussions sur le développement de PostgreSQL sur la liste pgsql-hackers :

[101] https://www.postgresql.org/community/contributors/
[103] https://www.postgresql.org/developer/committers/
[104] https://www.postgresql.org/list/pgsql-hackers/
[105] https://www.postgresql.org/list/pgsql-bugs/
[106] https://commitfest.postgresql.org/

- 2023 : http://rhaas.blogspot.com/2024/01/who-contributed-to-postgresql.html
- 2022 : http://rhaas.blogspot.com/2023/04/who-contributed-to-postgresql.html
- 2020/2021 : http://rhaas.blogspot.com/2022/01/who-contributed-to-postgresql.html
- 2019 : http://rhaas.blogspot.com/2020/05/who-contributed-to-postgresql.html
- 2018 : http://rhaas.blogspot.com/2019/01/who-contributed-to-postgresql.html
- 2017 : http://rhaas.blogspot.com/2018/06/who-contributed-to-postgresql.html
- 2016 : http://rhaas.blogspot.com/2017/04/who-contributes-to-postgresql.html

1.6.5 Répartition des développeurs

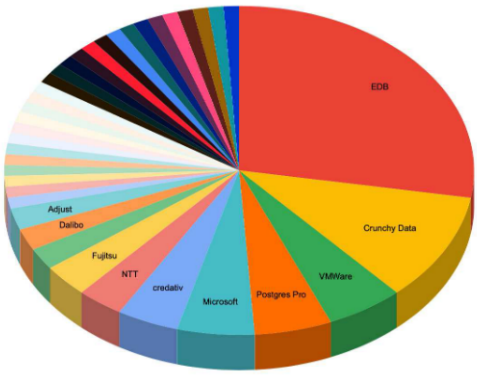


Figure 1/ .5: Répartition des développeurs

Voici une répartition des différentes sociétés qui ont contribué aux améliorations de la version 13. On y voit qu'un grand nombre de sociétés prend part à ce développement. La plus importante est EDB, mais même elle n'est responsable que d'un petit tiers des contributions.

(Source : Future Postgres Challenges[107], Bruce Momjian, 2021)

1.6.6 Utilisateurs

- Vous !
- **Le succès d'un logiciel libre dépend de ses utilisateurs.**

Il est impossible de connaître précisément le nombre d'utilisateurs de PostgreSQL. Toutefois ce nombre est en constante augmentation.

[107] https://momjian.us/main/writings/pgsql/challenges.pdf

Il existe différentes manières de s'impliquer dans une communauté Open-Source. Dans le cas de PostgreSQL, vous pouvez :

- déclarer un bug ;
- tester les versions bêta ;
- témoigner.

1.6.7 Pourquoi participer

- Rapidité des corrections de bugs
- Préparer les migrations / tester les nouvelles versions
- Augmenter la visibilité du projet
- Créer un réseau d'entraide

Au-delà de motivations idéologiques ou technologiques, il y a de nombreuses raisons objectives de participer au projet PostgreSQL.

Envoyer une description d'un problème applicatif aux développeurs est évidemment le meilleur moyen d'obtenir sa correction. Attention toutefois à être précis et complet lorsque vous déclarez un bug sur pgsql-bugs[108] ! Assurez-vous que vous pouvez le reproduire.

Tester les versions « candidates » dans votre environnement (matériel et applicatif) est la meilleure garantie que votre système d'information sera compatible avec les futures versions du logiciel.

Les retours d'expérience et les cas d'utilisations professionnelles sont autant de preuves de la qualité de PostgreSQL. Ces témoignages aident de nouveaux utilisateurs à opter pour PostgreSQL, ce qui renforce la communauté.

S'impliquer dans les efforts de traductions, de relecture ou dans les forums d'entraide ainsi que toute forme de transmission en général est un très bon moyen de vérifier et d'approfondir ses compétences.

1.6.8 Ressources web de la communauté

- Site officiel : https://www.postgresql.org/
- Actualité : https://planet.postgresql.org/
- Des extensions : https://pgxn.org/

[108] https://www.postgresql.org/list/pgsql-bugs/

Le site officiel de la communauté se trouve sur https://www.postgresql.org/. Ce site contient des informations sur PostgreSQL, la documentation des versions maintenues, les archives des listes de discussion, etc.

Le site « Planet PostgreSQL » est un agrégateur réunissant les blogs des *Core Hackers*, des contributeurs, des traducteurs et des utilisateurs de PostgreSQL.

Le site PGXN est l'équivalent pour PostgreSQL du CPAN de Perl, une collection en ligne de librairies et extensions accessibles depuis la ligne de commande.

1.6.9 Documentation officielle

- LA référence, même au quotidien
- Anglais : https://www.postgresql.org/docs/
- Français : https://docs.postgresql.fr/

La documentation officielle sur https://www.postgresql.org/docs/current est maintenue au même titre que le code du projet, et sert aussi au quotidien, pas uniquement pour des cas obscurs.

Elle est versionnée pour chaque version majeure.

La traduction française suit de près les mises à jour de la documentation officielle : https://docs.postgresql.fr/.

1.6.10 Serveurs francophones

- Site officiel : https://www.postgresql.fr/
- Documentation traduite : https://docs.postgresql.fr/
- Forum : https://forums.postgresql.fr/
- Actualité : https://planete.postgresql.fr/
- Association PostgreSQLFr : https://www.postgresql.fr/asso/accueil
- Groupe de Travail Inter-Entreprises (PGGTIE) : https://www.postgresql.fr/entreprises/accueil

Le site postgresql.fr est le site de l'association des utilisateurs francophones du logiciel. La communauté francophone se charge de la traduction de toutes les documentations.

1.6.11 Listes de discussions / Listes d'annonces

- pgsql-announce
- pgsql-general
- pgsql-admin
- pgsql-sql
- pgsql-performance
- pgsql-fr-generale
- pgsql-advocacy
- pgsql-bugs

Les mailing-lists sont les outils principaux de gouvernance du projet. Toute l'activité de la communauté (bugs, promotion, entraide, décisions) est accessible par ce canal. Les développeurs principaux du projets répondent parfois eux-mêmes. Si vous avez une question ou un problème, la réponse se trouve probablement dans les archives ! Pour s'inscrire ou consulter les archives : https://www.postgresql.org/list/.

Si vous pensez avoir trouvé un bug, vous pouvez le remonter sur la liste anglophone pgsql-bugs[109], par le formulaire dédié[110]. Pour faciliter la tâche de ceux qui tenteront de vous répondre, suivez bien les consignes sur les rapports de bug[111] : informations complètes, reproductibilité…

1.6.12 IRC

- Réseau LiberaChat
- IRC anglophone :
  - #postgresql
  - #postgresql-eu
- IRC francophone :
  - #postgresqlfr

Le point d'entrée principal pour le réseau LiberaChat est le serveur **irc.libera.chat**. La majorité des développeurs sont disponibles sur IRC et peuvent répondre à vos questions.

[109] https://www.postgresql.org/list/pgsql-bugs/
[110] https://www.postgresql.org/account/submitbug/
[111] https://docs.postgresql.fr/current/bogue-reporting.html

Des canaux de discussion spécifiques à certains projets connexes sont également disponibles, comme par exemple #slony.

Attention ! Vous devez poser votre question en public et ne pas solliciter de l'aide par message privé.

1.6.13 Wiki

- https://wiki.postgresql.org/

Le wiki est un outil de la communauté qui met à disposition une véritable mine d'informations.

Au départ, le wiki avait pour but de récupérer les spécifications écrites par des développeurs pour les grosses fonctionnalités à développer à plusieurs. Cependant, peu de développeurs l'utilisent dans ce cadre. L'utilisation du wiki a changé en passant plus entre les mains des utilisateurs qui y intègrent un bon nombre de pages de documentation (parfois reprises dans la documentation officielle). Le wiki est aussi utilisé par les organisateurs d'événements pour y déposer les slides des conférences. Elle n'est pas exhaustive et, hélas, souffre fréquemment d'un manque de mises à jour.

1.6.14 L'avenir de PostgreSQL

- PostgreSQL est devenu la base de données de référence
- Grandes orientations :
  - réplication logique
  - meilleur parallélisme
  - gros volumes
- Prochaine version, la 18
- Stabilité économique
- De plus en plus de (gros) clients
- Le futur de PostgreSQL dépend de vous !

Le projet avance grâce à de plus en plus de contributions. Les grandes orientations actuelles sont :

- une réplication de plus en plus sophistiquée ;

- une gestion plus étendue du parallélisme ;
- une volumétrie acceptée de plus en plus importante ;
- etc.

PostgreSQL est là pour durer. Le nombre d'utilisateurs, de toutes tailles, augmente tous les jours. Il n'y a pas qu'une seule entreprise derrière ce projet. Il y en a plusieurs, petites et grosses sociétés, qui s'impliquent pour faire avancer le projet, avec des modèles économiques et des marchés différents, garants de la pérennité du projet.

1.7 CONCLUSION

- Un projet de grande ampleur
- Un SGBD complet
- Souplesse, extensibilité
- De belles références
- Une solution **stable**, **ouverte**, **performante** et **éprouvée**
- Pas de dépendance envers UN éditeur

Certes, la licence PostgreSQL implique un coût nul (pour l'acquisition de la licence), un code source disponible et aucune contrainte de redistribution. Toutefois, il serait erroné de réduire le succès de PostgreSQL à sa gratuité.

Beaucoup d'acteurs font le choix de leur SGBD sans se soucier de son prix. En l'occurrence, ce sont souvent les qualités intrinsèques de PostgreSQL qui séduisent :

- sécurité des données (reprise en cas de crash et résistance aux bogues applicatifs) ;
- facilité de configuration ;
- montée en puissance et en charge progressive ;
- gestion des gros volumes de données ;
- pas de dépendance envers un unique éditeur ou prestataire.

1.7.1 Bibliographie

- Documentation officielle (préface)
- Articles fondateurs de M. Stonebracker (1987)
- *Présentation du projet PostgreSQL* (Guillaume Lelarge, 2008)
- *Looking back at PostgreSQL* (J.M. Hellerstein, 2019)

Quelques références :

- Préface de la documentation officielle : 2. Bref historique de PostgreSQL[112]
- *The Design of POSTGRES*[113] , Michael Stonebraker & Lawrence A. Rowe, 1987
- Présentation du projet PostgreSQL[114], Guillaume Lelarge, RMLL 2008

[112] https://docs.postgresql.fr/current/history.html
[113] http://db.cs.berkeley.edu/papers/ERL-M85-95.pdf
[114] https://web.archive.org/web/20160322070704/2008.rmll.info/Presentation-de-PostgreSQL.html

- *Looking Back at PostgreSQL*[115] , Joseph M. Hellerstein, 2019

Iconographie : La photo initiale est le logo officiel de PostgreSQL[116].

1.7.2 Questions

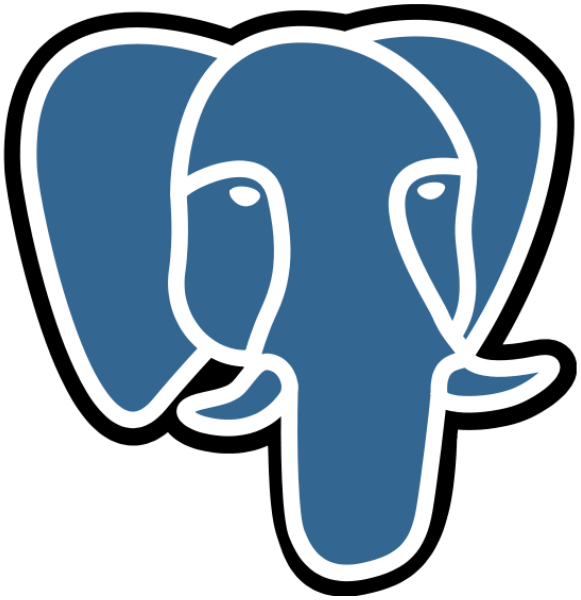
N'hésitez pas, c'est le moment !

[115] https://arxiv.org/pdf/1901.01973.pdf
[116] https://www.postgresql.org/about/policies/trademarks/

1.8 QUIZ

https://dali.bo/a1_quiz

2/ Découverte des fonctionnalités



2.1 AU MENU

- Fonctionnalités du moteur
- Objets SQL
- Connaître les différentes fonctionnalités et possibilités
- Découvrir des exemples concrets

Ce module propose un tour rapide des fonctionnalités principales du moteur : ACID, MVCC, transactions, journaux de transactions… ainsi que des objets SQL gérés (schémas, index, tablespaces, triggers…). Ce rappel des concepts de base permet d'avancer plus facilement lors des modules suivants.

2.2 FONCTIONNALITÉS DU MOTEUR

- Standard SQL
- ACID : la gestion transactionnelle
- Niveaux d'isolation
- Journaux de transactions
- Administration
- Sauvegardes
- Réplication
- Supervision
- Sécurité
- Extensibilité

Cette partie couvre les différentes fonctionnalités d'un moteur de bases de données. Il ne s'agit pas d'aller dans le détail de chacune, mais de donner une idée de ce qui est disponible. Les modules suivants de cette formation et des autres formations détaillent certaines de ces fonctionnalités.

2.2.1 Respect du standard SQL

- Excellent support du SQL ISO
- Objets SQL
  - tables, vues, séquences, routines, triggers
- Opérations
  - jointures, sous-requêtes, requêtes CTE, requêtes de fenêtrage, etc.

La dernière version du standard SQL est SQL:2023[1]. À ce jour, aucun SGBD ne la supporte complètement, *mais* :

- PostgreSQL progresse et s'en approche au maximum, au fil des versions ;
- la majorité de la norme est supportée, parfois avec des syntaxes différentes ;
- PostgreSQL est le SGDB le plus respectueux du standard.

[1] https://en.wikipedia.org/wiki/SQL:2023

2.2.2 ACID

Gestion transactionnelle : la force des bases de données relationnelles :

- **Atomicité** (*Atomic*)
- **Cohérence** (*Consistent*)
- **Isolation** (*Isolated*)
- **Durabilité** (*Durable*)

Les propriétés ACID sont le fondement même de toute bonne base de données. Il s'agit de l'acronyme des quatre règles que toute transaction (c'est-à-dire une suite d'ordres modifiant les données) doit respecter :

- **A** : Une transaction est appliquée en « tout ou rien ».
- **C** : Une transaction amène la base d'un état stable à un autre.
- **I** : Les transactions n'agissent pas les unes sur les autres.
- **D** : Une transaction validée sera conservée de manière permanente.

Les bases de données relationnelles les plus courantes depuis des décennies (PostgreSQL bien sûr, mais aussi Oracle, MySQL, SQL Server, SQLite…) se basent sur ces principes, même si elles font chacune des compromis différents suivant leurs cas d'usage, les compromis acceptés à chaque époque avec la performance et les versions.

Atomicité :

Une transaction doit être exécutée entièrement ou pas du tout, et surtout pas partiellement, même si elle est longue et complexe, même en cas d'incident majeur sur la base de données. L'exemple basique est une transaction bancaire : le montant d'un virement doit être sur un compte ou un autre, et en cas de problème ne pas disparaître ou apparaître en double. Ce principe garantit que les données modifiées par des transactions valides seront toujours visibles dans un état stable, et évite nombre de problèmes fonctionnels comme techniques.

Cohérence :

Un état cohérent respecte les règles de validité définies dans le modèle, c'est-à-dire les contraintes définies dans le modèle : types, plages de valeurs admissibles, unicité, liens entre tables (clés étrangères), etc. Le non-respect de ces règles par l'applicatif entraîne une erreur et un rejet de la transaction.

Isolation :

Des transactions simultanées doivent agir comme si elles étaient seules sur la base. Surtout, elles ne voient pas les données *non validées* des autres transactions. Ainsi une transaction peut travailler sur un état stable et fixe, et durer assez longtemps sans risque de gêner les autres transactions.

Il existe plusieurs « niveaux d'isolation » pour définir précisément le comportement en cas de lectures ou écritures simultanées sur les mêmes données et pour arbitrer avec les contraintes de per-

formances ; le niveau le plus contraignant exige que tout se passe comme si toutes les transactions se déroulaient successivement.

Durabilité :

Une fois une transaction validée par le serveur (typiquement : `COMMIT` ne retourne pas d'erreur, ce qui valide la cohérence et l'enregistrement physique), l'utilisateur doit avoir la garantie que la donnée ne sera pas perdue ; du moins jusqu'à ce qu'il décide de la modifier à nouveau. Cette garantie doit valoir même en cas d'événement catastrophique : plantage de la base, perte d'un disque… C'est donc au serveur de s'assurer autant que possible que les différents éléments (disque, système d'exploitation…) ont bien rempli leur office. C'est à l'humain d'arbitrer entre le niveau de criticité requis et les contraintes de performances et de ressources adéquates (et fiables) à fournir à la base de données.

NoSQL :

À l'inverse, les outils de la mouvance (« NoSQL », par exemple MongoDB ou Cassandra), ne fournissent pas les garanties ACID. C'est le cas de la plupart des bases non-relationnelles, qui reprennent le modèle BASE[2] (*Basically Available*, *Soft State*, *Eventually Consistent*, soit succintement : disponibilité d'abord ; incohérence possible entre les réplicas ; cohérence… à terme, après un délai). Un intérêt est de débarrasser le développeur de certaines lourdeurs apparentes liées à la modélisation assez stricte d'une base de données relationnelle. Cependant, la plupart des applications ont d'abord besoin des garanties de sécurité et cohérence qu'offrent un moteur transactionnel classique, et la décision d'utiliser un système ne les garantissant pas ne doit pas être prise à la légère ; sans parler d'autres critères comme la fragmentation du domaine par rapport au monde relationnel et son SQL (à peu près) standardisé. Avec le temps, les moteurs transactionnels ont acquis des fonctionnalités qui faisaient l'intérêt des bases NoSQL (en premier lieu la facilité de réplication et le stockage de JSON), et ces dernières ont tenté d'intégrer un peu plus de sécurité dans leur modèle.

2.2.3 MVCC

- MultiVersion Concurrency Control
- Le « noyau » de PostgreSQL
- Garantit les propriétés ACID
- Permet les accès concurrents sur la même table
  - une lecture ne bloque pas une écriture
  - une écriture ne bloque pas une lecture
  - une écriture ne bloque pas les autres écritures…
  - …sauf pour la mise à jour de la **même ligne**

[2] https://en.wikipedia.org/wiki/Eventual_consistency

MVCC (Multi Version Concurrency Control) est le mécanisme interne de PostgreSQL utilisé pour garantir la cohérence des données lorsque plusieurs processus accèdent simultanément à la même table.

MVCC maintient toutes les versions nécessaires de chaque ligne, ainsi **chaque transaction voit une image figée de la base** (appelée *snapshot*). Cette image correspond à l'état de la base lors du démarrage de la requête ou de la transaction, suivant le niveau d'*isolation* demandé par l'utilisateur à PostgreSQL pour la transaction.

MVCC fluidifie les mises à jour en évitant les blocages trop contraignants (verrous sur `UPDATE`) entre sessions et par conséquent de meilleures performances en contexte transactionnel.

C'est notamment MVCC qui permet d'exporter facilement une base *à chaud* et d'obtenir un export cohérent alors même que plusieurs utilisateurs sont potentiellement en train de modifier des données dans la base.

C'est la qualité de l'implémentation de ce système qui fait de PostgreSQL un des meilleurs SGBD au monde : chaque transaction travaille dans son image de la base, cohérent du début à la fin de ses opérations. Par ailleurs, les écrivains ne bloquent pas les lecteurs et les lecteurs ne bloquent pas les écrivains, contrairement aux SGBD s'appuyant sur des verrous de lignes. Cela assure de meilleures performances, moins de contention et un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

2.2.4 Transactions

- Une transaction = ensemble **atomique** d'opérations
- « Tout ou rien »
- `BEGIN` obligatoire pour grouper des modifications
- `COMMIT` pour valider
  - y compris le DDL
- Perte des modifications si :
  - `ROLLBACK` / perte de la connexion / arrêt (brutal ou non) du serveur
- `SAVEPOINT` pour sauvegarde des modifications d'une transaction à un instant `t`
- Pas de transactions imbriquées

L'exemple habituel et très connu des transactions est celui du virement d'une somme d'argent du compte de Bob vers le compte d'Alice. Le total du compte de Bob ne doit pas montrer qu'il a été débité de X euros tant que le compte d'Alice n'a pas été crédité de X euros. Nous souhaitons en fait que les deux opérations apparaissent aux yeux du reste du système comme une seule opération unitaire. D'où l'emploi d'une transaction explicite. En voici un exemple :

```
BEGIN;
UPDATE comptes SET solde=solde-200 WHERE proprietaire='Bob';
UPDATE comptes SET solde=solde+200 WHERE proprietaire='Alice';
COMMIT;
```

Contrairement à d'autres moteurs de bases de données, PostgreSQL accepte aussi les instructions DDL dans une transaction. En voici un exemple :

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);

SELECT age FROM capitaines;

 age
  35

ROLLBACK;
SELECT age FROM capitaines;

ERROR:  relation "capitaines" does not exist
LINE 1: SELECT age FROM capitaines;
                        ^
```

Nous voyons que la table `capitaines` a existé **à l'intérieur** de la transaction. Mais puisque cette transaction a été annulée (`ROLLBACK`), la table n'a pas été créée au final.

Cela montre aussi le support du DDL transactionnel au sein de PostgreSQL : PostgreSQL n'effectue aucun `COMMIT` implicite sur des ordres DDL tels que `CREATE TABLE`, `DROP TABLE` ou `TRUNCATE TABLE`. De ce fait, ces ordres peuvent être annulés au sein d'une transaction.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, age integer);
INSERT INTO capitaines VALUES (1, 'Haddock', 35);
SAVEPOINT insert_sp;
UPDATE capitaines SET age = 45 WHERE nom = 'Haddock';
ROLLBACK TO SAVEPOINT insert_sp;
COMMIT;

SELECT age FROM capitaines WHERE nom = 'Haddock';

 age
  35
```

Malgré le `COMMIT` après l'`UPDATE`, la mise à jour n'est pas prise en compte. En effet, le `ROLLBACK TO SAVEPOINT` a permis d'annuler cet `UPDATE` mais pas les opérations précédant le `SAVEPOINT`.

À partir de la version 12, il est possible de chaîner les transactions avec `COMMIT AND CHAIN` ou `ROLLBACK AND CHAIN`. Cela veut dire terminer une transaction et en démarrer une autre immédiatement après avec les mêmes propriétés (par exemple, le niveau d'isolation).

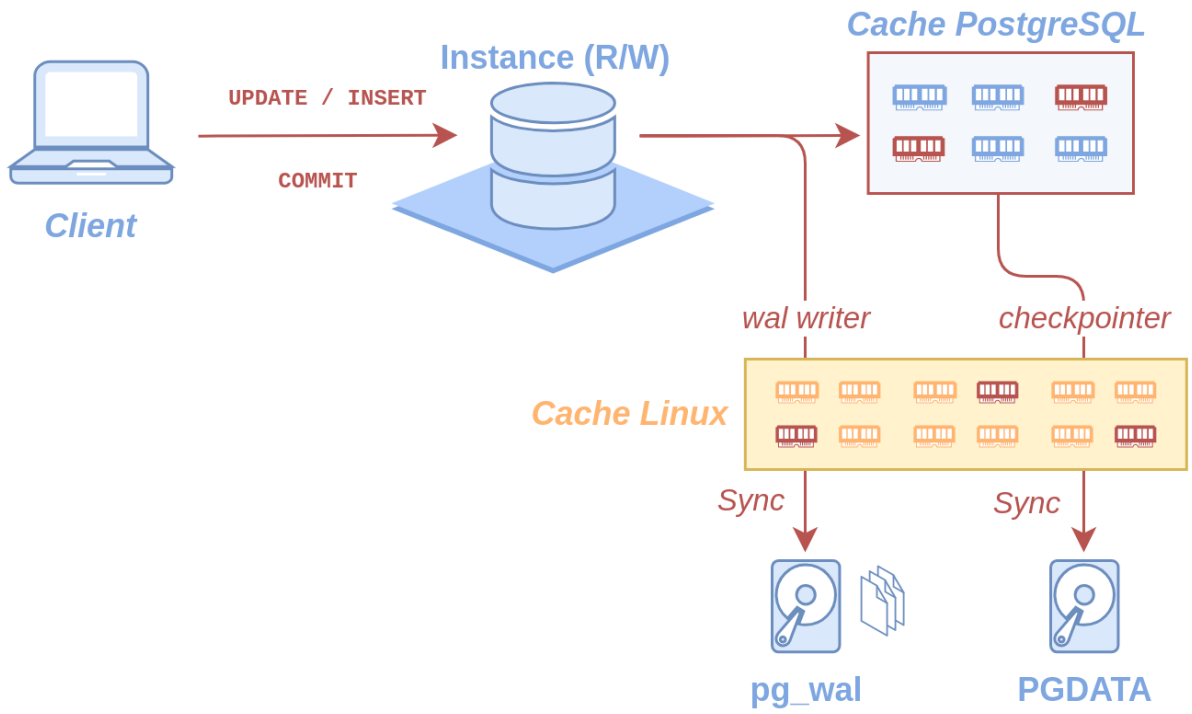
2.2.5 Niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Nous pouvons spécifier le niveau d'isolation au démarrage d'une transaction
  - `BEGIN ISOLATION LEVEL xxx;`
- Niveaux d'isolation supportés
  - `read commited` (défaut)
  - `repeatable read`
  - `serializable`

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé sera un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

Le standard SQL spécifie quatre niveaux, mais PostgreSQL n'en supporte que trois (il n'y a pas de `read uncommitted` : les lignes non encore committées par les autres transactions sont toujours invisibles).

2.2.6 Fiabilité : journaux de transactions (1)



2.2.7 Fiabilité : journaux de transactions (2)

- *Write Ahead Logs* (WAL)
- Chaque donnée est écrite **2 fois** sur le disque !
- Avantages :
  - sécurité infaillible (après `COMMIT`), intégrité, durabilité
  - écriture séquentielle rapide, et un seul *sync* sur le WAL
  - fichiers de données écrits en asynchrone
  - sauvegarde PITR et réplication fiables

Les journaux de transactions (appelés souvent WAL) sont une garantie contre les pertes de données. Il s'agit d'une technique standard de journalisation appliquée à toutes les transactions, pour garantir l'intégrité (la base reste cohérente quoiqu'il arrive) et la durabilité (ce qui est validé ne sera pas perdu).

Ainsi lors d'une modification de donnée, l'écriture au niveau du disque se fait généralement en deux temps :

- écriture des modifications dans le journal de transactions, avec écriture forcée sur disque (« synchronisation ») lors d'un `COMMIT` ;
- écriture dans le fichier de données bien plus tard, lors d'un « checkpoint ».

Ainsi en cas de crash :

- PostgreSQL redémarre ;
- il vérifie s'il reste des données non intégrées aux fichiers de données dans les journaux (mode *recovery*) ;
- si c'est le cas, ces données sont recopiées dans les fichiers de données afin de retrouver un état stable et cohérent ;
- PostgreSQL peut alors s'ouvrir sans perte des transactions validées lors du crash (une transaction interrompue, et donc jamais validée, est perdue).

Les écritures dans le journal se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture (sur un disque dur classique, c'est l'opération la plus coûteuse). De plus, comme nous n'écrivons que dans un seul fichier de transactions, la synchronisation sur disque, lors d'un `COMMIT`, peut se faire sur ce seul fichier, si le système de fichiers le supporte. Concrètement, ces journaux sont des fichiers de 16 Mo par défaut, avec des noms comme `000000010000002600000AF`, dans le répertoire `pg_wal/` de l'instance PostgreSQL (répertoire souvent sur une partition dédiée).

L'écriture définitive dans les fichiers de données est asynchrone, et généralement lissée, ce qui est meilleur pour les performances qu'une écriture immédiate. Cette opération est appelée « *checkpoint* » et périodique (5 minutes par défaut, ou plus).

Divers paramètres et fonctionnalités peuvent altérer ce comportement par défaut, par exemple pour des raisons de performances.

À côté de la sécurité et des performances, le mécanisme des journaux de transactions est aussi utilisé pour des fonctionnalités très intéressantes, comme le PITR et la réplication physique, basés sur le rejeu des informations stockées dans ces journaux.

Pour plus d'informations :

- *PostgreSQL et ses journaux de transactions*[3], Guillaume Lelarge, GNU/Linux Magazine n°108 (septembre 2008), encore d'actualité pour l'essentiel.

[3] https://public.dalibo.com/archives/publications/glmf108_postgresql_et_ses_journaux_de_transactions.pdf

2.2.8 Sauvegardes

- Sauvegarde des fichiers à froid
  - outils système
- Import/Export logique
  - `pg_dump`, `pg_dumpall`, `pg_restore`
- Sauvegarde physique à chaud
  - `pg_basebackup`
  - sauvegarde PITR

PostgreSQL supporte différentes solutions pour la sauvegarde.

La plus simple revient à sauvegarder à froid tous les fichiers des différents répertoires de données mais cela nécessite d'arrêter le serveur, ce qui occasionne une mise hors production plus ou moins longue, suivant la volumétrie à sauvegarder.

L'export logique se fait avec le serveur démarré. Plusieurs outils sont proposés : `pg_dump` pour sauvegarder une base, `pg_dumpall` pour sauvegarder toutes les bases. Suivant le format de l'export, l'import se fera avec les outils `psql` ou `pg_restore`. Les sauvegardes se font à chaud et sont cohérentes sans blocage de l'activité (seuls la suppression des tables et le changement de leur définition sont interdits).

Enfin, il est possible de sauvegarder les fichiers à chaud. Cela nécessite de mettre en place l'archivage des journaux de transactions. L'outil `pg_basebackup` est conseillé pour ce type de sauvegarde.

Il est à noter qu'il existe un grand nombre d'outils développés par la communauté pour faciliter encore plus la gestion des sauvegardes avec des fonctionnalités avancées comme le PITR (*Point In Time Recovery*) ou la gestion de la rétention, notamment pg_back (sauvegarde logique), pgBackRest ou barman (sauvegarde physique).

2.2.9 Réplication

- Réplication physique
  - instance complète
  - même architecture
- Réplication logique (PG 10+)
  - table par table / colonne par colonne avec ou sans filtre (PG 15)
  - voire opération par opération
- Asynchrone ou synchrone
- Asymétrique

PostgreSQL dispose de la réplication depuis de nombreuses années.

Le premier type de réplication intégrée est la réplication physique. Il n'y a pas de granularité, c'est forcément l'instance complète (toutes les bases de données), et au niveau des fichiers de données. Cette réplication est asymétrique : un seul serveur primaire effectue lectures comme écritures, et les serveurs secondaires n'acceptent que des lectures.

Le deuxième type de réplication est bien plus récent vu qu'il a été ajouté en version 10. Il s'agit d'une réplication logique, où les données elles-mêmes sont répliquées. Cette réplication est elle aussi asymétrique. Cependant, ceci se configure table par table (et non pas au niveau de l'instance comme pour la réplication physique). Avec la version 15, il devient possible de choisir quelles colonnes sont publiées et de filtrer les lignes à publier.

La réplication logique n'est pas intéressante quand nous voulons un serveur sur lequel basculer en cas de problème sur le primaire. Dans ce cas, il vaut mieux utiliser la réplication physique. Par contre, c'est le bon type de réplication pour une réplication partielle ou pour une mise à jour de version majeure.

Dans les deux cas, les modifications sont transmises en asynchrone (avec un délai possible). Il est cependant possible de la configurer en synchrone pour tous les serveurs ou seulement certains.

2.2.10 Extensibilité

- Extensions
  - `CREATE EXTENSION monextension ;`
  - nombreuses : contrib, packagées… selon provenance
  - notion de confiance (v13+)
  - dont langages de procédures stockées !
- Système des *hooks*
- *Background workers*

Faute de pouvoir intégrer toutes les fonctionnalités demandées dans PostgreSQL, ses développeurs se sont attachés à permettre à l'utilisateur d'étendre lui-même les fonctionnalités sans avoir à modifier le code principal.

Ils ont donc ajouté la possibilité de créer des extensions. Une extension contient un ensemble de types de données, de fonctions, d'opérateurs, etc. en un seul objet logique. Il suffit de créer ou de supprimer cet objet logique pour intégrer ou supprimer tous les objets qu'il contient. Cela facilite grandement l'installation et la désinstallation de nombreux objets. Les extensions peuvent être codées en différents langages, généralement en C ou en PL/SQL. Elles ont eu un grand succès.

La possibilité de développer des routines dans différents langages en est un exemple : perl, python, PHP, Ruby ou JavaScript sont disponibles. PL/pgSQL est lui-même une extension à proprement parler, toujours présente.

Autre exemple : la possibilité d'ajouter des types de données, des routines et des opérateurs a permis l'émergence de la couche spatiale de PostgreSQL (appelée PostGIS).

Les provenances, rôle et niveau de finition des extensions sont très variables. Certaines sont des utilitaires éprouvés fournis avec PostgreSQL (parmi les « contrib »). D'autres sont des utilitaires aussi complexes que PostGIS ou un langage de procédures stockées. Des éditeurs diffusent leur produit comme une extension plutôt que *forker* PostgreSQL (Citus, timescaledb…). Beaucoup d'extensions peuvent être installées très simplement depuis des paquets disponibles dans les dépôts habituels (de la distribution ou du PGDG), ou le site du concepteur. Certaines sont diffusées comme code source à compiler. Comme tout logiciel, il faut faire attention à en vérifier la source, la qualité, la réputation et la pérennité.

Une fois les binaires de l'extension en place sur le serveur, l'ordre `CREATE EXTENSION` suffit généralement dans la base cible, et les fonctionnalités sont immédiatement exploitables.

Les extensions sont habituellement installées par un administrateur (un utilisateur doté de l'attribut `SUPERUSER`). À partir de la version 13, certaines extensions sont déclarées de confiance `trusted`). Ces extensions peuvent être installées par un utilisateur standard (à condition qu'il dispose des droits de création dans la base et le ou les schémas concernés).

Les développeurs de PostgreSQL ont aussi ajouté des *hooks* pour accrocher du code à exécuter sur certains cas. Cela a permis entre autres de créer l'extension `pg_stat_statements` qui s'accroche au code de l'exécuteur de requêtes pour savoir quelles sont les requêtes exécutées et pour récupérer des statistiques sur ces requêtes.

Enfin, les *background workers* ont vu le jour. Ce sont des processus spécifiques lancés par le serveur PostgreSQL lors de son démarrage et stoppés lors de son arrêt. Cela a permis la création de PoWA (outil qui historise les statistiques sur les requêtes) et une amélioration très intéressante de `pg_prewarm` (sauvegarde du contenu du cache disque à l'arrêt de PostgreSQL, restauration du contenu au démarrage).

Des exemples d'extensions sont décrites dans nos modules Extensions PostgreSQL pour l'utilisateur[4], Extensions PostgreSQL pour la performance[5], Extensions PostgreSQL pour les DBA[6].

2.2.11 Sécurité

- Fichier `pg_hba.conf`
- Filtrage IP
- Authentification interne (MD5, SCRAM-SHA-256)
- Authentification externe (identd, LDAP, Kerberos…)
- Support natif de SSL

Le filtrage des connexions se paramètre dans le fichier de configuration `pg_hba.conf`. Nous pouvons y définir quels utilisateurs (déclarés auprès de PostgreSQL) peuvent se connecter à quelles bases, et depuis quelles adresses IP.

L'authentification peut se baser sur des mots de passe chiffrés propres à PostgreSQL (`md5` ou le plus récent et plus sécurisé `scram-sha-256` en version 10), ou se baser sur une méthode externe (auprès de l'OS, ou notamment LDAP ou Kerberos qui couvre aussi Active Directory).

Si PostgreSQL interroge un service de mots de passe centralisé, vous devez toujours créer les rôle dans PostgreSQL. Seule l'option `WITH PASSWORD` est inutile. Pour créer, configurer mais aussi supprimer les rôles depuis un annuaire, l'outil ldap2pg[7] existe.

L'authentification et le chiffrement de la connexion par SSL sont couverts.

[4] https://dali.bo/x1_html
[5] https://dali.bo/x2_html
[6] https://dali.bo/x3_html
[7] https://labs.dalibo.com/ldap2pg

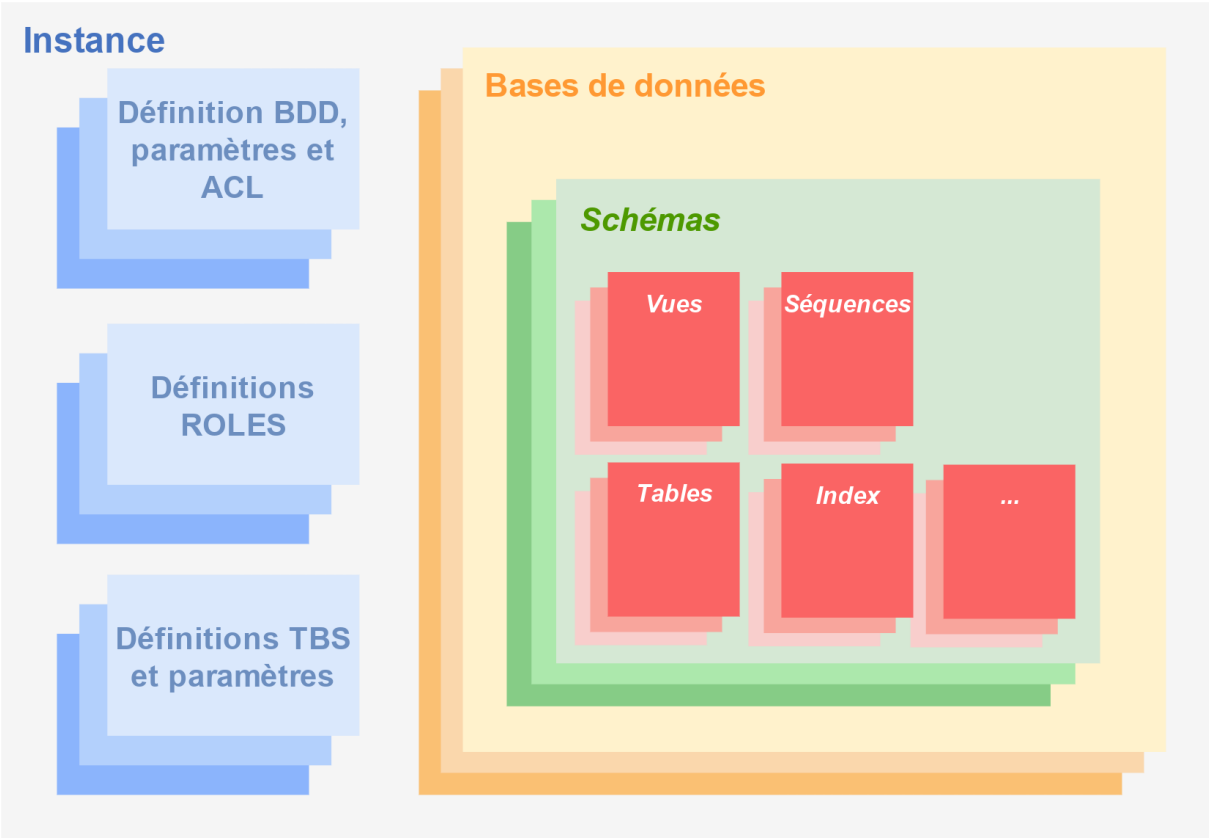
2.3 OBJETS SQL

- Instances
- Objets globaux :
  - Bases
  - Rôles
  - Tablespaces
- Objets locaux :
  - Schémas
  - Tables
  - Vues
  - Index
  - Routines
  - …

Le but de cette partie est de passer en revue les différents objets logiques maniés par un moteur de bases de données PostgreSQL.

Nous allons donc aborder la notion d'instance, les différents objets globaux et les objets locaux. Tous ne seront pas vus, mais le but est de donner une idée globale des objets et des fonctionnalités de PostgreSQL.

2.3.1 Organisation logique



Il est déjà important de bien comprendre une distinction entre les objets. Une instance est un ensemble de bases de données, de rôles et de tablespaces. Ces objets sont appelés des objets globaux parce qu'ils sont disponibles quelque soit la base de données de connexion. Chaque base de données contient ensuite des objets qui lui sont propres. Ils sont spécifiques à cette base de données et accessibles uniquement lorsque l'utilisateur est connecté à la base qui les contient. Il est donc possible de voir les bases comme des conteneurs hermétiques en dehors des objets globaux.

2.3.2 Instances

- Une instance
  - un répertoire de données
  - un port TCP
  - une configuration
  - plusieurs bases de données
- Plusieurs instances possibles sur un serveur

Une instance est un ensemble de bases de données. Après avoir installé PostgreSQL, il est nécessaire de créer un répertoire de données contenant un certain nombre de répertoires et de fichiers qui permettront à PostgreSQL de fonctionner de façon fiable. Le contenu de ce répertoire est créé initialement par la commande `initdb`. Ce répertoire stocke ensuite tous les objets des bases de données de l'instance, ainsi que leur contenu.

Chaque instance a sa propre configuration. Il n'est possible de lancer qu'un seul `postmaster` par instance, et ce dernier acceptera les connexions à partir d'un port TCP spécifique.

Il est possible d'avoir plusieurs instances sur le même serveur, physique ou virtuel. Dans ce cas, chaque instance aura son répertoire de données dédié et son port TCP dédié. Ceci est particulièrement utile quand l'on souhaite disposer de plusieurs versions de PostgreSQL sur le même serveur (par exemple pour tester une application sur ces différentes versions).

2.3.3 Rôles

- Utilisateurs / Groupes
  - Utilisateur : Permet de se connecter
- Différents attributs et droits

Une instance contient un ensemble de rôles. Certains sont prédéfinis et permettent de disposer de droits particuliers (lecture de fichier avec `pg_read_server_files`, annulation d'une requête avec `pg_signal_backend`, etc). Cependant, la majorité est composée de rôles créés pour permettre la connexion des utilisateurs.

Chaque rôle créé peut être utilisé pour se connecter à n'importe quelle base de l'instance, à condition que ce rôle en ait le droit. Ceci se gère directement avec l'attribution du droit `LOGIN` au rôle, et avec

la configuration du fichier d'accès `pg_hba.conf`.

Chaque rôle peut être propriétaire d'objets, auquel cas il a tous les droits sur ces objets. Pour les objets dont il n'est pas propriétaire, il peut se voir donner des droits, en lecture, écriture, exécution, etc par le propriétaire.

Nous parlons aussi d'utilisateurs et de groupes. Un utilisateur est un rôle qui a la possibilité de se connecter aux bases alors qu'un groupe ne le peut pas. Un groupe sert principalement à gérer plus simplement les droits d'accès aux objets.

2.3.4 Tablespaces

- Répertoire physique contenant les fichiers de données de l'instance
- Une base peut
  - se trouver sur un seul tablespace
  - être répartie sur plusieurs tablespaces
- Permet de gérer l'espace disque et les performances
- Pas de quota

Toutes les données des tables, vues matérialisées et index sont stockées dans le répertoire de données principal. Cependant, il est possible de stocker des données ailleurs que dans ce répertoire. Il faut pour cela créer un tablespace. Un tablespace est tout simplement la déclaration d'un autre répertoire de données utilisable par PostgreSQL pour y stocker des données :

```
CREATE TABLESPACE chaud LOCATION '/SSD/tbl/chaud';
```

Il est possible d'avoir un tablespace par défaut pour une base de données, auquel cas tous les objets logiques créés dans cette base seront enregistrés physiquement dans le répertoire lié à ce tablespace. Il est aussi possible de créer des objets en indiquant spécifiquement un tablespace, ou de les déplacer d'un tablespace à un autre. Un objet spécifique ne peut appartenir qu'à un seul tablespace (autrement dit, un index ne pourra pas être enregistré sur deux tablespaces). Cependant, pour les objets partitionnés, le choix du tablespace peut se faire partition par partition.

Le but des tablespaces est de fournir une solution à des problèmes d'espace disque ou de performances. Si la partition où est stocké le répertoire des données principal se remplit fortement, il est possible de créer un tablespace dans une autre partition et donc d'utiliser l'espace disque de cette partition. Si de nouveaux disques plus rapides sont à disposition, il est possible de placer les objets fréquemment utilisés sur le tablespace contenant les disques rapides. Si des disques SSD sont à disposition, il est très intéressant d'y placer les index, les fichiers de tri temporaires, des tables de travail…

Par contre, contrairement à d'autres moteurs de bases de données, PostgreSQL n'a pas de notion de quotas. Les tablespaces ne peuvent donc pas être utilisés pour contraindre l'espace disque utilisé par certaines applications ou certains rôles.

2.3.5 Bases

- Conteneur hermétique
- Un rôle ne se connecte pas à une instance
  - il se connecte forcément à une base
- Une fois connecté, il ne voit que les objets de cette base
  - contournement : foreign data wrappers, dblink

Une base de données est un conteneur hermétique. En dehors des objets globaux, le rôle connecté à une base de données ne voit et ne peut interagir qu'avec les objets contenus dans cette base. De même, il ne voit pas les objets locaux des autres bases. Néanmoins, il est possible de lui donner le droit d'accéder à certains objets d'une autre base (de la même instance ou d'une autre instance) en utilisant les *Foreign Data Wrappers* (`postgres_fdw`) ou l'extension `dblink`.

Un rôle ne se connecte pas à l'instance. Il se connecte forcément à une base spécifique.

2.3.6 Schémas

- Espace de noms
- Sous-ensemble de la base
- Non lié à un utilisateur
- Résolution des objets : `search_path`
- `pg_catalog`, `information_schema`
  - pour catalogues système (lecture seule !)

Les schémas sont des espaces de noms à l'intérieur d'une base de données permettant :

- de grouper logiquement les objets d'une base de données ;
- de séparer les utilisateurs entre eux ;

- de contrôler plus efficacement les accès aux données ;
- d'éviter les conflits de noms dans les grosses bases de données.

Un schéma n'a à priori aucun lien avec un utilisateur donné.

Un schéma est un espace logique sans lien avec les emplacements physiques des données (ne pas confondre avec les *tablespaces*).

Un utilisateur peut avoir accès à tous les schémas ou à un sous-ensemble, tout dépend des droits dont il dispose. Depuis la version 15, un nouvel utilisateur n'a le droit de créer d'objet nulle part. Dans les versions précédentes, il avait accès au schéma `public` de chaque base et pouvait y créer des objets.

Lorsque le schéma n'est pas indiqué explicitement pour les objets d'une requête, PostgreSQL recherche les objets dans les schémas listés par le paramètre `search_path` valable pour la session en cours .

Voici un exemple d'utilisation des schémas :

```
-- Création de deux schémas
CREATE SCHEMA s1;
CREATE SCHEMA s2;

-- Création d'une table sans spécification du schéma
CREATE TABLE t1 (id integer);

-- Comme le montre la méta-commande \d, la table est créée dans le schéma public

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres

-- Ceci est dû à la configuration par défaut du paramètre search_path
-- modification du search_path
SET search_path TO s1;

-- création d'une nouvelle table sans spécification du schéma
CREATE TABLE t2 (id integer);

-- Cette fois, le schéma de la nouvelle table est s1
-- car la configuration du search_path est à s1
-- Nous pouvons aussi remarquer que les tables capitaines et s1
-- ne sont plus affichées
-- Ceci est dû au fait que le search_path ne contient que le schéma s1 et
-- n'affiche donc que les objets de ce schéma.

postgres=# \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 s1     | t2   | table | postgres
```

```
-- Nouvelle modification du search_path
SET search_path TO s1, public;

-- Cette fois, les deux tables apparaissent

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
CREATE TABLE s2.t3 (id integer);

-- changement du search_path pour voir la table
SET search_path TO s1, s2, public;

-- La table apparaît bien, et le schéma d'appartenance est bien s2

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Création d'une nouvelle table en spécifiant cette fois le schéma
-- attention, cette table a un nom déjà utilisé par une autre table
CREATE TABLE s2.t2 (id integer);

-- La création se passe bien car, même si le nom de la table est identique,
-- le schéma est différent
-- Par contre, \d ne montre que la première occurence de la table
-- ici, nous ne voyons t2 que dans s1

postgres=# \d
                List of relations
 Schema |       Name        |   Type   |  Owner
--------+-------------------+----------+----------
 public | capitaines        | table    | postgres
 public | capitaines_id_seq | sequence | postgres
 public | t1                | table    | postgres
 s1     | t2                | table    | postgres
 s2     | t3                | table    | postgres

-- Changeons le search_path pour placer s2 avant s1
SET search_path TO s2, s1, public;

-- Maintenant, la seule table t2 affichée est celle du schéma s2

postgres=# \d
                List of relations
```

```
 Schema |        Name         |   Type    |  Owner
--------+---------------------+-----------+----------
 public | capitaines          | table     | postgres
 public | capitaines_id_seq   | sequence  | postgres
 public | t1                  | table     | postgres
 s2     | t2                  | table     | postgres
 s2     | t3                  | table     | postgres
```

Tous ces exemples se basent sur des ordres de création de table. Cependant, le comportement serait identique sur d'autres types de commande (`SELECT`, `INSERT`, etc) et sur d'autres types d'objets locaux.

Pour des raisons de sécurité, il est très fortement conseillé de laisser le schéma `public` en toute fin du `search_path`. En effet, avant la version 15, s'il est placé au début, comme tout le monde avait le droit de créer des objets dans `public`, quelqu'un de mal intentionné pouvait placer un objet dans le schéma `public` pour servir de proxy à un autre objet d'un schéma situé après `public`. Même si la version 15 élimine ce risque, il reste la bonne pratique d'adapter le `search_path` pour placer les schémas applicatifs en premier.

Les schémas `pg_catalog` et `information_schema` contiennent des tables utilitaires (« catalogues système ») et des vues. Les catalogues système représentent l'endroit où une base de données relationnelle stocke les métadonnées des schémas, telles que les informations sur les tables, et les colonnes, et des données de suivi interne. Dans PostgreSQL, ce sont de simples tables. Un simple utilisateur lit fréquemment ces tables, plus ou moins directement, mais n'a aucune raison d'y modifier des données. Toutes les opérations habituelles pour un utilisateur ou administrateur sont disponibles sous la forme de commandes SQL.

Ne modifiez jamais directement les tables et vues système dans les schémas `pg_catalog` et `information_schema` ; n'y ajoutez ni n'y effacez jamais rien !

Même si cela est techniquement possible, seules des exceptions particulièrement ésotériques peuvent justifier une modification directe des tables systèmes (par exemple, une correction de vue système, suite à un bug corrigé dans une version mineure). Ces tables n'apparaissent d'ailleurs pas dans une sauvegarde logique (`pg_dump`).

2.3.7 Tables

Par défaut, une table est :

- Permanente
  - si temporaire, vivra le temps de la session (ou de la transaction)
- Journalisée
  - si *unlogged*, perdue en cas de crash, pas de réplication
- Non partitionnée
  - partitionnement possible par intervalle, valeur ou hachage

Par défaut, les tables sont permanentes, journalisées et non partitionnées.

Il est possible de créer des tables temporaires (`CREATE TEMPORARY TABLE`). Celles-ci ne sont visibles que par la session qui les a créées et seront supprimées par défaut à la fin de cette session. Il est aussi possible de les supprimer automatiquement à la fin de la transaction qui les a créées. Il n'existe pas dans PostgreSQL de notion de table temporaire globale. Cependant, une extension[8] existe pour combler leur absence.

Pour des raisons de performance, il est possible de créer une table non journalisée (`CREATE UNLOGGED TABLE`). La définition de la table est journalisée mais pas son contenu. De ce fait, en cas de crash, il est impossible de dire si la table est corrompue ou non, et donc, au redémarrage du serveur, PostgreSQL vide la table de tout contenu. De plus, n'étant pas journalisée, la table n'est pas présente dans les sauvegardes PITR, ni repliquée vers d'éventuels serveurs secondaires.

Enfin, depuis la version 10, il est possible de partitionner les tables suivant un certain type de partitionnement : par intervalle, par valeur ou par hachage.

[8] https://github.com/darold/pgtt

2.3.8 Vues

- Masquer la complexité
  - structure : interface cohérente vers les données, même si les tables évoluent
  - sécurité : contrôler l'accès aux données de manière sélective
- Vues matérialisées
  - à rafraîchir à une certaine fréquence

Le but des vues est de masquer une complexité, qu'elle soit du côté de la structure de la base ou de l'organisation des accès. Dans le premier cas, elles permettent de fournir un accès qui ne change pas même si les structures des tables évoluent. Dans le second cas, elles permettent l'accès à seulement certaines colonnes ou certaines lignes. De plus, les vues étant exécutées avec les mêmes droits que l'utilisateur qui les a créées, cela permet un changement temporaire des droits d'accès très appréciable dans certains cas.

Voici un exemple d'utilisation :

```
SET search_path TO public;

-- création de l'utilisateur guillaume
-- il n'aura pas accès à la table capitaines
-- par contre, il aura accès à la vue capitaines_anon
CREATE ROLE guillaume LOGIN;

-- ajoutons une colonne à la table capitaines
-- et ajoutons-y des données
ALTER TABLE capitaines ADD COLUMN num_cartecredit text;
INSERT INTO capitaines (nom, age, num_cartecredit)
  VALUES ('Robert Surcouf', 20, '1234567890123456');

-- création de la vue
CREATE VIEW capitaines_anon AS
  SELECT nom, age, substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- ajout du droit de lecture à l'utilisateur guillaume
GRANT SELECT ON TABLE capitaines_anon TO guillaume;

-- connexion en tant qu'utilisateur guillaume
SET ROLE TO guillaume;

-- vérification qu'on lit bien la vue mais pas la table
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |    num_cc_anon
----------------+-----+----------------
```

```
 Robert Surcouf |  20 | 123456789******
```

```
-- tentative de lecture directe de la table
SELECT * FROM capitaines;
ERROR:  permission denied for relation capitaines
```

Il est possible de modifier une vue en lui ajoutant des colonnes à la fin, au lieu de devoir les détruire et recréer (ainsi que toutes les vues qui en dépendent, ce qui peut être fastidieux).

Par exemple :

```
SET ROLE postgres;

CREATE OR REPLACE VIEW capitaines_anon AS SELECT
  nom,age,substring(num_cartecredit,0,10)||'******' AS num_cc_anon,
  md5(substring(num_cartecredit,0,10)) AS num_md5_cc
  FROM capitaines;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';
```

```
      nom       | age |   num_cc_anon    |            num_md5_cc
----------------+-----+------------------+----------------------------------
 Robert Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

Nous pouvons aussi modifier les données au travers des vues simples, sans ajout de code et de trigger :

```
UPDATE capitaines_anon SET nom = 'Nicolas Surcouf' WHERE nom = 'Robert Surcouf';

SELECT * from capitaines_anon WHERE nom LIKE '%Surcouf';
```

```
       nom       | age |   num_cc_anon    |            num_md5_cc
-----------------+-----+------------------+----------------------------------
 Nicolas Surcouf |  20 | 123456789****** | 25f9e794323b453885f5181f1b624d0b
```

```
UPDATE capitaines_anon SET num_cc_anon = '123456789xxxxxx'
  WHERE nom = 'Nicolas Surcouf';
```

```
ERROR:  cannot update column "num_cc_anon" of view "capitaines_anon"
DETAIL:  View columns that are not columns of their base relation
         are not updatable.
```

PostgreSQL gère le support natif des vues matérialisées (`CREATE MATERIALIZED VIEW nom_vue_mat AS SELECT …`). Les vues matérialisées sont des vues dont le contenu est figé sur disque, permettant de ne pas recalculer leur contenu à chaque appel. De plus, il est possible de les indexer pour accélérer leur consultation. Il faut cependant faire attention à ce que leur contenu reste synchrone avec le reste des données.

Les vues matérialisées ne sont pas mises à jour automatiquement, il faut demander explicitement le rafraîchissement (`REFRESH MATERIALIZED VIEW`). Avec la clause `CONCURRENTLY`, s'il y a un index d'unicité, le rafraîchissement ne bloque pas les sessions lisant en même temps les données d'une vue matérialisée.

```
-- Suppression de la vue
DROP VIEW capitaines_anon;
```

```
-- Création de la vue matérialisée
CREATE MATERIALIZED VIEW capitaines_anon AS
  SELECT nom,
    age,
    substring(num_cartecredit, 0, 10) || '******' AS num_cc_anon
  FROM capitaines;

-- Les données sont bien dans la vue matérialisée
SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+----------------
 Nicolas Surcouf |  20 | 123456789******

-- Mise à jour d'une ligne de la table
-- Cette mise à jour est bien effectuée, mais la vue matérialisée
-- n'est pas impactée
UPDATE capitaines SET nom = 'Robert Surcouf' WHERE nom = 'Nicolas Surcouf';

SELECT * FROM capitaines WHERE nom LIKE '%Surcouf';

 id |      nom       | age | num_cartecredit
----+----------------+-----+-----------------
  1 | Robert Surcouf |  20 | 1234567890123456

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

       nom       | age |   num_cc_anon
-----------------+-----+----------------
 Nicolas Surcouf |  20 | 123456789******

-- Le résultat est le même mais le plan montre bien que PostgreSQL ne passe
-- plus par la table mais par la vue matérialisée :
EXPLAIN SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

                         QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on capitaines_anon  (cost=0.00..20.62 rows=1 width=68)
   Filter: (nom ~~ '%Surcouf'::text)

-- Après un rafraîchissement explicite de la vue matérialisée,
-- cette dernière contient bien les bonnes données
REFRESH MATERIALIZED VIEW capitaines_anon;

SELECT * FROM capitaines_anon WHERE nom LIKE '%Surcouf';

      nom       | age |   num_cc_anon
----------------+-----+----------------
 Robert Surcouf |  20 | 123456789******

-- Pour rafraîchir la vue matérialisée sans bloquer les autres sessions :

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;

ERROR:  cannot refresh materialized view "public.capitaines_anon" concurrently
HINT:  Create a unique index with no WHERE clause on one or more columns
       of the materialized view.
```

```
-- En effet, il faut un index d'unicité pour faire un rafraîchissement
-- sans bloquer les autres sessions.
CREATE UNIQUE INDEX ON capitaines_anon(nom);

REFRESH MATERIALIZED VIEW CONCURRENTLY capitaines_anon;
```

2.3.9 Index

- Algorithmes supportés
  - B-tree (par défaut)
  - Hash
  - GiST / SP-GiST
  - GIN
  - BRIN
  - Bloom
- Type
  - Mono ou multicolonne
  - Partiel
  - Fonctionnel
  - Couvrant

PostgreSQL propose plusieurs algorithmes d'index.

Pour une indexation standard, nous utilisons en général un index B-tree, de par ses nombreuses possibilités et ses très bonnes performances.

Les index hash sont peu utilisés, essentiellement dans la comparaison d'égalité de grandes chaînes de caractères.

Moins simples d'abord, les index plus spécifiques (GIN, GIST) sont spécialisés pour les grands volumes de données complexes et multidimensionnelles : indexation textuelle, géométrique, géographique, ou de tableaux de données par exemple.

Les index BRIN sont des index très compacts destinés aux grandes tables où les données sont fortement corrélées par rapport à leur emplacement physique sur les disques.

Les index bloom sont des index probabilistes visant à indexer de nombreuses colonnes interrogées simultanément. Ils nécessitent l'ajout d'une extension (nommée `bloom`). Contrairement aux index btree, les index bloom ne dépendent pas de l'ordre des colonnes.

Le module `pg_trgm` permet l'utilisation d'index dans des cas habituellement impossibles, comme les expressions rationnelles et les `LIKE '%...%'`.

Généralement, l'indexation porte sur la valeur d'une ou plusieurs colonnes. Il est néanmoins possible de n'indexer qu'une partie des lignes (index partiel) ou le résultat d'une fonction sur une ou plusieurs colonnes en paramètre. Enfin, il est aussi possible de modifier les index de certaines contraintes (unicité et clé primaire) pour inclure des colonnes supplémentaires.

Plus d'informations :

- Article Wikipédia sur les arbres B[9] ;
- Article Wikipédia sur les tables de hachage[10] ;
- Documentation officielle française[11].

2.3.10 Types de données

- Types de base
  - natif : `int`, `float`
  - standard SQL : `numeric`, `char`, `varchar`, `date`, `time`, `timestamp`, `bool`
- Type complexe
  - tableau
  - JSON (`jsonb`), XML
  - vecteur (données LLM, FTS)
- Types métier
  - réseau, géométrique, etc.
- Types créés par les utilisateurs
  - structure SQL, C, Domaine, Enum

PostgreSQL dispose d'un grand nombre de types de base, certains natifs (comme la famille des `integer` et celle des `float`), et certains issus de la norme SQL (`numeric`, `char`, `varchar`, `date`, `time`, `timestamp`, `bool`).

Il dispose aussi de types plus complexes. Les tableaux (`array`) permettent de lister un ensemble de valeurs discontinues. Les intervalles (`range`) permettent d'indiquer toutes les valeurs comprises entre une valeur de début et une valeur de fin. Ces deux types dépendent évidemment d'un type de

[9] https://fr.wikipedia.org/wiki/Arbre_B
[10] https://fr.wikipedia.org/wiki/Table_de_hachage
[11] https://docs.postgresql.fr/current/textsearch-indexes.html

base : tableau d'entiers, intervalle de dates, etc. Existent aussi les types complexes, les données XML et JSON (préférer le type optimisé `jsonb`).

PostgreSQL sait travailler avec des vecteurs pour des calculs avancé. De base, le type `tsvector` permet la recherche plein texte, avec calcul de proximité de mots dans un texte, pondération des résultats, etc. L'extension `pgvector` permet de stocker et d'indexer des vecteurs utilisé par les algorithmes LLM implémentés dans les IA génératives.

Enfin, il existe des types métiers ayant trait principalement au réseau (adresse IP, masque réseau), à la géométrie (point, ligne, boite). Certains sont apportés par des extensions.

Tout ce qui vient d'être décrit est natif. Il est cependant possible de créer ses propres types de données, soit en SQL soit en C. Les possibilités et les performances ne sont évidemment pas les mêmes.

Voici comment créer un type en SQL :

```
CREATE TYPE serveur AS (
  nom             text,
  adresse_ip      inet,
  administrateur  text
);
```

Ce type de données va pouvoir être utilisé dans tous les objets SQL habituels : table, routine, opérateur (pour redéfinir l'opérateur `+` par exemple), fonction d'agrégat, contrainte, etc.

Voici un exemple de création d'un opérateur :

```
CREATE OPERATOR + (
    leftarg = stock,
    rightarg = stock,
    procedure = stock_fusion,
    commutator = +
);
```

(Il faut au préalable avoir défini le type `stock` et la fonction `stock_fusion`.)

Il est aussi possible de définir des domaines. Ce sont des types créés par les utilisateurs à partir d'un type de base et en lui ajoutant des contraintes supplémentaires.

En voici un exemple :

```
CREATE DOMAIN code_postal_francais AS text CHECK (value ~ '^\d{5}$');
ALTER TABLE capitaines ADD COLUMN cp code_postal_francais;
UPDATE capitaines SET cp = '35400' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET cp = '1420' WHERE nom = 'Haddock';

ERROR:  value for domain code_postal_francais violates check constraint
        "code_postal_francais_check"

UPDATE capitaines SET cp = '01420' WHERE nom = 'Haddock';
SELECT * FROM capitaines;

 id |      nom        | age | num_cartecredit  |  cp
----+-----------------+-----+------------------+-------
  1 | Robert Surcouf  |  20 | 1234567890123456 | 35400
  1 | Haddock         |  35 |                  | 01420
```

Les domaines permettent d'intégrer la déclaration des contraintes à la déclaration d'un type, et donc de simplifier la maintenance de l'application si ce type peut être utilisé dans plusieurs tables : si la définition du code postal est insuffisante pour une évolution de l'application, il est possible de la modifier par un `ALTER DOMAIN`, et définir de nouvelles contraintes sur le domaine. Ces contraintes seront vérifiées sur l'ensemble des champs ayant le domaine comme type avant que la nouvelle version du type ne soit considérée comme valide.

Le défaut par rapport à des contraintes `CHECK` classiques sur une table est que l'information ne se trouvant pas dans la table, les contraintes sont plus difficiles à lister sur une table.

Enfin, il existe aussi les enums. Ce sont des types créés par les utilisateurs composés d'une liste ordonnée de chaînes de caractères.

En voici un exemple :

```
CREATE TYPE jour_semaine
  AS ENUM ('Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi',
  'Samedi', 'Dimanche');

ALTER TABLE capitaines ADD COLUMN jour_sortie jour_semaine;

UPDATE capitaines SET jour_sortie = 'Mardi' WHERE nom LIKE '%Surcouf';
UPDATE capitaines SET jour_sortie = 'Samedi' WHERE nom LIKE 'Haddock';

SELECT * FROM capitaines WHERE jour_sortie >= 'Jeudi';
```

```
 id |   nom    | age | num_cartecredit | cp | jour_sortie
----+----------+-----+-----------------+----+-------------
  1 | Haddock |  35 |                 |    | Samedi
```

Les *enums* permettent de déclarer une liste de valeurs statiques dans le dictionnaire de données plutôt que dans une table externe sur laquelle il faudrait rajouter des jointures : dans l'exemple, nous aurions pu créer une table `jour_de_la_semaine`, et stocker la clé associée dans `planning`. Nous aurions pu tout aussi bien positionner une contrainte `CHECK`, mais nous n'aurions plus eu une liste ordonnée.

Exemple d'utilisation :

- Conférence de Heikki Linakangas sur la création d'un type color[12].

[12] https://wiki.postgresql.org/images/1/11/FOSDEM2011-Writing_a_User_defined_type.pdf

2.3.11 Contraintes

- `CHECK`
  - `prix > 0`
- `NOT NULL`
  - `id_client NOT NULL`
- Unicité
  - `id_client UNIQUE`
- Clés primaires
  - `UNIQUE NOT NULL` ==> `PRIMARY KEY (id_client)`
- Clés étrangères
  - `produit_id REFERENCES produits(id_produit)`
- `EXCLUDE`
  - `EXCLUDE USING gist (room WITH =, during WITH &&)`

Les contraintes sont la garantie de conserver des données de qualité ! Elles permettent une vérification qualitative des données, beaucoup plus fine qu'en définissant uniquement un type de données.

Les exemples ci-dessus reprennent :

- un prix qui doit être strictement positif ;
- un identifiant qui ne doit pas être vide (sinon des jointures filtreraient des lignes) ;
- une valeur qui doit être unique (comme des numéros de clients ou de facture) ;
- une clé primaire (unique non nulle), qui permet d'identifier précisément une ligne ;
- une clé étrangère vers la clé primaire d'une autre table (là encore pour garantir l'intégrité des jointures) ;
- une contrainte d'exclusion interdisant que deux plages temporelles se recouvrent dans la réservation de la même salle de réunion.

Les contraintes d'exclusion permettent un test sur plusieurs colonnes avec différents opérateurs (et non uniquement l'égalité, comme dans le cas d'une contrainte unique, qui n'est qu'une contrainte d'exclusion très spécialisée). Si le test se révèle positif, la ligne est refusée.

Une contrainte peut porter sur plusieurs champs et un champ peut être impliqué dans plusieurs

contraintes :

```
CREATE TABLE commandes (
    no_commande     varchar(16) CHECK (no_commande ~ '^[A-Z0-9]*$'),
    id_entite_commerciale int REFERENCES entites_commerciales,
    id_client       int       REFERENCES clients,
    date_commande   date      NOT NULL,
    date_livraison  date      CHECK (date_livraison >= date_commande),
    PRIMARY KEY (no_commande, id_entite_commerciale)
);
```

```
\d commandes
                                Table « public.commandes »
        Colonne         |         Type          | … | NULL-able | Par défaut
-----------------------+-----------------------+---+-----------+------------
 no_commande            | character varying(16) |   | not null  |
 id_entite_commerciale  | integer               |   | not null  |
 id_client              | integer               |   |           |
 date_commande          | date                  |   | not null  |
 date_livraison         | date                  |   |           |
Index :
    "commandes_pkey" PRIMARY KEY, btree (no_commande, id_entite_commerciale)
Contraintes de vérification :
    "commandes_check" CHECK (date_livraison >= date_commande)
    "commandes_no_commande_check" CHECK (no_commande::text ~ '^[A-Z0-9]*$'::text)
Contraintes de clés étrangères :
    "commandes_id_client_fkey" FOREIGN KEY (id_client) REFERENCES clients(id_client)
    "commandes_id_entite_commerciale_fkey" FOREIGN KEY (id_entite_commerciale)
 ↪  REFERENCES entites_commerciales(id_entite_commerciale)
```

Les contraintes doivent être vues comme la dernière ligne de défense de votre application face aux bugs. En effet, le code d'une application change beaucoup plus souvent que le schéma, et les données survivent souvent à l'application, qui peut être réécrite entretemps. Quoi qu'il se passe, des contraintes judicieuses garantissent qu'il n'y aura pas d'incohérence logique dans la base.

Si elles sont gênantes pour le développeur (car elles imposent un ordre d'insertion ou de mise à jour), il faut se rappeler que les contraintes peuvent être « débrayées » le temps d'une transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED ;
…
COMMIT ;
```

Les contraintes ne seront validées qu'au `COMMIT`.

Sur le sujet, voir par exemple *Constraints: a Developer's Secret Weapon*[13] de Will Leinweber (pgDay Paris 2018) (slides[14], vidéo[15]).

Du point de vue des performances, les contraintes permettent au planificateur d'optimiser les requêtes. Par exemple, le planificateur sait ne pas prendre en compte certaines jointures, notamment grâce à l'existence d'une contrainte d'unicité. (Sur ce point, la version 15 améliore les contraintes

[13] https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2192-constraints-a-developers-secret-weapon/
[14] https://www.postgresql.eu/events/pgdayparis2018/sessions/session/1835/slides/70/2018-03-15%20constraints%20a%20developers%20secret%20weapon%20pgday%20paris.pdf
[15] https://youtu.be/hWh8QoV8z8k

d'unicité en permettant de choisir si la valeur NULL est considérée comme unique ou pas. Par défaut et historiquement, une valeur NULL n'étant pas égale à une valeur NULL, les valeurs NULL sont considérées distinctes, et donc on peut avoir plusieurs valeurs NULL dans une colonne ayant une contrainte d'unicité.)

2.3.12 Colonnes à valeur générée

- Valeur calculée à l'insertion
- `DEFAULT`
- Identité
  - `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY`
- Expression
  - `GENERATED ALWAYS AS ( generation_expr ) STORED`

Une colonne a par défaut la valeur `NULL` si aucune valeur n'est fournie lors de l'insertion de la ligne. Il existe néanmoins trois cas où le moteur peut substituer une autre valeur.

Le plus connu correspond à la clause `DEFAULT`. Dans ce cas, la valeur insérée correspond à la valeur indiquée avec cette clause si aucune valeur n'est indiquée pour la colonne. Si une valeur est précisée, cette valeur surcharge la valeur par défaut. L'exemple suivant montre cela :

```
CREATE TABLE t2 (c1 integer, c2 integer, c3 integer DEFAULT 10);
INSERT INTO t2 (c1, c2, c3) VALUES (1, 2, 3);
INSERT INTO t2 (c1) VALUES (2);
SELECT * FROM t2;
```

```
 c1 | c2 | c3
----+----+----
  1 |  2 |  3
  2 |    | 10
```

La clause `DEFAULT` ne peut pas être utilisée avec des clauses complexes, notamment des clauses comprenant des requêtes.

Pour aller un peu plus loin, à partir de PostgreSQL 12, il est possible d'utiliser `GENERATED ALWAYS AS ( expression )`. Cela permet d'avoir une valeur calculée pour la colonne, valeur qui ne peut pas être surchargée, ni à l'insertion, ni à la mise à jour (mais qui est bien stockée sur le disque).

Comme exemple, nous allons reprendre la table `capitaines` et lui ajouter une colonne ayant comme valeur la version modifiée du numéro de carte de crédit :

```
ALTER TABLE capitaines
  ADD COLUMN num_cc_anon text
```

```
  GENERATED ALWAYS AS (substring(num_cartecredit, 0, 10) || '******') STORED;

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;

      nom       | num_cartecredit  |   num_cc_anon
----------------+------------------+-----------------
 Robert Surcouf | 1234567890123456 | 123456789******
 Haddock        |                  |

INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi', 'test');

ERROR:  cannot insert into column "num_cc_anon"
DETAIL:  Column "num_cc_anon" is a generated column.

INSERT INTO capitaines VALUES
  (2, 'Joseph Pradere-Niquet', 40, '9876543210987654', '44000', 'Lundi');

SELECT nom, num_cartecredit, num_cc_anon FROM capitaines;

          nom          | num_cartecredit  |   num_cc_anon
-----------------------+------------------+-----------------
 Robert Surcouf        | 1234567890123456 | 123456789******
 Haddock               |                  |
 Joseph Pradere-Niquet | 9876543210987654 | 987654321******
```

Enfin, `GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY` permet d'obtenir une colonne d'identité, bien meilleure que ce que le pseudo-type `serial` propose. Si `ALWAYS` est indiqué, la valeur n'est pas modifiable.

```
ALTER TABLE capitaines
  ADD COLUMN id2 integer GENERATED ALWAYS AS IDENTITY;

SELECT nom, id2 FROM capitaines;

          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3

INSERT INTO capitaines (nom) VALUES ('Tom Souville');

SELECT nom, id2 FROM capitaines;

          nom          | id2
-----------------------+-----
 Robert Surcouf        |   1
 Haddock               |   2
 Joseph Pradere-Niquet |   3
 Tom Souville          |   4
```

Le type `serial` est remplacé par le type `integer` et une séquence comme le montre l'exemple suivant. C'est un problème dans la mesure où la déclaration qui est faite à la création de la table produit un résultat différent en base et donc dans les exports de données.

```
CREATE TABLE tserial(s serial);
```

```
                              Table "public.tserial"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-------------------------------------
 s      | integer |           | not null | nextval('tserial_s_seq'::regclass)
```

2.3.13 Langages

- Procédures & fonctions en différents langages
- Par défaut : SQL, C et PL/pgSQL
- Extensions officielles : Perl, Python
- Mais aussi Java, Ruby, Javascript…
- Intérêts : fonctionnalités, performances

Les langages officiellement supportés par le projet sont :

- PL/pgSQL ;
- PL/Perl[16] ;
- PL/Python[17] ;
- PL/Tcl.

Voici une liste non exhaustive des langages procéduraux disponibles, à différents degrés de maturité :

- PL/sh[18] ;
- PL/R[19] ;
- PL/Java[20] ;
- PL/lolcode ;
- PL/Scheme ;
- PL/PHP ;
- PL/Ruby ;
- PL/Lua[21] ;
- PL/pgPSM ;
- PL/v8[22] (Javascript).

Le wiki PostgreSQL contient un tableau des langages supportés[23].

[16] https://docs.postgresql.fr/current/plperl.html
[17] https://docs.postgresql.fr/current/plpython.html
[18] https://github.com/petere/plsh
[19] https://github.com/postgres-plr/plr
[20] https://tada.github.io/pljava/
[21] https://github.com/pllua/pllua
[22] https://github.com/plv8/plv8
[23] https://wiki.postgresql.org/wiki/PL_Matrix

Pour qu'un langage soit utilisable, il doit être activé au niveau de la base où il sera utilisé. Les trois langages activés par défaut sont le C, le SQL et le PL/pgSQL. Les autres doivent être ajoutés à partir des paquets de la distribution ou du PGDG, ou compilés à la main, puis l'extension installée dans la base :

```
CREATE EXTENSION plperl ;
CREATE EXTENSION plpython3u ;
-- etc.
```

Ces fonctions peuvent être utilisées dans des index fonctionnels et des triggers comme toute fonction SQL ou PL/pgSQL.

Chaque langage a ses avantages et inconvénients. Par exemple, PL/pgSQL est très simple à apprendre mais n'est pas performant quand il s'agit de traiter des chaînes de caractères. Pour ce traitement, il est souvent préférable d'utiliser PL/Perl, voire PL/Python. Évidemment, une routine en C aura les meilleures performances mais sera beaucoup moins facile à coder et à maintenir, et ses bugs seront susceptibles de provoquer un plantage du serveur.

Par ailleurs, les procédures peuvent s'appeler les unes les autres quel que soit le langage. S'ajoute l'intérêt de ne pas avoir à réécrire en PL/pgSQL des fonctions existantes dans d'autres langages ou d'accéder à des modules bien établis de ces langages.

2.3.14 Fonctions & procédures

- Fonction
  - renvoie une ou plusieurs valeurs
  - `SETOF` ou `TABLE` pour plusieurs lignes
- Procédure (v11+)
  - ne renvoie rien
  - peut gérer le transactionnel dans certains cas

Historiquement, PostgreSQL ne proposait que l'écriture de fonctions. Depuis la version 11, il est aussi possible de créer des procédures. Le terme « routine » est utilisé pour signifier procédure ou fonction.

Une fonction renvoie une donnée. Cette donnée peut comporter une ou plusieurs colonnes. Elle peut aussi avoir plusieurs lignes dans le cas d'une fonction `SETOF` ou `TABLE`.

Une procédure ne renvoie rien. Elle a cependant un gros avantage par rapport aux fonctions dans le fait qu'elle peut gérer le transactionnel. Elle peut valider ou annuler la transaction en cours. Dans ce cas, une nouvelle transaction est ouverte immédiatement après la fin de la transaction précédente.

2.3.15 Opérateurs

- Dépend d'un ou deux types de données
- Utilise une fonction prédéfinie :

```
CREATE OPERATOR //
  (FUNCTION=division0,
  LEFTARG=integer,
  RIGHTARG=integer);
```

Il est possible de créer de nouveaux opérateurs sur un type de base ou sur un type utilisateur. Un opérateur exécute une fonction, soit à un argument pour un opérateur unitaire, soit à deux arguments pour un opérateur binaire.

Voici un exemple d'opérateur acceptant une division par zéro sans erreur :

```
-- définissons une fonction de division en PL/pgSQL
CREATE FUNCTION division0 (p1 integer, p2 integer) RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
  IF p2 = 0 THEN
      RETURN NULL;
  END IF;

  RETURN p1 / p2;
END
$$;

-- créons l'opérateur
CREATE OPERATOR // (FUNCTION = division0, LEFTARG = integer, RIGHTARG = integer);

-- une division normale se passe bien

SELECT 10/5;

 ?column?
----------
        2

SELECT 10//5;

 ?column?
----------
        2

-- une division par 0 ramène une erreur avec l'opérateur natif
SELECT 10/0;

ERROR:  division by zero
```

```
-- une division par 0 renvoie NULL avec notre opérateur
SELECT 10//0;

 ?column?
----------

(1 row)
```

2.3.16 Triggers

- Opérations : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`
- Trigger sur :
  - une table, voire une colonne, et/ou avec condition
  - une vue
  - une opération DDL
  - une connexion
- Tables de transition
- Effet sur :
  - l'ensemble de la requête (`FOR STATEMENT`)
  - chaque ligne impactée (`FOR EACH ROW`)
- N'importe quel langage supporté

Les triggers peuvent être exécutés avant (`BEFORE`) ou après (`AFTER`) une opération.

Il est possible de les déclencher pour chaque ligne impactée (`FOR EACH ROW`) ou une seule fois pour l'ensemble de la requête (`FOR STATEMENT`). Dans le premier cas, il est possible d'accéder à la ligne impactée (ancienne et nouvelle version). Dans le deuxième cas, des tables de transition donnent à l'utilisateur une vision des lignes avant et après modification.

Par ailleurs, les triggers peuvent être écrits dans n'importe lequel des langages de routine supportés par PostgreSQL (C, PL/pgSQL, PL/Perl, etc.)

Exemple :

```
ALTER TABLE capitaines ADD COLUMN salaire integer;

CREATE FUNCTION verif_salaire()
RETURNS trigger AS $verif_salaire$
BEGIN
  -- Nous verifions que les variables ne sont pas vides
  IF NEW.nom IS NULL THEN
    RAISE EXCEPTION 'Le nom ne doit pas être null.';
```

```
  END IF;

  IF NEW.salaire IS NULL THEN
    RAISE EXCEPTION 'Le salaire ne doit pas être null.';
  END IF;

  -- pas de baisse de salaires !
  IF NEW.salaire < OLD.salaire THEN
    RAISE EXCEPTION 'Pas de baisse de salaire !';
  END IF;

  RETURN NEW;
END;
$verif_salaire$ LANGUAGE plpgsql;

CREATE TRIGGER verif_salaire BEFORE INSERT OR UPDATE ON capitaines
  FOR EACH ROW EXECUTE PROCEDURE verif_salaire();

UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 3000 WHERE nom = 'Robert Surcouf';
UPDATE capitaines SET salaire = 2000 WHERE nom = 'Robert Surcouf';

ERROR:  pas de baisse de salaire !
CONTEXTE : PL/pgSQL function verif_salaire() line 13 at RAISE
```

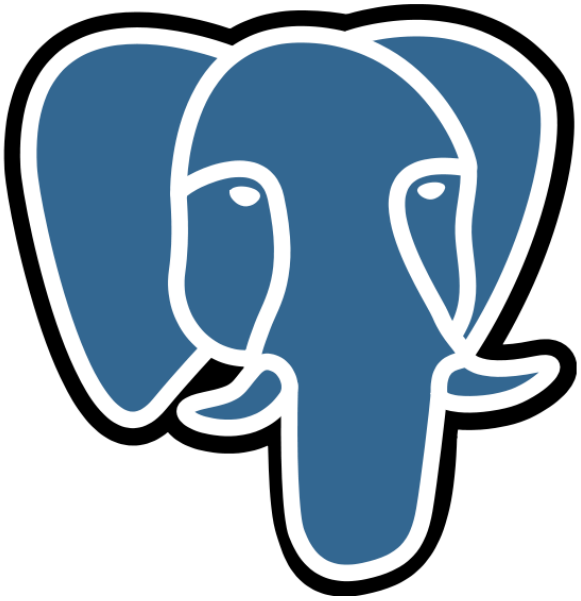
2.3.17 Questions

N'hésitez pas, c'est le moment !

2.4 QUIZ

https://dali.bo/a2_quiz

3/ Installation de PostgreSQL



3.1 INTRODUCTION

- Pré-requis
- Installation depuis les sources
- Installation depuis les binaires
  - installation à partir des paquets
  - installation sous Windows
- Installation dans les conteneurs
- Premiers réglages
- Mises à jours

Il existe trois façons d'installer PostgreSQL :

- Les installeurs graphiques :
  - uniquement Windows et macOS ;
  - avantages : installation facile, idéale pour les nouveaux venus ;
  - inconvénients : pas d'intégration avec le système de paquets du système d'exploitation.
- les paquets du système :
  - avantages : meilleure intégration avec les autres logiciels, idéal pour un serveur en production ;
  - inconvénients : aucun ?
- Le code source :
  - avantages : configuration très fine, ajout de patchs, intéressant pour les utilisateurs expérimentés et les testeurs, ou pour embarquer PostgreSQL au sein d'un ensemble de logiciels ;
  - inconvénients : nécessite un environnement de compilation, ainsi que de configurer utilisateurs et script de démarrage.

Nous allons maintenant détailler chaque façon d'installer PostgreSQL.

3.2 PRÉ-REQUIS MINIMAUX POUR UNE INSTANCE POSTGRESQL

3.2.1 Pré-requis minimaux

- À peu près n'importe quel OS actuel
  - Linux (conseillé)
  - Unix propriétaires (dont macOS), FreeBSD
  - Windows
- N'importe quelle machine
  - …selon les besoins
  - 64 bits conseillé
- Stockage fiable
- Pas d'antivirus !

Il n'existe pas de configuration minimale pour une installation de PostgreSQL. La configuration de base est très conservatrice (128 Mo de cache). PostgreSQL peut fonctionner sur n'importe quelle machine actuelle, sur x86, 32 ou 64 bits. La configuration dépend plutôt des performances et volumétries attendues.

Les plate-formes officiellement supportées[1] incluent les principaux dérivés d'Unix, en premier lieu Linux, mais aussi les FreeBSD, OpenBSD, macOS, Solaris, ou illumos ; ainsi que Windows.

Linux 64 bits est de loin la plate-forme privilégiée par les développeurs, celle disposant du plus d'outils annexes, et est donc recommandée pour faire tourner PostgreSQL.

Les versions 32 bits fonctionnent mais sont de plus en plus ignorées par les développeurs d'extensions ou les mainteneurs[2], et ne se voient plus vraiment en production.

Debian et Red Hat et leurs dérivés sont les principales distributions vues en production (à côté d'Alpine pour les images docker).

[1] https://www.postgresql.org/docs/current/supported-platforms.html
[2] https://www.postgresql.org/message-id/Zr3_tXWr0VQol-VH%40msg.df7cb.de

Si vous devez absolument installer un antivirus ou un outil anti-intrusion, il faut impérativement exclure de son analyse tous les répertoires, fichiers et processus de PostgreSQL. L'interaction avec des antivirus a régulièrement mené à des problèmes de performance, voire de corruption.

3.3 INSTALLATION À PARTIR DES SOURCES

3.3.1 Étapes d'une installation à partir des sources

- Téléchargement
- Vérification des prérequis
- Compilation
- Installation

Les utilisateurs compilent rarement PostgreSQL, et nous recommandons d'utiliser les paquets précompilés du PGDG. Certaines utilisations le nécessitent toutefois. C'est aussi l'occasion de voir quelques concepts techniques importants.

Nous allons aborder ici les différentes étapes à réaliser pour installer PostgreSQL à partir des sources :

- trouver les fichiers sources ;
- préparer le serveur pour accueillir PostgreSQL ;
- compiler le serveur ;
- vérifier le résultat de la compilation ;
- installer les fichiers compilés.

3.3.2 Téléchargement

- Disponible depuis https://www.postgresql.org/download/
  - `postgresql-<version>.tar.bz2`
- Git : https://git.postgresql.org/git/postgresql.git

Les fichiers sources et les instructions de compilation sont disponibles sur le site officiel du projet[3] (ou plus directement https://www.postgresql.org/ftp/source/ ou https://ftp.postgresql.org/pub/source). Le nom du fichier à télécharger se présente toujours sous la forme `postgresql-<version>.tar.bz2` où `<version>` représente la version de PostgreSQL (par exemple : https://ftp.postgresql.org/pub/source/v17.0/postgresql-17.0.tar.bz2)

[3] https://www.postgresql.org/download/

Lorsque la future version de PostgreSQL paraît en phase de test (versions bêta), souvent à partir de mai, les sources sont accessibles à l'adresse suivante : https://www.postgresql.org/developer/beta.

Il existe bien sûr un dépôt git[4] officiel, avec un miroir sur Github[5].

3.3.3 Phases de compilation/installation

```
# au choix
tar xvfj postgresql-17.0.tar.bz2
git clone --branch REL_17_0 https://git.postgresql.org/git/postgresql.git

./configure --prefix=/usr/local/pgsql   # beaucoup d'options !
make -j8
cd contrib/ && make -j8        # ne pas oublier les contrib
make check

sudo make install && cd contrib/ && sudo make install

pg_config --configure
```

La compilation de PostgreSQL sous Linux/Unix suit un processus classique.

Une fois l'archive ou le dépôt Git récupéré, si les librairies habituelles de développement sont présentes, la compilation est extrêmement classique. Le répertoire `contrib` contient des modules et extensions gérés par le projet, dont l'installation est chaudement conseillée. La compilation s'effectue de la même manière.

```
./configure
```

Des options courantes pour une machine de production sont :

- `--prefix=` **répertoire** : répertoire d'installation personnalisé (par défaut, il s'agit de `/usr/local/pgsql`) ;
- `--with-pgport` =**port** : port par défaut (si différent de 5432) ;
- `--with-openssl` : support d'OpenSSL pour bénéficier de connexions chiffrées ;
- `--enable-nls` : le support de la langue utilisateur pour les messages provenant du serveur et des applications ;
- `--with-perl`, `--with-python` : installe les langages PL correspondants.

On voudra généralement activer ces trois dernières options… et beaucoup d'autres.

[4] https://git.postgresql.org/gitweb/?p=postgresql.git;a=summary
[5] https://github.com/postgres/postgres

```
make --jobs=8 all
(cd contrib/ && make --jobs=8 all)
make --jobs=8 check
sudo make install && cd contrib/ && sudo make install
```

(Le `make` de GNU se nomme `gmake` sur certaines systèmes différents de Linux). Cette phase est la plus longue, mais ne dure que quelques minutes sur du matériel récent en parallélisant.

`make check` lance des tests de non régression pour vérifier que PostgreSQL fonctionne correctement sur la machine cible.

Pour les mises à jour, il est important de connaître les options utilisées lors de la précédente compilation. L'outil `pg_config` le permet. Un PostgreSQL 17.0 sur Debian 12 affichera ce qui suit (les distributions ont tendance à intégrer toutes les options) :

```
$ /usr/lib/postgresql/17/bin/pg_config --configure
 '--build=x86_64-linux-gnu' '--prefix=/usr' '--includedir=${prefix}/include'
 '--mandir=${prefix}/share/man' '--infodir=${prefix}/share/info'
 '--sysconfdir=/etc' '--localstatedir=/var' '--disable-option-checking'
 '--disable-silent-rules' '--libdir=${prefix}/lib/x86_64-linux-gnu'
 '--runstatedir=/run'
 '--disable-maintainer-mode' '--disable-dependency-tracking'
 '--with-tcl' '--with-perl' '--with-python' '--with-pam' '--with-openssl'
 '--with-libxml' '--with-libxslt' '--mandir=/usr/share/postgresql/17/man'
 '--docdir=/usr/share/doc/postgresql-doc-17'
 '--sysconfdir=/etc/postgresql-common' '--datarootdir=/usr/share/'
 '--datadir=/usr/share/postgresql/17'
 '--bindir=/usr/lib/postgresql/17/bin'
 '--libdir=/usr/lib/x86_64-linux-gnu/'
 '--libexecdir=/usr/lib/postgresql/'
 '--includedir=/usr/include/postgresql/'
 '--with-extra-version= (Debian 17.0-1.pgdg120+1)'
 '--enable-nls' '--enable-thread-safety' '--enable-debug' '--disable-rpath'
 '--with-uuid=e2fs' '--with-gnu-ld' '--with-gssapi' '--with-ldap'
 '--with-pgport=5432' '--with-system-tzdata=/usr/share/zoneinfo'
 'AWK=mawk' 'MKDIR_P=/bin/mkdir -p' 'PROVE=/usr/bin/prove'
 'PYTHON=/usr/bin/python3' 'TAR=/bin/tar' 'XSLTPROC=xsltproc --nonet'
 'CFLAGS=-g -O2 -fstack-protector-strong -Wformat -Werror=format-security
  ↪  -fno-omit-frame-pointer' 'LDFLAGS=-Wl,-z,relro -Wl,-z,now'
 '--enable-tap-tests' '--with-icu'
 '--with-llvm' 'LLVM_CONFIG=/usr/bin/llvm-config-16' 'CLANG=/usr/bin/clang-16'
 '--with-lz4' '--with-zstd'
 '--with-systemd' '--with-selinux' '--enable-dtrace'
 'build_alias=x86_64-linux-gnu'
 'CPPFLAGS=-Wdate-time -D_FORTIFY_SOURCE=2'
 'CXXFLAGS=-g -O2 -fstack-protector-strong -Wformat -Werror=format-security'
```

Il existe aussi une vue `pg_config` accessible sur un serveur démarré :

```
 TABLE pg_config ;
       name         |  setting
-------------------+----------------------------------------------------------
 BINDIR             | /usr/lib/postgresql/17/bin
 DOCDIR             | /usr/share/doc/postgresql-doc-17
 HTMLDIR            | /usr/share/doc/postgresql-doc-17
```

```
 INCLUDEDIR        | /usr/include/postgresql
 PKGINCLUDEDIR     | /usr/include/postgresql
 INCLUDEDIR-SERVER | /usr/include/postgresql/17/server
 LIBDIR            | /usr/lib/x86_64-linux-gnu
 PKGLIBDIR         | /usr/lib/postgresql/17/lib
 LOCALEDIR         | /usr/share/locale
 MANDIR            | /usr/share/postgresql/17/man
 SHAREDIR          | /usr/share/postgresql/17
 SYSCONFDIR        | /etc/postgresql-common
 PGXS              | /usr/lib/postgresql/17/lib/pgxs/src/makefiles/pgxs.mk
 CONFIGURE         |  '--build=x86_64-linux-gnu' '--prefix=/usr'
↪  '--includedir=${prefix}/include' '--mandir=${prefix}/share/man'
↪  '--infodir=${prefix}/share/info' '--sysconfdir=/etc' '--localstatedir=/var'
↪  '--disable-option-checking' '--disable-silent-rules'
↪  '--libdir=${prefix}/lib/x86_64-linux-gnu' '--runstatedir=/run'
↪  '--disable-maintainer-mode' '--disable-dependency-tracking' '--with-tcl'
↪  '--with-perl' '--with-python' '--with-pam' '--with-openssl' '--with-libxml'
↪  '--with-libxslt' '--mandir=/usr/share/postgresql/17/man'
↪  '--docdir=/usr/share/doc/postgresql-doc-17'
↪  '--sysconfdir=/etc/postgresql-common' '--datarootdir=/usr/share/'
↪  '--datadir=/usr/share/postgresql/17' '--bindir=/usr/lib/postgresql/17/bin'
↪  '--libdir=/usr/lib/x86_64-linux-gnu/' '--libexecdir=/usr/lib/postgresql/'
↪  '--includedir=/usr/include/postgresql/' '--with-extra-version= (Debian
↪  17.0-1.pgdg120+1)' '--enable-nls' '--enable-thread-safety' '--enable-debug'
↪  '--disable-rpath' '--with-uuid=e2fs' '--with-gnu-ld' '--with-gssapi'
↪  '--with-ldap' '--with-pgport=5432' '--with-system-tzdata=/usr/share/zoneinfo'
↪  'AWK=mawk' 'MKDIR_P=/bin/mkdir -p' 'PROVE=/usr/bin/prove'
↪  'PYTHON=/usr/bin/python3' 'TAR=/bin/tar' 'XSLTPROC=xsltproc --nonet' 'CFLAGS=-g
↪  -O2 -fstack-protector-strong -Wformat -Werror=format-security
↪  -fno-omit-frame-pointer' 'LDFLAGS=-Wl,-z,relro -Wl,-z,now' '--enable-tap-tests'
↪  '--with-icu' '--with-llvm' 'LLVM_CONFIG=/usr/bin/llvm-config-16'
↪  'CLANG=/usr/bin/clang-16' '--with-lz4' '--with-zstd' '--with-systemd'
↪  '--with-selinux' '--enable-dtrace' 'build_alias=x86_64-linux-gnu'
↪  'CPPFLAGS=-Wdate-time -D_FORTIFY_SOURCE=2' 'CXXFLAGS=-g -O2
↪  -fstack-protector-strong -Wformat -Werror=format-security'
 CC                | gcc
 CPPFLAGS          | -Wdate-time -D_FORTIFY_SOURCE=2 -D_GNU_SOURCE
↪  -I/usr/include/libxml2
 CFLAGS            | -Wall -Wmissing-prototypes -Wpointer-arith
↪  -Wdeclaration-after-statement -Werror=vla -Wendif-labels
↪  -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-function-type
↪  -Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv
↪  -fexcess-precision=standard -Wno-format-truncation -Wno-stringop-truncation -g
↪  -g -O2 -fstack-protector-strong -Wformat -Werror=format-security
↪  -fno-omit-frame-pointer
 CFLAGS_SL         | -fPIC
 LDFLAGS           | -Wl,-z,relro -Wl,-z,now -L/usr/lib/llvm-16/lib -Wl,--as-needed
 LDFLAGS_EX        |
 LDFLAGS_SL        |
 LIBS              | -lpgcommon -lpgport -lselinux -lzstd -llz4 -lxslt -lxml2 -lpam
↪  -lssl -lcrypto -lgssapi_krb5 -lz -lreadline -lm
 VERSION           | PostgreSQL 17.0 (Debian 17.0-1.pgdg120+1)
(23 lignes)
```

3.3.4 Utilisateur dédié de PostgreSQL

- Jamais **root**
- Utilisateur dédié
  - propriétaire des répertoires et fichiers
  - peut lancer PostgreSQL
  - traditionnellement : **postgres**
- Variables d'environnement (défaut) :

```
export PATH=/usr/local/pgsql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
export MANPATH=$MANPATH:/usr/local/pgsql/share/man
export PGDATA=/usr/local/pgsql/data     # Données
```

Le serveur PostgreSQL ne peut pas être exécuté par l'utilisateur **root**, pour des raisons de sécurité. Un utilisateur système sans droits particuliers suffit. Il sera le **seul** propriétaire des répertoires et fichiers gérés par le serveur PostgreSQL. Il sera aussi le compte qui permettra de lancer PostgreSQL (directement ou indirectement via les outils système comme systemd).

Cet utilisateur système est habituellement appelé **postgres** (et les paquets d'installation utilisent ce nom), mais ce n'est absolument pas une obligation.

Il est nécessaire de positionner un certain nombre de variables d'environnement dans `${HOME}/.profile` ou dans `/etc/profile`. Avec le chemin par défaut de la compilation, elles valent :

```
export PATH=/usr/local/pgsql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
export MANPATH=$MANPATH:/usr/local/pgsql/share/man
export PGDATA=/usr/local/pgsql/data
```

Chemins des binaires :

- `/usr/local/pgsql/bin` est le chemin par défaut (ou le chemin indiqué à l'option `--prefix` lors de l'étape `configure`). L'ajouter à `PATH` permet de rendre l'exécution de PostgreSQL possible depuis n'importe quel répertoire ;
- `LD_LIBRARY_PATH` indique au système où trouver les différentes bibliothèques nécessaires à l'exécution des programmes ;
- `MANPATH` indique le chemin de recherche des pages de manuel.

Emplacement des données :

`PDGATA` est une spécificité de PostgreSQL : cette variable indique le répertoire des fichiers de données de PostgreSQL, c'est-à-dire les données de l'utilisateur. (Plus rigoureusement : elle pointe l'emplacement du fichier `postgresql.conf`, généralement dans ce répertoire de données. Les paquets pour Debian/Ubuntu représentent une grosse exception, avec un `postgresql.conf` dans `/etc`, qui contient le paramètre `data_directory`, lequel pointe vers le vrai répertoire de données.) PostgreSQL n'impose de lui-même aucune contrainte sur le chemin.

3.3.5 Création du répertoire de données de l'instance

```
$ initdb -D /usr/local/pgsql/data
```

- Une seule instance sur le répertoire !
- Pas à la racine d'un point de montage
- Options d'emplacement :
  - `--data` pour les fichiers de données
  - `--waldir` pour les journaux de transactions
- Autres options :
  - `--data-checksums` : sommes de contrôle (conseillé !)
  - et : chemin des journaux, mot de passe, encodage…

La commande `initdb` doit être exécutée sous le compte de l'utilisateur système PostgreSQL décrit dans la section précédente (généralement **postgres**). (Les paquets d'installation fournissent d'autres outils, mais tous utilisent `initdb`.)

Répertoire :

`initdb` crée les fichiers d'une nouvelle instance avec une première base de données dans le répertoire indiqué. Si le répertoire n'existe pas, `initdb` tentera de le créer, s'il a les droits pour le faire. S'il existe, il doit être vide.

Typiquement, les chemins par défaut sont ceux-ci :

```
# chemin par défaut d'une version compilée sans --prefix
/usr/local/pgsql/data
# chemin par défaut des paquets Debian/Ubuntu (première instance v17)
/var/lib/postgresql/17/main
# chemin par défaut des paquets RPM du PGDG (première instance v17)
/var/lib/pgsql/17/data
```

Vous pouvez décider de placer le `PGDATA` n'importe où pourvu que l'utilisateur sous lequel tourne PostgreSQL puisse y accéder ; il y a cependant quelques conseils à connaître.

Attention : pour des raisons de sécurité et de fiabilité, les répertoires choisis pour les données de votre instance **ne doivent pas** être à la racine d'un point de montage. Que ce soit le répertoire PGDATA, le répertoire `pg_wal` ou les éventuels *tablespaces*. Si un ou plusieurs points de montage sont dédiés à l'utilisation de PostgreSQL, positionnez toujours les données dans un sous-répertoire, voire deux niveaux en-dessous du point de montage, couramment : point de montage/version majeure/nom instance. Exemples :

```
# Si /mnt/donnees est le point de montage
/mnt/donnees/17/dbprod
# Si /var/lib/postgresql est une partition
/var/lib/postgresql/17/main
# Si /var/lib/pgsql est une partition
/var/lib/pgsql/17/data
```

À ce propos, voir :

- chapitre *Use of Secondary File Systems*[6] ;
- le détail des raisons techniques[7].

Enfin :

Un répertoire de données ne doit être utilisé que par une seule instance (processus) à la fois !
PostgreSQL vérifie au démarrage qu'aucune autre instance du même serveur n'utilise les fichiers indiqués, mais cette protection n'est pas absolue, notamment avec des accès depuis des systèmes différents. Faites donc bien attention à ne lancer PostgreSQL qu'une seule fois sur un répertoire de données.

Si plusieurs instances cohabitent sur le serveur, elles devront avoir chacune leur répertoire.

Lancement de initdb :

Voici ce qu'affiche cette commande :

```
$ initdb --data /usr/local/pgsql/data --data-checksums

The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "fr_FR.UTF-8".
```

[6] https://www.postgresql.org/docs/current/creating-cluster.html
[7] https://bugzilla.redhat.com/show_bug.cgi?id=1247477#c1

```
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "french".

Data page checksums are enabled.

fixing permissions on existing directory /usr/local/pgsql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default "max_connections" ... 100
selecting default "shared_buffers" ... 128MB
selecting default time zone ... Europe/Paris
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or
 ↪  --auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /usr/local/pgsql/data -l logfile start
```

Nous pouvons ainsi suivre les actions de `initdb` :

- détection de l'utilisateur, de l'encodage et de la locale ;
- prise en compte des sommes de contrôle ;
- création ou vérification des droits du répertoire PGDATA (`/usr/local/pgsql/data` ici) ;
- création des sous-répertoires ;
- création et modification des fichiers de configuration ;
- exécution du script bootstrap ;
- synchronisation sur disque ;
- affichage de quelques informations supplémentaires (noter les droits par défaut très laxistes d'une version compilée).

Authentification par défaut :

Il est possible de changer la méthode d'authentification par défaut avec les paramètres en ligne de commande `--auth`, `--auth-host` et `--auth-local`. Les options `--pwprompt` ou `--pwfile` permettent d'assigner un mot de passe à l'utilisateur **postgres**.

Sommes de contrôle :

Il est conseillé d'activer les sommes de contrôle des fichiers de données avec l'option `--data-checksums`. Ces sommes de contrôle sont vérifiées à chaque lecture d'un bloc. Leur activation est chaudement recommandée pour détecter une corruption physique le plus tôt possible. Tous les processeurs Intel/AMD pas trop anciens supportent le jeu d'instruction SSE 4.2 (voir dans `/proc/cpuinfo`), il ne devrait pas y avoir d'impact notable sur les performances. Les journaux générés seront cependant plus nombreux. Il est possible de vérifier, activer ou désactiver toutes les sommes de contrôle grâce à l'outil `pg_checksums`. Cependant, l'opération d'activation sur une instance existante impose un arrêt

et une réécriture complète des fichiers. (L'opération était même impossible avant PostgreSQL 12.) Pensez-y donc dès l'installation, y compris si vous installez par les paquets.

Emplacement des journaux :

L'option `--waldir` indique l'emplacement des journaux de transaction. Par défaut, ils sont dans le sous-répertoire `pg_wal/` dans le PGDATA. Si vous désignez un autre répertoire, `pg_wal` sera un lien symbolique vers le répertoire désigné.

Taille des segments :

Les fichiers des journaux de transaction ont une taille par défaut de 16 Mo. Augmenter leur taille avec `--wal-segsize` n'a d'intérêt que pour les très grosses installations générant énormément de journaux pour optimiser leur archivage.

3.3.6 Lancement et arrêt

- Avec le script de l'OS (recommandé) ou `pg_ctl` :

```
systemctl [action] postgresql      # systemd
/etc/init.d/postgresql [action]    # SysV Init
service postgresql [action]        # idem
…

$ pg_ctl --pgdata /usr/local/pgsql/data --log logfile [action]
         --mode [smart|fast|immediate]
```

- `[action]` dépend du besoin :
  - `start` / `stop` / `restart`
  - `reload` pour recharger la configuration
  - `status`
  - `promote`, `logrotate`, `kill`…

(Re)démarrage et arrêt :

La méthode recommandée est d'utiliser un script de démarrage adapté à l'OS, (voir plus bas les outils les commandes `systemd` ou celles propres à Debian) , surtout si l'on a installé PostgreSQL par les paquets. Au besoin, des scripts d'exemple existent dans le répertoire des sources (`contrib/start-scripts/`[8]) pour Linux, FreeBSD et macOS. Ces scripts sont à exécuter en tant qu'utilisateur **root**. Sinon, il est possible d'exécuter `pg_ctl` avec l'utilisateur créé précédemment.

[8] https://github.com/postgres/postgres/tree/master/contrib/start-scripts

Les deux méthodes partagent certaines des actions présentées ci-dessus : `start`, `stop`, `restart` (aux sens évidents), ou `reload` (pour recharger la configuration sans redémarrer PostgreSQL ni couper les connexions).

L'option `--mode` permet de préciser le mode d'arrêt parmi les trois disponibles :

- `smart` : pour vider le cache de PostgreSQL sur disque, interdire de nouvelles connexions et attendre la déconnexion des clients et d'éventuelles sauvegardes ;
- `fast` (par défaut) : pour vider le cache sur disque et déconnecter les clients sans attendre (les transactions en cours sont annulées) ;
- `immediate` : équivalent à un arrêt brutal : tous les processus serveur sont tués et donc, au redémarrage, le serveur devra rejouer ses journaux de transactions.

Rechargement de la configuration :

Pour recharger la configuration après changement du paramétrage, la commande :

```
pg_ctl reload -D /repertoire_pgdata
```

est équivalente à cet ordre SQL :

```
SELECT pg_reload_conf() ;
```

Il faut aussi savoir que quelques paramètres nécessitent un redémarrage de PostgreSQL et non un simple rechargement, ils sont indiqués dans les commentaires de `postgresql.conf`.

3.4 INSTALLATION À PARTIR DES PAQUETS LINUX

3.4.1 Paquets

- Packages Debian
- Packages RPM

Pour une utilisation en environnement de production, il est généralement préférable d'installer les paquets binaires préparés spécialement pour la distribution utilisée. Les paquets sont préparés par des personnes différentes, suivant les recommendations officielles de la distribution. Il y a donc des différences, parfois importantes, entre les paquets.

3.4.2 Paquets Debian officiels

- Nombreux paquets disponibles :
  - serveur, client, contrib, docs
  - extensions, outils
- `apt install postgresql-<version majeure>`
  - installe les binaires
  - crée l'utilisateur **postgres**
  - exécute `initdb`
  - démarre le serveur

Sur Debian et les versions dérivées (Ubuntu notamment), l'installation de PostgreSQL a été découpée en plusieurs paquets (ici pour la version majeure 17) :

- le serveur : `postgresql-17` ;
- les clients : `postgresql-client-17` ;
- la documentation : `postgresql-doc-17`.

Ce paquet correspond toujours à la dernière version mineure trimestrielle (par exemple 17.1).

La version majeure dans le nom des paquets (`9.6`, `10`, `13`, `17`…) permet d'installer plusieurs versions majeures sur le même serveur physique ou virtuel.

> Par défaut, sans autre dépôt, une seule version majeure sera disponible dans une version de distribution. Par exemple, `apt install postgresql` sur Debian 12 installera en fait `postgresql-15` (il est en dépendance).

Il existe aussi des paquets pour les outils, les extensions, etc. Certains langages de procédures stockées sont disponibles dans des paquets séparés :

- PL/python dans `postgresql-plpython3-17`
- PL/perl dans `postgresql-plperl-17`
- PL/Tcl dans `postgresql-pltcl-17`
- etc.

Pour compiler des outils liés à PostgreSQL, il est recommandé d'installer également les bibliothèques de développement qui font partie du paquet `postgresql-server-dev-17`.

Quand le paquet `postgresql-17` est installé, plusieurs opérations sont immédiatement exécutées :

- téléchargement du paquet (dans la dernière version mineure) ;
- installation des binaires contenus dans le paquet ;
- création de l'utilisateur **postgres** (s'il n'existe pas déjà) ;
- paramétrage d'une première instance nommée `main` ;
- création du répertoire des données, lancement de l'instance.

Les exécutables sont installés dans :

```
/usr/lib/postgresql/17/bin
```

Chaque instance porte un nom, qui se retrouve dans le paramètre `cluster_name`, et permet d'identifier les processus dans un `ps` ou un `top`. Le nom de la première instance de chaque version majeure est par défaut `main`. Pour cette instance :

- les données sont dans :

```
/var/lib/postgresql/17/main
```

- les fichiers de configuration (pas tous ! certains restent dans le répertoire des données) sont dans :

```
/etc/postgresql/17/main
```

- les traces sont gérées par l'OS sous ce nom :

```
/var/log/postgresql/postgresql-17-main.log
```

- un fichier PID, la socket d'accès local, et l'espace de travail temporaire des statistiques d'activité figurent dans `/var/run/postgresql`.

Tout ceci vise à respecter le plus possible la norme FHS[9] (*Filesystem Hierarchy Standard*).

En cas de mise à jour d'un paquet, le serveur PostgreSQL est redémarré après mise à jour des binaires.

3.4.3 Paquets Debian : spécificités

- Plusieurs versions majeures installables
- Wrappers/scripts pour la gestion des différentes instances :
  - `pg_lsclusters`
  - `pg_ctlcluster`
    - ou : `systemctl stop|start postgresql-15@main`
  - `pg_createcluster`
  - etc.
- Respect de la FHS
- Configuration dans `/etc/postgresql/`

Numéroter les paquets permet d'installer plusieurs versions majeures de PostgreSQL (mais chacune seulement dans sa dernière version mineure) au besoin sur le même système, si les dépôts les contiennent.

Les mainteneurs des paquets Debian ont écrit des scripts pour faciliter la création, la suppression et la gestion de différentes instances sur le même serveur. Les principaux sont :

- `pg_lsclusters` liste les instances ;
- `pg_createcluster <version majeure> <nom instance>` crée une instance de la version majeure et du nom voulu ;
- `pg_dropcluster <version majeure> <nom instance>` détruit l'instance désignée ;
- `/etc/postgresql-common/createcluster.conf` permet de centraliser les paramètres par défaut des instances ;
- la gestion d'une instance est réalisée avec la commande `pg_ctlcluster` :

```
pg_ctlcluster <version majeure> <nom instance> start|stop|reload|status|promote
```

Ce dernier script interagit avec systemd, qui peut être utilisé pour arrêter ou démarrer séparément chaque instance. Ces deux commandes sont équivalentes :

```
sudo pg_ctlcluster 17 main start
sudo systemctl start postgresql@17-main
```

[9] https://fr.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

3.4.4 Paquets Debian communautaires

- La communauté met des paquets Debian à disposition :
  - https://apt.postgresql.org
- Synchrone avec le projet PostgreSQL
- Ajout du dépôt dans `/etc/apt/sources.list.d/pgdg.list`
- Utilisation chaudement conseillée

La distribution Debian préfère des paquets testés et validés, y compris sur des versions assez anciennes, que d'adopter la dernière version dès qu'elle est disponible. Par exemple, Debian 11 ne contiendra jamais que PostgreSQL 13 et ses versions mineures, et Debian 12 ne contiendra que PostgreSQL 15.

Pour faciliter les mises à jour, la communauté PostgreSQL met à disposition son propre dépôt de paquets Debian. Elle en assure le maintien et le support. Les paquets de la communauté ont la même provenance et le même contenu que les paquets officiels Debian, avec d'ailleurs les mêmes mainteneurs. La seule différence est que `apt.postgresql.org` est mis à jour plus fréquemment, en liaison directe avec la communauté PostgreSQL, et contient beaucoup plus d'outils et extensions.

Le wiki[10] indique quelle est la procédure, qui peut se résumer à :

```
sudo apt install -y postgresql-common
sudo /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

```
…
Setting up postgresql-common (248) ...
Creating config file /etc/postgresql-common/createcluster.conf with new version
Building PostgreSQL dictionaries from installed myspell/hunspell packages...
Removing obsolete dictionary files:
Created symlink /etc/systemd/system/multi-user.target.wants/postgresql.service →
 ↪  /lib/systemd/system/postgresql.service.
Processing triggers for man-db (2.11.2-2) ...
This script will enable the PostgreSQL APT repository on apt.postgresql.org on
your system. The distribution codename used will be bookworm-pgdg.
Press Enter to continue, or Ctrl-C to abort.
```

Si l'on continue, ce dépôt sera ajouté au système (ici sous Debian 12) :

```
$ cat /etc/apt/sources.list.d/pgdg.sources
Types: deb
Architectures: amd64
URIs: https://apt.postgresql.org/pub/repos/apt
Suites: bookworm-pgdg
Components: main
Signed-By: /usr/share/postgresql-common/pgdg/apt.postgresql.org.gpg
```

[10] https://wiki.postgresql.org/wiki/Apt

et la version choisie de PostgreSQL sera immédiatement installable :

```
sudo apt install postgresql-17
```

3.4.5 Paquets Red Hat communautaires : yum.postgresql.org

- Préférer les paquets distribués par la communauté :
  - https://yum.postgresql.org/
  - https://yum.postgresql.org/howto/
  - plus complets que les Appstream
- Ajout du dépôt comme paquet RPM

Les versions majeures de Red Hat et de ses dérivés (Rocky Linux, AlmaLinux, Fedora, CentOS, Scientific Linux…) sont très espacées, et la version par défaut de PostgreSQL n'est parfois plus supportée. Même les versions disponibles en AppStream (avec `dnf module`) sont parfois en retard ou ne contiennent que certaines versions majeures. Les dépôts de la communauté sont donc fortement conseillés. Ils contiennent aussi beaucoup plus d'utilitaires, toutes les versions majeures supportées de PostgreSQL simultanément, et collent au plus près des versions publiées par la communauté.

Ce dépôt convient pour les dérivés de Red Hat comme Fedora, CentOS, Rocky Linux…

3.4.6 Paquets Red Hat communautaires : installation

```
sudo dnf install -y \
     https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-
 ↪  x86_64/pgdg-redhat-repo-latest.noarch.rpm
sudo dnf -qy module disable postgresql

sudo dnf install -y postgresql17-server
# dont : utilisateur postgres

sudo /usr/pgsql-17/bin/postgresql-17-setup initdb
sudo systemctl enable postgresql-17
sudo systemctl start postgresql-17
```

L'installation de la configuration du dépôt de la communauté est très simple. Les commandes peuvent même être générées en fonction des versions sur https://www.postgresql.org/download/linux/redhat/. L'exemple ci-dessus installe PostgreSQL 17 sur Rocky 9.

3.4.7 Paquets Red Hat communautaires : spécificités

- Paquets séparés serveur, client, contrib
- `/usr/pgsql-XX/bin` : binaires
- Initialisation manuelle (`postgresql-17-setup initdb`)
  - vers : `/var/lib/pgsql/XX/data`
- Gestion par systemd
- Particularités :
  - plusieurs versions majeures installables
  - configuration : dans le répertoire de données

Sous Red Hat et les versions dérivées, les dépôts communautaires ont été découpés en plusieurs paquets, disponibles pour chacune des versions majeures supportées :

- le serveur : `postgresqlXX-server` ;
- les clients : `postgresqlXX` ;
- les modules contrib : `postgresqlXX-contrib` ;
- la documentation : `postgresqlXX-docs`.

où XX est la version majeure (par exemple `13` ou `17`).

Il existe aussi des paquets pour les outils, les extensions, etc. Certains langages de procédures stockées sont disponibles dans des paquets séparés :

- PL/python dans `postgresqlXX-plpython3` ;
- PL/perl dans `postgresqlXX-plperl` ;
- PL/Tcl dans `postgresqlXX-pltcl` ;
- etc.

Pour compiler des outils liés à PostgreSQL, il est recommandé d'installer également les bibliothèques de développement qui font partie du paquet `postgresqlXX-devel`.

Ce nommage sous-entend qu'il est possible d'installer plusieurs versions majeures sur le même serveur physique ou virtuel. Les exécutables sont installés dans le répertoire `/usr/pgsql-XX/bin`, les traces dans `/var/lib/pgsql/XX/data/log` (utilisation du `logger process` de PostgreSQL), les données dans `/var/lib/pgsql/XX/data`. Ce dernier est le répertoire par défaut des données, mais il est possible de le surcharger.

Sur un système type Red Hat sans dépôt communautaire, les noms des paquets ne comportent pas le numéro de version et installent tous les binaires cités ici dans `/usr/bin`.

Quand le paquet serveur est installé, plusieurs opérations sont exécutées : téléchargement du paquet, installation des binaires contenus dans le paquet, et création de l'utilisateur **postgres** (s'il n'existe pas déjà).

Le répertoire des données n'est pas créé. Cela reste une opération à réaliser par la personne qui a installé PostgreSQL sur le serveur. Lancer le script `/usr/psql-XX/bin/postgresqlXX-setup` en tant que **root** :

```
PGSETUP_INITDB_OPTIONS="--data-checksums" \
 /usr/pgsql-17/bin/postgresql-17-setup initdb
```

Plutôt que de respecter la norme FHS (*Filesystem Hierarchy Standard*), les mainteneurs ont fait le choix de respecter l'emplacement des fichiers utilisé par défaut par les développeurs PostgreSQL. La configuration de l'instance (`postgresql.conf` entre autres) est donc directement dans le PGDATA.

Pour installer plusieurs instances, il faudra créer manuellement des services systemd différents.

En cas de mise à jour d'un paquet, le serveur PostgreSQL n'est pas redémarré après mise à jour des binaires.

3.5 INDUSTRIALISATION AVEC PGLIFT

3.5.1 pglift : présentation rapide

- Déploiement et *infrastructure as code* :
  - ligne de commande
  - collections Ansible
- Couverture fonctionnelle :
  - Sauvegardes avec pgBackRest
  - Supervision avec Prometheus
  - Administration avec temBoard
  - Analyse avec PoWA
  - Haute disponibilité avec Patroni
  - Intégration système avec `systemd` ou `rsyslog`

pglift[11] est un outil permettant de déployer et d'exploiter PostgreSQL à grande échelle. Le projet fournit à la fois une interface en ligne de commande pour gérer le cycle de vie des instances et une collection de modules Ansible pour piloter une *infrastructure-as-code* dans un contexte de production.

L'élément fondamental de pglift est l'instance. Celle-ci est constituée d'une instance PostgreSQL et inclut des composants satellites, facultatifs, permettant d'exploiter PostgreSQL à grande échelle. Dans sa version 1.0, pglift supporte les composants suivants :

pgBackRest[12] permet de prendre en charge les sauvegardes physiques PITR (*Point In Time Recovery*) de PostgreSQL.

postgres_exporter[13] est un service de supervision permettant de remonter des informations à l'outil de surveillance Prometheus[14].

temBoard[15] est une console web de supervision et d'administration dédiée aux instances PostgreSQL.

PoWA[16] est une console web permettant d'analyser l'activité des instances PostgreSQL en direct.

Patroni[17] est un outil permettant de construire un agrégat d'instances PostgreSQL résilient offrant un service de haute disponibilité.

[11] https://pglift.readthedocs.io/
[12] https://pgbackrest.org/
[13] https://github.com/prometheus-community/postgres_exporter
[14] https://prometheus.io/
[15] https://temboard.readthedocs.io/
[16] https://powa.readthedocs.io/en/latest/
[17] https://patroni.readthedocs.io/

Tous les composants satellites supportés par pglift sont des logiciels libres. Le projet pglift est lui aussi nativement open source, sous licence GPLv3. Son développement se passe en public sur https://gitlab.com/dalibo/pglift/ pour l'API Python et l'interface en ligne de commande et sur https://gitlab.com/dalibo/pglift-ansible/ pour la collection Ansible `dalibo.pglift`.

Références :

- Présentation sur le blog Dalibo[18] (Denis Laxalde)
- Documentation[19]

3.5.2 pglift : fichier de configuration

```yaml
prefix: /srv # fichier /etc/pglift/settings.yaml
postgresql:
  auth:
    host: scram-sha-256
prometheus:
  execpath: /usr/bin/prometheus-postgres-exporter
pgbackrest:
  repository:
    mode: path
    path: /srv/pgsql-backups
powa: {}
systemd: {}
rsyslog: {}
```

À côté de PostgreSQL, l'instance inclut un ensemble d'outils nécessaires à son utilisation. L'intégration de ces outils satellites est configurée localement via un fichier YAML `settings.yaml`.

Ce fichier définit comment les différents composants de l'instance sont configurés, installés et exécutés. Il permet aussi de définir quels composants satellites facultatifs supportés par pgLift sont inclus dans l'instance. Si un élément est listé dans ce fichier sans paramètre associé, il sera exploité dans sa configuration par défaut.

Dans l'exemple ci-dessus, temBoard et Patroni ne sont pas installés, et PoWA est laissé à sa configuration par défaut.

[18] https://blog.dalibo.com/2023/10/17/pglift-intro.html
[19] https://pglift.readthedocs.io/

3.5.3 pglift : exemples de commandes

- Initialisation

```
pglift instance create main --pgbackrest-stanza=main
```

- Modification de configuration

```
pglift pgconf -i main set log_connections=on
```

- Sauvegarde physique

```
pglift instance backup main
```

- Utilisation des outils de l'instance

```
pglift instance exec main -- psql
pglift instance exec main -- pgbackrest info
```

Interface impérative en ligne de commande :

La commande suivante permet la création d'une instance pglift :

```
$ pglift instance create main --pgbackrest-stanza=main
INFO     initializing PostgreSQL
INFO     configuring PostgreSQL authentication
INFO     configuring PostgreSQL
INFO     starting PostgreSQL 16-main
INFO     creating role 'powa'
INFO     creating role 'prometheus'
INFO     creating role 'backup'
INFO     altering role 'backup'
INFO     creating 'powa' database in 16/main
INFO     creating extension 'btree_gist' in database powa
INFO     creating extension 'pg_qualstats' in database powa
INFO     creating extension 'pg_stat_statements' in database powa
INFO     creating extension 'pg_stat_kcache' in database powa
INFO     creating extension 'powa' in database powa
INFO     configuring Prometheus postgres_exporter 16-main
INFO     configuring pgBackRest stanza 'main' for
         pg1-path=/srv/pgsql/16/main/data
INFO     creating pgBackRest stanza main
INFO     starting Prometheus postgres_exporter 16-main
```

L'instance pglift inclut l'instance PostgreSQL ainsi que l'ensemble des modules définis dans le fichier de configuration `settings.yaml`. pglift gère aussi l'intégration au système avec `systemd` ou `rsyslog` comme dans notre exemple. Tout ceci fonctionne sans privilège **root** pour une meilleure séparation des responsabilités et une meilleure sécurité.

pglift permet de récupérer l'état d'une instance à un moment donné :

```
$ pglift instance get main -o json
{
  "name": "main",
  "version": "16",
  "port": 5432,
  "settings": {
    "unix_socket_directories": "/run/user/1000/pglift/postgresql",
    "shared_buffers": "1 GB",
    "wal_level": "replica",
    "archive_mode": true,
    "archive_command": "/usr/bin/pgbackrest --config-path=/etc/pgbackrest \
    --stanza=main --pg1-path=/srv/pgsql/16/main/data archive-push %p",
    "effective_cache_size": "4 GB",
    "log_destination": "syslog",
    "logging_collector": true,
    "log_directory": "/var/log/postgresql",
    "log_filename": "16-main-%Y-%m-%d_%H%M%S.log",
    "syslog_ident": "postgresql-16-main",
    "cluster_name": "main",
    "lc_messages": "C",
    "lc_monetary": "C",
    "lc_numeric": "C",
    "lc_time": "C",
    "shared_preload_libraries": "pg_qualstats, pg_stat_statements, pg_stat_kcache"
  },
  "data_checksums": false,
  "locale": "C",
  "encoding": "UTF8",
  "standby": null,
  "state": "started",
  "pending_restart": false,
  "wal_directory": "/srv/pgsql/16/main/wal",
  "prometheus": {
    "port": 9187
  },
  "data_directory": "/srv/pgsql/16/main/data",
  "powa": {},
  "pgbackrest": {
    "stanza": "main"
  }
}
```

ou de modifier l'instance :

```
# activation du paramètre log_connections
$ pglift pgconf -i main set log_connections=on
INFO     configuring PostgreSQL
INFO     instance 16/main needs reload due to parameter changes: log_connections
INFO     reloading PostgreSQL configuration for 16-main
log_connections: None -> True
# changement du port prometheus
$ pglift instance alter main --prometheus-port 8188
INFO     configuring PostgreSQL
INFO     reconfiguring Prometheus postgres_exporter 16-main
```

```
INFO     instance 16/main needs reload due to parameter changes: log_connections
INFO     reloading PostgreSQL configuration for 16-main
INFO     starting Prometheus postgres_exporter 16-main
$ pglift instance get main # vérification
 name  version  port  data_checksums  locale  encoding  pending_restart  prometheus
  ↪  pgbackrest
 main  16       5432  False           C       UTF8      False            port: 8188
  ↪  stanza: main
```

Les instances et objets PostgreSQL peuvent être manipulés à l'aide des outils *natifs* de PostgreSQL depuis la ligne de commande :

```
$ pglift instance exec main -- pgbench -i bench
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.18 s (drop tables 0.00 s, create tables 0.01 s, ... vacuum 0.04 s, primary
 ↪  keys 0.05 s).
$ pglift instance exec main -- pgbench bench
pgbench (16.0 (Debian 16.0-1.pgdg120+1))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
number of failed transactions: 0 (0.000%)
latency average = 1.669 ms
initial connection time = 4.544 ms
tps = 599.125277 (without initial connection time)
```

Ceci s'applique aussi à des outils tiers, par exemple avec pgBackRest :

```
$ pglift instance exec main -- pgbackrest info
stanza: main
    status: ok
    cipher: none

    db (current)
        wal archive min/max (16): 000000010000000000000001/000000010000000000000007

        full backup: 20231016-092726F
            timestamp start/stop: 2023-10-16 09:27:26+02 / 2023-10-16 09:27:31+02
            wal start/stop: 000000010000000000000004 / 000000010000000000000004
            database size: 32.0MB, database backup size: 32.0MB
            repo1: backup set size: 4.2MB, backup size: 4.2MB

        diff backup: 20231016-092726F_20231016-092821D
            timestamp start/stop: 2023-10-16 09:28:21+02 / 2023-10-16 09:28:24+02
            wal start/stop: 000000010000000000000007 / 000000010000000000000007
            database size: 54.5MB, database backup size: 22.6MB
```

```
            repo1: backup set size: 6MB, backup size: 1.8MB
            backup reference list: 20231016-092726F
```

Le tutoriel de la ligne de commande[20] de la documentation recense tous les exemples d'utilisation des commandes usuelles de *pglift*.

Interface déclarative avec Ansible :

pglift comporte une collection de modules Ansible, sous l'espace de noms `dalibo.pglift`.Voici un exemple de playbook illustrant ses capacités :

```
- name: Set up database instances
hosts: dbserver
tasks:
  - name: main instance
    dalibo.pglift.instance:
      name: main
      state: started
      port: 5444
      settings:
        max_connections: 100
        shared_buffers: 1GB
        shared_preload_libraries: 'pg_stat_statements, passwordcheck'
      surole_password: ''
      pgbackrest:
        stanza: main
        password: ''
      prometheus:
        password: ''
        port: 9186
      roles:
        - name: admin
          login: true
          password: ''
          connection_limit: 10
          validity: '2025-01-01T00:00'
          in_roles:
            - pg_read_all_stats
            - pg_signal_backend
      databases:
        - name: main
          owner: admin
          settings:
            work_mem: 3MB
          extensions:
            - name: unaccent
              schema: public
```

Le module `dalibo.pglift.instance` permet de gérer l'instance, et les objets reliés comme des rôles ou des bases de données. Les données sensibles peuvent être prises en charge par Ansible vault[21]. Les modules Ansible permettent un contrôle plus important que la ligne de commandes, grâce aux

[20] https://pglift.readthedocs.io/en/latest/tutorials/cli.html
[21] https://docs.ansible.com/ansible/latest/vault_guide/index.html

champs imbriqués, pouvant inclure la configuration de l'instance, des bases de données, des extensions, etc.

Comme tout module Ansible en général, cette interface est complètement déclarative, idempotente et sans état. Ces modules fonctionnent avec d'autres modules Ansible, tels que `community.postgresql`. Le tutoriel Ansible[22] de la documentation recense davantage d'exemples d'utilisation.

[22] https://pglift.readthedocs.io/en/latest/tutorials/ansible.html

3.6 UTILISER POSTGRESQL DANS UN CONTENEUR

3.6.1 Docker

```
docker run --name pg17 -e POSTGRES_PASSWORD=mdpsuperfort -d postgres

docker exec -it pg17 psql -U postgres
```

- Image officielle Docker Inc (pas PGDG !) : https://hub.docker.com/_/postgres
- Très pratique pour le développement
- Beaucoup de possibilités avancées avec `docker-compose`
- Ne jamais lancer 2 conteneurs Docker sur un même PGDATA
- Une base de données est-elle faite pour du docker ?
  - supervision système délicate

Les conteneurs comme docker et assimilés (podman, containerd, etc.) permettent d'isoler l'installation de PostgreSQL sur un poste sans avoir à gérer de machine virtuelle. Une fois configuré, le conteneur permet d'avoir un contexte d'exécution de PostgreSQL identique peu importe son support. C'est idéal dans le cas de machines « jetables », par exemple les chaînes de CI/CD.

Exemple :

À titre d'exemple, voici les étapes nécessaires à l'installation d'une instance PostgreSQL sous docker.

D'abord, récupérer l'image[23] de la dernière version de PostgreSQL maintenue par la communauté *PostgreSQL docker* :

```
docker pull postgres
```

La commande permet de créer et de lancer un nouveau conteneur à partir d'une image donnée, ici `postgres:17.0`. Certaines options sont passées en paramètres à `initdb` :

```
docker run \
--network net17 \
--name pg17 \
-p 127.0.0.1:16501:5432 \
--env-file password.env \
-e POSTGRES_INITDB_ARGS='--data-checksums --wal-segsize=1' \
-v '/var/lib/postgresql/docker/17/pg17':'/var/lib/postgresql/data' \
-d postgres:17.0 \
-c 'work_mem=10MB' \
-c 'shared_buffers=256MB' \
-c 'cluster_name=pg17'
```

[23] https://hub.docker.com/_/postgres

Cette commande permet au conteneur d'être attaché à un réseau dédié. Celui-ci doit avoir été créé au préalable avec la commande :

```
docker network create --subnet=192.168.122.0/24 net17
```

L'option `--name` permet de nommer le conteneur pour le rendre plus facilement accessible.

L'option `-p` permet de faire suivre le port 5432 ouvert par le container sur le port 16501 du serveur.

L'option `-d` permet de faire de l'image un démon en arrière-plan.

Les options `-e` définissent des variables d'environnement, moyen systématique de passer des informations au conteneur. Ici l'option est utilisée pour le mot de passe et certaines des options d' `initdb`. On préférera utiliser un fichier `.env` pour `docker compose` ou `docker run --env-file` pour éviter de définir un secret dans la ligne de commande.

L'option `--env-file` permet de passer en paramètre un fichier contenant des variables d'environnement. Ici nous passons un fichier contenant le mot de passe dont le contenu est le suivant :

```
# fichier password.env
POSTGRES_PASSWORD=mdpsuperfort
```

L'option `-v '/var/lib/postgresql/docker/17/pg17':'/var/lib/postgresql/data'` lie un répertoire du disque sur le `PGDATA` du container : ainsi les fichiers de la base survivront à la disparition du conteneur.

Les paramètres de PostgreSQL dans les `-c` sont transmis directement à la ligne de commande du postmaster.

Une fois l'image téléchargée et le conteneur instancié, il est possible d'accéder directement à psql via la commande suivante :

```
docker exec -it pg17 psql -U postgres
```

Ou directement via le serveur hôte s'il dispose de psql :

```
psql -h 127.0.0.1 -p 16501 -U postgres
```

En production, il est recommandé de dédier un serveur à chaque instance PostgreSQL. De ce fait, docker permet de reproduire cette isolation pour des cas d'usage de développement et prototypage où dédier une machine à PostgreSQL est soit trop coûteux, soit trop complexe.

Exemple avec docker compose :

Il est évidement possible de passer par l'outil `docker compose` pour déployer une instance PostgreSQL conteneurisée. Voici la commande `docker run` précédente portée sous `docker compose` dans un fichier YAML.

Fichier `docker-compose.yml`

```
version: '3.3'
```

```yaml
networks:
  net16:
    ipam:
      driver: default
      config:
        - subnet: 192.168.122.0/24

services:
    pg16:
      networks:
          - net17
      container_name: pg17
      ports:
          - '127.0.0.1:16501:5432'
      env_file:
          - password.env
      environment:
          - POSTGRES_INITDB_ARGS=--data-checksums --wal-segsize=1
      volumes:
          - /var/lib/postgresql/docker/17/pg17:/var/lib/postgresql/data'
      image: postgres:17.0
      command: [
      -c, work_mem=10MB,
      -c, shared_buffers=256MB,
      ]
```

La commande `docker compose --file docker-compose.yml up -d` permet la création, ou le lancement, des objets définis dans le fichier `docker-compose.yml`. L'option `-d` permet de lancer les conteneurs en arrière-plan.

```
# Au premier lancement
docker compose --file docker-compose.yml up -d
Creating network "postgresql_net17" with the default driver
Creating pg17 ... done

# Aux lancements suivants
docker compose --file docker-compose.yml up -d
Starting pg17 ... done
```

Pour arrêter, il faut utiliser l'option `stop` :

```
docker compose --file docker-compose.yml stop
Stopping pg17 ... done
```

Limites de l'utilisation de PostgreSQL sous docker :

Ne lancez jamais 2 instances de PostgreSQL sous docker sur le même PGDATA ! Les sécurités habituelles de PostgreSQL ne fonctionnent pas et les deux instances écriront toutes les deux dans les fichiers. La corruption est garantie.

En production, si l'utilisation de PostgreSQL sous docker est de nos jours très fréquente, ce n'est pas

toujours une bonne idée. Une base de données est l'inverse total d'une machine sans état et jetable, et pour les performances, il vaut mieux en première intention gonfler la base (*scale up*) que multiplier les instances (*scale out*), qui sont de toute façon toutes dépendantes de l'instance primaire. Si le choix est fait de fonctionner sous docker, voire Kubernetes, pour des raisons architecturales, la base de données est sans doute le dernier composant à migrer.

De plus, pour les performances, la supervision au niveau système devient très compliquée, et le DBA est encore plus aveugle qu'avec une virtualisation classique. L'ajout d'outillage ou d'extensions non fournies avec PostgreSQL devient un peu plus compliquée (`docker stats` n'est qu'un bon début).

3.6.2 Kubernetes

- Dans les très grands environnements
  - mise à l'échelle automatique
  - complexité à assumer
- Réclame de l'expérience sur PG **et** K8s
- Réservez un *worker* à PostgreSQL
- Quel opérateur ?
  - CrunchyData, CloudNativePG, Percona…
  - nombreux outils, HA…

La taille de certains SI, ou le besoin très important d'automatisation, peuvent justifier le déploiement d'un outil comme Kubernetes. Il permet entre autres l'*autoscaling* lors de variations de charges importantes.

Comme déjà dit ci-dessus : les bases de données sont tout l'inverse de machines jetables. Il est toutefois possible de les déployer sur Kubernetes, mais vos bases PostgreSQL sont sans doute les derniers éléments de votre infrastructure à migrer vers Kubernetes, si même leur criticité le permet. En effet, PostgreSQL est plutôt adapté à une scalabilité verticale (ajout de mémoire ou processeurs) qu'horizontale, même si la multiplication de secondaires en lecture seule ou du *sharding* peut être envisageable. Par contre, des bases de données non critiques et/ou à courte durée de vie (par exemple pour du développement), et surtout standardisées, ont intérêt à être gérées par Kubernetes.

L'utilisation de Kubernetes apporte une couche d'abstraction supplémentaire et demande nécessairement des connaissances à avoir sur son fonctionnement. La manière de travailler avec PostgreSQL changera aussi. On peut par exemple citer la modification des paramètres de configuration qui pourra, selon la façon dont est déployée l'instance dans Kubernetes, être très différente de la manière « traditionnelle ». Kubernetes est une plateforme de déploiement d'applications et ne répondra pas à votre place aux considérations liées à PostgreSQL :

- les problèmes de performance dus au SQL ou aux disques trop lents, existeront toujours ;

- les montées de versions seront toujours à faire régulièrement ;
- comprendre les concepts essentiels de PostgreSQL (instances, bases, schéma, etc) reste obligatoire ;
- le besoin d'arbitrage entre peu de grosses instances multibases, et de nombreuses petites instances séparée, reste présent ; même si la facilité de déploiement et la légèreté de PostgreSQL favorise les instances monobases.

Même si en théorie on peut s'en passer, il est fortement conseillé de passer par un opérateur Kubernetes dédié à PostgreSQL, un composant qui facilite énormément l'administration de PostgreSQL dans Kubernetes. Plusieurs fournisseurs en proposent un, chacun avec ses avantages, inconvénients et limites. Ils proposent notamment :

- des images docker toutes prêtes, basées sur les paquets du PGDG, ou des images maison, avec ou sans outils annexes (comme le pooler pgBouncer), et plus ou moins d'extensions ;
- la mise en place de sauvegarde PITR avec des outils spécifiques ;
- une mise en place automatique de haute disponibilité, avec la construction automatique d'instances en réplication (lecture seule) et des bascules automatiques, le tout étant basé sur Patroni ou sur les mécanismes de Kubernetes ;
- des mécanismes de surveillance à destination de Kubernetes pour la connaissance de la santé des instances ;
- des outils de supervision ;
- des mécanismes de mises à jour mineure et majeure ;
- et bien d'autres choses encore.

Citons notamment les opérateurs suivants :

- postgres-operator, de Zalando (avec WAL-E et Patroni, image docker propre) ;
- postgres-operator, de Crunchy Data (avec pgBackRest et Patroni, dépôts propres) ;
- CloudNativePG, initié à l'origine par EDB, le plus récent et peut-être le plus « à la mode » et « cloud native » (avec barman) ;
- Percona Operator for PostgreSQL (avec pgBackRest, dépôts propres) ;
- StackGres (avec Patroni)

Pour choisir un opérateur, les points d'attention sont :

- la licence (opérateur et images docker), car certains fournisseurs poussent bien sûr à la version payante ;
- la disponibilité et la transparence du code source ;
- la pérennité de l'opérateur ;
- la documentation ;
- la rapidité de mise à jour des versions de PostgreSQL et, à l'inverse, le support de vieilles versions de PostgreSQL que vous seriez obligés de conserver ;
- le niveau de capacité[24] de l'opérateur (Level III, IV, V) ;
- le comportement lors des mises à jour pour limiter les interruptions de service ;
- la facilité d'adaptation de l'image docker sous-jacente (par exemple avec une extension supplémentaire) ;

[24] https://operatorframework.io/operator-capabilities/

- les outils intégrés (sauvegarde, supervision…), leur facilité d'emploi, leur intégration avec l'existant (chaque éditeur a tendance à pousser ses outils propres) ;
- la présence de télémétrie, rarement bienvenue.

Le sujet Kubernetes est trop vaste et évolue trop vite pour être développé ici. Ajoutons juste qu'un conseil fréquent est de réserver un nœud (une machine physique ou virtuelle) à chaque instance PostgreSQL, notamment pour s'assurer que l'instance aura toujours à disposition des capacités RAM et CPU. Dédier un espace de stockage aux bases est également important, car une base de données est très sensible aux performances du stockage.

3.7 INSTALLATION SOUS WINDOWS

3.7.1 Comment installer sous Windows ?

- Un seul installeur graphique disponible, proposé par EnterpriseDB
- Ou archive des binaires

Le portage de PostgreSQL sous Windows a justifié à lui seul le passage de la branche 7 à la branche 8 du projet. Le système de fichiers NTFS est obligatoire car, contrairement à la VFAT, il gère les liens symboliques (appelés jonctions sous Windows).

L'installateur n'existe plus qu'en version 64 bits depuis PostgreSQL 11.

Étant donné la quantité de travail nécessaire pour le développement et la maintenance de l'installeur graphique, la communauté a abandonné l'installeur graphique qu'elle a proposé un temps. EnterpriseDB a continué de proposer gratuitement le sien, pour la version communautaire comme pour leur version payante. D'autres installateurs ont été proposés par d'autres éditeurs.

Il contient le serveur PostgreSQL avec les modules contrib ainsi que pgAdmin 4, et aussi un outil appelé StackBuilder permettant d'installer d'autres outils comme des pilotes JDBC, ODBC, C#, ou PostGIS.

Pour installer PostgreSQL sans installateur ni compilation, EBD propose aussi une archive des binaires compilés[25], même de versions aussi anciennes que la 9.3.

[25] https://www.enterprisedb.com/download-postgresql-binaries

3.7.2 Installeur graphique

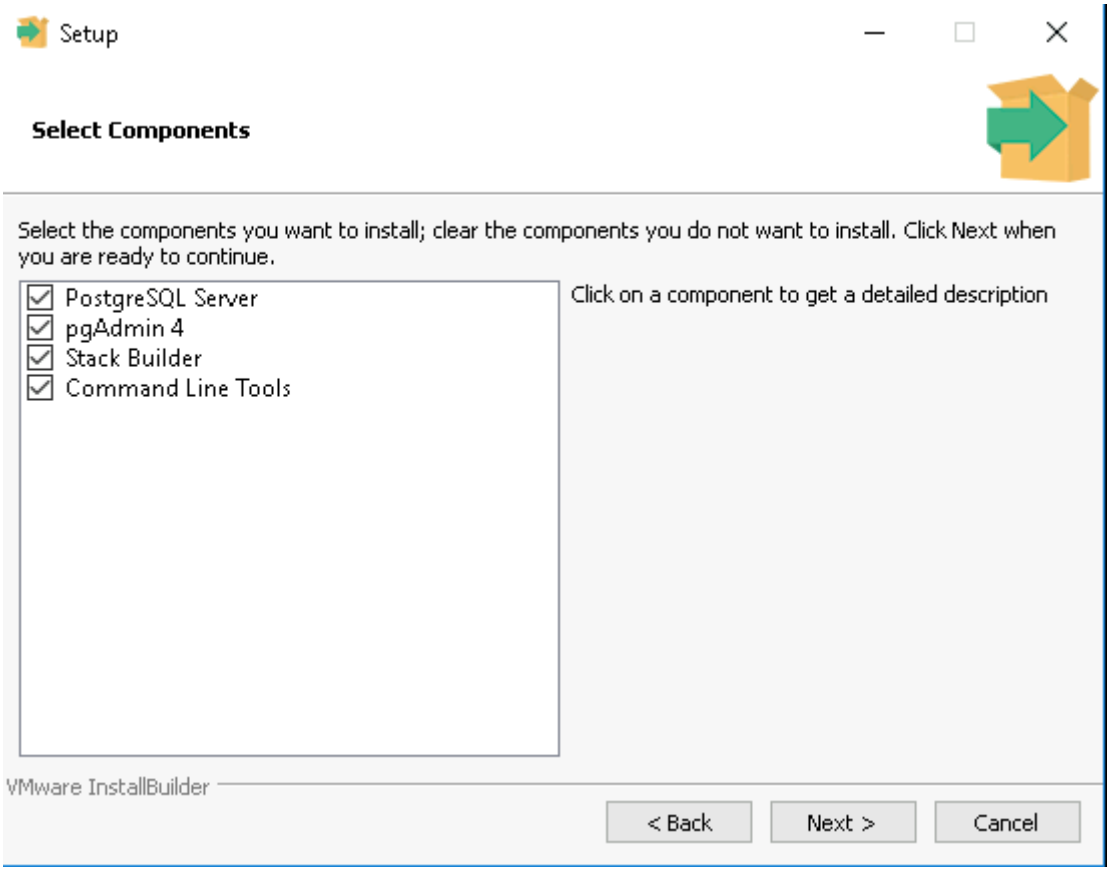


Figure 3/ .1: Installeur graphique - bienvenue

Son utilisation est tout à fait classique. Il y a plusieurs écrans de saisie d'informations :

- le répertoire d'installation des binaires ;
- le choix des outils (copie d'écran ci-dessus), notamment des outils en ligne de commande (à conserver impérativement), des pilotes et de pgAdmin 4 ;
- le répertoire des données de la première instance ;
- le mot de passe de l'utilisateur **postgres** ;
- le numéro de port ;
- la locale par défaut.

Le répertoire d'installation a une valeur par défaut généralement convenable car il n'existe pas vraiment de raison d'installer les binaires PostgreSQL en dehors du répertoire *Program Files*.

Par contre, le répertoire des données de l'instance PostgreSQL. n'a pas à être dans ce même répertoire *Program Files* ! Il est souvent modifié pour un autre disque que le disque système.

Le numéro de port est par défaut le 5432, sauf si d'autres instances sont déjà installées. Dans ce cas, l'installeur propose un numéro de port non utilisé.

Le mot de passe est celui de l'utilisateur **postgres** au sein de PostgreSQL. En cas de mise à jour, il faut saisir l'ancien mot de passe. Le service lui-même et tous ses processus tourneront avec le compte système générique **NETWORK SERVICE** (Compte de service réseau).

La commande `initdb` est exécutée pour créer le répertoire des données. Un service est ajouté pour lancer automatiquement le serveur au démarrage de Windows.

Un sous-menu du menu *Démarrage* contient le nécessaire pour interagir avec le serveur PostgreSQL, comme une icône pour recharger la configuration et surtout pgAdmin 4, qui se lancera dans un navigateur.

L'outil StackBuilder, lancé dans la foulée, permet de télécharger et d'installer d'autres outils : pilotes pour différents langages (Npgsql pour C#, pgJDBC, psqlODBC), PostGIS… installés dans le répertoire de l'utilisateur en cours.

L'installateur peut être utilisé uniquement en ligne de commande (voir les options avec `--help`).

Cet installeur existe aussi sous macOS. Autant il est préférable de passer par les paquets de sa distribution Linux, autant il est recommandé d'utiliser cet installeur sous macOS.

3.8 PREMIERS RÉGLAGES

3.8.1 Au menu

- Sécurité
- Configuration
  - accès
  - connexions
  - mémoire
  - journaux
  - traces
  - tâches de fond

3.8.2 Sécurité

- Politique d'accès :
  - pour l'utilisateur **postgres** système
  - pour le rôle **postgres**
- Règles d'accès à l'instance dans `pg_hba.conf`

Selon l'environnement et la politique de sécurité interne à l'entreprise, il faut potentiellement initialiser un mot de passe pour l'utilisateur système **postgres** :

```
$ passwd postgres
```

Sans mot de passe, il faudra passer par un système comme `sudo` pour pouvoir exécuter des commandes en tant qu'utilisateur **postgres**, ce qui sera nécessaire au moins au début.

Le fait de savoir qu'un utilisateur existe sur un serveur permet à un utilisateur hostile de tenter de forcer une connexion par force brute. Par exemple, ce billet de blog[26], montre que l'utilisateur **postgres** est dans le top 10 des logins attaqués.

La meilleure solution pour éviter ce type d'attaque est de ne pas définir de mot de passe pour l'utilisateur OS **postgres** et de se connecter uniquement par des échanges de clés SSH.

[26] https://blog.sucuri.net/2013/07/ssh-brute-force-the-10-year-old-attack-that-still-persists.html

Il est conseillé de ne fixer aucun mot de passe pour l'utilisateur système. Il en va de même pour le rôle **postgres** dans l'instance. Une fois connecté au système, nous pourrons utiliser le mode d'authentification local `peer` pour nous connecter au rôle **postgres**. Ce mode permet de limiter la surface d'attaque sur son instance.

En cas de besoin d'accès distant en mode superutilisateur, il sera possible de créer des rôles supplémentaires avec des droits superutilisateur. Ces noms ne doivent pas être facile à deviner par de potentiels attaquants. Il faut donc éviter les rôles **admin** ou **root**.

Si vous avez besoin de créer des mots de passe, ils doivent bien sûr être longs et complexes (par exemple en les générant avec les utilitaires `pwgen` ou `apg`).

Si vous avez utilisé l'installeur proposé par EnterpriseDB, l'utilisateur système et le rôle PostgreSQL ont déjà un mot de passe, celui demandé par l'installeur. Il n'est donc pas nécessaire de leur en configurer un autre.

Enfin, il est important de vérifier les règles d'accès au serveur contenues dans le fichier `pg_hba.conf`. Ces règles définissent les accès à l'instance en se basant sur plusieurs paramètres : utilisation du réseau ou du socket fichier, en SSL ou non, depuis quel réseau, en utilisant quel rôle, pour quelle base de données et avec quelle méthode d'authentification.

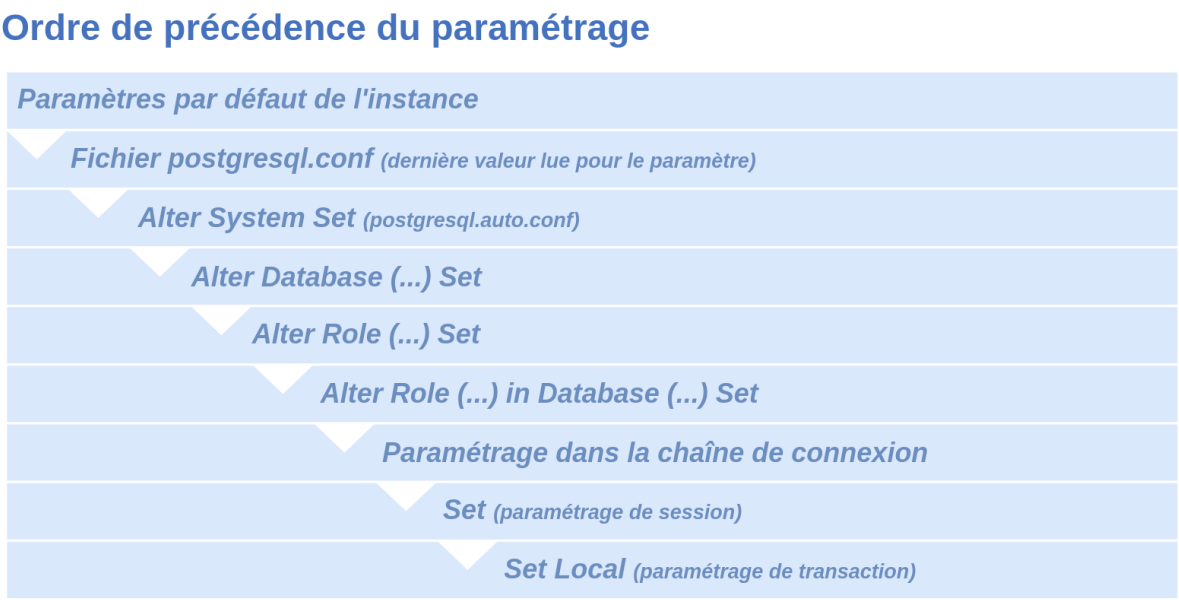
3.8.3 Configuration minimale

- Fichier `postgresql.conf`
- Configuration du moteur
- Plus de 300 paramètres
- Quelques paramètres essentiels

La configuration du moteur se fait via un seul fichier, le fichier `postgresql.conf`. Il se trouve généralement dans le répertoire des données du serveur PostgreSQL. Sous certaines distributions (Debian et affiliés principalement), il est déplacé dans `/etc/postgresql/`.

Ce fichier contient beaucoup de paramètres, plus de 300, mais seuls quelques-uns sont essentiels à connaître pour avoir une instance fiable et performante.

3.8.4 Précédence des paramètres



PostgreSQL offre une certaine granularité dans sa configuration, ainsi certains paramètres peuvent être surchargés par rapport au fichier `postgresql.conf`. Il est utile de connaître l'ordre de précédence. Par exemple, un utilisateur peut spécifier un paramètre dans sa session avec l'ordre `SET`, celui-ci sera prioritaire par rapport à la configuration présente dans le fichier `postgresql.conf`.

3.8.5 Configuration des connexions : accès au serveur

- `listen_addresses = '*'` (systématique)
- `port = 5432`
- `password_encryption` = `scram-sha-256` (v10+)

Ouvrir les accès :

Par défaut, une instance PostgreSQL n'écoute que sur l'interface de boucle locale (`localhost`) et pas sur les autres interfaces réseaux. Pour autoriser les connexions externes à PostgreSQL, il faut modifier le paramètre `listen_addresses`, en général ainsi :

```
listen_addresses = '*'
```

La valeur `*` est un joker indiquant que PostgreSQL doit écouter sur toutes les interfaces réseaux disponibles au moment où il est lancé. Il est aussi possible d'indiquer les interfaces, une à une, en les sé-

parant avec des virgules. Cette méthode est intéressante lorsqu'on veut éviter que l'instance écoute sur une interface donnée. Par prudence il est possible de se limiter aux interfaces destinées à être utilisées :

```
listen_addresses = 'localhost, 10.1.123.123'
```

La restriction par `listen_addresses` est un premier niveau de sécurité. Elle est complémentaire de la méthode plus fine par `pg_hba.conf`, par les IP clientes, utilisateur et base, qu'il faudra de toute façon déployer. De plus, modifier `listen_addresses` impose de redémarrer l'instance.

Port :

Le port par défaut des connexions TCP/IP est le `5432`. C'est la valeur traditionnelle et connue de tous les outils courants.

La modifier n'a d'intérêt que si vous voulez exécuter plusieurs instances PostgreSQL sur le même serveur (physique ou virtuel). En effet, plusieurs instances sur une même machine ne peuvent pas écouter sur le même couple adresse IP et port.

Une instance PostgreSQL n'écoute jamais que sur ce seul port, et tous les clients se connectent dessus. Il n'existe pas de notion de *listener* ou d'outil de redirection comme sur d'autres bases de données concurrentes, du moins sans outil supplémentaire (par exemple le pooler PgBouncer).

S'il y a plusieurs instances dans une même machine, elles devront posséder chacune un couple adresse IP/port unique. En pratique, il vaut mieux attribuer un port par instance. Bien sûr, PostgreSQL refusera de démarrer s'il voit que le port est déjà occupé.

Ne confondez pas la connexion à `localhost` (soit `::1` en IPv6 ou `127.0.0.1` en IPv4), qui utilise les ports TCP/IP, et la connexion dite `local`, passant par les *sockets* de l'OS (par défaut `/var/run/postgresql/.s.PGSQL.5432` sur les distributions les plus courantes). La distinction est importante dans `pg_hba.conf` notamment.

Chiffrement des mots de passe :

À partir de la version 10 et avant la version 14, le paramètre `password_encryption` est à modifier dès l'installation. Il définit l'algorithme de chiffrement utilisé pour le stockage des mots de passe. La valeur `scram-sha-256` permettra d'utiliser la nouvelle norme, plus sécurisée que l'ancien `md5`. Ce n'est plus nécessaire à partir de la version 14 car c'est la valeur par défaut. Avant toute modification, vérifiez quand même que vos outils clients sont compatibles. Au besoin, vous pouvez revenir à `md5` pour un utilisateur donné.

3.8.6 Configuration du nombre de connexions

- `max_connections = 100`
- 1 connexion = 1 processus serveur
- Compromis entre
  - CPU / nombre requêtes actives / RAM / complexité
- Danger si trop haut !
  - performances (même avec des connexions inactives)
  - risque de saturation
- Possibilité de réserver quelques connexions pour l'administration

Le nombre de connexions simultanées est limité par le paramètre `max_connections`. Dès que ce nombre est atteint, les connexions suivantes sont refusées avec un message d'erreur, et ce jusqu'à ce qu'un utilisateur connecté se déconnecte.

`max_connections` vaut par défaut 100, et c'est généralement suffisant en première intention.

Noter qu'il existe un paramètre `superuser_reserved_connections` (à 3 par défaut) qui réserve quelques connexions au superutilisateur pour qu'il puisse se connecter malgré une saturation. Depuis PostgreSQL 16, il existe un autre paramètre nommé `reserved_connections`, (à 0 par défaut) pour réserver quelques connexions aux utilisateurs à qui l'on aura attribué un rôle spécifique, nommé `pg_use_reserved_connections`. Ce peut être utile pour des utilisateurs non applicatifs (supervision et sauvegarde notamment) à qui l'on ne veut ou peut pas donner le rôle `SUPERUSER`.

Il peut être intéressant de diminuer `max_connections` pour interdire d'avoir trop de connexions actives. Cela permet de soulager les entrées-sorties, ou de monter `work_mem` (la mémoire de tri). À l'inverse, il est possible d'augmenter `max_connections` pour qu'un plus grand nombre d'utilisateurs ou d'applications puisse se connecter en même temps.

Au niveau mémoire, un processus consomme par défaut 2 Mo de mémoire vive. Cette consommation peut augmenter suivant son activité.

Il faut surtout savoir qu'à chaque connexion se voit associée un processus sur le serveur, processus qui n'est vraiment actif qu'à l'exécution d'une requête. Il s'agit donc d'arbitrer entre :

- le nombre de requêtes à exécuter à un instant T ;

- le nombre de CPU disponibles ;
- la complexité et la longueur des requêtes ;
- et même le nombre de processus que peut gérer l'OS.

L'établissement a un certain coût également. Il faut éviter qu'une application se connecte et se déconnecte sans cesse.

Il ne sert à rien d'autoriser des milliers de connexions s'il n'y a que quelques processeurs, ou si les requêtes sont lourdes. Si le nombre de requêtes réellement actives augmente fortement, le serveur peut s'effondrer. Restreindre les connexions permet de préserver le serveur, même si certaines connexions sont refusées.

Le paramétrage est compliqué par le fait qu'une même requête peut mobiliser plusieurs processeurs si elle est parallélisée. Certaines requêtes seront limitées par le CPU, d'autres par la bande passante des disques.

Enfin, même si une connexion inactive ne consomme pas de CPU et peu de RAM, elle a tout de même un impact. En effet, une connexion active va générer assez fréquemment ce qu'on appelle un snapshot (ou une image) de l'état des transactions de la base. La durée de création de ce snapshot dépend principalement du nombre de connexions, actives ou non, sur le serveur. Donc une connexion active consommera plus de CPU s'il y a 399 autres connexions, actives ou non, que s'il y a 9 connexions, actives ou non. Ce comportement est partiellement corrigé avec la version 14. Mais il vaut mieux éviter d'avoir des milliers de connexions ouvertes « au cas où ».

Intercaler un « pooler » comme PgBouncer entre les clients et l'instance peut se justifier dans certains cas :

- connexions/déconnexions très fréquentes ;
- centaines, voire milliers, de connexions généralement inactives ;
- limitation du nombre de connexions actives avec mise en attente au niveau du pooler (sans erreur).

3.8.7 Configuration de la mémoire partagée

- `shared_buffers = (?)GB`
  - 25 % de la RAM généralement
  - max 40 %
  - complémentaire du cache OS

Shared buffers :

Chaque fois que PostgreSQL a besoin de lire ou d'écrire des données, il les charge d'abord dans son cache interne. Ce cache ne sert qu'à ça : stocker des blocs disques qui sont accessibles à tous les

processus PostgreSQL, ce qui permet d'éviter de trop fréquents accès disques car ces accès sont lents. La taille de ce cache dépend d'un paramètre appelé `shared_buffers`.

La documentation officielle[27] conseille ceci pour dimensionner `shared_buffers` :

> Un bon point de départ est 25 % de la mémoire vive totale. Ne pas dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine dédiée de 32 Go de RAM, cela donne donc :

```
shared_buffers = 8GB
```

Le défaut de 128 Mo n'est donc pas adapté à un serveur sur une machine récente.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Le cache système limite la plupart du temps l'impact d'un mauvais paramétrage de `shared_buffers`, et il est moins grave de sous-dimensionner un peu `shared_buffers` que de le sur-dimensionner.

> Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres liés aux journaux et *checkpoints* comme `max_wal_size`. Il faut aussi s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…) et donc adapter `work_mem`.

> Modifier `shared_buffers` impose de redémarrer l'instance.

[27] https://docs.postgresql.fr/current/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-MEMORY

3.8.8 Configuration : mémoire des processus

- `work_mem`
  - par processus, voire nœud
  - valeur très dépendante de la charge et des requêtes
  - fichiers temporaires vs saturation RAM
- × `hash_mem_multiplier`
- `maintenance_work_mem`
- Pas de limite stricte à la consommation mémoire des sessions
  - Augmenter prudemment & superviser

Les processus de PostgreSQL ont accès à la mémoire partagée, définie principalement par `shared_buffers`, mais ils ont aussi leur mémoire propre. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

- Le paramètre le plus important est `work_mem`, qui définit la taille maximale de la mémoire de travail d'un `ORDER BY`, de certaines jointures, pour la déduplication… que peut utiliser un processus sur un nœud de requête, principalement lors d'opérations de tri ou regroupement.
- Autre paramètre capital, `maintenance_work_mem` qui est la mémoire utilisable pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, `REINDEX`, ajouts de clé étrangère…

Cette mémoire liée au processus est rendue immédiatement après la fin de l'ordre concerné.

- Il existe aussi `logical_decoding_work_mem` (défaut : 64 Mo), utilisable pour chacun des flux de réplication logique (s'il y en a, ils sont rarement nombreux).

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go, voire plus sur les machines récentes, car il concerne des opérations lourdes (indexation, nettoyage des index par `VACUUM`…). Leurs consommations de RAM s'additionnent, mais en pratique ces opérations sont rarement exécutées plusieurs fois simultanément.

Monter au-delà de 1 Go n'a d'intérêt que pour la création ou la réindexation de très gros index.

`autovacuum_work_mem` est la mémoire que s'autorise à prendre l'autovacuum pour les nettoyages d'index, et ce pour chaque *worker* de l'autovacuum (3 maximum par défaut). Par défaut, ce paramètre reprend `maintenance_work_mem`, et est généralement laissé tel quel.

Paramétrage de work_mem :

Pour `work_mem`, c'est beaucoup plus compliqué.

Si `work_mem` est trop bas, beaucoup d'opérations de tri, y compris nombre de jointures, ne s'effectueront pas en RAM. Par exemple, si une jointure par hachage impose d'utiliser 100 Mo en mémoire, mais que `work_mem` vaut 10 Mo, PostgreSQL écrira des dizaines de Mo sur disque à chaque appel de la jointure. Si, par contre, le paramètre `work_mem` vaut 120 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément la requête.

Trop de fichiers temporaires peuvent ralentir les opérations, voire saturer le disque. Un `work_mem` trop bas peut aussi contraindre le planificateur à choisir des plans d'exécution moins optimaux.

Par contre, si `work_mem` est trop haut, et que trop de requêtes le consomment simultanément, le danger est de saturer la RAM. Il n'existe en effet pas de limite à la consommation des sessions de PostgreSQL, ni globalement ni par session !
Or le paramétrage de l'overcommit sous Linux est par défaut très permissif, le noyau ne bloquera rien. La première conséquence de la saturation de mémoire est l'assèchement du cache système (complémentaire de celui de PostgreSQL), et la dégradation des performances. Puis le système va se mettre à swapper, avec à la clé un ralentissement général et durable. Enfin le noyau, à court de mémoire, peut être amené à tuer un processus de PostgreSQL. Cela mène à l'arrêt de l'instance, ou plus fréquemment à son redémarrage brutal avec coupure de toutes les connexions et requêtes en cours.

Toutefois, si l'administrateur paramètre correctement l'overcommit (voir https://dali.bo/j1_html#configuration-de-la-surréservation-mémoire), Linux refusera d'allouer la RAM et la requête tombera en erreur, mais le cache système sera préservé, et PostgreSQL ne tombera pas.

Suivant la complexité des requêtes, il est possible qu'un processus utilise plusieurs fois `work_mem` (par exemple si une requête fait une jointure et un tri, ou qu'un nœud est parallélisé). À l'inverse, beaucoup de requêtes ne nécessitent aucune mémoire de travail.

La valeur de `work_mem` dépend donc beaucoup de la mémoire disponible, des requêtes et du nombre de connexions actives.

Si le nombre de requêtes simultanées est important, `work_mem` devra être faible. Avec peu de requêtes simultanées, `work_mem` pourra être augmenté sans risque.

Il n'y a pas de formule de calcul miracle. Une première estimation courante, bien que très conservatrice, peut être :

> `work_mem` = mémoire / `max_connections`

On obtient alors, sur un serveur dédié avec 16 Go de RAM et 200 connexions autorisées :

```
work_mem = 80MB
```

Mais `max_connections` est fréquemment surdimensionné, et beaucoup de sessions sont inactives. `work_mem` est alors sous-dimensionné.

Plus finement, Christophe Pettus propose en première intention[28] :

> `work_mem` = 4 × mémoire libre / `max_connections`

Soit, pour une machine dédiée avec 16 Go de RAM, donc 4 Go de *shared buffers*, et 200 connections :

```
work_mem = 240MB
```

Dans l'idéal, si l'on a le temps pour une étude, on montera `work_mem` jusqu'à voir disparaître l'essentiel des fichiers temporaires dans les traces, tout en restant loin de saturer la RAM lors des pics de charge.

En pratique, le défaut de 4 Mo est très conservateur, souvent insuffisant. Généralement, la valeur varie entre 10 et 100 Mo. Au-delà de 100 Mo, il y a souvent un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisés pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Augmenter globalement la valeur du `work_mem` peut parfois mener à une consommation excessive de mémoire. Il est possible de ne la modifier que le temps d'une session pour les besoins d'une requête ou d'un traitement particulier :

```
SET work_mem TO '30MB' ;
```

hash_mem_multiplier :

`hash_mem_multiplier` est un paramètre multiplicateur, qui peut s'appliquer à certaines opérations (le hachage, lors de jointures ou agrégations). Par défaut, il vaut 1 en versions 13 et 14, et 2 à partir de la 15. Le seuil de consommation fixé par `work_mem` est alors multiplié d'autant. `hash_mem_multiplier` permet de donner plus de RAM à ces opérations sans augmenter globalement `work_mem`. Il peut lui aussi être modifié dans une session.

Il existe d'autres paramètres influant sur les besoins en mémoires, moins importants pour une première approche.

3.8.9 Configuration des journaux de transactions 1/2

`fsync` = `on` (si vous tenez à vos données)

[28] https://thebuild.com/blog/2023/03/13/everything-you-know-about-setting-work_mem-is-wrong/

À chaque fois qu'une transaction est validée (`COMMIT`), PostgreSQL écrit les modifications qu'elle a générées dans les journaux de transactions.

Afin de garantir la durabilité, PostgreSQL effectue des écritures synchrones des journaux de transaction, donc une écriture physique des données sur le disque. Cela a un coût important sur les performances en écritures s'il y a de nombreuses transactions mais c'est le prix de la sécurité.

Le paramètre `fsync` permet de désactiver l'envoi de l'ordre de synchronisation au système d'exploitation. Ce paramètre **doit** rester à `on` en production. Dans le cas contraire, un arrêt brutal de la machine peut mener à la perte des journaux non encore enregistrés et à la corruption de l'instance. D'autres paramètres et techniques existent pour gagner en performance (et notamment si certaines données peuvent être perdues) sans pour autant risquer de corrompre l'instance.

3.8.10 Configuration des journaux de transactions 2/2

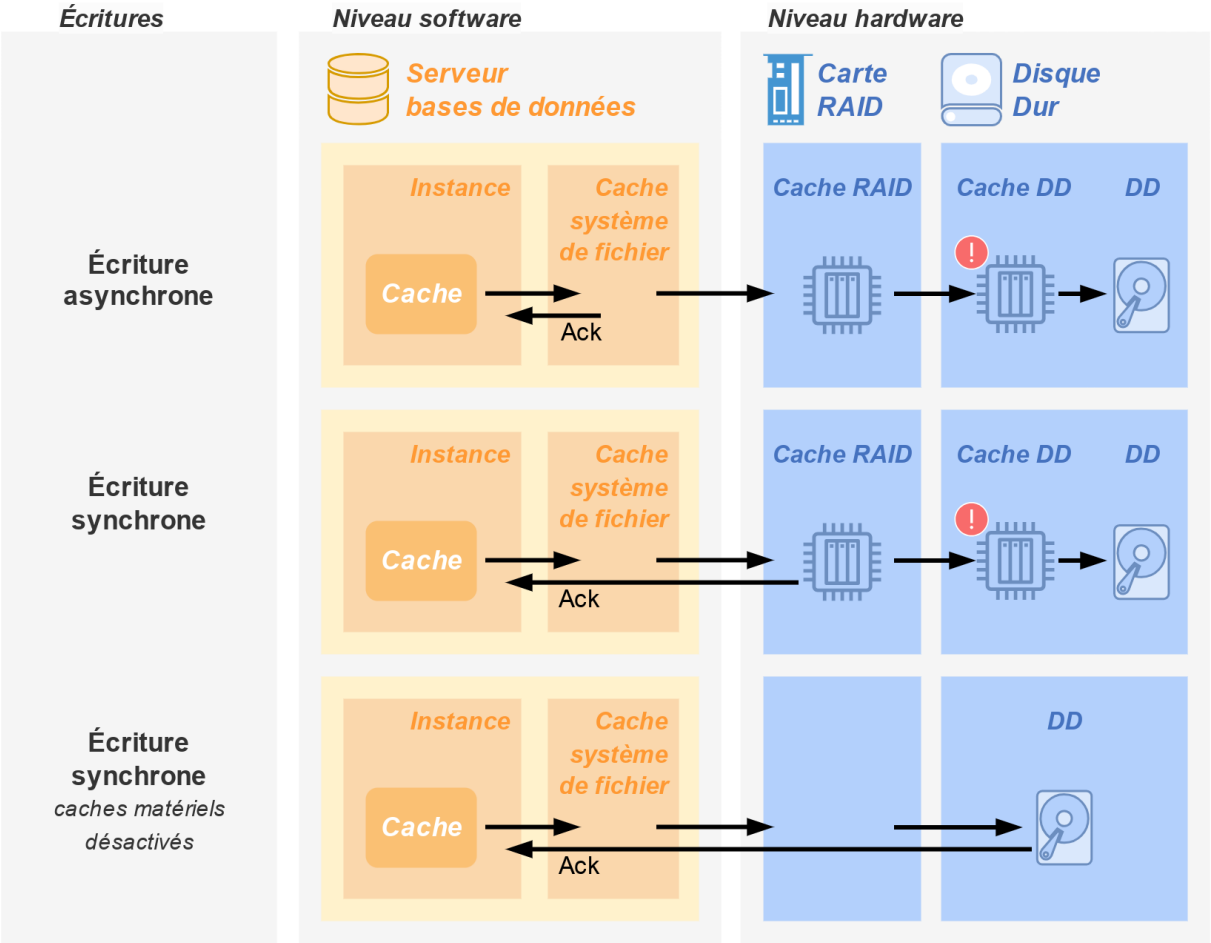


Figure 3/ .2: Niveaux de cache et fsync

Une écriture peut être soit synchrone soit asynchrone. Pour comprendre ce mécanisme, nous allons simplifier le cheminement de l'écriture d'un bloc :

- Dans le cas d'une écriture **asynchrone** : Un processus qui modifie un fichier écrit en fait d'abord dans le cache du système de fichiers du système d'exploitation (OS), cache situé en RAM (mémoire volatile). L'OS confirme tout de suite au processus que l'écriture a été réalisée pour lui rendre la main au plus vite : il y a donc un gain en performance important. Cependant, le bloc ne sera écrit sur disque que plus tard afin notamment de grouper les demandes d'écritures des autres processus, et de réduire les déplacements des têtes de lecture/écriture des disques, qui sont des opérations coûteuses en temps. Entre la confirmation de l'écriture et l'écriture réelle sur les disques, il peut se passer un certain délai : si une panne survient durant celui-ci, les données soi-disant écrites seront perdues, car pas encore physiquement sur le disque.
- Dans le cas d'une écriture **synchrone** : Un processus écrit dans le cache du système

d'exploitation, puis demande explicitement à l'OS d'effectuer la synchronisation (écriture physique) sur disque. Les blocs sont donc écrits sur les disques immédiatement et le processus n'a la confirmation de l'écriture qu'une fois cela fait. Il attendra donc pendant la durée de cette opération, mais il aura la garantie que la donnée est bien présente physiquement sur les disques. Cette synchronisation est très coûteuse et lente (encore plus avec un disque dur classique et ses têtes de disques à déplacer).

Un phénomène équivalent peut se produire à nouveau au niveau matériel (hors du contrôle de l'OS) : pour gagner en performance, les constructeurs ont rajouté un système de cache au sein des cartes RAID. L'OS (et donc le processus qui écrit) a donc confirmation de l'écriture dès que la donnée est présente dans ce cache, alors qu'elle n'est pas encore écrite sur disque. Afin d'éviter la perte de donnée en cas de panne électrique, ce cache est secouru par une batterie qui laissera le temps d'écrire le contenu du cache. Vérifiez qu'elle est bien présente sur vos disques et vos cartes contrôleur RAID.

3.8.11 Configuration des traces

- Selon système/distribution :
  - `log_destination`
  - `logging_collector`
  - emplacement et nom différent pour `postgresql-????.log`
- `log_line_prefix` à compléter :
  - `log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '`
- `lc_messages` = `C` (anglais)

PostgreSQL dispose de plusieurs moyens pour enregistrer les traces : soit il les envoie sur la sortie des erreurs (`stderr`, `csvlog` et `jsonlog`), soit il les envoie à syslog (`syslog`, seulement sous Unix), soit il les envoie au journal des événements (`eventlog`, sous Windows uniquement). Dans le cas où les traces sont envoyées sur la sortie des erreurs, il peut récupérer les messages via un démon appelé *logger process* qui va enregistrer les messages dans des fichiers. Ce démon s'active en configurant le paramètre `logging_collector` à `on`.

Tout cela est configuré par défaut différemment selon le système et la distribution. Red Hat active `logging_collector` et PostgreSQL dépose ses traces dans des fichiers journaliers `$PGDATA/log/postgresql-<jour de la semaine>.log`. Debian utilise `stderr` sans autre paramétrage et c'est le système qui dépose les traces dans `/var/log/postgresql/postgresql-VERSION-nominstance.log`. Les deux variantes fonctionnent. En fonction des habitudes et contraintes locales, il est possible de préférer et d'activer l'une ou l'autre.

L'entête de chaque ligne des traces doit contenir au moins la date et l'heure exacte (`%t` ou `%m` suivant la précision désirée) : des traces sans date et heure ne servent à rien. Des entêtes complets sont suggérés par la documentation de l'analyseur de log pgBadger :

```
log_line_prefix = '%t [%p]: [%l-1] db=%d,user=%u,app=%a,client=%h '
```

Beaucoup d'utilisateurs français récupèrent les traces de PostgreSQL en français. Bien que cela semble une bonne idée au départ, cela se révèle être souvent un problème. Non pas à cause de la qualité de la traduction, mais plutôt parce que les outils de traitement des traces fonctionnent uniquement avec des traces en anglais. Même un outil comme pgBadger, pourtant écrit par un Français, ne sait pas interpréter des traces en français. De plus, la moindre recherche sur Internet ramènera plus de liens si le message est en anglais. Positionnez donc `lc_messages` à `C`.

3.8.12 Configuration des tâches de fond

Laisser ces deux paramètres à `on` :

- `autovacuum`
- `track_counts`

En dehors du *logger process*, PostgreSQL dispose d'autres tâches de fond.

Les processus autovacuum jouent un rôle important pour disposer de bonnes performances : ils empêchent une fragmentation excessive des tables et index, et mettent à jour les statistiques sur les données (statistiques servant à l'optimiseur de requêtes).

La récupération des statistiques sur l'activité permet le bon fonctionnement de l'autovacuum et donne de nombreuses informations importantes à l'administrateur de bases de données.

Ces deux tâches de fond devraient toujours être activés.

3.8.13 Se faciliter la vie

- Création automatique de configuration
  - pgtune et https://pgtune.leopard.in.ua/
  - http://pgconfigurator.cybertec.at/
- Documentation et analyse de configuration
  - https://postgresqlco.nf

pgtune existe en plusieurs versions. La version en ligne de commande va détecter automatiquement le nombre de CPU et la quantité de RAM, alors que la version web nécessitera que ces informations soient saisies. Suivant le type d'utilisation, pgtune proposera une configuration adaptée. Cette configuration n'est évidemment pas forcément optimale par rapport à vos applications, tout simplement parce qu'il ne connaît que les ressources et le type d'utilisation, mais c'est généralement un bon point de départ.

pgconfigurator est un outil plus récent, un peu plus graphique, mais il remplit exactement le même but que pgtune.

Enfin, le site postgresql.co.nf[29] est un peu particulier. C'est en quelque sorte une encyclopédie sur les paramètres de PostgreSQL, mais il est aussi possible de lui faire analyser une configuration. Après analyse, des informations supplémentaires seront affichées pour améliorer cette configuration, que ce soit pour la stabilité du serveur comme pour ses performances.

[29] https://postgresqlco.nf

3.9 MISE À JOUR

3.9.1 Mise à jour

- Recommandations
- Mise à jour mineure
- Mise à jour majeure
- Mise à jour de l'OS

3.9.2 Recommandations

- Les *Release Notes*
- Intégrité des données
- Bien redémarrer le serveur !

Chaque nouvelle version de PostgreSQL est accompagnée d'une note expliquant les améliorations, les corrections et les innovations apportées par cette version, qu'elle soit majeure ou mineure. Ces notes contiennent toujours une section dédiée aux mises à jour dans laquelle se trouvent des conseils essentiels.

Les *Releases Notes* sont présentes dans l'annexe E de la documentation officielle[30].

Les données de votre instance PostgreSQL sont toujours compatibles d'une version mineure à l'autre. Ainsi, les mises à jour vers une version mineure supérieure peuvent se faire sans migration de données, sauf cas exceptionnel qui serait alors précisé dans les notes de version. Par exemple, de la 15.3 à la 15.4, il a été recommandé de reconstruire les index de type BRIN[31] pour prendre en compte une correction de bug les concernant. Autre exemple : à partir de la 10.3[32], `pg_dump` a imposé des noms d'objets qualifiés pour des raisons de sécurité, ce qui a posé problème pour certains réimports.

Pensez éventuellement à faire une sauvegarde préalable par sécurité.

À contrario, si la mise à jour consiste en un changement de version majeure (par exemple, de la 11 à la 17), il est nécessaire de s'assurer que les données seront transférées correctement sans incompatibilité. Là encore, il est important de lire les *Releases Notes* **avant** la mise à jour.

[30] https://docs.postgresql.fr/current/release.html
[31] https://www.postgresql.org/docs/release/15.4/
[32] https://www.postgresql.org/docs/release/10.3/

Le site why-upgrade[33], basé sur les *release notes*, permet de compiler les différences entre plusieurs versions de PostgreSQL.

Dans tous les cas, pensez à bien redémarrer le serveur. Mettre à jour les binaires ne suffit pas.

3.9.3 Mise à jour mineure

- Méthode
  - arrêter PostgreSQL
  - mettre à jour les binaires
  - redémarrer PostgreSQL
- Pas besoin de s'occuper des données, sauf cas exceptionnel
  - bien lire les *Release Notes* pour s'en assurer

Faire une mise à jour mineure est simple et rapide.

La première action est de lire les *Release Notes* pour s'assurer qu'il n'y a pas à se préoccuper des données. C'est généralement le cas mais il est préférable de s'en assurer avant qu'il ne soit trop tard.

La deuxième action est de faire la mise à jour. Tout dépend de la façon dont PostgreSQL a été installé :

- par compilation, il suffit de remplacer les anciens binaires par les nouveaux ;
- par paquets précompilés, il suffit d'utiliser le système de paquets (`apt` sur Debian et affiliés, `yum` ou `dnf` sur Red Hat et affiliés) ;
- par l'installeur graphique, en le ré-exécutant.

Ceci fait, un redémarrage du serveur est nécessaire. Il est intéressant de noter que les paquets Debian s'occupent directement de cette opération. Il n'est donc pas nécessaire de le refaire.

[33] https://why-upgrade.depesz.com/

3.9.4 Mise à jour majeure

- Bien lire les *Release Notes*
- Bien tester l'application avec la nouvelle version
  - rechercher les régressions en terme de fonctionnalités et de performances
  - penser aux extensions et aux outils
- Pour mettre à jour
  - mise à jour des binaires
  - et mise à jour/traitement des fichiers de données
- 3 méthodes
  - dump/restore
  - réplication logique
  - `pg_upgrade`

Faire une mise à jour majeure est une opération complexe à préparer prudemment.

La première action là-aussi est de lire les *Release Notes* pour bien prendre en compte les régressions potentielles en terme de fonctionnalités et/ou de performances. Cela n'arrive presque jamais mais c'est possible malgré toutes les précautions mises en place.

La deuxième action est de mettre en place un serveur de tests où se trouve la nouvelle version de PostgreSQL avec les données de production. Ce serveur sert à tester PostgreSQL mais aussi, et même surtout, l'application. Le but est de vérifier encore une fois les régressions possibles.

N'oubliez pas de tester les extensions non officielles, voire développées en interne, que vous avez installées. Elles sont souvent moins bien testées.

N'oubliez pas non plus de tester les outils d'administration, de monitoring, de modélisation. Ils nécessitent souvent une mise à jour pour être compatibles avec la nouvelle version installée.

Une fois que les tests sont concluants, arrive le moment de la mise en production. C'est une étape qui peut être longue car les fichiers de données doivent être traités. Il existe plusieurs méthodes que nous détaillerons après.

3.9.5 Mise à jour majeure par dump/restore

- Méthode historique
- Simple et sans risque
  - mais d'autant plus longue que le volume de données est important
- Outils :
  - `pg_dumpall -g` puis `pg_dump`
  - `psql` puis `pg_restore`

Il s'agit de la méthode la plus ancienne et la plus sûre. L'idée est de sauvegarder l'ancienne version avec l'outil de sauvegarde de la nouvelle version. `pg_dumpall` peut suffire, mais `pg_dump` est malgré tout recommandé.

Le problème de lenteur vient surtout de la restauration. `pg_restore` est un outil assez lent pour des volumétries importantes. Il convient pour des volumes de données peu conséquents (au plus une centaine de Go), au cas où l'on est patient, ou si les autres méthodes ne sont pas possibles.

Il est cependant possible d'accélérer la restauration grâce à la parallélisation (option `--jobs`). Ce n'est possible que si la sauvegarde a été faite avec `pg_dump -Fd` ou `-Fc`. Il est à noter que cette sauvegarde peut elle aussi être parallélisée (option `--jobs` là encore).

Il existe un piège peu connu lié à toute migration logique d'une grande base : le gel massif des lignes. Pour des raisons techniques de recyclage des numéros de transaction, PostgreSQL doit « geler » les lignes anciennes et jamais modifiées, ce qui implique de réécrire le bloc. Or, toutes les lignes insérées par une migration ont le même « âge ». Si elles ne sont pas modifiées, ces lignes risquent d'être toutes gelées et réécrites en même temps : ce peut être très brutal en terme de saturation disque, de journaux générés, etc. si la base est grosse. Le délai avant le déclenchement du gel automatique dépend de la consommation des numéros de transaction sur l'instance migrée, et varie de quelques semaines à des années.
Des ordres `VACUUM FREEZE` sur les plus grosses tables à des moments calmes permettent d'étaler ces écritures. Si ces ordre sont interrompus, l'essentiel de qu'il a pu geler ne sera plus à re-geler plus tard.
Pour les détails, voir https://dali.bo/m4_html#le-wraparound-1 et https://dali.bo/m5_html#paramétrage-du-freeze-1.

3.9.6 Mise à jour majeure par réplication logique

- Possible entre versions 10 et supérieures
- Bascule très rapide
- Et retour possible

La réplication logique rend possible une migration entre deux instances de version majeure différente avec une indisponibilité très courte. Le principe est de répliquer une base à l'identique vers une instance de version plus récente, alors que la production tourne.

La réplication logique n'est disponible en natif qu'à partir de PostgreSQL version 10, la base à migrer doit donc être en version 10 ou supérieure. (Autrefois, on utilisait des outils de réplication par triggers, plus complexes à manier.) Des clés primaires sur chaque table sont très fortement conseillées, mais pas forcément obligatoires.

L'idée est d'installer la nouvelle version de PostgreSQL normalement, sur le même serveur ou sur un autre serveur. On déclare la réplication de l'ancienne instance vers la nouvelle. Les utilisateurs peuvent continuer à travailler pendant le transfert initial des données. Ils verront au pire une baisse de performances, due à la lecture et à l'envoi des données vers le nouveau serveur. Une fois le transfert initial réalisé, les données modifiées entre-temps sont transférées vers le nouveau serveur. Toute cette opération peut s'étaler sur plusieurs jours.

Une fois les deux serveurs synchronisés, il ne reste plus qu'à déclencher un *switchover* (bascule) ; puis d'attendre que les dernières données soient répliquées, ce qui peut être très rapide ; et enfin de connecter les applications au nouveau serveur. La réplication peut alors être inversée pour garder l'ancienne production synchrone, permettant de rebasculer dessus en cas de problème sans perdre les données modifiées depuis la bascule. Une fois acté que le nouveau serveur donne pleine satisfaction, il suffit de débrancher la réplication logique des deux côtés.

Pour les grosses bases, il existe le même danger d'un gel brutal des lignes comme avec `pg_restore`.

3.9.7 Mise à jour majeure par pg_upgrade

- `pg_upgrade` : fourni avec PostgreSQL
- Prérequis : pas de changement de format des fichiers entre versions
- Nécessite les deux versions sur le même serveur
- Support des serveurs PostgreSQL à migrer :
  - version minimale 9.2 pour pg_upgrade v15
  - version minimale 8.4 sinon

`pg_upgrade` est certainement l'outil le plus rapide pour une mise à jour majeure.

Il profite du fait que les formats des fichiers de données n'évolue pas, ou de manière rétrocompatible, entre deux versions majeures. Il n'est donc pas nécessaire de tout réécrire.

Grossièrement, son fonctionnement est le suivant. Il récupère la déclaration des objets sur l'ancienne instance avec un `pg_dump` du schéma de chaque base et de chaque objet global. Il intègre la déclaration des objets dans la nouvelle instance. Il fait un ensemble de traitement sur les identifiants d'objets et de transactions. Puis, il copie les fichiers de données de l'ancienne instance vers la nouvelle instance. La copie est l'opération la plus longue, mais comme il n'est pas nécessaire de reconstruire les index et de vérifier les contraintes, cette opération est bien plus rapide que la restauration d'une sauvegarde style `pg_dump`. Pour aller encore plus rapidement, il est possible de créer des liens physiques à la place de la copie des fichiers. Ceci fait, la migration est terminée.

En 2010, Stefan Kaltenbrunner et Bruce Momjian avaient mesuré qu'une base de 150 Go mettait 5 heures à être mise à jour avec la méthode historique (sauvegarde/restauration). Elle mettait 44 minutes en mode copie et 42 secondes en mode lien lors de l'utilisation de `pg_upgrade`.

C'est une migration physique : le problème du gel ultérieur des lignes comme avec `pg_restore` ne se pose pas.

Vu ses performances, ce serait certainement l'outil à privilégier. Cependant, c'est un outil très complexe et quelques bugs particulièrement méchants ont terni sa réputation. Notre recommandation est de bien tester la mise à jour avant de le faire en production, et uniquement sur des bases suffisamment volumineuses dont le court délai de migration justifie l'utilisation de cet outil.

Lors du développement de la version 15, les développeurs ont supprimé certaines vieilles parties du code, ce qui le rend à présent incompatible avec des versions très anciennes (de la 8.4 à la 9.1, cette dernière remontant tout de même à 2012).

3.9.8 Mise à jour de l'OS

Si vous migrez aussi l'OS ou déplacez les fichiers d'une instance :

- compatibilité architecture
- compatibilité librairies
  - réindexation parfois nécessaire
  - ex : Debian 10 et glibc 2.28

Un projet de migration PostgreSQL est souvent l'occasion de mettre à jour le système d'exploitation. Vous pouvez également en profiter pour déplacer l'instance sur un autre serveur à l'OS plus récent en copiant (à froid) le PGDATA.

Il faut bien sûr que l'architecture physique (32/64 bits, *big/little indian*) reste la même. Cependant, même entre deux versions de la même distribution, certains composants du système d'exploitation peuvent avoir une influence, à commencer par la `glibc`. Cette librairie fondamentale de Linux définit l'ordre des caractères, ordre utilisé dans les index de champs textes. Une incompatibilité entre deux versions sur ce point oblige donc à reconstruire les index, sous peine d'incohérence avec les fonctions de comparaison sur le nouveau système et de corruption à l'écriture.

Daniel Vérité détaille sur son blog[34] le problème pour les mises à jour entre Debian 9 et 10, à cause de la mise à jour de la `glibc` en version 2.28. (Voir aussi le wiki PostgreSQL[35].) L'utilisation des collations ICU[36] dans les index contourne le problème, mais elles sont encore peu répandues. Dans beaucoup de cas, la collation `C.UTF-8` propre à PostgreSQL (version 17 au moins) peut éliminer facilement tout souci de ce genre.

Ce problème ne touche bien sûr pas les migrations ou les restaurations avec `pg_dump` / `pg_restore` : les données sont alors transmises de manière logique, indépendamment des caractéristiques physiques des instances source et cible, et les index sont systématiquement reconstruits sur la machine cible.

[34] https://blog-postgresql.verite.pro/2018/08/30/glibc-upgrade.html
[35] https://wiki.postgresql.org/wiki/Locale_data_changes
[36] https://blog-postgresql.verite.pro/2018/07/27/icu_ext.html

3.10 CONCLUSION

- L'installation est simple…
- …mais elle doit être soigneusement préparée
- Préférer les paquets officiels
- Attention aux données lors d'une mise à jour !

3.10.1 Pour aller plus loin

- Documentation officielle, chapitre `Installation`
- Documentation Dalibo, pour l'installation sur Windows

Vous pouvez retrouver la documentation en ligne sur https://docs.postgresql.fr/current/installation.html.

La documentation de Dalibo pour l'installation de PostgreSQL sur Windows est disponible sur https://public.dalibo.com/archives/etudes/installer_postgresql_9.0_sous_windows.pdf.

3.10.2 Questions

N'hésitez pas, c'est le moment !

3.11 QUIZ

https://dali.bo/b_quiz

3.12 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/b_solutions.

3.12.1 Installation depuis les paquets binaires du PGDG (Rocky Linux)

But : Installer PostgreSQL à partir des paquets communautaires.
Cette instance servira aux TP suivants.

Pré-installation

Quelle commande permet d'installer les paquets binaires de PostgreSQL ?

Quelle version est packagée ?

Quels paquets devront également être installés ?

Installation

Installer le dépôt.

Désactiver le module d'installation pour la version PostgreSQL de la distribution.

Installer les paquets de PostgreSQL16 : serveur, client, contribs.

Quel est le chemin des binaires ?

Création de la première instance

Créer une première instance avec les outils de la famille Red Hat en activant les sommes de contrôle (*checksums*).

Vérifier ce qui a été fait dans le journal `initdb.log`.

Démarrage

Démarrer l'instance.

> Activer le démarrage de l'instance au démarrage de la machine.

> Où sont les fichiers de données (`PGDATA`), et les traces de l'instance ?

Configuration

> Vérifier la configuration par défaut de PostgreSQL. Est-ce que le serveur écoute sur le réseau ?

> Quel est l'utilisateur sous lequel tourne l'instance ?

Connexion

> En tant que **root**, tenter une connexion avec `psql`.

> En tant que **postgres**, tenter une connexion avec `psql`. Quitter.

> À quelle base se connecte-t-on par défaut ?

> Créer une première base de données et y créer des tables.

3.12.2 Installation à partir des sources (optionnel)

But : Installer PostgreSQL à partir du code source

Note : Pour éviter tout problème lié au positionnement des variables d'environnement dans les exercices suivants, l'installation depuis les sources se fera avec un utilisateur dédié, différent de l'utilisateur utilisé par l'installation depuis les paquets de la distribution.

Outils de compilation

> Installer les outils de compilation suivants, si ce n'est déjà fait.

NB : Ceci a été testé avec PostgreSQL 17 sur Rocky Linux 9 et Debian 12. Des dépendances un peu différentes pourraient apparaître pour d'autres versions.

Sous Rocky Linux, il faudra utiliser `dnf` :

```
sudo dnf -y group install "Development Tools"
sudo dnf -y install perl readline-devel openssl-devel wget bzip2
```

Sous Debian ou Ubuntu :

```
sudo apt install -y build-essential libreadline-dev zlib1g-dev flex bison \
  libxml2-dev libxslt-dev libssl-dev pkg-config
```

> Créer l'utilisateur système **srcpostgres** avec `/opt/pgsql` pour répertoire `HOME`.

> Se connecter en tant que l'utilisateur **srcpostgres**.

Téléchargement

> - Télécharger l'archive des fichiers sources de la dernière version stable depuis postgresql.org.
> - (Alternative : récupérer le dépôt git de la dernière branche stable.)
> - Les placer dans `/opt/pgsql/src`.

Compilation et installation

> L'installation des binaires compilés se fera dans `/opt/pgsql/17/`.
>
> - Configurer en conséquence l'environnement de compilation (`./configure`).
> - Compiler PostgreSQL. Ne pas oublier les contribs.

> Installer les fichiers obtenus.

> Où se trouvent les binaires installés de PostgreSQL ?

Configurer le système

> Ajouter les variables d'environnement `PATH` et `LD_LIBRARY_PATH` au `~srcpostgres/.bash_profile` de l'utilisateur **srcpostgres** pour accéder facilement à ces binaires.

Création d'une instance

> Avec `initdb`, initialiser une instance dans `/opt/pgsql/17/data` en spécifiant **postgres** comme nom de super-utilisateur, et en activant les sommes de contrôle.

> Démarrer l'instance.

> - Tenter une première connexion avec `psql`.
> - Pourquoi cela échoue-t-il ?

> Se connecter en tant qu'utilisateur **postgres**. Ressortir.

> Dans `.bash_profile`, configurer la variable d'environnement `PGUSER` pour se connecter toujours en tant que **postgres**.

Première base

> Créer une première base de donnée nommée `test`.

> Se connecter à la base `test` et créer quelques tables.

Arrêt

> Arrêter cette instance.

3.13 TRAVAUX PRATIQUES (SOLUTIONS)

3.13.1 Installation depuis les paquets binaires du PGDG (Rocky Linux)

Pré-installation

> Quelle commande permet d'installer les paquets binaires de PostgreSQL ?

Le présent TP utilise Rocky Linux en version 8 ou 9. C'est une distribution communautaire qui se veut succéder au projet CentOS, clone de Red Hat, interrompu en 2021[37].

(Une version plus complète, ainsi que l'utilisation de paquets Debian, sont traités dans l'annexe « Installation de PostgreSQL depuis les paquets communautaires ».)

> Quelle version est packagée ?

La dernière version stable de PostgreSQL disponible au moment de la rédaction de ces lignes est la 17.0. Par contre, la dernière version disponible dans les dépôts dépend de votre distribution. C'est la raison pour laquelle **les dépôts du PGDG sont à privilégier**.

> Quels paquets devront également être installés ?

Le paquet `libpq` doit également être installé. Il est aussi nécessaire d'installer les paquets `llvmjit` (pour la compilation à la volée), qui réclame elle-même la présence du dépôt EPEL, mais c'est une fonctionnalité optionnelle qui ne sera pas traitée ici.

Installation

> Installer le dépôt en vous inspirant des consignes sur :
> https://www.postgresql.org/download/linux/redhat
> mais en ajoutant les contribs et les sommes de contrôle.

Préciser :

- PostgreSQL 17
- Red Hat Enterprise, Rocky or Oracle version 8 (ou 9 selon le cas)
- x86_64

Nous allons reprendre ligne à ligne ce script et le compléter.

Se connecter avec l'utilisateur **root** sur la machine de formation, et recopier le script proposé par le guide. Dans la commande ci-dessous, les deux lignes **doivent être copiées et collées ensemble**.

```
# Rocky Linux 8
dnf install -y https://download.postgresql.org\
/pub/repos/yum/reporpms/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# Rocky Linux 9
dnf install -y https://download.postgresql.org\
/pub/repos/yum/reporpms/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

[37] https://blog.centos.org/2020/12/future-is-centos-stream

> Désactiver le module d'installation pour la version PostgreSQL de la distribution.

Cette opération est nécessaire pour Rocky Linux 8 ou 9.

```
dnf -qy module disable postgresql
```

> Installer les paquets de PostgreSQL17 : serveur, client, contribs.

```
dnf install -y postgresql17-server postgresql17-contrib
# optionnel
dnf install -y postgresql17-llvmjit
```

Il s'agit respectivement

- des binaires du serveur ;
- des « contribs » et extensions optionnelles (mais chaudement conseillées) ;
- et du paquet nécessaire à la compilation à la volée (JIT).

Le paquet `postgresql17` (outils client) fait partie des dépendances et est installé automatiquement.

> Quel est le chemin des binaires ?

Ils se trouvent dans `/usr/pgsql-17/bin/` (chemin propre à ces paquets) :

```
ls -1 /usr/pgsql-17/bin/
```

```
clusterdb
createdb
;
postgres
postgresql-17-check-db-dir
postgresql-17-setup
psql
reindexdb
vacuumdb
vacuumlo
```

Noter qu'il existe des liens dans `/usr/bin` pointant vers la version la plus récente des outils en cas d'installation de plusieurs versions :

```
which psql
```

```
/usr/bin/psql
```

```
file /usr/bin/psql
```

```
/usr/bin/psql: symbolic link to /etc/alternatives/pgsql-psql
```

```
file /etc/alternatives/pgsql-psql
```

```
/etc/alternatives/pgsql-psql: symbolic link to /usr/pgsql-17/bin/psql
```

Création de la première instance

> Créer une première instance avec les outils de la famille Red Hat en activant les sommes de contrôle (*checksums*).

La création d'une instance passe par un outil spécifique à ces paquets.

> Cet outil doit être appelé en tant que **root** (et non **postgres**).

Optionnellement, on peut ajouter des paramètres d'initialisation à cette étape. La mise en place des sommes de contrôle est généralement conseillée pour être averti de toute corruption des fichiers.

Toujours en temps que **root** :

```
export PGSETUP_INITDB_OPTIONS="--data-checksums"
/usr/pgsql-17/bin/postgresql-17-setup initdb
```

```
Initializing database ... OK
```

L'`export` est nécessaire pour activer les sommes de contrôle.

> Vérifier ce qui a été fait dans le journal `initdb.log`.

La sortie de la commande précédente est redirigée vers le fichier `initdb.log` situé dans le répertoire qui contient celui de la base (`PGDATA`). Il est possible d'y vérifier l'ensemble des étapes réalisées, notamment l'activation des sommes de contrôle.

```
$ cat /var/lib/pgsql/17/initdb.log
```

```
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

fixing permissions on existing directory /var/lib/pgsql/17/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default "max_connections" ... 100
selecting default "shared_buffers" ... 128MB
selecting default time zone ... UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

Success. You can now start the database server using:

    /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/17/data/ -l logfile start
```

Ne pas tenir compte de la dernière ligne, qui est une suggestion qui ne tient pas compte des outils prévus pour cet OS.

Démarrage

> Démarrer l'instance.

Attention, si vous avez créé une instance, par exemple à partir des sources, elle doit impérativement être arrêtée pour pouvoir démarrer la nouvelle instance ! Elles ne peuvent pas être démarrées en même temps, sauf à modifier le port dans la configuration de l'une d'entre elles.

En tant que **root** :

```
systemctl start postgresql-17
```

Si aucune erreur ne s'affiche, tout va bien à priori.

Pour connaître l'état de l'instance :

```
systemctl status postgresql-17
```

```
● postgresql-17.service - PostgreSQL 17 database server
     Loaded: loaded (/usr/lib/systemd/system/postgresql-17.service; disabled;
 ↪  preset: disabled)
     Active: active (running) since Tue 2024-10-15 18:44:08 UTC; 6s ago
       Docs: https://www.postgresql.org/docs/17/static/
    Process: 72901 ExecStartPre=/usr/pgsql-17/bin/postgresql-17-check-db-dir
 ↪  ${PGDATA} (code=exited, status=0/SUCCESS)
   Main PID: 72906 (postgres)
      Tasks: 7 (limit: 2676)
     Memory: 17.7M
        CPU: 37ms
     CGroup: /system.slice/postgresql-17.service
             ├─72906 /usr/pgsql-17/bin/postgres -D /var/lib/pgsql/17/data/
             ├─72907 "postgres: logger "
             ├─72908 "postgres: checkpointer "
             ├─72909 "postgres: background writer "
             ├─72911 "postgres: walwriter "
             ├─72912 "postgres: autovacuum launcher "
             └─72913 "postgres: logical replication launcher "

Oct 15 18:44:08 rocky9 systemd[1]: Starting PostgreSQL 17 database server...
Oct 15 18:44:08 rocky9 postgres[72906]: 2024-10-15 18:44:08.144 UTC [72906] LOG:
 ↪  redirecting log output to logging collecto>
Oct 15 18:44:08 rocky9 postgres[72906]: 2024-10-15 18:44:08.144 UTC [72906] HINT:
 ↪  Future log output will appear in director>
Oct 15 18:44:08 rocky9 systemd[1]: Started PostgreSQL 17 database server.
```

> Activer le démarrage de l'instance au démarrage de la machine.

Le packaging Red Hat ne prévoie pas l'activation du service au boot, il faut le demander explicitement :

```
systemctl enable postgresql-17
```

```
Created symlink /etc/systemd/system/multi-user.target.wants/postgresql-17.service →
↪  /usr/lib/systemd/system/postgresql-17.service.
```

> Où sont les fichiers de données (`PGDATA`), et les traces de l'instance ?

Les données et fichiers de configuration sont dans `/var/lib/pgsql/17/data/`.

```
ls -1 /var/lib/pgsql/17/data/
```

```
base
current_logfiles
global
log
pg_commit_ts
pg_dynshmem
pg_hba.conf
pg_ident.conf
pg_logical
pg_multixact
pg_notify
pg_replslot
pg_serial
pg_snapshots
pg_stat
pg_stat_tmp
pg_subtrans
pg_tblspc
pg_twophase
PG_VERSION
pg_wal
pg_xact
postgresql.auto.conf
postgresql.conf
postmaster.opts
postmaster.pid
```

Les traces sont par défaut dans le sous-répertoire `log/` du PGDATA.

```
ls -l /var/lib/pgsql/17/data/log/
```

```
total 4
-rw-------. 1 postgres postgres 1092 Oct 15 18:49 postgresql-Tue.log
```

NB : Dans les paquets RPM, le nom exact du fichier dépend du jour de la semaine.

Configuration

> Vérifier la configuration par défaut de PostgreSQL. Est-ce que le serveur écoute sur le réseau ?

Il est possible de vérifier dans le fichier `postgresql.conf` que par défaut, le serveur écoute uniquement l'interface réseau `localhost` (la valeur est commentée car c'est celle par défaut) :

```
grep listen_addresses /var/lib/pgsql/17/data/postgresql.conf
```

```
#listen_addresses = 'localhost'         # what IP address(es) to listen on;
```

Il faudra donc modifier ainsi et rédémarrer pour que des utilisateurs puissent se connecter depuis d'autres machines :

```
listen_addresses = '*'          # what IP address(es) to listen on;
```

```
systemctl restart postgresql-17
```

Il est aussi possible de vérifier au niveau système en utilisant la commande `netstat` (qui nécessite l'installation du paquet `net-tools`) :

```
netstat -anp|grep -E '(Active|Proto|postgres)'
```

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address State   PID/Program name
tcp        0      0 0.0.0.0:5432    0.0.0.0:*       LISTEN  73049/postgres
tcp6       0      0 :::5432         :::*            LISTEN  73049/postgres
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags    Type    State      I-Node PID/Program name     Path
unix  2      [ ACC ]  STREAM  LISTENING  87790  73049/postgres
 ↪  /run/postgresql/.s.PGSQL.5432
unix  2      [ ACC ]  STREAM  LISTENING  87791  73049/postgres      /tmp/.s.PGSQL.5432
unix  3      [ ]      STREAM  CONNECTED  88816  73050/postgres: log
…
```

(La présence de lignes `tcp6` dépend de la configuration de la machine.)

On notera que la *socket* écoute à deux endroits, dans `/run/postgresql/` et dans `/tmp`. C'est un paramétrage par défaut lié aux paquets RPM.

> Quel est l'utilisateur sous lequel tourne l'instance ?

C'est l'utilisateur nommé **postgres** :

```
ps -U postgres -f -o pid,user,cmd
```

```
    PID USER     CMD
    PID USER     CMD
  73049 postgres /usr/pgsql-17/bin/postgres -D /var/lib/pgsql/17/data/
  73050 postgres  \_ postgres: logger
  73051 postgres  \_ postgres: checkpointer
  73052 postgres  \_ postgres: background writer
  73054 postgres  \_ postgres: walwriter
  73055 postgres  \_ postgres: autovacuum launcher
  73056 postgres  \_ postgres: logical replication launcher
```

Il possède aussi le PGDATA :

```
ls -l /var/lib/pgsql/17/
```

```
total 8
drwx------.  2 postgres postgres    6 Sep 26 20:10 backups
drwx------. 20 postgres postgres 4096 Oct 15 19:29 data
-rw-------.  1 postgres postgres  914 Oct 15 18:41 initdb.log
```

postgres est le nom traditionnel sur la plupart des distributions, mais il n'est pas obligatoire (par exemple, le TP par compilation utilise un autre utilisateur).

Connexion

> En tant que **root**, tenter une connexion avec `psql`.

```
# psql
```

```
psql: error: connection to server on socket "/run/postgresql/.s.PGSQL.5432" failed:
↪  FATAL:  role "root" does not exist
```

Cela échoue car `psql` tente de se connecter avec l'utilisateur système en cours, soit **root**. Ça ne marchera pas mieux cependant en essayant de se connecter avec l'utilisateur **postgres** :

```
psql -U postgres
```

```
psql: error: connection to server on socket "/run/postgresql/.s.PGSQL.5432" failed:
↪  FATAL:  Peer authentication failed for user "postgres"
```

En effet, le `pg_hba.conf` est configuré de telle manière que l'utilisateur de PostgreSQL et celui du système doivent porter le même nom (connexion `peer` ici).

> En tant que **postgres**, tenter une connexion avec `psql`. Quitter.

```
sudo -iu postgres psql
```

```
psql (17.0)
Type "help" for help.

postgres=# exit
```

La connexion fonctionne donc indirectement depuis tout utilisateur pouvant effectuer un `sudo`.

> À quelle base se connecte-t-on par défaut ?

```
sudo -iu postgres psql
```

```
psql (17.0)
Type "help" for help.

postgres=# \conninfo
You are connected to database "postgres" as user "postgres" via socket in
↪  "/run/postgresql" at port "5432".
postgres=#
```

Là encore, la présence d'une base nommée `postgres` est une tradition et non une obligation.

Première base

> Créer une première base de données et y créer des tables.

```
sudo -iu postgres psql
```

```
postgres=# CREATE DATABASE test ;
CREATE DATABASE
```

Alternativement :

```
sudo -iu postgres createdb test
```

Se connecter explicitement à la bonne base :

```
sudo -iu postgres psql -d test
```

```
test=# CREATE TABLE mapremieretable (x int);
CREATE TABLE

test=# \d+

                           Liste des relations
 Schéma |       Nom       | Type  | Propriétaire | Taille  | Description
--------+-----------------+-------+--------------+---------+-------------
 public | mapremieretable | table | postgres     | 0 bytes |
```

3.13.2 Installation à partir des sources (optionnel)

Outils de compilation

> Installer les outils de compilation suivants, si ce n'est déjà fait.

Ces actions doivent être effectuées en tant qu'utilisateur privilégié (soit directement en tant que **root**, soit en utilisant la commande `sudo`).

NB : Ceci a été testé avec PostgreSQL 17 sur Rocky Linux 9 et Debian 12. Des dépendances un peu différentes pourraient apparaître pour d'autres versions.

Sous Rocky Linux, il faudra utiliser `dnf` :

```
sudo dnf -y group install "Development Tools"
sudo dnf -y install perl readline-devel openssl-devel wget bzip2
```

Sous Debian ou Ubuntu :

```
sudo apt install -y build-essential libreadline-dev zlib1g-dev flex bison \
  libxml2-dev libxslt-dev libssl-dev pkg-config
```

Une fois ces outils installés, tout ce qui suit devrait fonctionner sur toute version de Linux.

> Créer l'utilisateur système **srcpostgres** avec `/opt/pgsql` pour répertoire `HOME`.

```
sudo useradd --home-dir /opt/pgsql --system --create-home srcpostgres
sudo usermod --shell /bin/bash srcpostgres
```

> Se connecter en tant que l'utilisateur **srcpostgres**.

Se connecter en tant qu'utilisateur **srcpostgres** :

```
sudo su - srcpostgres
```

Téléchargement

> - Télécharger l'archive des fichiers sources de la dernière version stable depuis postgresql.org.
> - (Alternative : récupérer la dernière version stable dans le dépôt git.)
> - Les placer dans `/opt/pgsql/src`.

En tant qu'utilisateur **srcpostgres**, créer un répertoire dédié aux sources :

```
mkdir ~srcpostgres/src
cd ~/src
```

Aller sur https://postgresql.org[38], cliquer *Download* et récupérer le lien vers l'archive des fichiers sources de la dernière version stable (PostgreSQL 17.0 au moment où ceci est écrit). Il est possible de le faire en ligne de commande :

[38] https://www.postgresql.org/ftp/source/

```
wget https://ftp.postgresql.org/pub/source/v17.0/postgresql-17.0.tar.bz2
```

Il faut décompresser l'archive :

```
tar xjvf postgresql-17.0.tar.bz2
cd postgresql-17.0
```

Alternativement, le dépôt git se récupère ainsi :

```
git clone --branch REL_17_0 --depth 1  \
https://git.postgresql.org/git/postgresql.git postgresql-17.0
cd postgresql-17.0
```

ou de manière moins économe :

```
git clone REL_17_0 https://git.postgresql.org/git/postgresql.git
cd postgresql
git checkout REL_17_0
cd postgresql
```

(Au besoin `git tag` liste les versions disponibles.)

Compilation et installation

> L'installation des binaires compilés se fera dans `/opt/pgsql/17/`.
>
> - Configurer en conséquence l'environnement de compilation (`./configure`).
> - Compiler PostgreSQL. Ne pas oublier les contribs.

Configuration :

```
./configure --prefix /opt/pgsql/17
```

```
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking which template to use... linux
checking whether NLS is wanted... no
checking for default port number... 5432
…
…
config.status: linking src/include/port/linux.h to src/include/pg_config_os.h
config.status: linking src/makefiles/Makefile.linux to src/Makefile.port
```

Des fichiers sont générés, notamment le *Makefile*.

La compilation se lance de manière classique. Elle peut prendre un certain temps sur les machines un peu anciennes :

```
make    # make -j3 selon le nombre de coeurs
```

```
make -C ./src/backend generated-headers
make[1]: Entering directory '/opt/pgsql/postgresql-17.0/src/backend'
make -C ../include/catalog generated-headers
make[2]: Entering directory '/opt/pgsql/postgresql-17.0/src/include/catalog'
…
make[1]: Leaving directory '/opt/pgsql/postgresql-17.0/src'
```

```
make -C config all
make[1]: Entering directory '/opt/pgsql/postgresql-17.0/config'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/opt/pgsql/postgresql-17.0/config'
```

```
cd contrib/
make
```

```
make -C ../src/backend generated-headers
make[1]: Entering directory '/opt/pgsql/postgresql-17.0/src/backend'
make -C ../include/catalog generated-headers
make[2]: Entering directory '/opt/pgsql/postgresql-17.0/src/include/catalog'
…
make[2]: Entering directory '/opt/pgsql/postgresql-17.0/src/backend/nodes'
make[2]: Nothing to be done for 'generated-header-symlinks'.
make[2]: Leaving directory '/opt/pgsql/postgresql-17.0/src/backend/nodes'
```

> Installer les fichiers obtenus.

L'installation peut se faire en tant que **srcpostgres** (et non **root**) car nous avons défini comme cible le répertoire `/opt/pgsql/17/` qui lui appartient :

```
cd ~srcpostgres/postgresql-17.0
make install
cd contrib
make install
```

Dans ce TP, nous nous sommes attachés à changer le moins possible d'utilisateur système. Il se peut que vous ayez à installer les fichiers obtenus en tant qu'utilisateur **root** dans d'autres environnements en fonction de la politique de sécurité adoptée.

> Où se trouvent les binaires installés de PostgreSQL ?

Les binaires installés sont situés dans le répertoire `/opt/pgsql/17/bin`.

```
ls -1 /opt/pgsql/17/bin
```

```
clusterdb
createdb
createuser
…
pg_verifybackup
pg_waldump
pg_walsummary
pgbench
postgres
psql
reindexdb
vacuumdb
vacuumlo
```

Configurer le système

> Ajouter les variables d'environnement `PATH` et `LD_LIBRARY_PATH` au `~srcpostgres/.bash_profile` de l'utilisateur **srcpostgres** pour accéder facilement à ces binaires.

Ajouter les lignes suivantes à la fin du fichier `~srcpostgres/.bash_profile` (ce fichier peut ne pas exister préalablement, et un autre fichier peut être nécessaire selon l'environnement utilisé) :

```
export PGDATA=/opt/pgsql/17/data
export PATH=/opt/pgsql/17/bin:$PATH
export LD_LIBRARY_PATH=/opt/pgsql/17/lib:$LD_LIBRARY_PATH
```

Il faut ensuite recharger le fichier à l'aide de la commande suivante (ne pas oublier le point et l'espace au début de la commande) ; ou se déconnecter et se reconnecter.

```
. ~srcpostgres/.bash_profile
```

Vérifier que les chemins sont bons :

```
which psql
```

```
~/17/bin/psql
```

Création d'une instance

> Avec `initdb`, initialiser une instance dans `/opt/pgsql/17/data` en spécifiant **postgres** comme nom de super-utilisateur, et en activant les sommes de contrôle.

```
$ initdb -D $PGDATA -U postgres --data-checksums
```

```
The files belonging to this database system will be owned by user "srcpostgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

creating directory /opt/pgsql/17/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default "max_connections" ... 100
selecting default "shared_buffers" ... 128MB
selecting default time zone ... UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or
↪  --auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /opt/pgsql/17/data -l logfile start
```

> Démarrer l'instance.

Attention : s'il y a déjà une autre instance sur cette machine et qu'elle est démarrée, le port 5432 est occupé et votre nouvelle instance ne pourra pas fonctionner. Arrêter l'autre instance ou modifier le port 5432 d'une des instances.

```
pg_ctl -D $PGDATA -l $PGDATA/server.log start
```

```
waiting for server to start.... done
server started
```

```
cat $PGDATA/server.log
```

```
2024-10-15 17:56:13.354 UTC [69286] LOG:  starting PostgreSQL 17.0 on
 ↪  x86_64-pc-linux-gnu, compiled by gcc (GCC) 11.4.1 20231218 (Red Hat 11.4.1-3),
 ↪  64-bit
2024-10-15 17:56:13.354 UTC [69286] LOG:  listening on IPv4 address "127.0.0.1",
 ↪  port 5432
2024-10-15 17:56:13.354 UTC [69286] LOG:  could not bind IPv6 address "::1": Cannot
 ↪  assign requested address
2024-10-15 17:56:13.358 UTC [69286] LOG:  listening on Unix socket
 ↪  "/tmp/.s.PGSQL.5432"
2024-10-15 17:56:13.362 UTC [69289] LOG:  database system was shut down at
 ↪  2024-10-15 17:54:39 UTC
2024-10-15 17:56:13.364 UTC [69286] LOG:  database system is ready to accept
 ↪  connections
```

> - Tenter une première connexion avec `psql`.
> - Pourquoi cela échoue-t-il ?

```
psql
```

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
 ↪  role "srcpostgres" does not exist
```

Par défaut, `psql` demande à se connecter avec un nom d'utilisateur (au sens PostgreSQL) identique à l'utilisateur système en cours, mais la base de données ne connaît pas d'utilisateur **srcpostgres**. Par défaut, elle ne connaît que **postgres**.

> Se connecter en tant qu'utilisateur **postgres**. Ressortir.

```
psql -U postgres
```

```
psql (17.0)
Type "help" for help.
```

```
postgres=# exit
```

Noter que la connexion fonctionne parce que le `pg_hba.conf` livré avec les sources est par défaut très laxiste (méthode `trust` en local et *via* `localhost` !). (Il y a d'ailleurs eu un avertissement lors de la création de la base.)

> Dans `.bash_profile`, configurer la variable d'environnement `PGUSER` pour se connecter toujours en tant que **postgres**. Retester la connextion directe avec `psql`.

Ajouter ceci à la fin du fichier `~srcpostgres/.bash_profile` :

```
export PGUSER=postgres
```

Et recharger le fichier à l'aide de la commande suivante (ne pas oublier le point et l'espace au début de la commande) :

```
. ~/.bash_profile
```

La connexion doit fonctionner sur le champ :

```
psql
```

```
psql (17.0)
Type "help" for help.

postgres=# \conninfo
You are connected to database "postgres" as user "postgres" via socket in "/tmp" at
 ↪  port "5432"
postgres=#
\q
```

Première base

> Créer une première base de donnée nommée `test`.

En ligne de commande shell :

```
createdb --echo test
```

```
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE test;
```

Alternativement, depuis `psql` :

```
postgres=# CREATE DATABASE test ;
```

```
CREATE DATABASE
```

> Se connecter à la base `test` et créer quelques tables.

```
psql -d test
```

```
test=# CREATE TABLE premieretable (x int) ;
CREATE TABLE
```

Arrêt

> Arrêter cette instance.

```
$ pg_ctl stop
```

```
waiting for server to shut down.... done
server stopped
```

```
tail $PGDATA/server.log
```

```
2024-10-15 18:00:33.830 UTC [69286] LOG:  received fast shutdown request
2024-10-15 18:00:33.835 UTC [69286] LOG:  aborting any active transactions
2024-10-15 18:00:33.836 UTC [69286] LOG:  background worker "logical replication
 ↪  launcher" (PID 69292) exited with exit code 1
2024-10-15 18:00:33.837 UTC [69287] LOG:  shutting down
2024-10-15 18:00:33.840 UTC [69287] LOG:  checkpoint starting: shutdown immediate
2024-10-15 18:00:33.918 UTC [69287] LOG:  checkpoint complete: wrote 975 buffers
 ↪  (6.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.036 s, sync=0.032 s,
 ↪  total=0.081 s; sync files=311, longest=0.015 s, average=0.001 s; distance=4585
 ↪  kB, estimate=4585 kB; lsn=0/1BA9900, redo lsn=0/1BA9900
2024-10-15 18:00:33.924 UTC [69286] LOG:  database system is shut down
```

3.14 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

3.14.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 17) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

```
# dnf -qy module disable postgresql
```

Installation de PostgreSQL 17 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql17-server postgresql17-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql17-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb
# cat /var/lib/pgsql/17/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/17/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-17/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/17/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-17
# systemctl stop postgresql-17
# systemctl status postgresql-17
# systemctl reload postgresql-17
# systemctl restart postgresql-17
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-17
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/17/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-17.service \
        /etc/systemd/system/postgresql-17-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/17/infocentre
```

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb postgresql-17-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/17/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-17-infocentre
# systemctl [enable|disable] postgresql-17-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

3.14.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence : https://apt.postgresql.org/

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 17 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-17 postgresql-client-17
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/17/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/17/main`
Fichiers de configuration	dans `/etc/postgresql/17/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 17 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/17/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 17 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 17 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 17 infocentre \
  --port=12345 \
  --datadir=/PGDATA/17/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/17/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/17/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 17 infocentre start
```

3.14.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (17.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (17.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- Pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

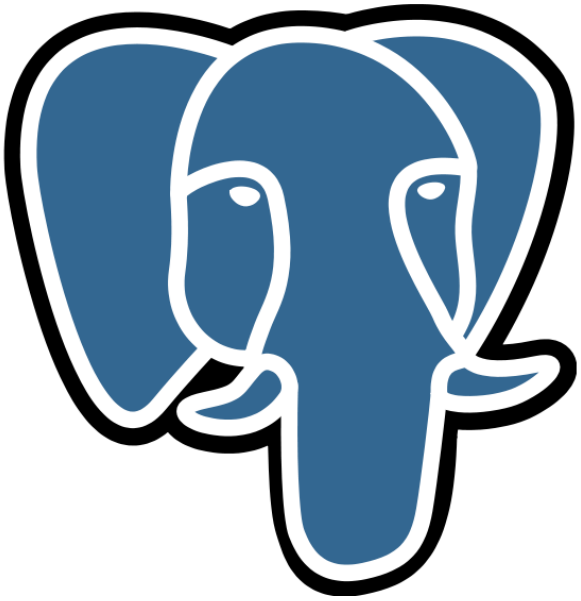
- en cas de problème, consulter les traces (dans `/var/lib/pgsql/17/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-17
```

```
root:~# pg_ctlcluster 17 main reload
```

```
postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

4/ Outils graphiques et console



4.1 PRÉAMBULE

Les outils graphiques et console :

- les outils graphiques d'administration
- la console
- les outils de contrôle de l'activité
- les outils DDL
- les outils de maintenance

Ce module nous permet d'approcher le travail au quotidien du DBA et de l'utilisateur de la base de données. Nous verrons les différents outils disponibles, en premier lieu la console texte, `psql`.

4.1.1 Plan

- Outils en ligne de commande de PostgreSQL
- Réaliser des scripts
- Outils graphiques

4.2 OUTILS CONSOLE DE POSTGRESQL

- Plusieurs outils PostgreSQL en ligne de commande existent
  - une console interactive
  - des outils de maintenance
  - des outils de sauvegardes/restauration
  - des outils de gestion des bases

Les outils console de PostgreSQL que nous allons voir sont fournis avec la distribution de PostgreSQL. Ils permettent de tout faire : exécuter des requêtes manuelles, maintenir l'instance, sauvegarder et restaurer les bases.

4.2.1 Outils : Gestion des bases

- `createdb` : ajouter une nouvelle base de données
- `createuser` : ajouter un nouveau compte utilisateur
- `dropdb` : supprimer une base de données
- `dropuser` : supprimer un compte utilisateur

Chacune de ces commandes est un « alias », un raccourci qui permet d'exécuter certaines commandes SQL directement depuis le shell sans se connecter explicitement au serveur. L'option `--echo` permet de voir les ordres SQL envoyés.

Par exemple, la commande système `dropdb` est équivalente à la commande SQL `DROP DATABASE`. L'outil `dropdb` se connecte à la base de données nommée `postgres` et exécute l'ordre SQL et enfin se déconnecte.

La création d'une base se fait en utilisant l'outil `createdb` et en lui indiquant le nom de la nouvelle base, de préférence avec un utilisateur dédié. `createuser` crée ce que l'on appelle un « rôle », et appelle `CREATE ROLE` en SQL. Nous verrons plus loin les droits de connexion, de superutilisateur, etc.

Une création de base depuis le shell peut donc ressembler à ceci :

```
$ createdb --echo --owner erpuser  erp_prod
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE erp_prod OWNER erpuser;
```

Alors qu'une création de rôle peut ressembler à ceci :

```
$ createuser --echo --login --no-superuser erpuser
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE erpuser NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Et si le `pg_hba.conf` le permet :

```
$ psql -U erpuser erp_prod  < script_installation.sql
```

4.2.2 Outils : Sauvegarde / Restauration

- Sauvegarde logique, pour une instance
  - `pg_dumpall` : sauvegarder l'instance PostgreSQL
- Sauvegarde logique, pour une base de données
  - `pg_dump` : sauvegarder une base de données
  - `pg_restore` : restaurer une base de données PostgreSQL
- Sauvegarde physique :
  - `pg_basebackup`
  - avec : `pg_verifybackup` et `pg_combinebackup` (v17)

Ces commandes sont essentielles pour assurer la sécurité des données du serveur.

Comme son nom l'indique, `pg_dumpall` sauvegarde l'instance complète, autrement dit toutes les bases mais aussi les objets globaux. À partir de la version 12, il est cependant possible d'exclure une ou plusieurs bases de cette sauvegarde.

Pour ne sauvegarder qu'une seule base, il est préférable de passer par l'outil `pg_dump`, qui possède plus d'options. Il faut évidemment lui fournir le nom de la base à sauvegarder. Pour sauvegarder notre base b1, il suffit de lancer la commande suivante :

```
$ pg_dump -f b1.sql b1
```

Pour la restauration d'une sauvegarde, l'outil habituel est `pg_restore`. `psql` est utilisé pour la restauration d'une sauvegarde faite en mode texte (script SQL).

Ces deux outils réalisent des sauvegardes logiques, donc au niveau des objets logiques (tables, index, etc).

La sauvegarde physique (donc au niveau des fichiers) à chaud est possible avec `pg_basebackup`, qui copie un serveur en fonctionnement, journaux de transaction inclus. Son fonctionnement est nettement plus complexe qu'un simple `pg_dump`. `pg_basebackup` est utilisé par les outils de sauvegarde

PITR, et pour créer des serveurs secondaires. `pg_verifybackup` permet de vérifier une sauvegarde réalisée avec `pg_basebackup`. À partir de PostgreSQL 17, `pg_basebackup` sait effectuer des sauvegardes incrémentales, à recombiner avec `pg_combinebackup`.

4.2.3 Outils : Maintenance

- Maintenance des bases
  - `vacuumdb` : récupérer l'espace inutilisé, statistiques
  - `clusterdb` : réorganiser une table en fonction d'un index
  - `reindexdb` : réindexer

`reindexdb`, là encore, est un alias lançant des ordres `REINDEX`. Une réindexation périodique des index peut être utile. Par exemple, pour lancer une réindexation de la base **b1** en affichant la commande exécutée :

```
$ reindexdb --echo --concurrently 11
SELECT pg_catalog.set_config('search_path', '', false);
REINDEX DATABASE CONCURRENTLY b1;
WARNING:  cannot reindex system catalogs concurrently, skipping all
```

`vacuumdb` permet d'exécuter les différentes variantes du `VACUUM` (`FULL`, `ANALYZE`, `FREEZE`…) depuis le shell, principalement le nettoyage des lignes mortes, la mise à jour des statistiques sur les données, et la reconstruction de tables. L'usage est ponctuel, le démon autovacuum s'occupant de cela en temps normal.

`clusterdb` lance un ordre `CLUSTER`, soit une reconstruction de la table avec tri selon un index. L'usage est très spécifique.

Rappelons que ces opérations posent des verrous qui peuvent être très gênants sur une base active.

4.2.4 Outils : Maintenance de l'instance

- `initdb` : création d'instance
- `pg_ctl` : lancer, arrêter, relancer, promouvoir l'instance
- `pg_upgrade` : migrations majeures
- `pg_config`, `pg_controldata` : configuration

Ces outils sont rarement utilisés directement, car on passe généralement par les outils du système d'exploitation et ceux fournis par les paquets, qui les utilisent. Ils peuvent toutefois servir et il faut les connaître.

`initdb` crée une instance, c'est-à-dire crée tous les fichiers nécessaires dans le répertoire indiqué (`PGDATA`). Les options permettent d'affecter certains paramètres par défaut. La plus importante (car on ne peut corriger plus tard qu'à condition que l'instance soit arrêtée, donc en arrêt de production) est l'option `--data-checksums` activant les sommes de contrôle, dont l'activation est généralement conseillée.

`pg_ctl` est généralement utilisé pour démarrer/arrêter une instance, pour recharger les fichiers de configuration après modification, ou pour promouvoir une instance secondaire en primaire. Toutes les actions possibles sont documentées ici[1].

`pg_upgrade` est utilisée pour convertir une instance existante lors d'une migration entre versions majeures. La version minimale supportée est la 8.4, sauf à partir de pg_upgrade 15 (version 9.2 dans ce cas).

`pg_config` fournit des informations techniques sur les binaires installés (chemins notamment).

`pg_controldata` fournit des informations techniques de base sur une instance.

4.2.5 Autres outils en ligne de commande

- `pgbench` pour des tests
- Outils liés à la réplication/sauvegarde physique, aux tests, analyse…

`pgbench` est l'outil de base pour tester la charge et l'influence de paramètres. Créez les tables de travail avec l'option `-i`, fixez la volumétrie avec `-s`, et lancez `pgbench` en précisant le nombre de clients, de transactions… L'outil vous calcule le nombre de transactions par secondes et diverses informations statistiques. Les requêtes utilisées sont basiques mais vous pouvez fournir les vôtres.

D'autres outils sont liés à l'archivage (`pg_receivewal`) et/ou à la réplication par *log shipping* (`pg_archivecleanup`) ou logique (`pg_recvlogical`), au sauvetage d'instances secondaires (`pg_rewind`), à la vérification de la disponibilité (`pg_isready`), à des tests de la configuration matérielle (`pg_test_fsync`, `pg_test_timing`), ou d'intégrité (`pg_checksums`), à l'analyse des journaux de transactions (`pg_waldump`).

[1] https://www.postgresql.org/docs/current/app-pg-ctl.html

4.3 CHAÎNES DE CONNEXION

Pour se connecter à une base :

- paramètres propres aux outils
- via la libpq
  - variables d'environnement
  - par chaînes clés/valeur
  - par chaînes URI
  - idem en Python,PHP,Perl
- JDBC/.NET/ODBC ont des syntaxes spécifiques

Les types de connexion connus de PostgreSQL et de sa librairie cliente (libpq) sont, au choix, les paramètres explicites (assez variables suivants les outils), les variables d'environnement, les chaînes clés/valeur, et les URI (`postgresql://…`).

Les pilotes PHP[2]…), Perl[3], ou Python[4] utilisent la libpq, comme bien sûr les outils du projet PostgreSQL, et ce qui suit est directement utilisable.

Nous ne traiterons pas ici des syntaxes propres aux outils n'utilisant pas la libpq, comme les pilotes JDBC[5] (dont les URI sont proches mais différente de celles décrites plus bas) ou .NET[6], ou encore les différents pilotes ODBC (comme psqlODBC[7], très limité).

4.3.1 Paramètres

Outils habituels, et très souvent :

```
$ psql -h serveur -d mabase -U nom -p 5432
```

Option	Variable	Valeur par défaut
`-h` HÔTE	$PGHOST	`/tmp`, `/var/run/postgresql`

[2] https://www.php.net/manual/en/function.pg-connect.php
[3] https://metacpan.org/pod/DBD::Pg#data_sources
[4] https://www.psycopg.org/docs/module.html#psycopg2.connect
[5] https://jdbc.postgresql.org/documentation/head/connect.html
[6] https://www.npgsql.org/doc/connection-string-parameters.html
[7] https://odbc.postgresql.org/

Option	Variable	Valeur par défaut
`-p` PORT	$PGPORT	`5432`
`-U` NOM	$PGUSER	nom de l'utilisateur OS
`-d` base	$PGDATABASE	nom de l'utilisateur PG
	$PGOPTIONS	options de connexions

Les options de connexion permettent d'indiquer comment trouver l'instance (serveur, port), puis d'indiquer l'utilisateur et la base de données concernés parmi les différentes de l'instance. Ces deux derniers champs doivent passer le filtre du `pg_hba.conf` du serveur pour que la connexion réussisse.

Lorsque l'une de ces options n'est pas précisée, la bibliothèque cliente PostgreSQL cherche une variable d'environnement correspondante et prend sa valeur. Si elle ne trouve pas de variable, elle se rabat sur une valeur par défaut.

Les paramètres et variables d'environnement qui suivent sont utilisés par les outils du projet, et de nombreux autres outils de la communauté.

La problématique du mot de passe est laissée de côté pour le moment.

Hôte :

Le paramètre `-h <hôte>` ou la variable d'environnement `$PGHOST` permettent de préciser le nom ou l'adresse IP de la machine qui héberge l'instance.

Sans précision, sur Unix, le client se connecte sur la socket Unix, généralement dans `/var/run/postgresql` (défaut sur Debian et Red Hat) ou `/tmp` (défaut de la version compilée). Le réseau n'est alors pas utilisé, et il y a donc une différence entre `-h localhost` (via ::1 ou 127.0.0.1 donc) et `-h /var/run/postgresql` (défaut), ce qui peut avoir un résultat différent selon la configuration du `pg_hba.conf`. Par défaut, l'accès par le réseau exige un mot de passe.

Sous Windows, le comportement par défaut est de se connecter à **localhost**.

Serveur :

`-p <port>` ou `$PGPORT` permettent de préciser le port sur lequel l'instance écoute les connexions entrantes. Sans indication, le port par défaut est le 5432.

Utilisateur :

`-U <nom>` ou `$PGUSER` permettent de préciser le nom de l'utilisateur, connu de PostgreSQL, qui doit avoir été créé préalablement sur l'instance.

Sans indication, le nom d'utilisateur PostgreSQL est le nom de l'utilisateur système connecté.

Base de données :

`-d base` ou `$PGDATABASE` permettent de préciser le nom de la base de données utilisée pour la connexion.

Sans précision, le nom de la base de données de connexion sera celui de l'utilisateur PostgreSQL (qui peut découler de l'utilisateur connecté au système).

Exemples :

- Chaîne de connexion classique, implicitement au port 5432 en local sur le serveur :

```
$ psql -U jeanpierre -d comptabilite
```

- Connexion à un serveur distant pour une sauvegarde :

```
$ pg_dump -h serveur3 -p 5435 -U postgres -d basecritique -f fichier.dump
```

- Connexion sans paramètre via l'utilisateur système **postgres**, et donc implicitement en tant qu'utilisateur **postgres** de l'instance à la base **postgres** (qui existent généralement par défaut).

```
$ sudo -iu postgres  psql
```

Dans les configurations par défaut courantes, cette commande est généralement la seule qui fonctionne sur une instance fraîchement installée.

- Utilisation des variables d'environnement pour alléger la syntaxe dans un petit script de maintenance :

```
#! /bin/bash
export PGHOST=/var/run/postgresql
export PGPORT=5435
export PGDATABASE=comptabilite
export PGUSER=superutilisateur
# liste des bases
psql -l
# nettoyage
vacuumdb
# une sauvegarde
pg_dump -f fichier.dump
```

Raccourcis

Généralement, préciser `-d` n'est pas nécessaire quand la base de données est le premier argument non nommé de la ligne de commande. Par exemple :

```
$ psql -U jeanpierre comptabilite
```

```
$ sudo -iu postgres  vacuumdb nomdelabase
```

Il faut faire attention à quelques différences entre les outils : pour `pgbench`, `-d` désigne le mode debug et le nom de la base est un argument non nommé ; alors que pour `pg_restore`, le `-d` est nécessaire pour restaurer vers une base.

Variable d'environnement PGOPTIONS

La variable d'environnement `$PGOPTIONS` permet de positionner la plupart des paramètres de sessions disponibles, qui surchargent les valeurs par défaut.

Par exemple, pour exécuter une requête avec un paramétrage différent de `work_mem` (mémoire de tri) :

```
$ PGOPTIONS="-c work_mem=100MB" psql -p 5433 -h serveur nombase  < grosse_requete.sql
```

ou pour importer une sauvegarde sans être freiné par un serveur secondaire synchrone :

```
$ PGOPTIONS="-c synchronous_commit=local" pg_restore -d nombase sauvegarde.dump
```

4.3.2 Autres variables d'environnement

- $PGAPPNAME
- $PGSSLMODE
- $PGPASSWORD
- …

Il existe aussi :

- `$PGSSLMODE` pour définir le niveau de chiffrement SSL de la connexion avec les valeurs `disable`, `prefer` (le défaut), `require`… (il existe d'autres variables d'environnement pour une gestion fine du SSL) ;
- `$PGAPPNAME` permet de définir une chaîne de caractère arbitraire pour identifier la connexion dans les outils et vues d'administration (paramètre de session `application_name`) : mettez-y par exemple le nom du script ;
- `$PGPASSWORD` peut stocker le mot de passe pour ne pas avoir à l'entrer à la connexion (voir plus loin).

Toutes ces variables d'environnement, ainsi que de nombreuses autres, et leurs diverses valeurs possibles, sont documentées[8].

4.3.3 Chaînes libpq clés/valeur

```
psql "host=serveur1  user=jeanpierre  dbname=comptabilite"

psql -d "host=serveur1  port=5432  user=jeanpierre  dbname=comptabilite
     sslmode=require  application_name='chargement'
     options='-c work_mem=30MB' "
```

En lieu du nom de la base, une chaîne peut inclure tous les paramètres nécessaires. Cette syntaxe permet de fournir plusieurs paramètres d'un coup.

[8] https://docs.postgresql.fr/current/libpq-envars.html

Les paramètres disponibles sont listés dans la documentation[9]. On retrouvera de nombreux paramètres équivalents aux variables d'environnement[10], ainsi que d'autres.

Dans cet exemple, on se connecte en exigeant le SSL, en positionnant `application_name` pour mieux repérer la session, et en modifiant la mémoire de tri, ainsi que le paramétrage de la synchronisation sur disque pour accélérer les choses :

```
$ psql "host=serveur1  port=5432  user=jeanpierre  dbname=comptabilite
       sslmode=require  application_name='chargement'
       options='-c work_mem=99MB' -c synchronous_commit=off' " \
       <  script_chargement.sql
```

Tous les outils de l'écosystème ne connaissent pas cette syntaxe (par exemple pgCluu).

4.3.4 Chaînes URI

```
psql -d "postgresql://jeanpierre@serveur1:5432/comptabilite"

psql \
"postgres://jeanpierre@serveur1/comptabilite?sslmode=require\
&options=-c%20synchronous_commit%3Doff"

psql -d postgresql://serveur1/comptabilite?user=jeanpierre\&port=5432
```

Une autre possibilité existe : des chaînes sous forme URI comme il en existe pour beaucoup de pilotes et d'outils. La syntaxe est de la forme :

```
postgresql://[user[:motdepasse]@][lieu][:port][/dbname][?param1=valeur21&param2=valeur2…]
```

Là encore cette chaîne remplace le nom de la base dans les outils habituels. `postgres://` et `postgresql://` sont tous deux acceptés.

Cette syntaxe est très souple. Une difficulté réside dans la gestion des caractères spéciaux, signes `=` et des espaces :

```
# Ces deux appels sont équivalents
$ psql -d postgresql:///var/lib/postgresql?dbname=pgbench\&user=pgbench\&port=5436
$ psql -h /var/lib/postgresql -d pgbench -U pgbench -p 5436

# Ces deux appels sont équivalents
$ psql "postgresql://bob@vm1/proddb?&options=-c%20synchronous_commit%3Doff"
$ psql -U bob -h vm1 -d proddb -c 'synchronous_commit=off'
```

[9] https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-PARAMKEYWORDS
[10] https://docs.postgresql.fr/current/libpq-envars.html

Tous les outils de l'écosystème ne connaissent pas cette syntaxe (par exemple pgCluu).

4.3.5 Connexion avec choix automatique du serveur

```
psql "host=serveur1,serveur2,serveur3
      port=5432,5433,5434
      user=jeanpierre  dbname=comptabilite
      target_session_attrs=read-write
      load_balance_hosts=random"   # v16
```

Il est possible d'indiquer plusieurs hôtes et ports. Jusque PostgreSQL 15 inclus, l'hôte sélectionné est le premier qui répond avec les conditions demandées. À partir de PostgreSQL 16, on peut ajouter `load_balance_hosts=random` pour que la libpq choisisse un serveur au hasard, pour une répartition de charge basique, mais très simple à mettre en place.

Si l'authentification ne passe pas, la connexion tombera en erreur. Il est possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule grâce au paramètre `target_session_attrs`.

Par exemple, on se connectera ainsi au premier serveur de la liste ouvert en écriture et disponible parmi les trois précisés :

```
psql postgresql://jeanpierre@serveur1:5432,serveur2:5433,serveur3:5434/\
comptabilite?target_session_attrs=read-write
```

qui équivaut à :

```
psql "host=serveur1,serveur2,serveur3 port=5432,5433,5434
      target_session_attrs=read-write  user=jeanpierre  dbname=comptabilite"
```

Depuis la version 14, dans le but de faciliter la connexion aux différents serveurs d'une grappe, `target_session_attrs` possède d'autres options[11] que `read-write`, à savoir : `any`, `read-only`, `primary`, `standby`, `prefer-standby`. Par exemple, la commande suivante permet de se connecter à un secondaire au hasard parmi ceux qui répondent. `serveur3` est doublé pour prendre une charge double des autres. S'il ne reste plus que le primaire, la connexion se fera à celui-ci.

```
psql "host=serveur1,serveur2,serveur3,serveur3 port=5432,5433,5434,5434
      target_session_attrs=prefer-standby
      load_balance_hosts=random
      user=jeanpierre  dbname=comptabilite"
```

[11] https://docs.postgresql.fr/current/libpq-connect.html#LIBPQ-CONNECT-TARGET-SESSION-ATTRS

4.3.6 Authentification d'un client (outils console)

- En interactif (`psql`)
  - `-W | --password`
  - `-w | --no-password`
- Variable `$PGPASSWORD`
- À préférer : fichier `.pgpass`
  - `chmod 600 .pgpass`
  - `nom_hote:port:database:nomutilisateur:motdepasse`

Options -W et -w de psql

L'option `-W` oblige à saisir le mot de passe de l'utilisateur. C'est le comportement par défaut si le serveur demande un mot de passe. Si les accès aux serveurs sont configurés sans mot de passe et que cette option est utilisée, le mot de passe sera demandé et fourni à PostgreSQL lors de la demande de connexion. Mais PostgreSQL ne vérifiera pas s'il est bon si la méthode d'authentification ne le réclame pas.

L'option `-w` empêche la saisie d'un mot de passe. Si le serveur a une méthode d'authentification qui nécessite un mot de passe, ce dernier devra être fourni par le fichier `.pgpass` ou par la variable d'environnement `$PGPASSWORD`. Dans tous les autres cas, la connexion échoue.

Variable $PGPASSWORD

Si `psql` détecte une variable `$PGPASSWORD` initialisée, il se servira de son contenu comme mot de passe qui sera soumis pour l'authentification du client.

Fichier .pgpass

Le fichier `.pgpass`, situé dans le répertoire personnel de l'utilisateur ou celui référencé par `$PGPASSFILE`, est un fichier contenant les mots de passe à utiliser si la connexion requiert un mot de passe (et si aucun mot de passe n'a été spécifié). Sur Microsoft Windows, le fichier est nommé `%APPDATA%\postgresql\pgpass.conf` (où `%APPDATA%` fait référence au sous-répertoire `Application Data` du profil de l'utilisateur).

Ce fichier devra être composé de lignes du format :

```
nom_hote:port:nom_base:nom_utilisateur:mot_de_passe
```

Chacun des quatre premiers champs pourraient être une valeur littérale ou `*` (qui correspond à tout). La première ligne réalisant une correspondance pour les paramètres de connexion sera utilisée (du

coup, placez les entrées plus spécifiques en premier lorsque vous utilisez des jokers). Si une entrée a besoin de contenir `*` ou `\`, échappez ce caractère avec `\`. Un nom d'hôte `localhost` correspond à la fois aux connexions `host` (`TCP`) et aux connexions `local` (`socket` de domaine *Unix*) provenant de la machine locale.

Les droits sur `.pgpass` doivent interdire l'accès aux autres et au groupe ; réalisez ceci avec la commande :

```
chmod 0600 ~/.pgpass
```

Attention : si les droits du fichier sont moins stricts, le fichier sera ignoré !

Les droits du fichier ne sont actuellement pas vérifiés sur Microsoft Windows.

4.4 LA CONSOLE PSQL

- Un outil simple pour
  - les opérations courantes
  - les tâches de maintenance
  - l'exécution des scripts
  - les tests

```
postgres$ psql
  base=#
```

La console `psql` permet d'effectuer l'ensemble des tâches courantes d'un utilisateur de bases de données. Si ces tâches peuvent souvent être effectuées à l'aide d'un outil graphique, la console présente l'avantage de pouvoir être utilisée en l'absence d'environnement graphique ou de scripter les opérations à effectuer sur la base de données. Elle a la garantie d'être également toujours disponible.

Nous verrons également qu'il est possible d'administrer la base de données depuis cette console.

Enfin, elle permet de tester l'activité du serveur, l'existence d'une base, la présence d'un langage de programmation…

4.4.1 Obtenir de l'aide et quitter

- Obtenir de l'aide sur les commandes internes `psql`
  - `\?`
- Obtenir de l'aide sur les ordres SQL
  - `\h <COMMANDE>`
- Quitter
  - `\q` ou `ctrl-D`
  - `quit` ou `exit` (v11)

`\?` liste les commandes propres à `psql` (par exemple `\d` ou `\du` pour voir les tables ou les utilisateurs), trop nombreuses pour pouvoir être mémorisées.

`\h <COMMANDE>` affiche l'aide en ligne des commandes SQL. Sans argument, la liste des commandes disponibles est affichée. La version 12 ajoute en plus l'URL vers la page web documentant cette commande.

Exemple :

```
postgres=# \h ALTER TA

Commande :    ALTER TABLE
Description : modifier la définition d'une table
Syntaxe :
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
…
URL: https://www.postgresql.org/docs/15/sql-altertable.html
```

`\q` ou `Ctrl-D` permettent de quitter `psql`. Depuis la version 11, il est aussi possible d'utiliser `quit` ou `exit`.

4.4.2 Gestion de la connexion

- Modifier le mot de passe d'un utilisateur
  - `\password  nomutilisateur`
- Quelle est la connexion courante ?
  - `\conninfo`
  - `SELECT current_user,session_user,system_user;`
- Se connecter à une autre base :
  - `\c ma base`
  - `\c mabase  utilisateur serveur 5432`

`\c` permet de changer d'utilisateur et/ou de base de données sans quitter le client. `\conninfo` permet de savoir où l'on est connecté.

Exemple :

```
CREATE DATABASE formation;
CREATE DATABASE prod;
CREATE USER stagiaire1;
CREATE USER stagiaire2;
CREATE USER admin;
```

```
postgres=# \c formation stagiaire1

You are now connected to database "formation" as user "stagiaire1".

formation=> \c - stagiaire2

You are now connected to database "formation" as user "stagiaire2".

formation=> \c prod admin

You are now connected to database "prod" as user "admin".

prod=> \conninfo

You are connected to database "prod" as user "admin"
                  on host "localhost" (address "::1") at port "5412".
```

Au niveau SQL, un équivalent partiel de `\conninfo` serait :

```
SELECT  current_user,
        current_catalog,                          -- base
        inet_server_addr(), inet_server_port(),   -- serveur
        system_user                               -- depuis PG16
        \gx
```

```
-[ RECORD 1 ]----+-----------------------
current_user     | dalibo
current_catalog  | pgbench
inet_server_addr | 192.168.74.5
inet_server_port | 5433
system_user      | scram-sha-256:postgres
```

Un superutilisateur pourra affecter un mot de passe à un autre utilisateur grâce à la commande `\password`, qui en fait encapsule un `ALTER USER … PASSWORD …`. Le gros intérêt de `\password` est d'envoyer le mot de passe chiffré au serveur. Ainsi, même si les traces contiennent toutes les requêtes SQL exécutées, il est impossible de retrouver les mots de passe via le fichier de traces. Ce n'est pas le cas avec un `CREATE ROLE` ou un `ALTER ROLE` manuel (à moins de chiffrer soi-même le mot de passe).

4.4.3 Catalogue système : objets utilisateurs

Lister :

- les bases de données
  - `\l`, `\l+`
- les tables
  - `\d`, `\d+`, `\dt`, `\dt+`
- les index
  - `\di`, `\di+`
- les schémas
  - `\dn`
- les fonctions & procédures
  - `\df[+]`
- etc…

Ces commandes permettent d'obtenir des informations sur les objets utilisateurs de tout type stockés dans la base de données. Pour les commandes qui acceptent un motif, celui-ci permet de restreindre les résultats retournés à ceux dont le nom d'opérateur correspond au motif précisé.

Le client psql en version 15 est compatible avec toutes les versions du serveur depuis la 9.2 incluse.

`\l` dresse la liste des bases de données sur le serveur. Avec `\l+`, les commentaires, les tablespaces par défaut et les tailles des bases sont également affichés, ce qui peut être très pratique.

`\dt` affiche les tables, `\di` les index, `\dn` les schémas, `\ds` les séquences, `\dv` les vues, etc. Là encore, on peut ajouter `+` pour d'autres informations comme la taille, et même `S` pour inclure les objets système normalement masqués.

Exemple :

Si l'on crée une simple base grâce à `pgbench` avec un utilisateur dédié :

```
$ psql
```

```
=# CREATE ROLE testeur LOGIN ;
=# \password testeur
Saisir le nouveau mot de passe :
Saisir le mot de passe à nouveau :
```

Utilisateurs en place :

```
=# \du
                              Liste des rôles
         Nom du rôle          |        Attributs        |       Membre de
-----------------------------+-------------------------+-------------------
 dalibo                      |                         | {}
 nagios                      | Superutilisateur        | {}
 postgres                    | Superutilisateur, (...) | {}
 testeur                     |                         | {}
```

Création de la base :

```
CREATE DATABASE pgbench OWNER testeur ;
```

Bases de données en place :

```
=# \l
                           Liste des bases de données
      Nom       | Propriétaire | Encodage | Collationnement | Type caract. | …
----------------+--------------+----------+-----------------+--------------+----
 pgbench        | testeur      | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 postgres       | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
```

Création des tables de cette nouvelle base :

```
$ pgbench --initialize --scale=10 -U testeur -h localhost pgbench
```

Connexion à la nouvelle base :

```
$ psql -h localhost -U testeur -d pgbench
```

Les tables :

```
=# \d
                Liste des relations
 Schéma |       Nom        | Type  | Propriétaire
--------+------------------+-------+--------------
 public | pgbench_accounts | table | testeur
 public | pgbench_branches | table | testeur
 public | pgbench_history  | table | testeur
 public | pgbench_tellers  | table | testeur
(4 lignes)

=# \dt+ pgbench_*s
                           Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Persistence |  M…  | Taille |
--------+------------------+-------+--------------+-------------+------+--------+--
 public | pgbench_accounts | table | testeur      | permanent   | heap | 128 MB |
 public | pgbench_branches | table | testeur      | permanent   | heap | 40 kB  |
 public | pgbench_tellers  | table | testeur      | permanent   | heap | 40 kB  |
(3 lignes)
```

Une table (affichage réduit pour des raisons de mise en page) :

```
=# \d+ pgbench_accounts
                           Table « public.pgbench_accounts »
 Colonne  |     Type      | Col..nt | NULL-able | … | Stockage | Compr. | …
----------+---------------+---------+-----------+---+----------+--------+--
```

```
 aid      | integer       |          | not null  |   | plain    |        |
 bid      | integer       |          |           |   | plain    |        |
 abalance | integer       |          |           |   | plain    |        |
 filler   | character(84) |          |           |   | extended |        |
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Méthode d'accès : heap
Options: fillfactor=100
```

Les index :

```
=> \di
                            Liste des relations
 Schéma |          Nom          | Type  | Propriétaire |      Table
--------+-----------------------+-------+--------------+------------------
 public | pgbench_accounts_pkey | index | testeur      | pgbench_accounts
 public | pgbench_branches_pkey | index | testeur      | pgbench_branches
 public | pgbench_tellers_pkey  | index | testeur      | pgbench_tellers
(3 lignes)
```

Les schémas, utilisateur ou système :

```
=> \dn+
                            Liste des schémas
  Nom   | Propriétaire |    Droits d´accès    |      Description
--------+--------------+----------------------+------------------------
 public | postgres     | postgres=UC/postgres+| standard public schema
        |              | =UC/postgres         |
(1 ligne)

=> \dnS
         Liste des schémas
        Nom         | Propriétaire
--------------------+--------------
 information_schema | postgres
 pg_catalog         | postgres
 pg_toast           | postgres
 public             | postgres
(4 lignes)
```

Les tablespaces (ici ceux par défaut) :

```
=> \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
 pg_global  | postgres     |
(2 lignes)
```

4.4.4 Catalogue système : rôles et accès

- Lister les rôles (utilisateurs et groupes)
  - `\du[+]`
- Lister les droits d'accès
  - `\dp`
- Lister les droits d'accès par défaut
  - `\ddp`
  - `ALTER DEFAULT PRIVILEGES`
- Lister les configurations par rôle et par base
  - `\drds`

On a vu que `\du` (`u` pour *user*) affiche les rôles existants. Rappelons qu'un rôle peut être aussi bien un utilisateur (si le rôle a le droit de `LOGIN`) qu'un groupe d'utilisateurs, voire les deux à la fois.

Dans les versions antérieures à la 8.1, il n'y avait pas de rôles, et les groupes et les utilisateurs étaient deux notions distinctes. Certaines commandes ont conservé le terme de *user*, mais il s'agit bien de rôles dans tous les cas.

Les droits sont accessibles par les commandes `\dp` (`p` pour « permissions ») ou `\z`.

Dans cet exemple, le rôle **admin** devient membre du rôle système **pg_signal_backend** :

```
=# GRANT pg_signal_backend TO admin;

=# \du
                                   List of roles
 Nom du rôle |                      Attributs          |       Membre de
-------------+-------------------------------------------+---------------------
 admin       |                                           | {pg_signal_backend}
 postgres    | Superuser, Create role, Create DB,        |
             | Replication, Bypass RLS                   | {}
 …
```

Toujours dans la base **pgbench** :

```
=# GRANT SELECT ON TABLE pgbench_accounts TO dalibo ;
```

```
=# GRANT ALL ON TABLE pgbench_history TO dalibo ;
=# \z
```

```
=> \z
                                    Droits d'accès
 Schéma |       Nom        | Type  |     Droits d'accès      | Droits … | …
--------+------------------+-------+-------------------------+----------+---
 public | pgbench_accounts | table | testeur=arwdDxt/testeur+|          |
        |                  |       | dalibo=r/testeur        |          |
 public | pgbench_branches | table |                         |          |
 public | pgbench_history  | table | testeur=arwdDxt/testeur+|          |
        |                  |       | dalibo=arwdDxt/testeur  |          |
 public | pgbench_tellers  | table |                         |          |
(4 lignes)
```

La commande `\ddp` permet de connaître les droits accordés par défaut à un utilisateur sur les nouveaux objets avec l'ordre `ALTER DEFAULT PRIVILEGES`.

```
=# ALTER DEFAULT PRIVILEGES GRANT ALL ON TABLES TO dalibo ;
```

```
=# \ddp
                 Droits d´accès par défaut
 Propriétaire | Schéma | Type  |     Droits d´accès
--------------+--------+-------+------------------------
 testeur      |        | table | dalibo=arwdDxt/testeur +
              |        |       | testeur=arwdDxt/testeur
(1 ligne)
```

Enfin, la commande `\drds` permet d'obtenir la liste des paramètres appliqués spécifiquement à un utilisateur ou une base de données.

```
=# ALTER DATABASE pgbench SET work_mem TO '15MB';
=# ALTER ROLE testeur SET log_min_duration_statement TO '0';
```

```
=# \drds
              Liste des paramètres
  Rôle   | Base de données |         Réglages
---------+-----------------+--------------------------
 testeur |                 | log_min_duration_statement=0
         | pgbench         | work_mem=15MB
…
```

4.4.5 Visualiser le code des objets

- Voir les vues ou les fonctions & procédures
  - `\dv`, `\df`
- Code d'une vue
  - `\sv`
- Code d'une procédure stockée
  - `\sf`

Ceci permet de visualiser le code de certains objets sans avoir besoin de l'éditer. Par exemple avec cette vue système :

```
=# \dv pg_tables
             Liste des relations
   Schéma   |    Nom     | Type | Propriétaire
------------+------------+------+--------------
 pg_catalog | pg_tables | vue  | postgres
(1 ligne)


=# \sv+ pg_tables
1       CREATE OR REPLACE VIEW pg_catalog.pg_tables AS
2        SELECT n.nspname AS schemaname,
3           c.relname AS tablename,
4           pg_get_userbyid(c.relowner) AS tableowner,
5           t.spcname AS tablespace,
6           c.relhasindex AS hasindexes,
7           c.relhasrules AS hasrules,
8           c.relhastriggers AS hastriggers,
9           c.relrowsecurity AS rowsecurity
10         FROM pg_class c
11           LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
12           LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
13        WHERE c.relkind = ANY (ARRAY['r'::"char", 'p'::"char"])
```

Ou cette fonction :

```
=# CREATE FUNCTION nb_sessions_actives () RETURNS int
LANGUAGE sql
AS $$
  SELECT COUNT(*) FROM pg_stat_activity
  WHERE backend_type = 'client backend' AND state='active' ;
$$ ;
```

```
=# \df nb_sessions_actives
                                        Liste des fonctions
 Schéma |         Nom         | Type … du résultat | Type …des paramètres | Type
--------+---------------------+--------------------+----------------------+------
 public | nb_sessions_actives | integer            |                      | func
(1 ligne)
```

```
=# \sf nb_sessions_actives
CREATE OR REPLACE FUNCTION public.nb_sessions_actives()
 RETURNS integer
 LANGUAGE sql
AS $function$
  SELECT COUNT(*) FROM pg_stat_activity
  WHERE backend_type = 'client backend' AND state='active' ;
$function$
```

Il est même possible de lancer un éditeur depuis `psql` pour modifier directement la vue ou la fonction avec respectivement `\ev` ou `\ef`.

4.4.6 Configuration

- Lister les paramètres correspondants au motif indiqué
  - `\dconfig` (v15+)

La méta-commande `\dconfig` permet de récupérer la configuration d'un paramètre. Par exemple :

```
b1=# \dconfig log_r*
  List of configuration parameters
          Parameter          | Value
-----------------------------+-------
 log_recovery_conflict_waits | off
 log_replication_commands    | off
 log_rotation_age            | 1d
 log_rotation_size           | 0
(4 rows)
```

En ajoutant le signe `+`, il est possible d'obtenir plus d'informations :

```
b1=# \dconfig+ log_r*
                        List of configuration parameters
          Parameter          | Value |  Type   |  Context   | Access privileges
-----------------------------+-------+---------+------------+-------------------
 log_recovery_conflict_waits | off   | bool    | sighup     |
 log_replication_commands    | off   | bool    | superuser  |
 log_rotation_age            | 1d    | integer | sighup     |
 log_rotation_size           | 0     | integer | sighup     |
(4 rows)
```

4.4.7 Exécuter des requêtes

- Exécuter une requête :

```
SELECT * FROM pg_tables ;
SELECT * FROM pg_tables \g
SELECT * FROM pg_tables \gx   -- une ligne par champ
INSERT INTO … VALUES (1) \; INSERT INTO … VALUES (2) ; -- 1 transaction
```

- Rappel des requêtes:
  - flèche vers le haut
  - `\g`
  - `Ctrl-R` suivi d'un extrait de texte représentatif

Les requêtes SQL doivent se terminer par `;` ou, pour marquer la parenté de PostgreSQL avec Ingres, `\g`. Cette dernière commande permet de relancer un ordre.

Depuis version 10, il est aussi possible d'utiliser `\gx` pour bénéficier de l'affichage étendu sans avoir à jouer avec `\x on` et `\x off`.

La console `psql`, lorsqu'elle est compilée avec la bibliothèque `libreadline` ou la bibliothèque `libedit` (cas des distributions Linux courantes), dispose des mêmes possibilités de rappel de commande que le shell bash.

Exemple :

```
postgres=# SELECT * FROM pg_tablespace LIMIT 5;

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |

postgres=# SELECT * FROM pg_tablespace LIMIT 5\g

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |

postgres=# \g

  oid  |  spcname   | spcowner | spcacl | spcoptions
-------+------------+----------+--------+------------
  1663 | pg_default |       10 |        |
```

```
  1664 | pg_global  |       10 |        |
 16502 | ts1        |       10 |        |
```

```
postgres=# \gx
```

```
-[ RECORD 1 ]----------
oid        | 1663
spcname    | pg_default
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 2 ]----------
oid        | 1664
spcname    | pg_global
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 3 ]----------
oid        | 16502
spcname    | ts1
spcowner   | 10
spcacl     |
spcoptions |
```

Plusieurs ordres sur la même ligne séparés par des `;` seront exécutés et affichés à la suite. Par contre, s'ils sont séparés par `\;`, ils seront envoyés ensemble et implicitement dans une même transaction (sauf utilisation explicite de `BEGIN` / `COMMIT` / `ROLLBACK`). Avant la version 15, ne sera affiché que le résultat du dernier ordre.

Dans cet exemple, la division par zéro fait tomber en erreur et annule l'insertion de la valeur 2 car les deux sont dans la même transaction :

```
=# CREATE TABLE demo_insertion (i float);
CREATE TABLE
=# INSERT INTO demo_insertion VALUES(1.0); INSERT INTO demo_insertion VALUES(1.0/0);
INSERT 0 1
ERROR:  division by zero
=# SELECT * FROM demo_insertion \; INSERT INTO demo_insertion VALUES (2.0) \;
   INSERT INTO demo_insertion VALUES (2.0/0) ;
 i
 1
(1 ligne)

INSERT 0 1
ERROR:  division by zero
=# SELECT * FROM demo_insertion ;
 i
 1
(1 ligne)
```

4.4.8 Afficher le résultat d'une requête

- `\x` pour afficher un champ par ligne
- Affichage par paginateur si l'écran ne suffit pas
- Préférer `less` :
  - `set PAGER='less -S'`
- Ou outil dédié :
  - `\setenv PAGER 'pspg'`

En mode interactif, `psql` cherche d'abord à afficher directement le résultat :

```
postgres=# SELECT relname,reltype, relchecks, oid,oid FROM pg_class LIMIT 3;

      relname       | reltype | relchecks |  oid  |  oid
--------------------+---------+-----------+-------+-------
 pg_statistic       |   11319 |         0 |  2619 |  2619
 t3                 |   16421 |         0 | 16419 | 16419
 capitaines_id_seq  |   16403 |         0 | 16402 | 16402
 t1                 |   16415 |         0 | 16413 | 16413
 t2                 |   16418 |         0 | 16416 | 16416
```

S'il y a trop de colonnes, on peut préférer n'avoir qu'un champ par ligne grâce au commutateur `\x` :

```
postgres=# \x on

Expanded display is on.

postgres=# SELECT relname,reltype, relchecks, oid,oid FROM pg_class LIMIT 3;

-[ RECORD 1 ]----------------
relname   | pg_statistic
reltype   | 11319
relchecks | 0
oid       | 2619
oid       | 2619
-[ RECORD 2 ]----------------
relname   | t3
reltype   | 16421
relchecks | 0
oid       | 16419
oid       | 16419
-[ RECORD 3 ]----------------
relname   | capitaines_id_seq
reltype   | 16403
relchecks | 0
```

```
oid       | 16402
oid       | 16402
```

`\x on` et `\x off` sont alternativement appelés si l'on tape plusieurs fois `\x`. `\x auto` délègue à `psql` la décision du meilleur affichage, en général à bon escient. `\gx` à la place de `;` bascule l'affichage pour une seule requête.

S'il n'y a pas la place à l'écran, `psql` appelle le paginateur par défaut du système. Il s'agit souvent de `more`, parfois de `less`. Ce dernier est bien plus puissant, permet de parcourir le résultat en avant et en arrière, avec la souris, de chercher une chaîne de caractères, de tronquer les lignes trop longues (avec l'option `-S`) pour naviguer latéralement.

Le paramétrage du paginateur s'effectue par des variables d'environnement :

```
export PAGER='less -S'
psql
```

ou :

```
PAGER='less -S'  psql
```

ou dans `psql` directement, ou `.psqlrc` :

```
\setenv PAGER 'less -S'
```

Mais `less` est généraliste. Un paginateur dédié à l'affichage de résultats de requêtes a été récemment développé par Pavel Stěhule[12] et son paquet figure dans les principales distributions.

Pour les gens qui consultent souvent des données dans les tables depuis la console, `pspg` permet de naviguer dans les lignes avec la souris en figeant les entêtes ou quelques colonnes ; de poser des signets sur des lignes ; de sauvegarder les lignes. (Pour plus de détail, voir cette présentation[13] et la page Github du projet[14]). La mise en place est similaire :

```
\setenv PAGER 'pspg'
```

À l'inverse, la pagination se désactive complètement avec :

```
\pset pager
```

(et bien sûr en mode non interactif, par exemple quand la sortie de `psql` est redirigée vers un fichier).

[12] http://okbob.blogspot.fr/2017/07/i-hope-so-every-who-uses-psql-uses-less.html
[13] http://blog-postgresql.verite.pro/2017/11/21/test-pspg.html
[14] https://github.com/okbob/pspg

4.4.9 Afficher les détails d'une requête

- `\gdesc`
- Afficher la liste des colonnes correspondant au résultat d'exécution d'une requête
  - noms
  - type de données

Après avoir exécuté une requête, ou même à la place de l'exécution, il est possible de connaître le nom des colonnes en résultat ainsi que leur type de données.

Requête :

```
SELECT nspname, relname
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

Description des colonnes :

```
postgres=# \gdesc

 Column   | Type
---------+------
 nspname | name
 relname | name
```

Ou sans exécution :

```
postgres=# SELECT * FROM generate_series (1, 1000) \gdesc

     Column       |  Type
-----------------+---------
 generate_series | integer
```

4.4.10 Exécuter le résultat d'une requête

- Exécuter le résultat d'une requête
  - `\gexec`

Parfois, une requête permet de créer des requêtes sur certains objets. Par exemple, si nous souhaitons exécuter un `VACUUM` sur toutes les tables du schéma `public`, nous allons récupérer la liste des tables avec cette requête :

```
SELECT nspname, relname FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

```
 nspname |     relname
---------+-----------------
 public  | pgbench_branches
 public  | pgbench_tellers
 public  | pgbench_accounts
 public  | pgbench_history
(4 lignes)
```

Plutôt que d'éditer manuellement cette liste de tables pour créer les ordres SQL nécessaires, autant modifier la requête pour qu'elle prépare elle-même les ordres SQL :

```
SELECT 'VACUUM ' || quote_ident(nspname) || '.' || quote_ident(relname)
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relkind = 'r';
```

```
           ?column?
--------------------------------
 VACUUM public.pgbench_branches
 VACUUM public.pgbench_tellers
 VACUUM public.pgbench_accounts
 VACUUM public.pgbench_history
(4 lignes)
```

Une fois que nous avons vérifié la validité des requêtes SQL, il ne reste plus qu'à les exécuter. C'est ce que permet la commande `\gexec` :

```
=# \gexec
```

```
VACUUM
VACUUM
VACUUM
VACUUM
```

4.4.11 Manipuler le tampon de requêtes

- Éditer
  - dernière requête : `\e`
  - vue : `\ev nom_vue`
  - fonction PL/pgSQL : `\ef nom_fonction`
- Historique :
  - `\s`

`\e nomfichier.sql` édite le tampon de requête courant ou un fichier existant indiqué à l'aide d'un éditeur externe.

L'éditeur désigné avec les variables d'environnement `$EDITOR` ou `$PSQL_EDITOR` notamment. Sur Unix, c'est généralement par défaut une variante de `vi` mais n'importe quel éditeur fait l'affaire :

```
PSQL_EDITOR=nano  psql
```

ou dans `psql` ou dans le `.psqlrc` :

```
\setenv PSQL_EDITOR 'gedit'
```

`\p` affiche le contenu du tampon de requête.

`\r` efface la requête qui est en cours d'écriture. La précédente requête reste accessible avec `\p`.

`\w nomfichier.sql` provoque l'écriture du tampon de requête dans le fichier indiqué (à modifier par la suite avec `\e` par exemple).

`\ev v_mavue` édite la vue indiquée. Sans argument, cette commande affiche le squelette de création d'une nouvelle vue.

`\ef f_mafonction` est l'équivalent pour une fonction.

`\s` affiche l'historique des commandes effectuées dans la session. `\s historique.log` sauvegarde cet historique dans le fichier indiqué.

4.4.12 Entrées/sorties

- Charger et exécuter un script SQL
  - `\i fichier.sql`
- Rediriger la sortie dans un fichier
  - `\o resultat.out`
- Écrire un texte sur la sortie standard
  - `\echo "Texte…"`

`\i fichier.sql` lance l'exécution des commandes placées dans le fichier passé en argument. `\ir` fait la même chose sauf que le chemin est relatif au chemin courant.

`\o resultat.out` envoie les résultats de la requête vers le fichier indiqué (voire vers une commande UNIX via un *pipe*).

Exemple :

```
=# \o tables.txt
=# SELECT * FROM pg_tables ;
```

(Si l'on intercale `\H`, on peut avoir un formatage en HTML.)

Attention : cela va concerner toutes les commandes suivantes. Entrer `\o` pour revenir au mode normal.

`\echo "Texte"` affiche le texte passé en argument sur la sortie standard. Ce peut être utile entre les étapes d'un script.

4.4.13 Gestion de l'environnement système

- Chronométrer les requêtes
  - `\timing on`
- Exécuter une commande OS
  - `\! ls -l` (sur le client !)
- Changer de répertoire courant
  - `\cd /tmp`
- Affecter la valeur d'une variable d'environnement (v15+)
  - `\getenv toto PATH`

`\timing on` active le chronométrage de toutes les commandes. C'est très utile mais alourdit l'affichage. Sans argument, la valeur actuelle est basculée de `on` à `off` et vice-versa.

```
=# \timing on
Chronométrage activé.

=# VACUUM ;
VACUUM
Temps : 26,263 ms
```

`\! <commande>` ouvre un shell interactif en l'absence d'argument ou exécute la commande indiquée **sur le client** (pas le serveur !) :

`\cd` (et non `\! cd` !) permet de changer de répertoire courant, là encore **sur le client**. Cela peut servir pour définir le chemin d'un script à exécuter ou d'un futur export.

`\getenv` permet de récupérer la valeur d'une valeur d'environnement système et de l'affecter à une variable.

Exemple :

```
\! cat nomfichier.out
\! ls -l /tmp
\! mkdir /home/dalibo/travail
\cd /home/dalibo/travail
\! pwd
/home/dalibo/travail
```

4.4.14 Variables internes psql

- Positionner des variables internes
  - `\set NOMVAR nouvelle_valeur`
- Variables internes usuelles
  - `ON_ERROR_STOP` : `on` / `off`
  - `ON_ERROR_ROLLBACK` : `on` / `off` / `interactive`
  - `ROW_COUNT` : nombre de lignes renvoyées par la dernière requête (v11)
  - `ERROR` : `true` si dernière requête en erreur (v11)
- Ne pas confondre avec `SET` (au niveau du serveur) !

Les variables déclarées avec `\set` sont positionnées au niveau de `psql`, outil client. Elles ne sont pas connues du serveur et n'existent pas dans les autres outils clients (pgAdmin, DBeaver…).
Il ne faut pas les confondre avec les paramètres définis sur le serveur au niveau de la session avec `SET`. Ceux-ci sont transmis directement au serveur quand ils sont entrés dans un outil client, quel qu'il soit.

`\set` affiche les variables internes et utilisateur.

`\set NOMVAR nouvelle_valeur` permet d'affecter une valeur.

La liste des variables prédéfinies est disponible dans la documentation de psql[15]. Beaucoup modifient le comportement de `psql`.

Exemple :

```
postgres=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'b1'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
FETCH_COUNT = '0'
HIDE_TABLEAM = 'off'
HIDE_TOAST_COMPRESSION = 'off'
HISTCONTROL = 'none'
```

[15] https://docs.postgresql.fr/current/app-psql.html#app-psql-variables

```
HISTSIZE = '500'
HOST = '/var/run/postgresql'
IGNOREEOF = '0'
LAST_ERROR_MESSAGE = ''
LAST_ERROR_SQLSTATE = '00000'
ON_ERROR_ROLLBACK = 'off'
ON_ERROR_STOP = 'off'
PORT = '5432'
PROMPT1 = '%/%R%x%# '
PROMPT2 = '%/%R%x%# '
PROMPT3 = '>> '
QUIET = 'off'
SERVER_VERSION_NAME = '15.1'
SERVER_VERSION_NUM = '150001'
SHOW_ALL_RESULTS = 'on'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
USER = 'postgres'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 15.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
↪  20210514 (Red Hat 8.5.0-15), 64-bit'
VERSION_NAME = '15.1'
VERSION_NUM = '150001'
```

Les variables `ON_ERROR_ROLLBACK` et `ON_ERROR_STOP` sont discutées dans la partie relative à la gestion des erreurs.

La version 11 ajoute quelques variables internes. `ROW_COUNT` indique le nombre de lignes de résultat de la dernière requête exécutée :

```
=# SELECT * FROM pg_namespace;

      nspname       | nspowner |                nspacl
--------------------+----------+--------------------------------------
 pg_toast           |       10 |
 pg_temp_1          |       10 |
 pg_toast_temp_1    |       10 |
 pg_catalog         |       10 | {postgres=UC/postgres,=U/postgres}
 public             |       10 | {postgres=UC/postgres,=UC/postgres}
 information_schema |       10 | {postgres=UC/postgres,=U/postgres}

=# \echo :ROW_COUNT
6

=# SELECT :ROW_COUNT ;
6
```

alors que `ERROR` est un booléen indiquant si la dernière requête était en erreur ou pas :

```
=# CREATE TABLE t1(id integer);
CREATE TABLE

=# CREATE TABLE t1(id integer);

ERROR:  relation "t1" already exists
```

```
postgres=# \echo :ERROR
true
postgres=# CREATE TABLE t2(id integer);
CREATE TABLE
postgres=# \echo :ERROR
false
```

4.4.15 Variables utilisateur psql

- Définir une variable utilisateur
  - `\set NOMVAR nouvelle_valeur`
- Invalider une variable
  - `\unset NOMVAR`
- Stockage du résultat d'une requête :
  - si résultat est une valeur unique
  - Exemple :

```
SELECT now() AS maintenant \gset
SELECT :'maintenant' ;
```

En dehors des variables internes évoquées dans le chapitre précédent, il est possible à un utilisateur de `psql` de définir ses propres variables.

```
-- initialiser avec une constante
\set active 'y'
\echo :active
y
-- initialiser avec le résultat d'une commande système
\set uptime `uptime`
\echo :uptime
```

```
 09:38:58 up 53 min,  1 user,  load average: 0,12, 0,07, 0,07
```

Et pour supprimer la variable :

```
\unset uptime
```

Il est possible de stocker le résultat d'une requête dans une variable pour sa réutilisation dans un script avec la commande `\gset`. Le nom de la variable est celui de la colonne ou de son alias. La valeur retournée par la requête doit être unique sous peine d'erreur.

```
SELECT now() AS "curdate" \gset
\echo :curdate
```

```
2020-08-16 10:53:51.795582+02
```

Il est possible aussi de donner un préfixe au nom de la variable :

```
SELECT now() AS "curdate" \gset v_
\echo :v_curdate
```

```
2020-08-16 10:54:20.484344+02
```

4.4.16 Tests conditionnels

- `\if`
- `\elif`
- `\else`
- `\endif`

Ces quatre instructions permettent de tester la valeur de variables `psql`, ce qui permet d'aller bien plus loin dans l'écriture de scripts SQL. Le client doit être en version 10 au moins (pas forcément le serveur).

Par exemple, si on souhaite savoir si on se trouve sur un serveur standby ou sur un serveur primaire, il suffit de configurer la variable `PROMPT1` à partir du résultat de l'interrogation de la fonction `pg_is_in_recovery()`. Pour cela, il faut enregistrer ce code dans le fichier `.psqlrc` :

```
SELECT pg_is_in_recovery() as est_standby \gset
\if :est_standby
    \set PROMPT1 'standby %x$ '
\else
    \set PROMPT1 'primaire %x$ '
\endif
```

Puis, en lançant `psql` sur un serveur primaire, on obtient :

```
psql (15.1)
Type "help" for help.

primaire $
```

alors qu'on obtient sur un serveur secondaire :

```
psql (15.1)
Type "help" for help.

standby $
```

4.4.17 Personnaliser psql

- Fichier de configuration `~/.psqlrc`
  - voire `~/.psqlrc-X.Y` ou `~/.psqlrc-X`
  - ignoré avec `-X`
- Exemple de `.psqlrc` :

```
\set ON_ERROR_ROLLBACK interactive  -- paramétrage de session
\timing on
\set PROMPT1 '%M:%> %n@%/%R%#%x'    -- invite
\set cfg 'SHOW ALL ;'               -- requête utilisable avec :cfg
\set cls '\\! clear;'               -- nettoyer l'écran avec :cls
```

`psql` est personnalisable par le biais de plusieurs variables internes. Il est possible de pérenniser ces personnalisations par le biais d'un fichier `~/.psqlrc` (ou `%APPDATA%\postgresql\psqlrc.conf` sous Windows, ou dans un répertoire désigné par `$PSQLRC`). Il peut exister des fichiers par version (par exemple `~/.psqlrc-15` ou `~/.psqlrc-15.0`), voire un fichier global[16].

Modifier l'invite est utile pour toujours savoir où et comment l'on est connecté. Tous les détails sont dans la documentation[17]. Par exemple, ajouter `%>` dans l'invite affiche le port. On peut aussi afficher toutes les variables listées par `\set` (par exemple ainsi : `%:PORT:` ou `%:USER:`). En cas de reconnexion, psql en régénère certaines, mais pas toutes.

Exemple de fichier `.psqlrc` :

```
\set QUIET 1
\timing on
\pset pager on
\setenv pager
\pset null 'ø'
-- Mots clés autocomplétés en majuscule
\set COMP_KEYWORD_CASE upper
-- Affichage
\x auto
\pset linestyle 'unicode'
\pset border 2
\pset unicode_border_linestyle double
\pset unicode_column_linestyle double
\pset unicode_header_linestyle double
-- Prompt dynamique
-- %> indique le port, %m le serveur, %n l'utilisateur, %/ la base…
```

[16] https://docs.postgresql.fr/current/app-psql.html#id-1.9.4.20.10
[17] https://docs.postgresql.fr/current/app-psql.html#app-psql-prompting

```
\set PROMPT1 '%m:%> %[%033[1;33;40m%]%n@%/%R%[%033[0m%]%#%x '
-- serveur secondaire ? (NB : non mis à jour lors d'une reconnexion !)
SELECT pg_is_in_recovery() as est_standby \gset
\if :est_standby
   \set PROMPT1 :PROMPT1 '(standby) '
\else
   \set PROMPT1 :PROMPT1
\endif
\set QUIET 0
```

```
$ psql -h serveur -p5435 -U jeanpierre -d mabase
```

```
[serveur]:5435 jeanpierre@postgres=# (standby) SELECT pi(), now(), null ;
```

pi	now	?column?
3.141592653589793	2022-01-07 16:08:39.925262+01	ø

```
(1 ligne)
```

Il est aussi possible d'y rajouter du paramétrage de session avec `SET` pour adapter le fuseau horaire, par exemple.

Des requêtes très courantes peuvent être ajoutées dans le `.psqlrc`, par exemple celles-ci :

```
-- Paramètres en cours avec leur source
-- Ceci impérativement sur une seule ligne !
\set config 'SELECT name, current_setting(name), CASE source WHEN $$configuration
 ↪  file$$ THEN
regexp_replace(sourcefile, $$^/.*/$$, $$$$)||$$:$$||sourceline ELSE source END FROM
 ↪  pg_settings
WHERE source <> $$default$$ OR name LIKE $$%.%$$;'
```

(Requête inspirée de Christoph Berg[18]).

```
\set extensions 'SELECT * FROM pg_available_extensions ORDER BY name ;'
```

…utilisables ainsi dans `psql` :

```
=# :config

=# :extensions
```

Attention : le `.psqlrc` n'est exécuté qu'au démarrage de `psql`, mais pas lors d'une reconnexion avec `\c` ! Les prompts dynamiques à base de variables utilisateur sont donc susceptibles d'être alors faux ! Pour relancer le script, utiliser :

```
\i ~/.psqlrc
```

[18] https://www.postgresql.org/message-id/YlFQLzlPi4QD0wSi%40msg.df7cb.de

Dans un script, il vaut mieux ignorer ce fichier de configuration grâce à `--no-psqlrc` (`-X`) pour revenir à l'environnement par défaut et éviter de polluer l'affichage :

```
$ psql -X -f script.sql
$ psql -X -At -c 'SELECT name, setting FROM pg_settings ;'
```

4.5 ÉCRITURE DE SCRIPTS SHELL

- Script SQL
- Script Shell
- Exemple sauvegarde

4.5.1 Exécuter un script SQL avec psql

- Avec `-c` :

```
psql -c 'SELECT * FROM matable' -c 'SELECT fonction(123)' ;
```

- Avec un script :

```
psql -f nom_fichier.sql
```

```
psql < nom_fichier.sql
```

- Depuis `psql` :
  - `\i nom_fichier.sql`

L'option `-c` permet de spécifier du SQL en ligne de commande sans avoir besoin de faire un script. Plusieurs ordres seront enchaînés dans la même session.

Il est généralement préférable d'enregistrer les ordres dans des fichiers si on veut les exécuter plusieurs fois sans se tromper. L'option `-f` est très utile dans ce cas. La redirection avec `<` est une alternative répandue.

4.5.2 Gestion des transactions

- `psql` est en mode auto-commit par défaut
  - variable `AUTOCOMMIT`
- Ouvrir une transaction explicitement
  - `BEGIN;`
- Terminer une transaction
  - `COMMIT;` ou `ROLLBACK;`
- Ouvrir une transaction implicitement
  - option `-1` (`--single-transaction`)

Par défaut, `psql` est en mode « autocommit », c'est-à-dire que tous les ordres SQL sont automatiquement validés après leur exécution.

Pour exécuter une suite d'ordres SQL dans une seule et même transaction, il faut soit ouvrir explicitement une transaction avec `BEGIN;` et la valider avec `COMMIT;` ou l'annuler avec `ROLLBACK;`. Les autres outils clients sont généralement dans ce même cas.

L'ordre

```
=# \set AUTOCOMMIT off
```

a pour effet d'insérer systématiquement un `BEGIN` avant chaque ordre s'il n'y a pas déjà une transaction ouverte ; il faudra valider ensuite avec `COMMIT` **avant** de déconnecter. Il est déconseillé de changer le comportement par défaut (`on`), même s'il peut désorienter au premier abord des personnes ayant connu une base de données concurrente.

Une autre possibilité est d'utiliser `psql -1` ou `psql --single-transation` : les ordres sont automatiquement encadrés d'un `BEGIN` et d'un `COMMIT`. La présence d'ordres `BEGIN`, `COMMIT` ou `ROLLBACK` explicites modifiera ce comportement en conséquence.

4.5.3 Écrire un script SQL

- Attention à l'encodage des caractères
  - `\encoding`
  - `SET client_encoding`
- Écriture des requêtes

L'encodage a moins d'importance depuis qu'UTF-8 s'est généralisé, mais il y a encore parfois des problèmes dans de vieilles bases ou certains vieux outils.

Rappelons que les bases modernes devraient toutes utiliser l'encodage UTF-8 (c'est le défaut).

`\encoding [ENCODAGE]` permet, en l'absence d'argument, d'afficher l'encodage du client. En présence d'un argument, il permet de préciser l'encodage du client.

Exemple :

```
postgres=# \encoding
UTF8
postgres=# \encoding LATIN9
postgres=# \encoding
LATIN9
```

Cela a le même effet que d'utiliser l'ordre SQL `SET client_encoding TO LATIN9;`.

En terme de présentation, il est commun d'écrire les mots clés SQL en majuscules et d'écrire les noms des objets et fonctions manipulés dans les requêtes en minuscule. Le langage SQL est un langage au même titre que Java ou PHP, la présentation est importante pour la lisibilité des requêtes, même si les variations personnelles sont nombreuses.

4.5.4 Les blocs anonymes

- 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`[19].)

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).

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

4.5.5 Utiliser des variables

```
\set nom_table 'ma_table'
SELECT * FROM :"nom_table";

\set valeur_col1 'test'
SELECT * FROM :"nom_table" WHERE col1 = :'valeur_col1';

\prompt 'invite' nom_variable
\unset variable

psql -v VARIABLE=valeur
```

psql permet de manipuler des variables internes personnalisées dans les scripts. Ces variables peuvent être particulièrement utiles pour passer des noms d'objets ou des termes à utiliser dans une requête par le biais des options de ligne de commande (`-v variable=valeur`).

Noter la position des guillemets quand la variable est une chaîne de caractères !

Exemple :

Une fois connecté à la base **pgbench**, on déclare deux variables propres au client :

```
pgbench=# \set nomtable pgbench_accounts
pgbench=# \set taillemini 1000000
```

Elles apparaissent bien dans la liste des variables :

```
pgbench=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'pgbench'
…
nomtable = 'pgbench_accounts'
taillemini = '1000000'
```

Elles s'utilisent ainsi :

```
SELECT pg_relation_size (:'nomtable') ;

 pg_relation_size
------------------
        134299648

SELECT relname, pg_size_pretty(pg_relation_size (oid))
FROM pg_class
WHERE relkind = 'r' AND pg_relation_size (oid) > :taillemini ;
```

```
     relname       | pg_size_pretty
------------------+----------------
 pgbench_accounts | 128 MB
```

La substitution s'effectue bien au niveau du client. Si l'on trace tout au niveau du serveur, ces requêtes apparaissent :

```
SELECT pg_relation_size ('pgbench_accounts') ;

SELECT relname, pg_size_pretty(pg_relation_size (oid))
FROM pg_class WHERE relkind = 'r'
AND pg_relation_size (oid) > 1000000 ;
```

4.5.6 Gestion des erreurs

- Ignorer les erreurs dans une transaction
  - `ON_ERROR_ROLLBACK`
- Gérer des erreurs SQL en shell
  - `ON_ERROR_STOP`

La variable interne `ON_ERROR_ROLLBACK` n'a de sens que si elle est utilisée dans une transaction. Elle peut prendre trois valeurs :

- `off` (défaut) ;
- `on` ;
- `interactive`.

Lorsque `ON_ERROR_ROLLBACK` est à `on`, psql crée un `SAVEPOINT` systématiquement avant d'exécuter une requête SQL. Ainsi, si la requête SQL échoue, psql effectue un `ROLLBACK TO SAVEPOINT` pour annuler cette requête. Sinon il relâche le `SAVEPOINT`.

Lorsque `ON_ERROR_ROLLBACK` est à `interactive`, le comportement de psql est le même seulement si il est utilisé en interactif. Si psql exécute un script, ce comportement est désactivé. Cette valeur permet de se protéger d'éventuelles fautes de frappe.

Utiliser cette option n'est donc pas neutre, non seulement en terme de performances, mais également en terme d'intégrité des données. Il ne faut donc pas utiliser cette option à la légère.

Enfin, la variable interne `ON_ERROR_STOP` a deux objectifs : arrêter l'exécution d'un script lorsque psql rencontre une erreur et retourner un code retour shell différent de 0. Si cette variable reste à `off`, psql retournera toujours la valeur 0 même s'il a rencontré une erreur dans l'exécution d'une requête.

Une fois activée, psql retournera un code d'erreur 3 pour signifier qu'il a rencontré une erreur dans l'exécution du script.

L'exécution d'un script qui comporte une erreur retourne le code 0, signifiant que psql a pu se connecter à la base de données et exécuté le script :

```
$ psql -f script_erreur.sql postgres
psql:script_erreur.sql:1: ERROR:  relation "vin" does not exist
LINE 1: SELECT * FROM vin;
                      ^
$ echo $?
0
```

Lorsque la variable `ON_ERROR_STOP` est activée, psql retourne un code erreur 3, signifiant qu'il a rencontré une erreur :

```
$ psql -v ON_ERROR_STOP=on -f script_erreur.sql postgres
psql:script_erreur.sql:1: ERROR:  relation "vin" does not exist
LINE 1: SELECT * FROM vin;
                      ^
$ echo $?
3
```

psql retourne les codes d'erreurs suivant au shell :

- 0 au shell s'il se termine normalement ;
- 1 s'il y a eu une erreur fatale de son fait (pas assez de mémoire, fichier introuvable) ;
- 2 si la connexion au serveur s'est interrompue ou arrêtée ;
- 3 si une erreur est survenue dans un script et si la variable `ON_ERROR_STOP` a été initialisée.

4.5.7 Formatage des résultats

- Sortie simplifiée pour exploitation automatisée : `-XAt`
  - `-t` (`--tuples-only`)
  - `-A` (`--no-align`)
  - `-X` (`--no-psqlrc`)
  - séparateurs : `-F` (`--field-separator`) et `-R` (`--record-separator`)
- Formats HTML ou CSV
  - `-H | --html`
  - `--csv` (à partir de la version 12)

`psql` peut servir à afficher des rapports basiques et possède quelques options de formatage.

L'option `--csv` suffit à répondre à ce besoin, à partir d'un client en version 12 au moins.

S'il faut définir plus finement le format, il existe des options. `-A` impose une sortie non alignée des données. En ajoutant `-t`, qui supprime l'entête, et `-X`, qui demande à ignorer le `.psqlrc`, la sortie peut être facilement exploitée par des outils classiques comme `awk` ou `sed` :

```
$ psql -XAt -c 'select datname, pg_database_size(datname) from pg_database'
postgres|87311139
powa|765977379
template1|9028387
template0|8593923
pgbench|166134563
```

Le séparateur `|` peut être remplacé par un autre avec `-F`, ou un octet nul avec `-z`, et le retour chariot de fin de ligne par une chaîne définie avec `-R`, ou un octet nul avec `-0`.

`-H` permet une sortie en HTML pour une meilleure lisibilité par un humain.

4.5.8 Résultats en pivot (tableau croisé)

- `\crosstabview [colV [colH [colD [colonnedetriH]]]]`
- Exécute la requête en tampon
  - au moins 3 colonnes

Par exemple, pour voir les différents types de clients connectés aux bases (clients système inclus), le résultat n'est pas très lisible :

```
=# \pset null ø

=# SELECT datname, backend_type, COUNT(*) as nb  FROM pg_stat_activity
   GROUP BY 1,2
   ORDER BY datname NULLS LAST, backend_type ;

 datname  |         backend_type         | nb
----------+------------------------------+----
 pgbench  | client backend               |  2
 postgres | client backend               |  1
 powa     | powa                         |  1
 ø        | archiver                     |  1
 ø        | autovacuum launcher          |  1
 ø        | background writer            |  1
 ø        | checkpointer                 |  1
 ø        | logical replication launcher |  1
 ø        | pg_wait_sampling collector   |  1
```

```
 ø         | walwriter                    |  1
(10 lignes)
```

On peut le reformater ainsi :

```
=# \crosstabview backend_type datname

         backend_type         | postgres | ø | powa | pgbench
------------------------------+----------+---+------+---------
 client backend               |        2 |   |      |       1
 walwriter                    |          | 1 |      |
 autovacuum launcher          |          | 1 |      |
 logical replication launcher |          | 1 |      |
 powa                         |          |   |    1 |
 background writer            |          | 1 |      |
 archiver                     |          | 1 |      |
 checkpointer                 |          | 1 |      |
(9 lignes)
```

4.5.9 Formatage dans les scripts SQL

- Donner un titre au résultat de la requête
  - `\pset title 'Résultat de la requête'`
- Formater le résultat
  - `\pset format html` (ou `csv`...)
- Diverses options peu utilisées

Il est possible de réaliser des modifications sur le format de sortie des résultats de requête directement dans le script SQL ou en mode interactif dans psql.

Afficher `\pset` permet de voir ces différentes options. La complétion automatique après `\pset` affiche les paramètres et valeurs possibles.

Par exemple, l'option `format` est par défaut à `aligned` mais possède d'autres valeurs :

```
=# \pset format <TAB>

aligned          csv              latex            troff-ms         wrapped
asciidoc         html             latex-longtable  unaligned
```

D'autres options existent, peu utilisées. La liste complète des options de formatage et leur description est disponible dans la documentation de la commande psql[20].

[20] https://docs.postgresql.fr/current/app-psql.html

4.5.10 Scripts & Crontab

- `cron`
  - Attention aux variables d'environnement !
- Ou tout ordonnanceur

La planification d'un script périodique s'effectue de préférence avec les outils du système, donc sous Unix avec `cron` ou une de ses variantes, même si n'importe quel ordonnanceur peut convenir.

Avec `cron`, il faut se rappeler qu'à l'exécution d'un script, l'environnement de l'utilisateur n'est pas initialisé, ou plus simplement, les fichiers de personnalisation (par ex. `.bashrc`) de l'environnement ne sont pas lus. Seule la valeur `$HOME` est initialisée. Un script fonctionnant parfaitement dans une session peut échouer une fois planifié. Il faut donc prévoir ce cas, initialiser les variables d'environnement requises de façon adéquate, et bien tester.

Par exemple, pour charger l'environnement de l'utilisateur :

```
#!/bin/bash

. ${HOME}/.bashrc

…
```

Rappelons que chaque utilisateur du système peut avoir ses propres crontab. L'utilisateur peut les visualiser avec la commande `crontab -l` et les éditer avec la commande `crontab -e`.

4.5.11 Exemple de script de sauvegarde

- Sauvegarder une base et classer l'archive (squelette) :

```
#!/bin/bash
# Paramètre : la base
t=$(mktemp)                      # fichier temporaire
pg_dump -Fc "$1" > $t            # sauvegarde
d=$(eval date +%d%m%y-%H%M%S)    # date
mv $t /backup/"${1}_${d}.dump"   # déplacement
exit 0
```

- …et ajouter la gestion des erreurs !
- …et les surveiller

Il est délicat d'écrire un script fiable. Ce script d'exemple possède plusieurs problèmes potentiels si le paramètre (la base) manque, si la sauvegarde échoue, si l'espace disque manque dans `/tmp`, si le déplacement échoue, si la partition cible n'est pas montée…

Parmi les outils existants, nous évoquerons notamment pg_back lors des sauvegardes[21].

Par convention, un script doit renvoyer 0 s'il s'est déroulé correctement.

[21] https://dali.bo/i1_html

4.6 OUTILS GRAPHIQUES

- Outils graphiques d'administration
  - temBoard
  - pgAdminIII et pgAdmin 4
  - pgmodeler

Il existe de nombreux outils graphiques permettant d'administrer des bases de données PostgreSQL. Certains sont libres, d'autres propriétaires. Certains sont payants, d'autres gratuits. Ils ont généralement les mêmes fonctionnalités de base, mais vont se distinguer sur certaines fonctionnalités un peu plus avancées comme l'import et l'export de données.

Nous allons étudier ici plusieurs outils proposés par la communauté, temBoard, pgAdmin.

4.6.1 temBoard

- Adresse: https://labs.dalibo.com/temboard
- Licence: PostgreSQL
- Notes: Serveur sur Linux, client web

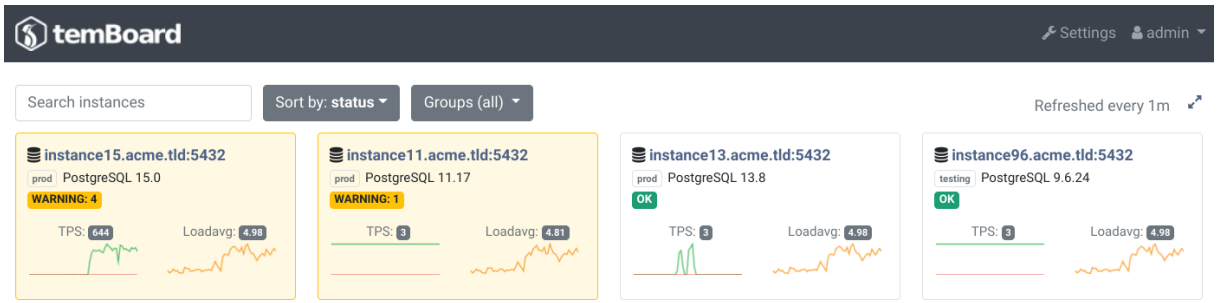
4.6.2 temBoard - PostgreSQL Remote Control

- Multi-instances
- Surveillance OS / PostgreSQL
- Suivi de l'activité
- Gestion des performances de PostgreSQL
- Configuration de chaque instance

temBoard est un outil permettant à un DBA de mener à bien la plupart de ses tâches courantes.

Le serveur web est installé de façon centralisée et un agent est déployé pour chaque instance.

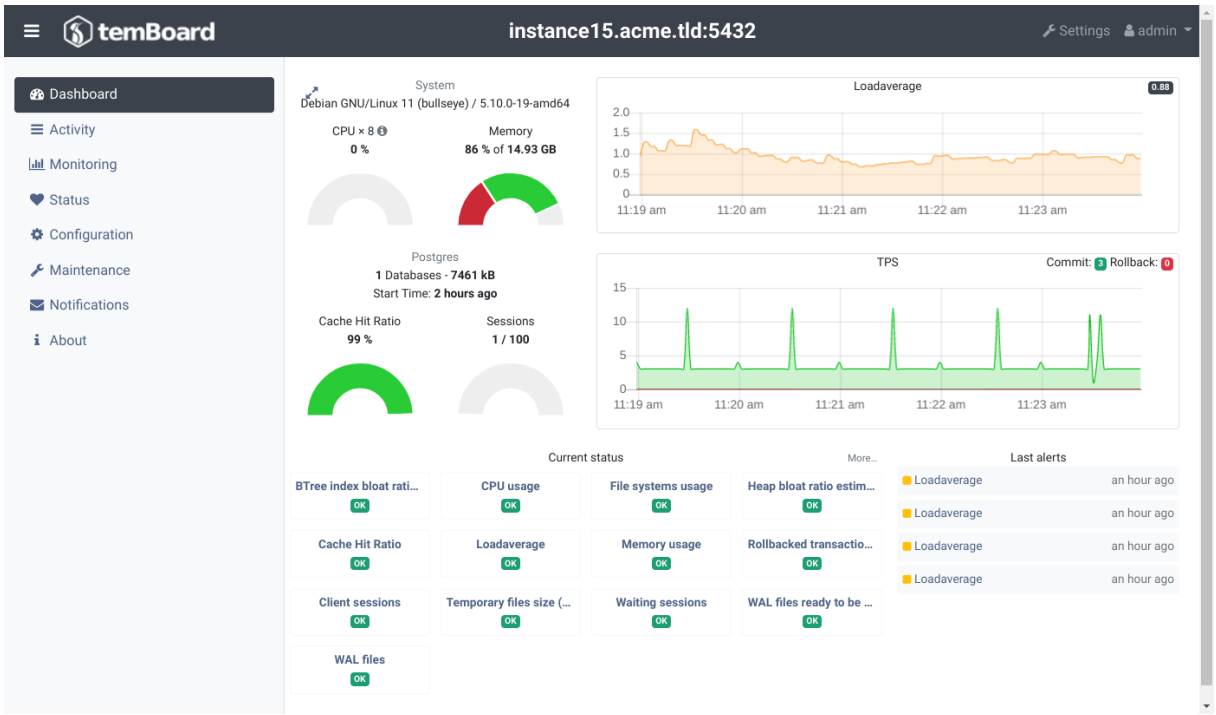
4.6.3 temBoard - Vue parc



temboard fournit un tableau de bord global.

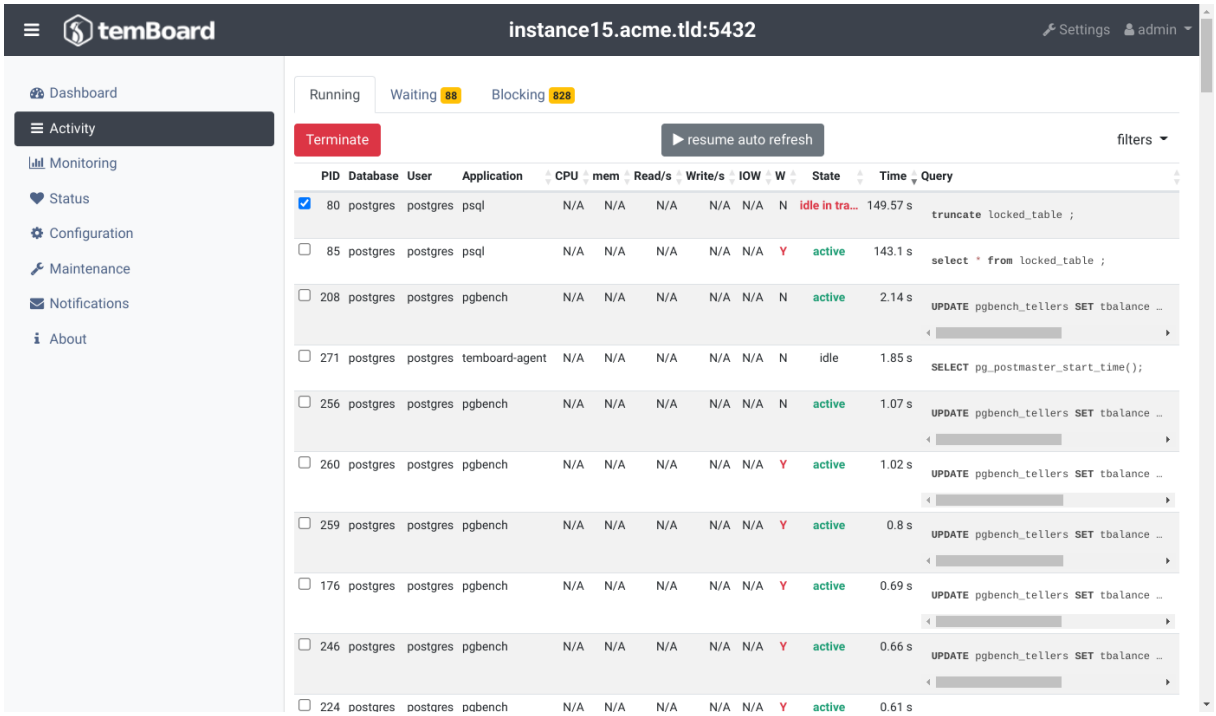
temBoard peut administrer plusieurs centaines d'instances.

4.6.4 temBoard - Tableau de bord



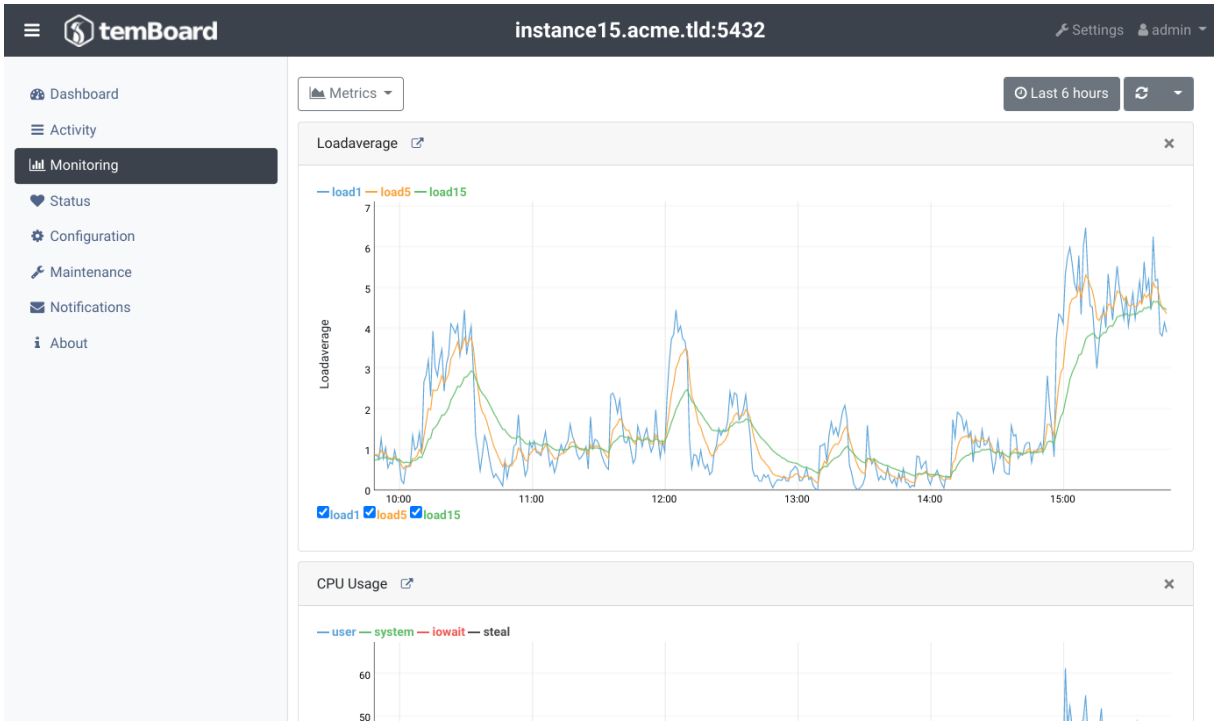
temboard présente un tableau de bord par instance.

4.6.5 temBoard - Activity



La section **Activity** permet de lister toutes les requêtes courantes (**Running**), les requêtes bloquées (**Waiting**) ou bloquantes (**Blocking**). Il est possible à partir de cette vue de terminer une session.

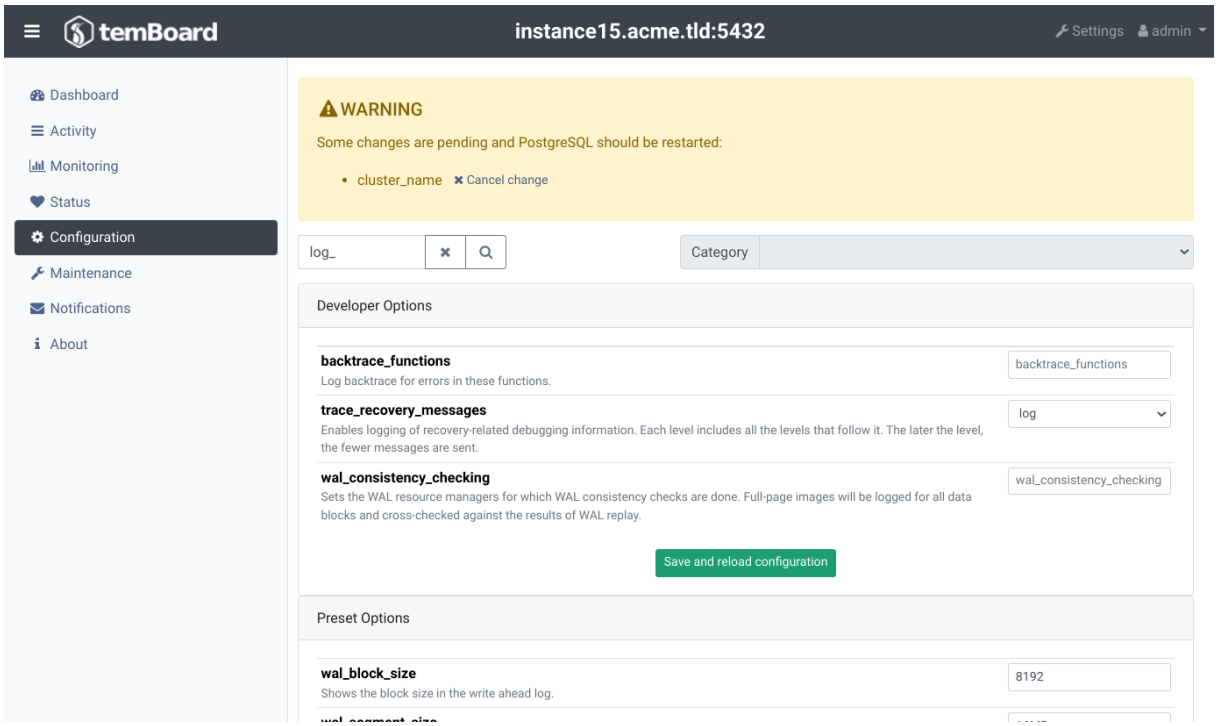
4.6.6 temBoard - Supervision



La section **Monitoring** permet de visualiser les graphiques historisés au niveau du système d'exploitation (CPU, mémoire, …) ainsi qu'au niveau de l'instance PostgreSQL.

temBoard inclut un système d'alerte par courriel et SMS lorsqu'une métrique dépasse un seuil.

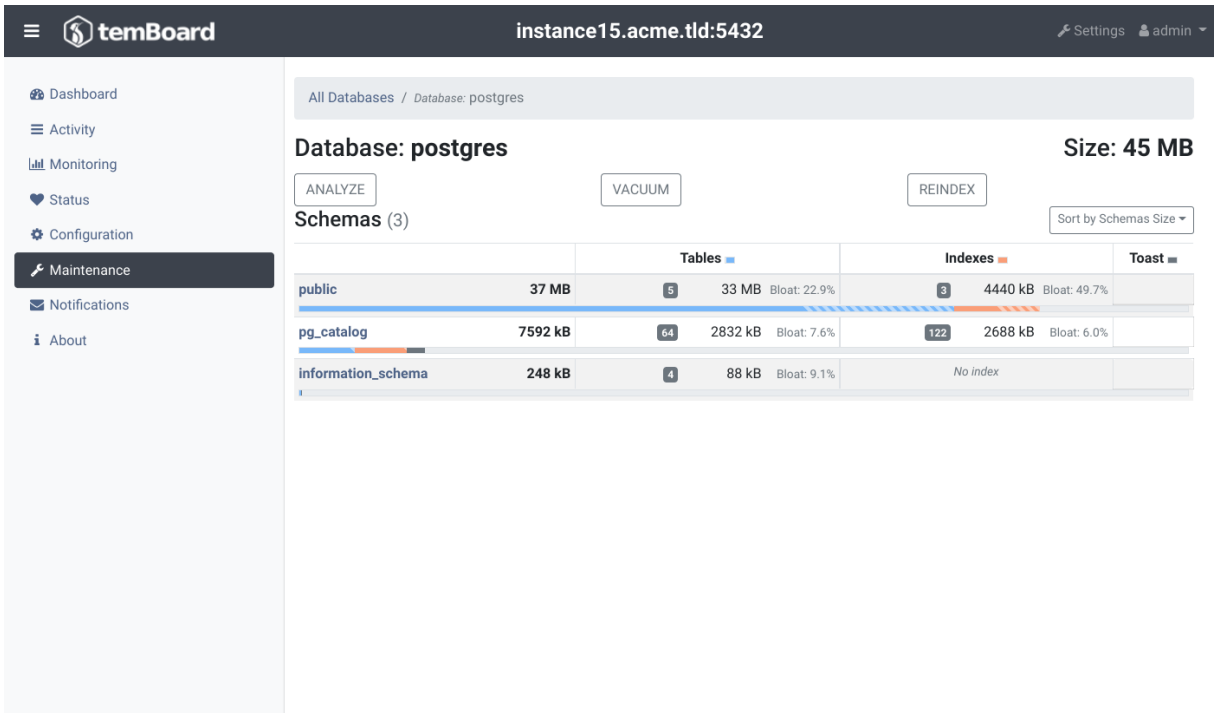
4.6.7 temBoard - Configuration



La section *Configuration* permet naviguer dans les paramètres de PostgreSQL de modifier ces paramètres

Suivant les cas, il sera proposé de recharger la configuration ou de redémarrer l'instance pour appliquer ces changements.

4.6.8 temBoard - Maintenance



temBoard estime la fragmentation du stockage des tables et des index.

Une interface permet de visualiser la fragmentation et d'agir dessus avec une granularité fine.

4.6.9 pgAdmin 4

- https://www.pgadmin.org
- Application web
- Licence : PostgreSQL
- Multiplateforme, multilangue

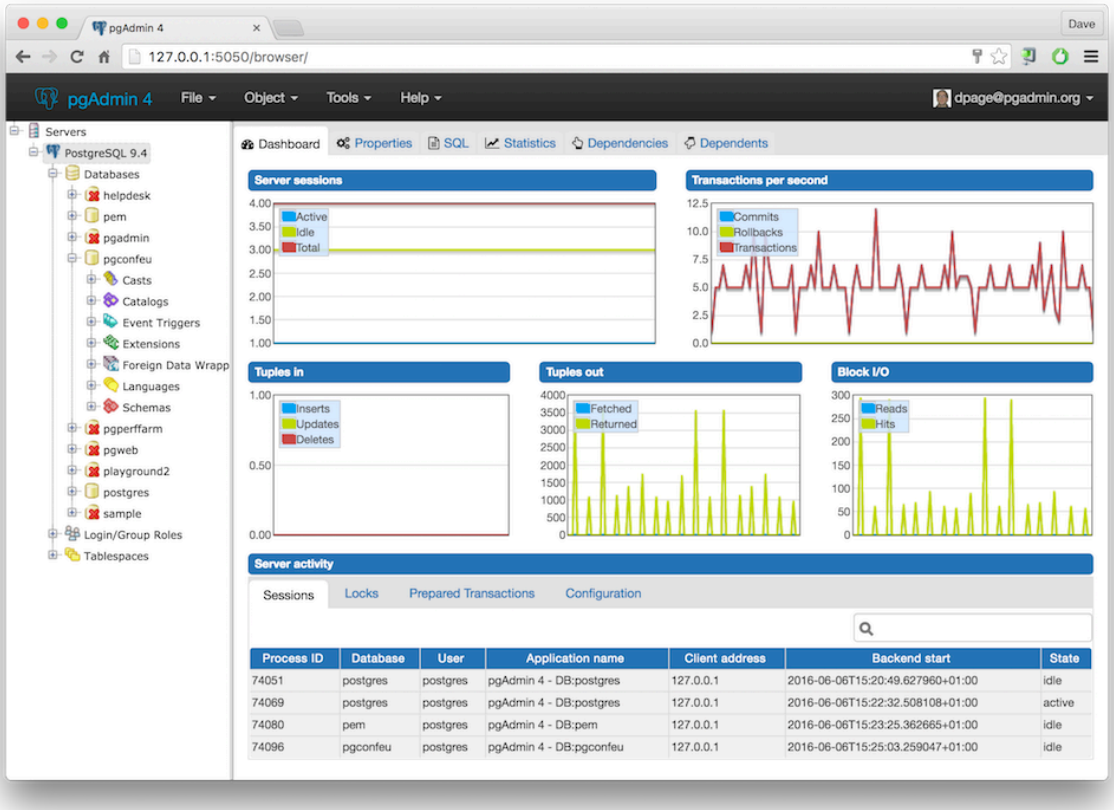
pgAdmin 4 est une application web, même si une version émulant un client lourd existe. Après un début difficile, le produit est à présent mature. Il reprend l'essentiel des fonctionnalités de pgAdmin III. Il est bien entendu compatible avec les dernières versions de PostgreSQL.

Il peut être déployé sur Windows et macOS X et bien sûr Linux, où il faudra utiliser les dépôts fournis par le projet[22], ou l'image docker.

Il est disponible sous licence PostgreSQL.

[22] https://www.pgadmin.org/download/

4.6.10 pgAdmin 4 : tableau de bord



Une des nouvelles fonctionnalités de pgAdmin 4 est l'apparition d'un tableau de bord remontant quelques métriques intéressantes. Il permet aussi la visualisation des géométries PostGIS, ce qui est parfois très utile.

4.6.11 DBeaver

- https://dbeaver.io
- Version Community sous Licence Apache 2.0
- Application Java Multiplateforme
- Version web *CloudBeaver* aussi disponible
- Supporte PostgreSQL (et ~ 80 autres SBGD)
- Version Pro payante et plus complète

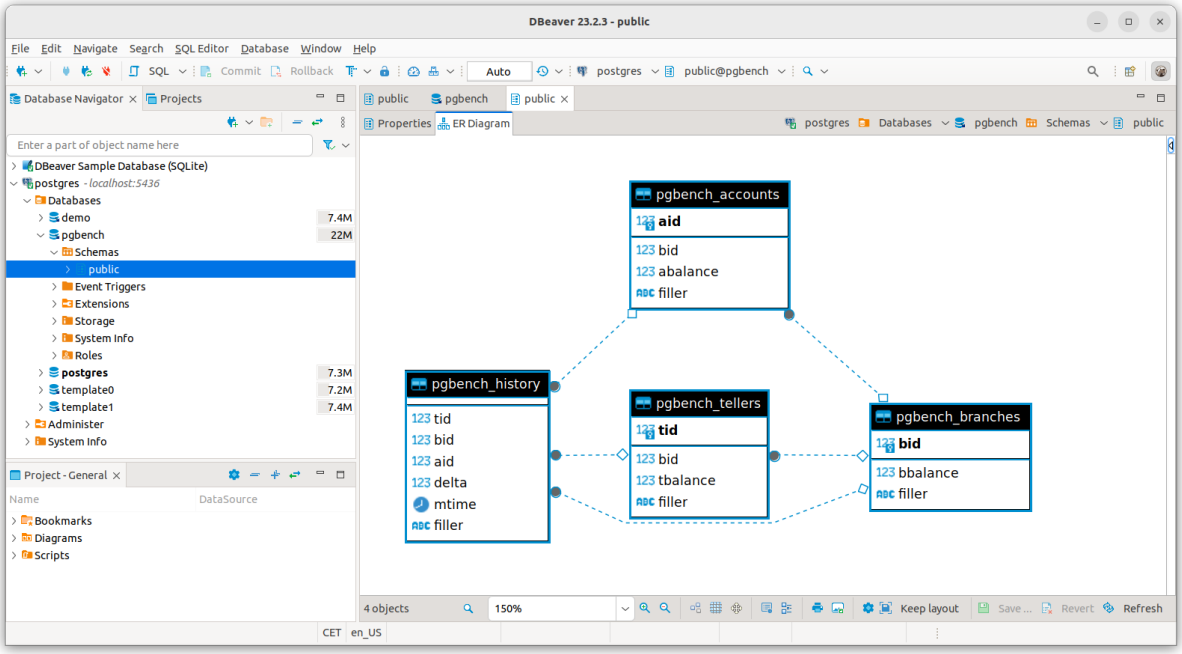
DBeaver est un outil graphique basé sur Eclipse. Il permet l'administration de plus de 80 SGBDs différents, y compris PostgreSQL. Le projet a sa propre entreprise depuis 2017 ainsi qu'une version payante plus complète. La version open-source *Community Edition* est disponible sous licence Apache 2.0. Une version web de DBeaver est disponible sous le nom de *CloudBeaver* (avec une page de démonstration publique[23]).

La version Pro, disponible sous différentes déclinaisons, propose des connecteurs propriétaires supplémentaires vers des systèmes de bases de données non relationnels (MongoDB, Redis, Cassandra, etc.), une gestion native des bases de données Cloud (AWS, Azure, etc.), des fonctionnalités dédiées aux entreprises ainsi qu'un support de la part de l'éditeur.

Il existe des paquets d'installation pour les systèmes d'exploitation les plus courants[24].

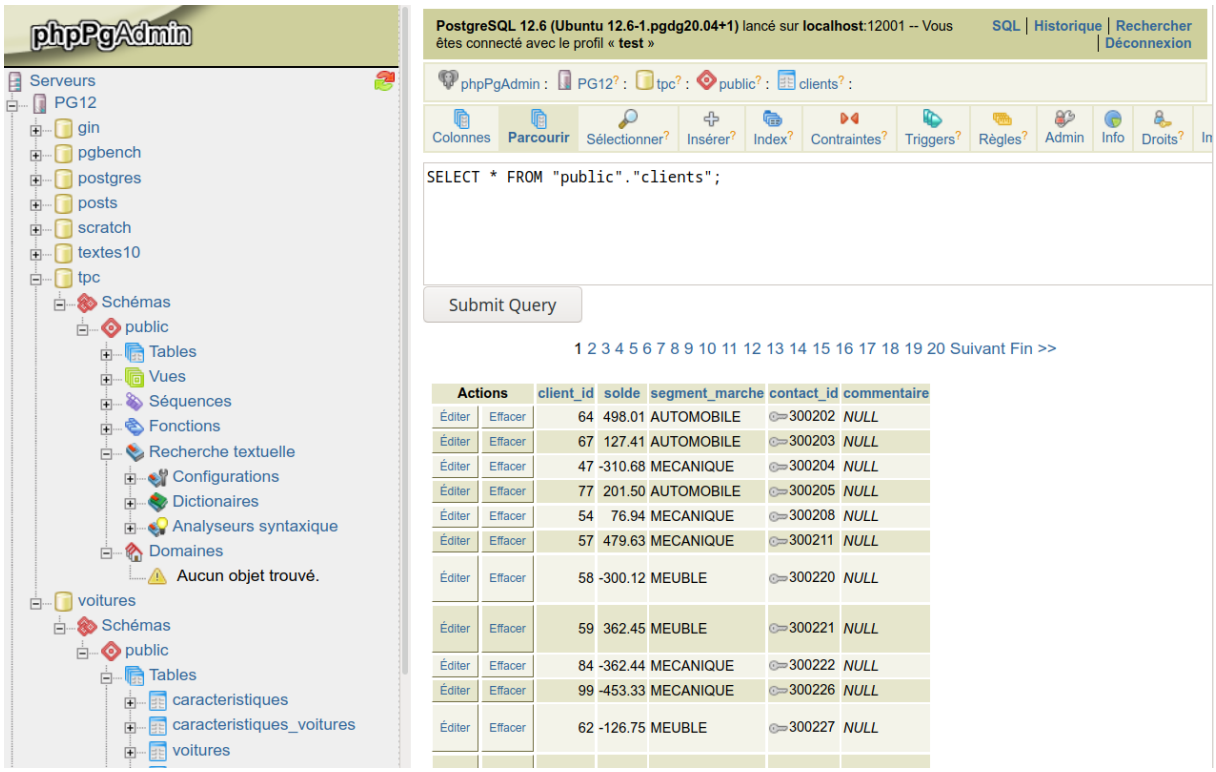
[23] https://demo.cloudbeaver.io/
[24] https://dbeaver.io/download/

4.6.12 DBeaver : fenêtre principale



La vue principale est divisée en deux sections, le côté gauche référence les différentes instances et le côté droit permet de consulter les différentes informations d'une instance en particulier. Ici, nous avons une vue entité-association des tables crées avec l'outil `pgbench` quand l'option `--foreign-keys` est spécifiée.

4.6.13 phpPgAdmin



4.6.14 phpPgAdmin : fonctionnalités

- Licence: GNU Public License
- Application web, simple
  - consultation, édition
  - sauvegarde, export
- Pérennité ?

PhpPgAdmin est une application web en PHP, légère et simple d'emploi, que l'on peut ouvrir à un utilisateur non informaticien pour y modifier des données.

Le projet PhpPgAdmin a malheureusement des problèmes de maintenance récurrents. Le « nouveau » dépôt officiel[25] semble abandonné. Ce fork[26] semble le plus actif, et il sert de base entre autres aux

[25] https://github.com/phppgadmin/phppgadmin
[26] https://github.com/ReimuHakurei/phpPgAdmin

paquets Debian et OpenSUSE. Il est compatible avec PostgreSQL 16 et PHP 8.2.

Notre conseil est de préférer la version web de pgAdmin 4, qui est beaucoup plus lourd, mais plus puissant et plus pérenne. Du moins s'il convient au public voulu.

4.6.15 AdminerEvo

4.6.16 AdminerEvo : fonctionnalités

- https://docs.adminerevo.org/
- Application web pour utilisateurs
- Basique mais simple & efficace
- Et simple : 1 fichier PHP
- Multibases, multilangues
- Licence : Apache License ou GPL 2

AdminerEvo prend la suite de Adminer[27] qui n'est plus maintenu et ne doit plus être utilisé. C'est une

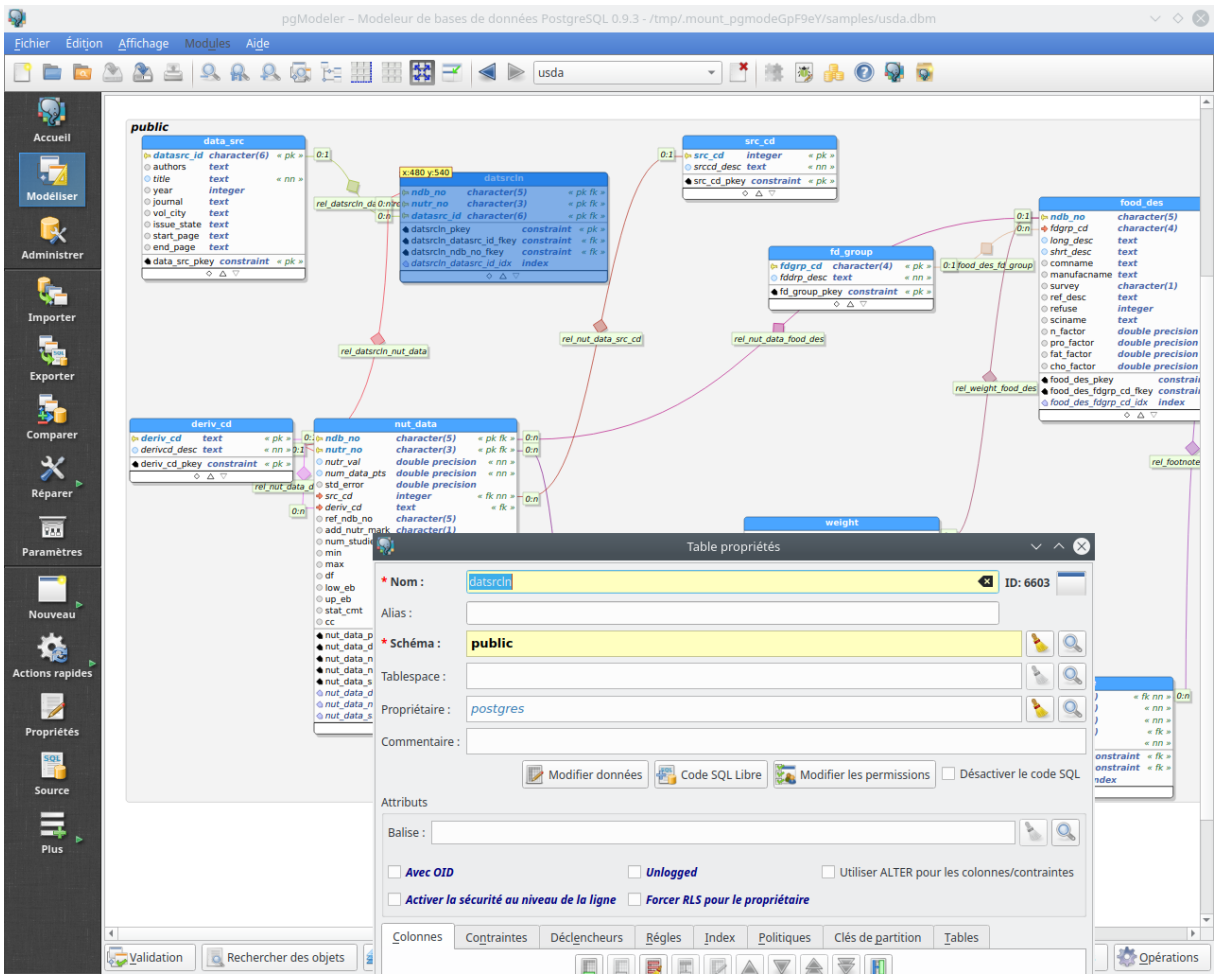
[27] https://www.adminer.org/

application web à destination des utilisateurs, pouvant gérer plusieurs types de bases, dont PostgreSQL.

Il utilise en un unique fichier PHP (éventuellement personnalisable par CSS). On récupérera simplement le dépôt Github[28].

L'interface est très basique. Elle peut sembler datée, voire primitive, mais elle est très simple et regroupe efficacement l'essentiel des fonctionnalités. C'est un candidat au remplacement de phpPgAdmin pour des utilisateurs non techniques.

4.6.17 pgModeler



[28] https://github.com/adminerevo/adminerevo

4.6.18 pgModeler

- Site officiel : https://pgmodeler.io/
- Licence : GPLv3
- Modélisation de base de données
- Fonctionnalité d'import export
- Comparaison de base

pgModeler permet de modéliser une base de données. Son intérêt par rapport à d'autres produits concurrents est qu'il est spécialisé pour PostgreSQL. Il en supporte donc toutes les spécificités, comme l'héritage de tables, les types composites, les types tableaux… C'est une excellente solution pour modéliser une base en partant de zéro, ou pour extraire une visualisation graphique d'une base existante.

Il est à noter que, bien que le projet soit libre, son installation par les sources peut être laborieuse, et les paquets ne sont pas forcément disponibles. L'équipe de développement propose des paquets binaires à prix modique.

4.7 CONCLUSION

- Les outils en ligne de commande sont « rustiques » mais puissants
- Ils peuvent être remplacés par des outils graphiques
- En cas de problème, il est essentiel de les maîtriser.

4.7.1 Questions

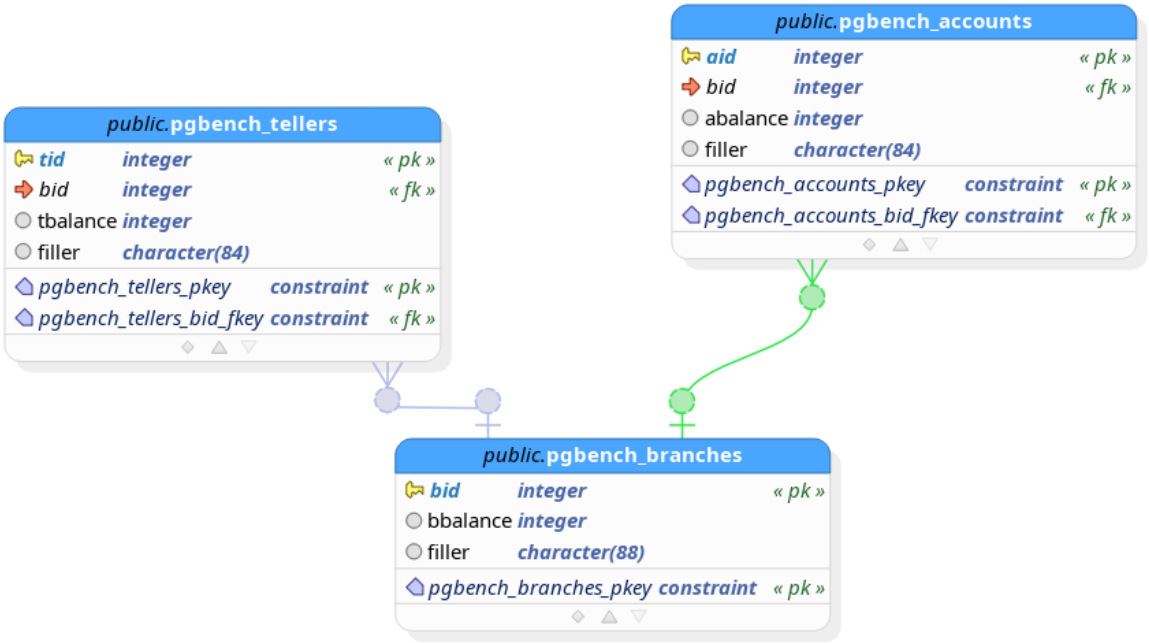
N'hésitez pas, c'est le moment !

4.8 QUIZ

https://dali.bo/de_quiz

4.9 INTRODUCTION À PGBENCH

pgbench est un outil de test livré avec PostgreSQL. Son but est de faciliter la mise en place de benchmarks simples et rapides. Par défaut, il installe une base très simple, génère une activité plus ou moins intense et calcule le nombre de transactions par seconde et la latence. C'est ce qui sera fait ici dans cette introduction. On peut aussi lui fournir ses propres scripts.



La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[29].

4.9.1 Installation

L'outil est installé avec les paquets habituels de PostgreSQL, client ou serveur suivant la distribution.

Sur les distributions à paquets RPM (RockyLinux…), l'outil n'est pas dans le chemin par défaut, il faudra fournir le chemin complet (qui ne sera pas répété ici):

```
/usr/pgsql-17/bin/pgbench
```

Il est préférable de créer un rôle non privilégié dédié, qui possédera la base de donnée :

```
CREATE ROLE pgbench LOGIN PASSWORD 'unmotdepassebienc0mplexe';
CREATE DATABASE pgbench OWNER pgbench ;
```

[29] https://youtu.be/aTwh_CgRaE0

Le `pg_hba.conf` doit éventuellement être adapté. La base par défaut s'initialise ainsi (ajouter `--port` et `--host` au besoin) :

```
pgbench -U -d pgbench --initialize --scale=100  pgbench
```

`--scale` permet de faire varier proportionnellement la taille de la base. À 100, la base pèsera 1,5 Go, avec 10 millions de lignes dans la table principale `pgbench_accounts` :

```
pgbench@pgbench=# \d+
                            Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Taille  | Description
--------+------------------+-------+--------------+---------+-------------
 public | pg_buffercache   | vue   | postgres     | 0 bytes |
 public | pgbench_accounts | table | pgbench      | 1281 MB |
 public | pgbench_branches | table | pgbench      | 40 kB   |
 public | pgbench_history  | table | pgbench      | 0 bytes |
 public | pgbench_tellers  | table | pgbench      | 80 kB   |
```

4.9.2 Générer de l'activité

Pour simuler une activité de 20 clients simultanés, répartis sur 4 processeurs, pendant 100 secondes :

```
pgbench -U pgbench -c 20 -j 4 -T100 pgbench
```

Pour afficher, ajouter `--debug` :

```
UPDATE pgbench_accounts SET abalance = abalance + -3455 WHERE aid = 3789437;
SELECT abalance FROM pgbench_accounts WHERE aid = 3789437;
UPDATE pgbench_tellers SET tbalance = tbalance + -3455 WHERE tid = 134;
UPDATE pgbench_branches SET bbalance = bbalance + -3455 WHERE bid = 78;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
 VALUES (134, 78, 3789437, -3455, CURRENT_TIMESTAMP);
```

À la fin, s'affichent notamment le nombre de transactions (avec et sans le temps de connexion) et la durée moyenne d'exécution du point de vue du client (`latency`) :

```
scaling factor: 100
query mode: simple
number of clients: 20
number of threads: 4
duration: 10 s
number of transactions actually processed: 20433
latency average = 9.826 ms
tps = 2035.338395 (including connections establishing)
tps = 2037.198912 (excluding connections establishing)
```

Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS='-c synchronous_commit=off -c commit_siblings=20' \
            pgbench -U pgbench -c 20 -j 4 -T100 pgbench  2>/dev/null
```

```
latency average = 6.992 ms
tps = 2860.465176 (including connections establishing)
tps = 2862.964803 (excluding connections establishing)
```

Des tests rigoureux doivent durer bien sûr beaucoup plus longtemps que 100 s, par exemple pour tenir compte des effets de cache, des checkpoints périodiques, etc.

4.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/de_solutions.

But :

- Acquérir certains automatismes dans l'utilisation de `psql`
- Créer des premières bases de données

L'ensemble des informations permettant de résoudre ces exercices a été abordé au cours de la partie théorique. Il est également possible de trouver les réponses dans le manuel de psql (`man psql`) ou dans l'aide en ligne de l'outil.

Il est important de bien discerner les différents utilisateurs impliqués au niveau système et PostgreSQL.

> Ouvrir plusieurs fenêtres ou consoles : au moins une avec l'utilisateur habituel (**dalibo** ici), une avec **root**, une avec l'utilisateur système **postgres**, une pour suivre le contenu des traces (`postgresql*.log`).

Nouvelle base bench :

> En tant qu'utilisateur système **postgres**, et avec l'utilitaire en ligne de commande `createdb`, créer une base de données nommée **bench** (elle appartiendra à **postgres**).

> Avec `psql`, se connecter à la base **bench** en tant qu'utilisateur **postgres**.

> Lister les bases de l'instance.

> Se déconnecter de PostgreSQL.

Voir les tables :

> Pour remplir quelques tables dans la base **bench**, on utilise un outil de *bench* livré avec PostgreSQL :
>
> ```
> /usr/pgsql-17/bin/pgbench -i --foreign-keys bench
> ```

> Quelle est la taille de la base après alimentation ?

> Afficher la liste des tables de la base **bench** et leur taille.

> Quelle est la structure de la table `pgbench_accounts` ?

> Afficher l'ensemble des autres objets non système (index, séquences, vues…) de la base.

Nouvel utilisateur :

> Toujours en tant qu'utilisateur système **postgres**, avec l'utilitaire `createuser`, créer un rôle **dupont** (il doit avoir l'attribut `LOGIN` !).

> Sous psql, afficher la liste des rôles (utilisateurs).

Voir les objets système :

> Dans la base **bench**, afficher l'ensemble des tables systèmes (schéma `pg_catalog`).

> Afficher l'ensemble des vues systèmes (schéma `pg_catalog`).

Manipuler les données :

> Le but est de créer une copie de la table `pgbench_tellers` de la base **bench** avec `CREATE TABLE AS`. Afficher l'aide de cette commande.

> Créer une table `pgbench_tellers_svg`, copie de la table `pgbench_tellers`.

> Sortir le contenu de la table `pgbench_tellers` dans un fichier `/tmp/pgbench_tellers.csv` (commande `\copy`).

> Quel est le répertoire courant ? Sans quitter `psql`, se déplacer vers `/tmp/`, et en lister le contenu.

> Afficher le contenu du fichier `/tmp/pgbench_tellers.csv` depuis `psql`.

> Créer un fichier `décompte.sql`, contenant 3 requêtes pour compter le nombre de lignes dans les 3 tables de **bench** :
>
> - Il devra écrire dans le fichier `/tmp/décompte.txt`.
> - Le faire exécuter par `psql`.

Détruire la base :

> Supprimer la base **bench**.

4.11 TRAVAUX PRATIQUES (SOLUTIONS)

Les copies d'écran qui suivent proviennent d'un PostgreSQL 15 sous RockyLinux 8 paramétré en anglais. Elles peuvent différer légèrement selon la version de PostgreSQL, l'OS et la langue.

> Ouvrir plusieurs fenêtres ou consoles : au moins une avec l'utilisateur habituel (**dalibo** ici), une avec **root**, une avec l'utilisateur système **postgres**, une pour suivre le contenu des traces (`postgresql*.log`).

Pour devenir **root** :

```
$ sudo su -
```

Pour devenir **postgres** :

```
$ sudo -iu postgres
```

Pour voir le contenu des traces défiler, se connecter dans une nouvelle fenêtre à nouveau en tant que **postgres**, et aller chercher le fichier de traces. Sur Red Hat/CentOS, il est par défaut dans `$PGDATA/log` et son nom exact varie chaque jour :

```
$ sudo -iu postgres
$ ls -l /var/lib/pgsql/15/data/log
-rw-------. 1 postgres postgres 4462 Jan  6 11:12 postgresql-Fri.log
$ tail -f  /var/lib/pgsql/15/data/log/postgresql-Tue.log
```

Par défaut ne s'afficheront que peu de messages : arrêt/redémarrage, erreur de connexion… Laisser la fenêtre ouverte en arrière-plan ; elle servira à analyser les problèmes.

Nouvelle base bench :

> En tant qu'utilisateur système **postgres**, et avec l'utilitaire en ligne de commande `createdb`, créer une base de données nommée **bench** (elle appartiendra à **postgres**).

Si vous n'êtes pas déjà **postgres** :

```
$ sudo -iu postgres
```

```
$ createdb --echo bench
SELECT pg_catalog.set_config('search_path', '', false)
CREATE DATABASE bench;
```

Noter que `createdb` ne fait que générer un ordre SQL. On peut aussi aussi directement exécuter cet ordre depuis `psql` sous un compte superutilisateur.

> Avec `psql`, se connecter à la base **bench** en tant qu'utilisateur **postgres**.

```
$ psql -d bench -U postgres
psql (15.1)
Type "help" for help.

bench=#
```

> Lister les bases de l'instance.

```
bench=# \l
```

```
bench=# \l
                                                    List of databases
  Name     |  Owner   | Encoding |   Collate   |    Ctype    | …|   Access privileges
----------+----------+----------+-------------+-------------+--+-------------------
 ↪  ---
bench     | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  |
postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  |
template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  | =c/postgres         +
          |          |          |             |             |  | postgres=CTc/postgres
template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  | =c/postgres         +
          |          |          |             |             |  | postgres=CTc/postgres
(4 rows)
```

(Deux colonnes indiquant à la source des librairies pour les collations ont été masquées.)

Noter que depuis le shell, le même résultat est renvoyé par :

```
$ psql -l
```

> Se déconnecter de PostgreSQL.

```
bench=# \q
```

(`exit` et **Ctrl-D** fonctionnent également.)

Voir les tables :

> Pour remplir quelques tables dans la base **bench**, on utilise un outil de *bench* livré avec PostgreSQL :
>
> `/usr/pgsql-17/bin/pgbench -i --foreign-keys bench`

L'outil est livré avec PostgreSQL, mais n'est pas dans les chemins par défaut sur Red Hat/CentOS/Rocky Linux.

La connexion doit fonctionner depuis n'importe quel compte système, il s'agit d'une connexion cliente tout comme `psql`.

> Quelle est la taille de la base après alimentation ?

`\l+` renvoie ceci :

```
$ psql
postgres=# \l+
                                                List of databases
  Name     |  Owner   | Encoding |   Collate   |    …    |   Access privileges   |
 ↪  Size   | …
----------+----------+----------+-------------+---------+-----------------------+---
 ↪  ------+----
```

```
bench     | postgres | UTF8     | en_US.UTF-8 |         |                      | 23 MB
 ↪  | …
postgres  | postgres | UTF8     | en_US.UTF-8 |         |                      | 6477
 ↪  kB | …
template0 | postgres | UTF8     | en_US.UTF-8 |         | =c/postgres         +| 7297
 ↪  kB | …
          |          |          |             |         | postgres=CTc/postgres |
 ↪  | …
template1 | postgres | UTF8     | en_US.UTF-8 |         | =c/postgres         +| 7377
 ↪  kB | …
         |         |         |            |        | postgres=CTc/postgres |        |
```

La base **bench** fait donc 23 Mo.

> Afficher la liste des tables de la base **bench** et leur taille.

`\dt` affiche les tables, `\dt+` ajoute quelques informations dont la taille.

```
postgres=# \c bench
You are now connected to database "bench" as user "postgres".

bench=# \dt+
 Schema |       Name       | Type  |  Owner   | Persistence | Access method |  Size
 ↪  | Description
--------+------------------+-------+----------+-------------+---------------+-------
 ↪  --+-------------
 public | pgbench_accounts | table | postgres | permanent   | heap          | 13 MB
 ↪  |
 public | pgbench_branches | table | postgres | permanent   | heap          | 40 kB
 ↪  |
 public | pgbench_history  | table | postgres | permanent   | heap          | 0 bytes
 ↪  |
 public | pgbench_tellers  | table | postgres | permanent   | heap          | 40 kB
 ↪  |
```

(Il est courant d'utiliser `\d` et non `\dt`. S'afficheront alors aussi les vues et les séquences.)

> Quelle est la structure de la table `pgbench_accounts` ?

`\d` (voire `d+`) est sans doute un des ordres les plus utiles à connaître :

```
bench=# \d pgbench_accounts
              Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Foreign-key constraints:
    "pgbench_accounts_bid_fkey" FOREIGN KEY (bid) REFERENCES pgbench_branches(bid)
```

```
Referenced by:
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                                FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
```

La table a quatre colonnes : `aid`, `bid`, `abalance`, `filler`.

La première porte la clé primaire (et ne peut donc être à `NULL`).

La seconde est une clé étrangère pointant vers `pgbench_branches`.

La table `pgbench_history` porte une clé étrangère pointant vers la clé primaire de cette table.

> Afficher l'ensemble des autres objets non système (index, séquences, vues…) de la base.

Les index :

```
bench=# \di+
                                                    List of relations
Schema |         Name          | Type  |  Owner   |      Table       | Persistence | …
 ↪  |  Size   | …
-------+-----------------------+-------+----------+------------------+-------------
 ↪  +---+---------+--
public | pgbench_accounts_pkey | index | postgres | pgbench_accounts | permanent   |
 ↪  | 2208 kB |
public | pgbench_branches_pkey | index | postgres | pgbench_branches | permanent   |
 ↪  | 16 kB   |
public | pgbench_tellers_pkey  | index | postgres | pgbench_tellers  | permanent   |
 ↪  | 16 kB   |
```

Ces index sont ceux nécessités par les clés primaires.

Il n'y a ni séquence ni vue :

```
bench=# \ds
N'a trouvé aucune relation.
bench=# \dv
N'a trouvé aucune relation.
```

Nouvel utilisateur :

> Toujours en tant qu'utilisateur système **postgres**, avec l'utilitaire `createuser`, créer un rôle **dupont** (il doit avoir l'attribut `LOGIN` !).

```
$ createuser --echo --login dupont
SELECT pg_catalog.set_config('search_path', '', false)
CREATE ROLE dupont NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

> Sous psql, afficher la liste des rôles (utilisateurs).

`\du` affiche les rôles (sauf ceux système).

Il n'y a que le superutilisateur **postgres** (par défaut), et **dupont** créé tout à l'heure mais sans droit particulier :

```
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 dupont    |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Voir les objets système :

> Dans la base **bench**, afficher l'ensemble des tables systèmes (schéma `pg_catalog`).

```
bench=# \dt pg_catalog.*
                    List of relations
   Schema   |           Name           | Type  |  Owner
------------+--------------------------+-------+----------
 pg_catalog | pg_aggregate             | table | postgres
 pg_catalog | pg_aggregate             | table | postgres
 pg_catalog | pg_am                    | table | postgres
…
 pg_catalog | pg_statistic             | table | postgres
…
(64 rows)
```

Notons que pour afficher uniquement les tables système, on préférera le raccourci `\dtS`.

> Afficher l'ensemble des vues systèmes (schéma `pg_catalog`).

Certaines des vues ci-dessous sont très utiles dans la vie de DBA :

```
bench=# \dv pg_catalog.*
                       List of relations
   Schema   |              Name                | Type |  Owner
------------+----------------------------------+------+----------
 pg_catalog | pg_available_extension_versions  | view | postgres
 pg_catalog | pg_available_extensions          | view | postgres
 pg_catalog | pg_backend_memory_contexts       | view | postgres
 pg_catalog | pg_config                        | view | postgress
 pg_catalog | pg_cursors                       | view | postgres
 pg_catalog | pg_file_settings                 | view | postgres
 pg_catalog | pg_group                         | view | postgres
 pg_catalog | pg_hba_file_rules                | view | postgres
 pg_catalog | pg_ident_file_mappings           | view | postgres
 pg_catalog | pg_indexes                       | view | postgres
 pg_catalog | pg_locks                         | view | postgres
 pg_catalog | pg_matviews                      | view | postgres
…
 pg_catalog | pg_roles                         | view | postgres
…
 pg_catalog | pg_settings                      | view | postgres
 pg_catalog | pg_shadow                        | view | postgres
…
 pg_catalog | pg_stat_activity                 | view | postgres
…
 pg_catalog | pg_stat_archiver                 | view | postgres
```

```
…
 pg_catalog | pg_stat_database                | view | postgres
…
 pg_catalog | pg_stat_progress_analyze        | view | postgres
 pg_catalog | pg_stat_progress_basebackup     | view | postgres
 pg_catalog | pg_stat_progress_cluster        | view | postgres
 pg_catalog | pg_stat_progress_copy           | view | postgres
 pg_catalog | pg_stat_progress_create_index   | view | postgres
 pg_catalog | pg_stat_progress_vacuum         | view | postgres
…
 pg_catalog | pg_stat_replication             | view | postgres
 pg_catalog | pg_stat_replication_slots       | view | postgres
…
 pg_catalog | pg_statio_all_tables            | view | postgres
…
 pg_catalog | pg_statio_user_indexes          | view | postgres
 pg_catalog | pg_statio_user_sequences        | view | postgres
 pg_catalog | pg_statio_user_tables           | view | postgres
 pg_catalog | pg_stats                        | view | postgres
…
 (75 rows)
```

Là encore, `\dvS` est un équivalent pour les tables systèmes.

Manipuler les données :

> Le but est de créer une copie de la table `pgbench_tellers` de la base **bench** avec `CREATE TABLE AS`. Afficher l'aide de cette commande.

```
bench=# \h CREATE TABLE AS
postgres=# \c bench
You are now connected to database "bench" as user "postgres".
bench=# \h CREATE TABLE AS
Command:     CREATE TABLE AS
Description: define a new table from the results of a query
Syntax:
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
 ↪  table_name
    [ (column_name [, ...] ) ]
    [ USING method ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]

URL: https://www.postgresql.org/docs/15/sql-createtableas.html
```

> Créer une table `pgbench_tellers_svg`, copie de la table `pgbench_tellers`.

```
bench=# CREATE TABLE pgbench_tellers_svg AS SELECT * FROM pgbench_tellers ;
SELECT 10
```

> Sortir le contenu de la table `pgbench_tellers` dans un fichier `/tmp/pgbench_tellers.csv` (commande `\copy`).

```
bench=# \copy pgbench_tellers TO '/tmp/pgbench_tellers.csv'
COPY 10
```

Rappelons que la commande `\copy` est propre à `psql` (outil client), et est exécutée sur le client, avec l'accès au système de fichiers du client. `\copy` ne doit pas être confondue avec `COPY`, commande similaire exécutée par le serveur, et n'ayant accès qu'au système de fichiers du serveur. Il est important de bien connaître la distinction même si le client est utilisé ici sur le serveur avec l'utilisateur système **postgres**.

> Quel est le répertoire courant ? Sans quitter `psql`, se déplacer vers `/tmp/`, et en lister le contenu.

Le répertoire courant est celui en cours quand `psql` a été lancé. Selon les cas, ce peut être `/home/dalibo`, `/var/lib/pgsql/`… On peut se déplacer avec `\cd`.

```
bench=# \! pwd
/home/dalibo

bench=# \cd /tmp

bench=# \! ls
pgbench_tellers.csv
systemd-private-1b08135528d846088bb892f5a82aec9e-bolt.service-1hjHUH
…
bench=#
```

> Afficher le contenu du fichier `/tmp/pgbench_tellers.csv` depuis `psql`.

Son contenu est le suivant :

```
bench=# \! cat /tmp/pgbench_tellers.csv
1       1       0       \N
2       1       0       \N
3       1       0       \N
4       1       0       \N
5       1       0       \N
6       1       0       \N
7       1       0       \N
8       1       0       \N
9       1       0       \N
10      1       0       \N
```

On aurait pu l'ouvrir avec un éditeur de texte ou n'importe quel autre programme présent sur le client :

```
bench=# \! vi /tmp/pgbench_tellers.csv
```

> Créer un fichier `décompte.sql`, contenant 3 requêtes pour compter le nombre de lignes dans les 3 tables de **bench** :
>
> - Il devra écrire dans le fichier `/tmp/décompte.txt`.
> - Le faire exécuter par `psql`.

Le fichier doit contenir ceci :

```
\o /tmp/décompte.txt
SELECT COUNT(*) FROM pgbench_accounts ;
SELECT COUNT(*) FROM pgbench_tellers ;
SELECT COUNT(*) FROM pgbench_branches ;
```

La première ligne ordonne d'écrire la sortie des ordres dans le fichier indiqué.

Il peut être appelé par :

```
$ psql -d bench -f /tmp/décompte.sql
```

ou :

```
$ psql -d bench < /tmp/décompte.sql
```

Vérifier ensuite le contenu de `/tmp/décompte.txt`.

Détruire la base :

> Supprimer la base **bench**.

Depuis la ligne de commande du système d'exploitation, en tant qu'utilisateur système **postgres** :

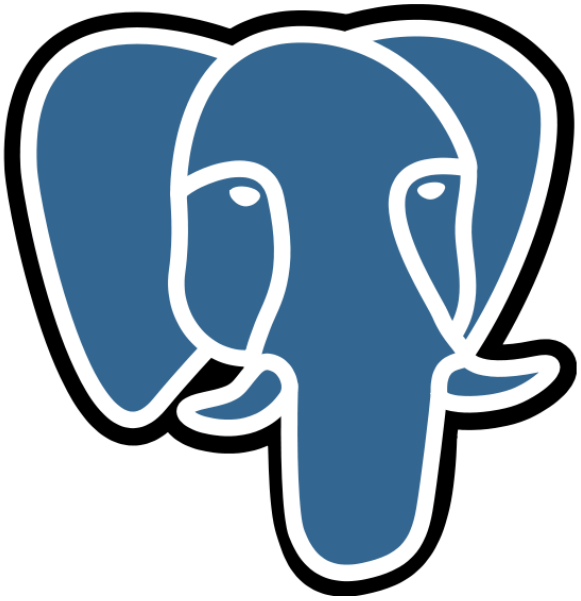
```
$ dropdb --echo bench
SELECT pg_catalog.set_config('search_path', '', false);
DROP DATABASE bench;
```

Alternativement, si l'on est connecté en tant que superutilisateur à l'instance (pas sous la base à supprimer !) :

```
postgres=# DROP DATABASE bench ;
DROP DATABASE
```

Noter l'absence de demande de confirmation !

5/ Tâches courantes



5.1 INTRODUCTION

- Gestion des bases
- Gestion des rôles
- Gestion des droits
- Tâches du DBA
- Sécurité

5.2 BASES

- Liste des bases
- Modèle (Template)
- Création
- Suppression
- Modification / configuration

Pour gérer des bases, il faut savoir les créer, les configurer et les supprimer. Il faut surtout comprendre qui a le droit de faire quoi, et comment. Ce chapitre détaille chacune des opérations possibles concernant les bases sur une instance.

5.2.1 Liste des bases

- Catalogue système : `pg_database`
- Commande `psql` :
  - `\l`
  - `\l+`

La liste des bases de données est disponible grâce à un catalogue système appelé `pg_database`. Il suffit d'un `SELECT` pour récupérer les méta-données sur chaque base :

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM pg_database \gx

-[ RECORD 1 ]--+-----------------------------------
oid            | 5
datname        | postgres
datdba         | 10
encoding       | 6
datlocprovider | c
datistemplate  | f
datallowconn   | t
datconnlimit   | -1
datfrozenxid   | 8885214
datminmxid     | 1
```

```
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion | 2.36
datacl         |
-[ RECORD 2 ]--+-----------------------------------
oid            | 18770
datname        | cave
datdba         | 18769
encoding       | 6
datlocprovider | c
datistemplate  | f
datallowconn   | t
datconnlimit   | -1
datfrozenxid   | 722
datminmxid     | 1
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion | 2.36
datacl         |
-[ RECORD 3 ]--+-----------------------------------
oid            | 1
datname        | template1
datdba         | 10
encoding       | 6
datlocprovider | c
datistemplate  | t
datallowconn   | t
datconnlimit   | -1
datfrozenxid   | 722
datminmxid     | 1
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion | 2.36
datacl         | {=c/postgres,postgres=CTc/postgres}
-[ RECORD 4 ]--+-----------------------------------
oid            | 4
datname        | template0
datdba         | 10
encoding       | 6
datlocprovider | c
datistemplate  | t
datallowconn   | f
datconnlimit   | -1
datfrozenxid   | 722
datminmxid     | 1
dattablespace  | 1663
datcollate     | fr_FR.UTF-8
```

```
datctype       | fr_FR.UTF-8
daticulocale   |
daticurules    |
datcollversion |
datacl         | {=c/postgres,postgres=CTc/postgres}
…
```

Voici la signification des principales colonnes :

- `oid` : identifiant système de la base ;
- `datname` : nom de la base ;
- `datdba` : l'identifiant de l'utilisateur propriétaire de cette base (voir l'OID correspondant à cet identifiant dans le catalogue système `pg_roles`) ;
- `encoding` : avec la fonction `pg_encoding_to_char()`, on peut voir que les valeurs 6 correspondent à UTF8, ce qui est généralement recommandé ;
- `datlocprovider` : `c` indique que les collations sont fournies par la biliothèque `libc` du système, un `i` indiquerait l'utilisation de la bibliothèque ICU (indépendante du système), et dans ce dernier cas les champs `daticulocale` et `daticurules` sont remplies ;
- `datistemplate` : à *true* si cette base est utilisable comme modèle ;
- `datallowconn` : à *true* s'il est autorisé de se connecter à cette base ;
- `datconnlimit` : limite du nombre de connexions simultanées pour les utilisateurs standards sur cette base (`0` interdit toute connexion, et `-1` ne pose pas de limite, jusque `max_connections` du moins) ;
- `datfrozenxid` : plus ancien identifiant de transaction géré par cette base (cela a une importance dans le recyclage des numéros de transaction) ;
- `dattablespace` : identifiant du tablespace par défaut de cette base (1663 indique généralement le tablespace `pg_default`, de fait le répertoire PGDATA ; les emplacements des tablespaces peuvent se trouver dans la table `pg_tablespace` ou par la fonction `pg_tablespace_location()`) ;
- `datacl` : droits pour cette base (un champ vide indique qu'il n'y a pas de droits spécifiques).

Pour avoir une vue plus simple, il est préférable d'utiliser la métacommande `\l` dans `psql` (vue raccourcie pour la mise en page) :

```
postgres=# \l
                                          List of databases
   Name     |   Owner   | Enc. | Locale P. |   Collate   |…|   Access privileges
------------+-----------+------+-----------+-------------+-+-----------------------
 cave       | caviste   | UTF8 | libc      | fr_FR.UTF-8 | |
 pgbench    | pgbench   | UTF8 | libc      | fr_FR.UTF-8 | |
 postgres   | postgres  | UTF8 | libc      | fr_FR.UTF-8 | |
 template0  | postgres  | UTF8 | libc      | fr_FR.UTF-8 | | =c/postgres          +
            |           |      |           |             | | postgres=CTc/postgres
 template1  | postgres  | UTF8 | libc      | fr_FR.UTF-8 | | =c/postgres          +
            |           |      |           |             | | postgres=CTc/postgres
 tpc        | tpc_owner | UTF8 | libc      | fr_FR.UTF-8 | |
…
```

Avec `\l+`, il est possible d'avoir plus d'informations (notamment la taille de la base ou le commentaire).

Noter que si l'on veut savoir où psql va chercher ces informations, il est possible de lui demander d'afficher la requête qu'il envoie au serveur :

```
$ psql --echo-hidden -c '\l+'

********* QUERY **********
SELECT
  d.datname as "Name",
  pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
  pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
  CASE d.datlocprovider WHEN 'c' THEN 'libc' WHEN 'i' THEN 'icu' END AS "Locale
 ↪  Provider",
  d.datcollate as "Collate",
  d.datctype as "Ctype",
  d.daticulocale as "ICU Locale",
  d.daticurules as "ICU Rules",
  pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges",
  CASE WHEN pg_catalog.has_database_privilege(d.datname, 'CONNECT')
       THEN pg_catalog.pg_size_pretty(pg_catalog.pg_database_size(d.datname))
       ELSE 'No Access'
  END as "Size",
  t.spcname as "Tablespace",
  pg_catalog.shobj_description(d.oid, 'pg_database') as "Description"
FROM pg_catalog.pg_database d
  JOIN pg_catalog.pg_tablespace t on d.dattablespace = t.oid
ORDER BY 1;
**************************
…
```

(suivent les résultats.)

La requête affichée montre bien que `psql` accède au catalogue `pg_database`, ainsi qu'à des fonctions système permettant d'éviter d'avoir à faire soi-même les jointures.

5.2.2 Modèle (template)

- Toute création de base se fait à partir d'un modèle
  - `template1`
- Permet de personnaliser sa création de base
- Mais il est aussi possible d'utiliser une autre base

Toute création de base se fait à partir d'un modèle. Par défaut, PostgreSQL utilise le modèle `template1`.

Tout objet ajouté dans le modèle est copié dans la nouvelle base. Cela concerne le schéma (la structure) comme les données. Il est donc intéressant d'ajouter des objets directement dans `template1` pour que ces derniers soient copiés dans les prochaines bases qui seront créées. Pour éviter malgré tout que cette base soit trop modifiée, il est possible de créer des bases qui seront ensuite utilisées comme modèle.

5.2.3 Création d'une base

- **SQL** : `CREATE DATABASE`
  - droit nécessaire: `SUPERUSER` ou `CREATEDB`
  - prérequis : base inexistante
- **Outil système** : `createdb`

L'ordre `CREATE DATABASE` est le seul moyen avec PostgreSQL de créer une base de données. Il suffit d'y ajouter le nom de la base à créer pour que la création se fasse. Il est néanmoins possible d'y ajouter un certain nombre d'options :

- `OWNER`, pour préciser le propriétaire de la base de données (si cette option n'est pas utilisée, le propriétaire est celui qui exécute la commande) ;
- `TEMPLATE`, pour indiquer le modèle à copier (par défaut `template1`) ;
- `ENCODING`, pour forcer un autre encodage que celui du serveur (à noter qu'il faudra utiliser le modèle `template0` dans ce cas) ;
- `LC_COLLATE` et `LC_CTYPE`, pour préciser respectivement l'ordre de tri des données textes et le jeu de caractères (par défaut, il s'agit de la locale utilisée lors de l'initialisation de l'instance) ;
- `STRATEGY` (depuis la version 15), pour indiquer la stratégie employée pour créer la base de donnée. Deux choix sont disponibles :
  - `FILE_COPY` : C'est la méthode historique, seule possible jusqu'en version 14 incluse. Le contenu des répertoires d'une base de données est intégralement copié pour initialiser la nouvelle base, avec juste une trace dans les journaux de transaction. Elle implique deux *checkpoints* parfois gênants, mais peut être intéressante pour copier de grosses bases en générant moins de journaux ;
  - `WAL_LOG` : C'est la méthode par défaut à partir de la version 15. L'opération est entièrement journalisée, et la liste des objets à copier et générée via le catalogue de PostgreSQL. Cette méthode évite les checkpoints et sécurise la copie en garantissant que toutes les opérations sont tracées, tout en évitant la copie accidentelle de fichier orphelins de la base modèle vers la base cible. Cette opération peut écrire beaucoup de journaux dans le cas où la base modèle est grosse, mais elle est idéale pour les créations de nouvelles bases presque vides ;

- `TABLESPACE`, pour stocker la base dans un autre tablespace que le répertoire des données ;
- `ALLOW_CONNECTIONS`, pour autoriser ou non les connexions à la base ;
- `CONNECTION LIMIT`, pour limiter le nombre de connexions d'utilisateurs standards simultanées sur cette base (illimité par défaut, tout en respectant le paramètre `max_connections`) ;
- `IS_TEMPLATE`, pour configurer ou non le mode template.

La copie se fait par clonage de la base de données modèle sélectionnée. Tous les objets et toutes les données faisant partie du modèle seront copiés sans exception. Par exemple, avant la 9.0, on ajoutait le langage PL/pgSQL dans la base de données `template1` pour que toutes les bases créées à partir de `template1` disposent directement de ce langage. Ce n'est plus nécessaire à partir de la 9.0 car le langage PL/pgSQL est activé dès la création de l'instance. Mais il est possible d'envisager d'autres usages de ce comportement (par exemple installer une extension ou une surcouche comme PostGIS dans chaque base).

À noter qu'il peut être nécessaire de sélectionner le modèle `template0` en cas de sélection d'un autre encodage que celui par défaut (comme la connexion est interdite sur `template0`, il y a peu de chances que des données textes avec un certain encodage aient été enregistrées dans cette base).

Voici l'exemple le plus simple de création d'une base :

```
CREATE DATABASE b1 ;
```

Cet ordre crée la base de données **b1**. Elle aura toutes les options par défaut. Autre exemple :

```
CREATE DATABASE b2 OWNER u1;
```

Cette commande SQL crée la base **b2** et s'assure que le propriétaire de cette base soit l'utilisateur **u1** (il faut que ce dernier existe au préalable).

Tous les utilisateurs n'ont pas le droit de créer une base de données. L'utilisateur qui exécute la commande SQL doit avoir soit l'attribut `SUPERUSER` soit l'attribut `CREATEDB`. S'il utilise un autre modèle que celui par défaut, il doit être propriétaire de ce modèle ou le modèle doit être marqué comme étant un modèle officiel (autrement dit la colonne `datistemplate` doit être à `true`).

Voici un exemple complet :

```
postgres=# CREATE DATABASE b1;

CREATE DATABASE

postgres=# CREATE USER u1;

CREATE ROLE

postgres=# CREATE DATABASE b2 OWNER u1;

CREATE DATABASE

postgres=# CREATE USER u2 CREATEDB;

CREATE ROLE
```

NB : pour que la connexion qui suit fonctionne, et sans mot de passe, il faut paramétrer `pg_hba.conf` pour autoriser la connexion de cet utilisateur. Ce sera traité plus bas.

```
postgres=# \c postgres u2

You are now connected to database "postgres" as user "u2".

postgres=> CREATE DATABASE b3;

CREATE DATABASE

postgres=> CREATE DATABASE b4 TEMPLATE b2;
ERROR:  permission denied to copy database "b2"
postgres=> CREATE DATABASE b4 TEMPLATE b3;

CREATE DATABASE

postgres=> \c postgres postgres

You are now connected to database "postgres" as user "postgres".

postgres=# ALTER DATABASE b2 IS_TEMPLATE=true;

ALTER DATABASE

postgres=# \c postgres u2

You are now connected to database "postgres" as user "u2".

postgres=> CREATE DATABASE b5 TEMPLATE b2;

CREATE DATABASE

postgres=> \c postgres postgres
postgres=# \l

postgres=# \l
                                                   List of databases
  Name     |  Owner   | Enc… |   Collate   |    Ctype    | … |   Access privileges
----------+----------+------+-------------+-------------+---+---------------------
b1         | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b2         | u1       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b3         | u2       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b4         | u2       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
b5         | u2       | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
postgres   | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   |
template0  | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   | =c/postgres          +
           |          |      |             |             |   | postgres=CTc/postgres
template1  | postgres | UTF8 | en_US.UTF-8 | en_US.UTF-8 |   | =c/postgres          +
           |          |      |             |             |   | postgres=CTc/postgres
```

L'outil système `createdb` se connecte à la base de données `postgres` et exécute la commande `CREATE DATABASE`, exactement comme ci-dessus. Appelée sans aucun argument, `createdb` crée une base de donnée portant le nom de l'utilisateur connecté (si cette dernière n'existe pas). L'option `--echo` de cette commande permet de voir exactement ce que `createdb` exécute :

```
$ createdb --echo --owner u1 b6

SELECT pg_catalog.set_config('search_path', '', false)
CREATE DATABASE b6 OWNER u1;
```

À titre d'exemple de trace de l'activité d'un client, voici ce qui peut être tracé :

```
[unknown] - LOG:  connection received: host=[local]
[unknown] - LOG:  connection authorized: user=postgres database=postgres
createdb - LOG:  duration: 1.018 ms statement: SELECT
 ↪  pg_catalog.set_config('search_path', ''Z, false)
createdb - CREATE DATABASE b6 OWNER u1;
createdb - LOG:  disconnection: session time: 0:00:00.277 user=postgres
 ↪  database=postgres host=[local]
```

Cela réclame ce paramétrage des traces :

```
log_min_duration_statement = 0
log_connections = on
log_disconnections = on
```

5.2.4 Suppression d'une base

- **SQL** : `DROP DATABASE`
  - droit nécessaire : `SUPERUSER` ou propriétaire de la base
  - prérequis : aucun utilisateur connecté sur la base
  - ou déconnexion forcée
- **Outil système** : `dropdb`

Supprimer une base de données supprime tous les objets et toutes les données contenues dans la base. La destruction d'une base de données ne peut pas être annulée.

La suppression se fait uniquement avec l'ordre `DROP DATABASE`. Seuls les superutilisateurs et le propriétaire d'une base peuvent supprimer cette base. Cependant, pour que cela fonctionne, il faut qu'aucun utilisateur ne soit connecté à cette base. Si quelqu'un est connecté, un message d'erreur apparaîtra :

```
postgres=# DROP DATABASE b6;
```

```
ERROR:  database "b6" is being accessed by other users
DETAIL:  There are 1 other session(s) using the database.
```

Il faut donc attendre que les utilisateurs se déconnectent, ou leur demander de le faire, voire les déconnecter autoritairement :

```
postgres=# SELECT pg_terminate_backend(pid)
       FROM pg_stat_activity
       WHERE datname='b6';
```

```
 pg_terminate_backend
--------------------
 t
```

```
postgres=# DROP DATABASE b6;
```

```
DROP DATABASE
```

Là-aussi, PostgreSQL propose un outil système appelé `dropdb` pour faciliter la suppression des bases. Cet outil se comporte comme `createdb`. Il se connecte à la base `postgres` et exécute l'ordre SQL correspondant à la suppression de la base :

```
$ dropdb --echo b5
```

```
SELECT pg_catalog.set_config('search_path', '', false)
DROP DATABASE b5;
```

Contrairement à `createdb`, sans nom de base, `dropdb` ne fait rien.

Pour forcer la déconnexion des utilisateurs, il est possible d'utiliser la clause `WITH (FORCE)` de l'ordre `DROP DATABASE` ou l'option en ligne de commande `--force` de l'outil `dropdb`.

5.2.5 Modification / configuration

- `ALTER DATABASE`
  - pour modifier quelques méta-données
  - pour ajouter, modifier ou supprimer une configuration
- Catalogue système `pg_db_role_setting`

Avec la commande `ALTER DATABASE`, il est possible de modifier quelques méta-données :

- le nom de la base ;
- son propriétaire ;
- la limite de connexions ;
- le tablespace de la base.

Dans le cas d'un changement de nom ou de tablespace, aucun utilisateur ne doit être connecté à la base pendant l'opération.

Il est aussi possible d'ajouter, de modifier ou de supprimer une configuration spécifique pour une base de données en utilisant la syntaxe suivante :

```
ALTER DATABASE base SET paramètre TO valeur;
```

La configuration spécifique de chaque base de données surcharge toute configuration reçue sur la ligne de commande du processus `postgres` père ou du fichier de configuration `postgresql.conf`. L'ajout d'une configuration avec `ALTER DATABASE` sauvegarde le paramétrage mais ne l'applique

pas immédiatement. Il n'est appliqué que pour les prochaines connexions. Notez que les utilisateurs peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut, pas d'un réglage forcé.

Voici un exemple complet :

```
b1=# SHOW work_mem;

 work_mem
----------
 4MB

b1=# ALTER DATABASE b1 SET work_mem TO '2MB';

ALTER DATABASE

b1=# SHOW work_mem;

 work_mem
----------
 4MB

b1=# \c b1

You are now connected to database "b1" as user "postgres".

b1=# SHOW work_mem;

 work_mem
----------
 2MB
```

Cette configuration est présente même après un redémarrage du serveur. Elle n'est pas enregistrée dans le fichier de configuration `postgresql.conf`, mais dans un catalogue système appelé `pg_db_role_setting` :

```
b1=# ALTER DATABASE b2 SET work_mem TO '32MB';

ALTER DATABASE

b1=# ALTER USER u1 SET maintenance_work_mem TO '256MB';

ALTER ROLE

b1=# SELECT * FROM pg_db_role_setting;

 setdatabase | setrole |          setconfig
-------------+---------+-----------------------------
       16384 |       0 | {work_mem=2MB}
       16386 |       0 | {work_mem=32MB}
           0 |   16385 | {maintenance_work_mem=256MB}

b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;
```

```
 Base | Utilisateur |        Configuration
------+-------------+-----------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
      | u1          | {maintenance_work_mem=256MB}
```

```
b1=# ALTER DATABASE b3 SET work_mem to '10MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER DATABASE b3 SET maintenance_work_mem to '128MB';
```

```
ALTER DATABASE
```

```
b1=# ALTER DATABASE b3 SET random_page_cost to 3;
```

```
ALTER DATABASE
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;
```

```
 Base | Utilisateur |                        Configuration
------+-------------+-------------------------------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB,random_page_cost=3}
      | u1          | {maintenance_work_mem=256MB}
```

Pour annuler la configuration d'un paramètre, utilisez :

```
ALTER DATABASE base RESET paramètre;
```

Par exemple :

```
b1=# ALTER DATABASE b3 RESET random_page_cost;
```

```
ALTER DATABASE
```

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;
```

```
 Base | Utilisateur |               Configuration
------+-------------+---------------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB}
      | u1          | {maintenance_work_mem=256MB}
```

Si vous copiez avec `CREATE DATABASE … TEMPLATE` une base dont certains paramètres ont été configurés spécifiquement pour elle, ces paramètres ne sont pas appliqués à la nouvelle base de données.

5.3 RÔLES

- Utilisateur/groupe
- Liste des rôles
- Création
- Suppression
- Modification
- Gestion des mots de passe

Un rôle peut être vu soit comme un utilisateur de la base de données, soit comme un groupe d'utilisateurs de la base de données, suivant la façon dont le rôle est conçu et configuré. Les rôles peuvent être propriétaires d'objets de la base de données (par exemple des tables) et peuvent affecter des droits sur ces objets à d'autres rôles pour contrôler l'accès à ces objets. De plus, il est possible de donner l'appartenance d'un rôle à un autre rôle, l'autorisant ainsi à utiliser les droits affectés au rôle dont il est membre.

Nous allons voir dans cette partie comment gérer les rôles, en allant de leur création à leur suppression, en passant par leur configuration.

5.3.1 Utilisateurs et groupes

- « Rôle » = utilisateurs et groupes
- `CREATE/DROP/ALTER ROLE`
- Historique
  - `CREATE/DROP/ALTER USER`
  - `CREATE/DROP/ALTER GROUP`

Avant PostgreSQL 8.1, les notions d'utilisateur et de groupe étaient séparées. Pour conserver la compatibilité avec les anciennes applications, les ordres SQL pour les utilisateurs et les groupes ont été conservés. Il est donc toujours possible de les utiliser mais il est actuellement conseillé de passer par les ordres SQL pour les rôles. En fait, un utilisateur est juste un rôle qui a par défaut le droit de se connecter.

5.3.2 Liste des rôles

- Catalogue système : `pg_roles`
- Dans `psql` :
  - `\du`
  - `\drg` (v15+)

La liste des rôles est disponible grâce à un catalogue système appelé `pg_roles`. Il suffit d'un SELECT pour récupérer les méta-données sur chaque rôle :

```
postgres=# \x

Expanded display is on.

postgres=# SELECT * FROM pg_roles LIMIT 3;

-[ RECORD 1 ]--+---------------------
rolname        | postgres
rolsuper       | t
rolinherit     | t
rolcreaterole  | t
rolcreatedb    | t
rolcanlogin    | t
rolreplication | t
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | t
rolconfig      |
oid            | 10
-[ RECORD 2 ]--+---------------------
rolname        | pg_monitor
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | f
rolreplication | f
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | f
rolconfig      |
oid            | 3373
-[ RECORD 3 ]--+---------------------
rolname        | pg_read_all_settings
rolsuper       | f
```

```
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | f
rolreplication | f
rolconnlimit   | -1
rolpassword    | ********
rolvaliduntil  |
rolbypassrls   | f
rolconfig      |
oid            | 3374
```

Voici la signification des différentes colonnes :

- `rolname`, le nom du rôle ;
- `rolsuper`, le rôle a-t-il l'attribut `SUPERUSER` ? ;
- `rolinherit`, le rôle hérite-t-il automatiquement des droits des rôles dont il est membre ? ;
- `rolcreaterole`, le rôle a-t-il le droit de créer des rôles ? ;
- `rolcreatedb`, le rôle a-t-il le droit de créer des bases ? ;
- `rolcanlogin`, le rôle a-t-il le droit de se connecter ? ;
- `rolreplication`, le rôle peut-il être utilisé dans une connexion de réplication ? ;
- `rolconnlimit`, limite du nombre de connexions simultanées pour ce rôle (`0` indiquant « pas de connexions possibles », `-1` permet d'indiquer qu'il n'y a pas de limite en dehors de la valeur du paramètre `max_connections`) ;
- `rolpassword`, mot de passe du rôle (non affiché) ;
- `rolvaliduntil`, date limite de validité du mot de passe ;
- `rolbypassrls`, le rôle court-circuite-t-il les droits sur les lignes ? ;
- `rolconfig`, configuration spécifique du rôle ;
- `oid`, identifiant système du rôle.

Pour avoir une vue plus simple, il est préférable d'utiliser la métacommande `\du` dans `psql` :

```
postgres=# \du
```

```
List of roles
-[ RECORD 1 ]-----------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
-[ RECORD 2 ]-----------------------------------------------------------
Role name  | u1
Attributes |
Member of  | {}
-[ RECORD 3 ]-----------------------------------------------------------
Role name  | u2
Attributes | Create DB
Member of  | {}
```

Il est à noter que les rôles systèmes ne sont pas affichés. Les rôles systèmes sont tous ceux commençant par `pg_`.

La métacommande `\du` ne fait qu'accéder aux tables systèmes. Par exemple :

```
$ psql postgres
```

```
psql (17.0)
Type "help" for help.
```

```
postgres=# \set ECHO_HIDDEN true
postgres=# \x on
postgres=# \du
******** REQUÊTE *********
SELECT r.rolname, r.rolsuper, r.rolinherit,
  r.rolcreaterole, r.rolcreatedb, r.rolcanlogin,
  r.rolconnlimit, r.rolvaliduntil
, r.rolreplication
, r.rolbypassrls
FROM pg_catalog.pg_roles r
WHERE r.rolname !~ '^pg_'
ORDER BY 1;
**************************
```

```
List of roles
-[ RECORD 7 ]-----------------------------------------
Nom du rôle | postgres
Attributs   | Superutilisateur, Créer un rôle, Créer
            | une base, Réplication, Contournement RLS
[...]
```

`\du` affiche aussi les groupes jusqu'à la version 14 de PostgreSQL. Pour les voir avec PostgreSQL 15 ou postérieur, utiliser `\drg` :

```
CREATE ROLE devs ;
CREATE ROLE alphonse LOGIN ;
CREATE ROLE alfred LOGIN ;
GRANT devs TO alphonse ;
GRANT devs TO alfred ;
```

```
                 Liste des droits de rôles
 Nom du rôle | Membre de |   Options    | Donneur de droits
-------------+-----------+--------------+-------------------
 alfred      | devs      | INHERIT, SET | postgres
 alphonse    | devs      | INHERIT, SET | postgres
```

5.3.3 Création d'un rôle

- **SQL** : `CREATE ROLE`
  - droit nécessaire : `SUPERUSER` ou `CREATEROLE`
  - prérequis : utilisateur inexistant
- **Outil système** : `createuser`
  - attribut `LOGIN` par défaut

L'ordre `CREATE ROLE` est le seul moyen avec PostgreSQL de créer un rôle. Il suffit d'y ajouter le nom du rôle à créer pour que la création se fasse. Il est néanmoins possible d'y ajouter un certain nombre d'options :

- `SUPERUSER`, pour que le nouveau rôle soit superutilisateur (autrement dit, ce rôle a le droit de tout faire une fois connecté à une base de données) ;
- `CREATEDB`, pour que le nouveau rôle ait le droit de créer des bases de données ;
- `CREATEROLE`, pour que le nouveau rôle ait le droit de créer un rôle ;
- `INHERIT`, pour que le nouveau rôle hérite automatiquement des droits des rôles dont il est membre ;
- `LOGIN`, pour que le nouveau rôle ait le droit de se connecter (c'est automatique avec `CREATE USER` ou `createuser`, pas avec `CREATE GROUP` ou `CREATE ROLE`) ;
- `REPLICATION`, pour que le nouveau rôle puisse se connecter en mode réplication ;
- `BYPASSRLS`, pour que le nouveau rôle puisse ne pas être vérifié pour les sécurités au niveau ligne ;
- `CONNECTION LIMIT`, pour limiter le nombre de connexions simultanées pour ce rôle ;
- `PASSWORD`, pour préciser le mot de passe de ce rôle ;
- `VALID UNTIL`, pour indiquer la date limite de validité du mot de passe ;
- `IN ROLE`, pour indiquer à quel rôle ce rôle appartient ;
- `IN GROUP`, pour indiquer à quel groupe ce rôle appartient ;
- `ROLE`, pour indiquer les membres de ce rôle ;
- `ADMIN`, pour indiquer les membres de ce rôles (les nouveaux membres ayant en plus la possibilité d'ajouter d'autres membres à ce rôle) ;
- `USER`, pour indiquer les membres de ce rôle ;
- `SYSID`, pour préciser l'identifiant système, mais est ignoré.

Par défaut, un rôle n'a aucun attribut (ni superutilisateur, ni le droit de créer des rôles ou des bases, ni la possibilité de se connecter en mode réplication, ni la possibilité de se connecter).

Voici quelques exemples simples :

```
postgres=# CREATE ROLE u3;

CREATE ROLE

postgres=# CREATE ROLE u4 CREATEROLE;

CREATE ROLE

postgres=# CREATE ROLE u5 LOGIN IN ROLE u2;

CREATE ROLE

postgres=# CREATE ROLE u6 ROLE u5;

CREATE ROLE

postgres=# \du

List of roles
-[ RECORD 1 ]-----------------------------------------------------------
Role name  | postgres
Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS
Member of  | {}
-[ RECORD 2 ]-----------------------------------------------------------
Role name  | u1
Attributes |
Member of  | {}
-[ RECORD 3 ]-----------------------------------------------------------
Role name  | u2
Attributes | Create DB
Member of  | {}
-[ RECORD 4 ]-----------------------------------------------------------
Role name  | u3
Attributes | Cannot login
Member of  | {}
-[ RECORD 5 ]-----------------------------------------------------------
Role name  | u4
Attributes | Create role, Cannot login
Member of  | {}
-[ RECORD 6 ]-----------------------------------------------------------
Role name  | u5
Attributes |
Member of  | {u2,u6}
-[ RECORD 7 ]-----------------------------------------------------------
Role name  | u6
Attributes | Cannot login
Member of  | {}
```

Tous les rôles n'ont pas le droit de créer un rôle. Le rôle qui exécute la commande SQL doit avoir soit l'attribut `SUPERUSER` soit l'attribut `CREATEROLE`. Un utilisateur qui a l'attribut `CREATEROLE` pourra créer tout type de rôles sauf des superutilisateurs.

Voici un exemple complet :

```
postgres=# CREATE ROLE u7 LOGIN CREATEROLE;

CREATE ROLE
```

```
postgres=# \c postgres u7

You are now connected to database "postgres" as user "u7".

postgres=> CREATE ROLE u8 LOGIN;

CREATE ROLE

postgres=> CREATE ROLE u9 LOGIN CREATEDB;

CREATE ROLE

postgres=> CREATE ROLE u10 LOGIN SUPERUSER;

ERROR:  must be superuser to create superusers

postgres=> \du

                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 u1        |                                                            | {}
 u2        | Create DB                                                  | {}
 u3        | Cannot login                                               | {}
 u4        | Create role, Cannot login                                  | {}
 u5        |                                                            | {u2,u6}
 u6        | Cannot login                                               | {}
 u7        | Create role                                                | {}
 u8        |                                                            | {}
 u9        | Create DB                                                  | {}
```

Il est toujours possible d'utiliser les ordres SQL `CREATE USER` et `CREATE GROUP`. PostgreSQL les comprend comme étant l'ordre `CREATE ROLE`. Dans le premier cas (`CREATE USER`), il ajoute automatiquement l'option `LOGIN`.

Un outil shell existe aussi pour créer un utilisateur, c'est-à-dire là encore un rôle avec l'attribut `LOGIN` :

```
$ createuser --echo u10 --superuser

SELECT pg_catalog.set_config('search_path', '', false)
CREATE ROLE u10 SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
```

5.3.4 Suppression d'un rôle

- **SQL** : `DROP ROLE`
  - droit nécessaire : `SUPERUSER` ou `CREATEROLE`
  - prérequis : rôle existant, rôle ne possédant pas d'objet
  - penser à `REASSIGN OWNED` et `DROP OWNED`
- **Outil système** : `dropuser`

La suppression d'un rôle se fait uniquement avec l'ordre `DROP ROLE`. Seuls les utilisateurs disposant des attributs `SUPERUSER` et `CREATEROLE` peuvent supprimer des rôles. Cependant, pour que cela fonctionne, il faut que le rôle à supprimer ne soit pas propriétaire d'objets dans l'instance. S'il est propriétaire, un message d'erreur apparaîtra :

```
postgres=> DROP ROLE u1;

ERROR:  role "u1" cannot be dropped because some objects depend on it
DETAIL:  owner of database b2
```

Il faut donc changer le propriétaire des objets en question ou supprimer les objets. Vous pouvez utiliser respectivement les ordres `REASSIGN OWNED` et `DROP OWNED` pour cela.

Un rôle qui n'a pas l'attribut `SUPERUSER` ne peut pas supprimer un rôle qui a cet attribut :

```
postgres=> DROP ROLE u10;

ERROR:  must be superuser to drop superusers
```

Par contre, il est possible de supprimer un rôle qui est connecté. Le rôle connecté gardera des possibilités limitées après sa suppression. Par exemple, il peut toujours lire quelques tables systèmes mais il ne peut plus créer d'objets.

Il existe aussi un outil système appelé `dropuser` pour faciliter la suppression d'un rôle :

```
$ dropuser --echo u10

SELECT pg_catalog.set_config('search_path', '', false)
DROP ROLE u10;
```

5.3.5 Modification d'un rôle

- `ALTER ROLE`
  - pour modifier quelques méta-données
  - pour ajouter, modifier ou supprimer une configuration
- Catalogue système : `pg_db_role_setting`

Avec la commande `ALTER ROLE`, il est possible de modifier quelques méta-données du rôle :

- son nom ;
- son mot de passe ;
- sa limite de validité ;
- ses attributs :
  - `SUPERUSER` ;
  - `CREATEDB` ;
  - `CREATEROLE` ;
  - `CREATEUSER` ;
  - `INHERIT` ;
  - `LOGIN` ;
  - `REPLICATION` ;
  - `BYPASSRLS`.

Toutes ces opérations peuvent s'effectuer alors que le rôle est connecté à la base.

Il est aussi possible d'ajouter, de modifier ou de supprimer une configuration spécifique pour un rôle en utilisant la syntaxe suivante :

```
ALTER ROLE rôle SET paramètre TO valeur;
```

La configuration spécifique de chaque rôle surcharge toute configuration reçue sur la ligne de commande du processus postgres père ou du fichier de configuration `postgresql.conf`, mais aussi la configuration spécifique de la base de données où le rôle est connecté. L'ajout d'une configuration avec `ALTER ROLE` sauvegarde le paramétrage mais ne l'applique pas immédiatement. Il n'est appliqué que pour les prochaines connexions. Notez que les rôles peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut, pas d'un réglage forcé.

Voici un exemple complet :

```
$ psql -U u2 postgres

psql (13.0)
Type "help" for help.
```

```
postgres=> SHOW work_mem;

 work_mem
----------
 4MB

postgres=> ALTER ROLE u2 SET work_mem TO '20MB';

ALTER ROLE

postgres=> SHOW work_mem;

 work_mem
----------
 4MB

postgres=> \c - u2

You are now connected to database "postgres" as user "u2".

postgres=> SHOW work_mem;

 work_mem
----------
 20MB
```

Cette configuration est présente même après un redémarrage du serveur. Elle n'est pas enregistrée dans le fichier de configuration `postgresql.conf` mais dans un catalogue système appelé `pg_db_role_setting` :

```
b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;

 Base | Utilisateur |               Configuration
------+-------------+--------------------------------------------
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB}
      | u1          | {maintenance_work_mem=256MB}
      | u2          | {work_mem=20MB}
```

Il est aussi possible de configurer un paramétrage spécifique pour un utilisateur et une base donnés :

```
postgres=# ALTER ROLE u2 IN DATABASE b1 SET work_mem to '10MB';

ALTER ROLE

postgres=# \c postgres u2

You are now connected to database "postgres" as user "u2".

postgres=> SHOW work_mem;
```

```
 work_mem
----------
 20MB

postgres=> \c b1 u2

You are now connected to database "b1" as user "u2".

b1=> SHOW work_mem;

 work_mem
----------
 10MB

b1=> \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> SHOW work_mem;

 work_mem
----------
 2MB

b1=> \c postgres u1

You are now connected to database "postgres" as user "u1".

postgres=> SHOW work_mem;

 work_mem
----------
 4MB

b1=# SELECT d.datname AS "Base", r.rolname AS "Utilisateur",
     setconfig AS "Configuration"
     FROM pg_db_role_setting
     LEFT JOIN pg_database d ON d.oid=setdatabase
     LEFT JOIN pg_roles r ON r.oid=setrole
     ORDER BY 1, 2;

 Base | Utilisateur |               Configuration
------+-------------+---------------------------------------------
 b1   | u2          | {work_mem=10MB}
 b1   |             | {work_mem=2MB}
 b2   |             | {work_mem=32MB}
 b3   |             | {work_mem=10MB,maintenance_work_mem=128MB}
      | u1          | {maintenance_work_mem=256MB}
      | u2          | {work_mem=20MB}
```

Pour annuler la configuration d'un paramètre pour un rôle, utilisez :

```
ALTER ROLE rôle RESET paramètre;
```

Attention : la prise en compte de ces options dans les sauvegardes est un point délicat. Il est détaillé dans notre module de formation sur les sauvegardes logiques.

Après sa création, il est toujours possible d'ajouter et de supprimer un rôle dans un autre rôle. Pour cela, il est possible d'utiliser les ordres `GRANT` et `REVOKE` :

```
GRANT role_groupe TO role_utilisateur;
```

Il est aussi possible de passer par la commande `ALTER GROUP` de cette façon :

```
ALTER GROUP role_groupe ADD USER role_utilisateur;
```

5.3.6 Mot de passe

Le mot de passe ne concerne pas toutes les méthodes d'authentification

- Par défaut : l'utilisateur n'a pas de mot de passe
  - donc pas de connexion possible
- Modification :

```
ALTER ROLE u1 PASSWORD 'supersecret';  -- dangereux
```

- Attention à ne pas l'afficher dans les traces
  - fournir un mot de passe déjà chiffré
  - utiliser `\password` ou `createuser`

```
createuser --login --echo --pwprompt u1
```

Certaines méthodes d'authentification n'ont pas besoin de mot de passe (comme `peer` pour les connexions depuis le serveur même), ou délèguent l'authentification à un système extérieur (comme `ldap`). Par défaut, les utilisateurs n'ont pas de mot de passe. Si la méthode en exige un, ils ne pourront pas se connecter.

Il est très fortement conseillé d'utiliser une méthode d'authentification avec saisie du mot de passe.

On peut le créer ainsi :

```
ALTER ROLE u1 PASSWORD 'supersecret';
```

À partir de là, avec une méthode d'authentification bien configurée, le mot de passe sera demandé. Il faudra, dans cet exemple, saisir « supersecret » pour que la connexion se fasse.

ATTENTION ! Le mot de passe peut apparaître en clair dans les traces ! Notamment si `log_min_duration_statement` vaut 0.

```
$ grep PASSWORD $PGDATA/log/traces.log

psql - LOG: duration: 1.865 ms
       statement: ALTER ROLE u1 PASSWORD 'supersecret';
```

La vue système `pg_stat_activity` ou l'extension `pg_stat_statements`, voire d'autres outils, sont aussi susceptibles d'afficher la commande et donc le mot de passe en clair.

Il est donc essentiel de s'arranger pour que seules des personnes de confiance aient accès aux traces et vues systèmes. Il est certes possible de demander temporairement la désactivation des traces pendant le changement de mot de passe (si l'on est superutilisateur) :

```
SET log_min_duration_statement TO -1;
SET log_statement TO none;
ALTER ROLE u1 PASSWORD 'supersecret';

$ grep PASSWORD $PGDATA/log/postgresql-Mon.log
[rien]
```

Cependant, cela ne règle pas le cas de `pg_stat_statements` et `pg_stat_activity`.

De manière générale, il est donc chaudement conseillé de ne renseigner que des mots de passe chiffrés. C'est très simple en mode interactif avec `psql`, la métacommande `\password` opère le chiffrement :

```
\password u1

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
```

L'ordre effectivement envoyé au serveur et éventuellement visible dans les traces sera :

```
ALTER USER u1 PASSWORD 'SCRAM-SHA-256$4096:KcHoLSZE…Hj81r3w='
```

De même si on crée le rôle depuis le shell avec `createuser` :

```
createuser --login --echo --pwprompt u1

Saisir le mot de passe pour le nouveau rôle :
Le saisir de nouveau :
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE u1 PASSWORD
 ↪  'SCRAM-SHA-256$4096:/LVaGESxmDwyF92urT…hS0kOxIpwWAbTpW1i9peGIfg='
NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN NOREPLICATION NOBYPASSRLS;
```

Lors des changements de mots de passe, ne pas oublier qu'il reste un risque de fuite aussi au niveau des outils système, par exemple l'historique du shell, l'affichage de la liste des processus ou les traces système !

5.3.7 Sécurité des mots de passe

```
password_encryption = "scram-sha-256"
```

- SCRAM-SHA-256 : défaut (≥ v14)
- MD5 : déprécié, mais défaut jusque v13 !
- Sécurité :
  - date limite sur le mot de passe (pas le rôle)
  - pas de vérification de la force du mot de passe
  - pas de limite de tentatives échouées (tester `fail2ban`)
  - itérations SCRAM (v16)

Le mot de passe est chiffré en interne, et visible dans les sauvegardes avec `pg_dumpall -g`, ou dans la vue système `pg_authid`.

Il existe deux méthodes de chiffrement : SCRAM-SHA-256 et l'historique MD5, à ne plus utiliser. Elles sont éventuellement utilisables simultanément pour des utilisateurs différents.

L'exemple suivant montre que la méthode de chiffrement peut différer selon les rôles, en fonction de la valeur du paramètre `password_encryption` au moment de la mise en place du mot de passe :

```
SET password_encryption TO "scram-sha-256";

CREATE ROLE u12 LOGIN PASSWORD 'supersecret';

SELECT * FROM pg_authid  WHERE rolname IN ('u1', 'u12') ORDER BY 1;

-[ RECORD 1 ]--+-------------------------------------------------------------
rolname        | u1
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
```

```
rolpassword     | md5fb75f17111cea61e62b54ab950dd1268
rolvaliduntil   |
-[ RECORD 2 ]--+-------------------------------------------------------------
rolname         | u12
rolsuper        | f
rolinherit      | t
rolcreaterole   | f
rolcreatedb     | f
rolcanlogin     | t
rolreplication  | f
rolbypassrls    | f
rolconnlimit    | -1
rolpassword     | SCRAM-SHA-256$4096:0/uC6oDNuQWO8H9pMaVg8g==$nDUpGSefHOZMd
                  TcbWR13NPELJubGg7PduiJjX/Hyt/M=:PSUzE+rP5g4f6mb5sFDRq/Hds
                  OrLvfYew9ZIdzO/GDw=
rolvaliduntil   |
```

Chiffrement SCRAM-SHA-256 :

SCRAM-SHA-256 n'est la méthode par défaut que depuis PostgreSQL 14, bien que disponible depuis PostgreSQL 10. Elle est plus complexe et plus sûre que l'ancienne méthode MD5. Notamment, le même mot de passe entré plusieurs fois pour un même utilisateur, même sur la même instance, donnera des versions chiffrées à chaque fois différentes, mais interchangeables.

Un chiffrement SCRAM-SHA-256 est de la forme :

```
SCRAM-SHA-256$<sel>:<nombre d'itérations>$<hash>
```

Pour quelques détails d'implémentation et une comparaison avec MD5, voir par exemple cette présentation de Jonathan Katz[1].

Générer soi-même des mots de passe chiffrés en SCRAM-SHA-256 en-dehors de `psql` est plus compliqué qu'avec MD5, et les outils comme `\password` s'appuient souvent sur les fonctions fournies par la bibliothèque `libpq`. Cette dernière sert aussi de base à la bibliothèque python psycopg3[2] et sa fonction PGconn.encrypt_password()[3]. Indépendamment de la `libpq`, il existe aussi un script python de Jonathan Katz[4] (version 3.6 minimum).

Depuis la version 16, il est possible d'ajuster le nombre d'itérations avec le paramètre `scram_iterations`. Le défaut de 4096 est un compromis. Monter plus haut permet de lutter contre les attaques par force brute. Réduire ce paramètre permet de parer aux problèmes de performances depuis certains terminaux peu puissants, ou en cas de connexions très fréquentes (le début de la discussion entre les développeurs[5] illustre bien ces deux contraintes).

Chiffrement MD5 :

`md5` est la méthode de chiffrement par défaut jusque PostgreSQL 13 inclus. On peut la rencontrer sur des instances de versions supérieures qui ont été migrées.

[1] https://fr.slideshare.net/jkatz05/safely-protect-postgresql-passwords-tell-others-to-scram
[2] https://github.com/psycopg/psycopg
[3] https://www.psycopg.org/psycopg3/docs/api/pq.html#psycopg.pq.PGconn.encrypt_password
[4] https://gist.github.com/jkatz/e0a1f52f66fa03b732945f6eb94d9c21#file-encypt_password-py-L20
[5] https://www.postgresql.org/message-id/F72E7BC7-189F-4B17-BF47-9735EB72C364@yesql.se

La méthode `md5` est officiellement dépréciée. Migrez les mots de passe de vos utilisateurs.

Cette méthode calcule la somme MD5 du mot de passe concaténé au nom du rôle (pour que deux utilisateurs de même mot de passe n'ait pas la même version chiffrée) ; puis « md5 » est ajouté devant.

De manière plus générale, l'algorithme MD5 est considéré comme trop faible de nos jours. Un autre problème de sécurité est que la version chiffrée d'un mot de passe est identique sur deux instances différentes pour un même nom d'utilisateur, ce qui ouvre la possibilité d'attaques par *rainbow tables*[6]. Un inconvénient plus mineur du chiffrement MD5 est qu'il utilise le nom de l'utilisateur : en cas de changement de ce nom, il faudra de nouveau rentrer le mot de passe pour que son stockage chiffré soit correct.

Pour éviter de fournir un mot de passe en clair à PostgreSQL, il était facile de le chiffrer en MD5 avant de le fournir à PostgreSQL :

```
$ echo -n "supersecretu1" | md5sum
fb75f17111cea61e62b54ab950dd1268  -
```

```
ALTER ROLE u1 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Dans les traces on ne trouvera que la version chiffrée :

```
psql - LOG:  duration: 2.100 ms  statement: ALTER ROLE u1
                                 PASSWORD 'md5fb75f17111cea61e62b54ab950dd1268';
```

Cohabitation de mots de passe SCRAM-SHA-256 et MD5 :

Lors de la mise à jour d'une ancienne instance, il n'est pas forcément possible de ré-entrer immédiatement tous les mots de passe existants chiffrés en MD5. De plus, certains outils clients anciens pourraient ne pas supporter SCRAM-SHA-256 (voir le wiki[7]). Changer la méthode d'authentification selon l'utilisateur est donc utile.

La méthode d'authentification SCRAM-SHA-256 doit rester le défaut, et tout mot de passe réentré sera chiffré en SCRAM-SHA-256 :

```
password_encryption = "scram-sha-256"
```

Si le mot de passe est stocké au format SCRAM-SHA-256, une authentification paramétrée sur `md5` ou `password` dans le fichier `pg_hba.conf` fonctionnera aussi. Cela facilite la migration progressive des utilisateurs de `md5` à `scram-sha-256`, qui peuvent réentrer leur mot de passe quand ils veulent.

Par contre, indiquer la méthode `scram-sha-256` dans `pg_hba.conf` impose un chiffrement en SCRAM-SHA-256 et interdit d'utiliser MD5.

On peut mixer les deux méthodes dans `pg_hba.conf` si le besoin se fait sentir, par exemple pour n'utiliser `md5` que pour une seule application (avec un compte dédié) avec un ancien client :

[6] https://fr.wikipedia.org/wiki/Rainbow_table
[7] https://wiki.postgresql.org/wiki/List_of_drivers

```
# vieille application (avant le cas général)
host   compta  mathusalem   192.168.66.66/32    md5
# authentification habituelle
host   all     all          192.168.66.0/24     scram-sha-256
```

Dates de fin de validité (expiration) des mots de passe :

On peut définir une date de fin de validité pour les mots de passe mais pas pour les rôles eux-mêmes. Il reste donc possible de se connecter en local par la méthode `peer` même si le mot de passe a expiré.

Sécurité des mots de passe :

Enfin, il est à noter que PostgreSQL ne vérifie pas la faiblesse d'un mot de passe. Il est certes possible d'installer une extension appelée `passwordcheck` (voir sa documentation[8]).

```
postgres=# ALTER ROLE u1 PASSWORD 'supersecret';
```

```
ERROR:  password must contain both letters and nonletters
```

Il est possible de modifier le code source de cette extension pour y ajouter les règles convenant à votre cas, ou d'utiliser la bibliothèque `Cracklib`. Des extensions de ce genre, extérieures au projet, existent. Cependant, ces outils exigent que le mot de passe soit fourni en clair, et donc sujet à une fuite (dans les traces par exemple), ce qui, répétons-le, est fortement déconseillé !

Un rôle peut tenter de se connecter autant de fois qu'il le souhaite, ce qui expose à des attaques de type force brute. Il est possible d'interdire toute connexion à partir d'un certain nombre de connexions échouées si vous utilisez une méthode d'authentification externe qui le gère (comme PAM, LDAP ou Active Directory). Vous pouvez aussi obtenir cette fonctionnalité en utilisant un outil comme `fail2ban`. Sa configuration est détaillée dans notre base de connaissances[9].

[8] https://docs.postgresql.fr/current/passwordcheck.html
[9] https://kb.dalibo.com/fail2ban

5.4 DROITS SUR LES OBJETS

- Droits sur les objets
- Droits sur les méta-données
- Héritage des droits
- Changement de rôle

Pour bien comprendre l'intérêt des utilisateurs, il faut bien comprendre la gestion des droits. Les droits sur les objets vont permettre aux utilisateurs de créer des objets ou de les utiliser. Les commandes `GRANT` et `REVOKE` sont essentielles pour cela. Modifier la définition d'un objet demande un autre type de droit, que les commandes précédentes ne permettent pas d'obtenir.

Donner des droits à chaque utilisateur peut paraître long et difficile. C'est pour cela qu'il est généralement préférable de donner des droits à une entité spécifique dont certains utilisateurs hériteront.

5.4.1 Droits sur les objets

- Donner un droit :

```
GRANT USAGE ON SCHEMA unschema TO utilisateur ;
GRANT SELECT,DELETE,INSERT ON TABLE matable TO utilisateur ;
```

- Retirer un droit :

```
REVOKE UPDATE ON TABLE matable FROM utilisateur ;
```

- Droits spécifiques pour chaque type d'objets :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA …
ALTER DEFAULT PRIVILEGES FOR ROLE …
```

- Avoir le droit de donner le droit :

```
WITH GRANT OPTION
```

- Groupe implicite : `public`
- Schéma par défaut : `public` lisible par tous (≤ 14) !

```
REVOKE ALL ON SCHEMA public FROM public ;
```

Par défaut, seul le propriétaire a des droits sur son objet. Les superutilisateurs n'ont pas de droit spécifique sur les objets mais étant donné leur statut de superutilisateur, ils peuvent tout faire sur tous les objets.

Le propriétaire d'un objet peut décider de donner certains droits sur cet objet à certains rôles. Il le fera avec la commande `GRANT` :

```
GRANT droits ON type_objet nom_objet TO role
```

Les droits disponibles dépendent du type d'objet visé. Par exemple, il est possible de donner le droit `SELECT` sur une table mais pas sur une fonction. Une fonction ne se lit pas, elle s'exécute. Il est donc possible de donner le droit `EXECUTE` sur une fonction.

La liste complète des droits figure dans la documentation officielle[10].

Il faut donner les droits aux différents objets séparément. De plus, donner le droit `ALL` sur une base de données donne tous les droits sur la base de données, autrement dit l'objet base de donnée, pas sur les objets à l'intérieur de la base de données. `GRANT` n'est pas une commande récursive. Prenons un exemple :

```
b1=# CREATE ROLE u20 LOGIN;

CREATE ROLE

b1=# CREATE ROLE u21 LOGIN;

CREATE ROLE

b1=# \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> CREATE SCHEMA s1;

ERROR:  permission denied for database b1

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# GRANT CREATE ON DATABASE b1 TO u20;

GRANT

b1=# \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> CREATE SCHEMA s1;

CREATE SCHEMA

b1=> CREATE TABLE s1.t1 (c1 integer);

CREATE TABLE
```

[10] https://docs.postgresql.fr/current/sql-grant.html

```
b1=> INSERT INTO s1.t1 VALUES (1), (2);

INSERT 0 2

b1=> SELECT * FROM s1.t1;

 c1
----
  1
  2

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> SELECT * FROM s1.t1;

ERROR:  permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                      ^

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> GRANT SELECT ON TABLE s1.t1 TO u21;

GRANT

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> SELECT * FROM s1.t1;

ERROR:  permission denied for schema s1
LINE 1: SELECT * FROM s1.t1;
                      ^

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> GRANT USAGE ON SCHEMA s1 TO u21;

GRANT

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> SELECT * FROM s1.t1;

 c1
----
  1
  2

b1=> INSERT INTO s1.t1 VALUES (3);

ERROR:  permission denied for relation t1
```

Le problème de ce fonctionnement est qu'il faut indiquer les droits pour chaque utilisateur, ce qui peut devenir difficile et long. Imaginez avoir à donner le droit `SELECT` sur les 400 tables d'un schéma… Il est néanmoins possible de donner les droits sur tous les objets d'un certain type dans un schéma. Voici un exemple :

```
GRANT SELECT ON ALL TABLES IN SCHEMA s1 to u21;
```

Notez aussi que, lors de la création d'une base, PostgreSQL ajoute automatiquement un schéma nommé `public`. Avant la version 15, tous les droits sont donnés sur ce schéma à un pseudo-rôle, lui aussi appelé `public`, et dont tous les rôles existants et à venir sont membres d'office. À partir de la version 15, le schéma `public` appartient au propriétaire de la base et aucun droit par défaut n'est donné aux autres utilisateurs.

Avec une version antérieure à la version 15, n'importe quel utilisateur peut donc, par défaut, créer des tables dans le schéma `public` de toute base où il peut se connecter (mais ne peut lire les tables créées là par d'autres, sans droit supplémentaire) !

Dans une logique de sécurisation, avant la version 15, il faut donc penser à enlever les droits à `public`. Une fausse bonne idée est de tout simplement supprimer le schéma `public`, ou de le récréer (par défaut, sans droits pour le groupe `public`). Cependant, une sauvegarde logique serait restaurée dans une base qui, par défaut, aurait à nouveau un schéma `public` ouvert à tous. Une révocation explicite des droits se retrouvera par contre dans une sauvegarde :

```
REVOKE ALL ON SCHEMA public FROM public ;
```

(Noter la subtilité de syntaxe : `GRANT... TO...` et `REVOKE... FROM...`)

Nombre de scripts et outils peuvent tomber en erreur sans ces droits. Il faudra remonter cela aux auteurs en tant que bugs.

Cette modification peut être faite aussi dans la base `template1` (qui sert de modèle à toute nouvelle base), sur toute nouvelle instance.

Enfin il est possible d'ajouter des droits pour des objets qui n'ont pas encore été créés. En fait, la commande `ALTER DEFAULT PRIVILEGES` permet de donner des droits par défaut à certains rôles. De cette façon, sur un schéma qui a tendance à changer fréquemment, il n'est plus nécessaire de se préoccuper des droits sur les objets.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO public ;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT INSERT ON TABLES TO utilisateur ;
```

Lorsqu'un droit est donné à un rôle, par défaut, ce rôle ne peut pas le donner à un autre. Pour lui donner en plus le droit de donner ce droit à un autre rôle, il faut utiliser la clause `WITH GRANT OPTION` comme le montre cet exemple :

```
b1=# CREATE TABLE t2 (id integer);

CREATE TABLE

b1=# INSERT INTO t2 VALUES (1);

INSERT 0 1

b1=# SELECT * FROM t2;

 id
----
  1

b1=# \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> SELECT * FROM t2;

ERROR:  permission denied for relation t2

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# GRANT SELECT ON TABLE t2 TO u1;

GRANT

b1=# \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> SELECT * FROM t2;

 id
----
  1

b1=> \c b1 u2

You are now connected to database "b1" as user "u2".

b1=> SELECT * FROM t2;

ERROR:  permission denied for relation t2

b1=> \c b1 u1

You are now connected to database "b1" as user "u1".

b1=> GRANT SELECT ON TABLE t2 TO u2;

WARNING:  no privileges were granted for "t2"
GRANT

b1=> \c b1 postgres

You are now connected to database "b1" as user "postgres".

b1=# GRANT SELECT ON TABLE t2 TO u1 WITH GRANT OPTION;
```

```
GRANT
b1=# \c b1 u1
You are now connected to database "b1" as user "u1".
b1=> GRANT SELECT ON TABLE t2 TO u2;
GRANT
b1=> \c b1 u2
You are now connected to database "b1" as user "u2".
b1=> SELECT * FROM t2;
 id
----
  1
```

5.4.2 Afficher les droits

- Colonne `*acl` sur les tables systèmes
  - `datacl` pour `pg_database`
  - `relacl` pour `pg_class`
- Codage `role1=xxxx/role2` (format `aclitem`)
  - `role1` : rôle concerné par les droits
  - `xxxx` : droits parmi `xxxx`
  - `role2` : rôle qui a donné les droits
- Sous `psql` :
  - `\dp` et `\z`

Les colonnes `*acl` des catalogues systèmes indiquent les droits sur un objet. Leur contenu est un codage au format `aclitem` indiquant le rôle concerné, ses droits, et qui lui a fourni ces droits (ou le propriétaire de l'objet, si celui qui a fourni les droits est un superutilisateur).

Les droits sont codés avec des lettres. Les voici avec leur signification :

- `r` pour la lecture (`SELECT`) ;
- `w` pour les modifications (`UPDATE`) ;
- `a` pour les insertions (`INSERT`) ;

- `d` pour les suppressions (`DELETE`) ;
- `D` pour la troncation (`TRUNCATE`) ;
- `x` pour l'ajout de clés étrangères ;
- `t` pour l'ajout de triggers ;
- `X` pour l'exécution de routines ;
- `U` pour l'utilisation (d'un schéma par exemple) ;
- `C` pour la création d'objets permanents (tables ou vues par exemple) ;
- `c` pour la connexion (spécifique aux bases de données) ;
- `T` pour la création d'objets temporaires (tables ou index temporaires) ;
- `m` pour le droit de « maintenir », c'est-à-dire lancer des `VACUUM`, `ANALYZE`, rafraîchir, réindexer… (à partir de PostgreSQL 17) ;
- `s` pour la modification d'un paramètre superutilisateur avec `SET` ;
- `A` pour la modification d'un paramètre avec `ALTER SYSTEM`.

```
CREATE TABLE public.demo1 (
    champ1 text,
    champ2 text);
ALTER TABLE public.demo1 OWNER TO postgres;
CREATE TABLE public.demo2 (
    champ3 text);
ALTER TABLE public.demo2 OWNER TO postgres;
GRANT SELECT,INSERT ON TABLE public.demo1 TO jack;
GRANT SELECT ON TABLE public.demo1 TO paul;
GRANT DELETE,TRUNCATE ON TABLE public.demo1 TO pierre;
GRANT UPDATE(champ2) ON TABLE public.demo1 TO pierre;
GRANT MAINTAIN ON demo1 TO jack,pierre;  /* v17 */
GRANT SELECT ON TABLE public.demo2 TO pierre;
GRANT SELECT(champ3) ON TABLE public.demo2 TO paul;
```

```
# \z
                                        Droits d'accès
 Schéma |  Nom   | Type  |       Droits d'accès       | Droits … la colonne | …
--------+-------+-------+----------------------------+---------------------+--
 public | demo1 | table | postgres=arwdDxtm/postgres+| champ2:            +|
        |       |       | jack=armi/postgres        +|   pierre=w/postgres |
        |       |       | paul=r/postgres           +|                     |
        |       |       | pierre=dDm/postgres        |                     |
 public | demo2 | table | postgres=arwdDxtm/postgres+| champ3:            +|
        |       |       | pierre=r/postgres          |   paul=r/postgres   |
```

5.4.3 Droits sur les métadonnées

- Seul le propriétaire peut changer la structure d'un objet
  - le renommer
  - le changer de schéma ou de tablespace
  - lui ajouter/retirer des colonnes
- Un seul propriétaire
  - peut être un utilisateur ou un groupe
- `REASSIGN OWNER` / `DROP OWNED`

Les droits sur les objets ne concernent pas le changement des méta-données et de la structure de l'objet. Seul le propriétaire (et les superutilisateurs) peut le faire. S'il est nécessaire que plusieurs personnes puissent utiliser la commande `ALTER` sur l'objet, il faut que ces différentes personnes aient un rôle qui soit membre du rôle propriétaire de l'objet. Prenons un exemple :

```
b1=# \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;

ERROR:  must be owner of relation t1

b1=> \c b1 u20

You are now connected to database "b1" as user "u20".

b1=> GRANT u20 TO u21;

GRANT ROLE

b1=> \du

                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 u1        |                                                            | {}
 u11       | Create DB                                                  | {}
 u12       |                                                            | {}
 u2        | Create DB                                                  | {}
 u20       |                                                            | {}
 u21       |                                                            | {u20}
 u3        | Cannot login                                               | {}
 u4        | Create role, Cannot login                                  | {}
 u5        |                                                            | {u2,u6}
 u6        | Cannot login                                               | {}
```

```
 u7         | Create role                                                | {}
 u8         |                                                            | {}
 u9         | Create DB                                                  | {}

b1=> \c b1 u21

You are now connected to database "b1" as user "u21".

b1=> ALTER TABLE s1.t1 ADD COLUMN c2 text;

ALTER TABLE
```

Pour assigner un propriétaire différent aux objets ayant un certain propriétaire, il est possible de faire appel à l'ordre `REASSIGN OWNED`[11]. De même, il est possible de supprimer tous les objets appartenant à un utilisateur avec l'ordre `DROP OWNED`[12]. Voici un exemple de ces deux commandes :

```
b1=# \d

       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t1   | table | u2
 public | t2   | table | u21

b1=# REASSIGN OWNED BY u21 TO u1;

REASSIGN OWNED

b1=# \d

       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t1   | table | u2
 public | t2   | table | u1

b1=# DROP OWNED BY u1;

DROP OWNED

b1=# \d

       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | t1   | table | u2
```

[11] https://docs.postgresql.fr/current/sql-reassign-owned.html
[12] https://docs.postgresql.fr/17/sql-drop-owned.html

5.4.4 Droits plus globaux 1/2

- Rôles systèmes d'administration
  - `pg_maintain` (17+, évite `GRANT MAINTAIN`)
  - `pg_signal_backend`
  - `pg_checkpoint` (15+)
  - `pg_reserved_connections` (16+)
- Rôles systèmes de supervision
  - `pg_read_all_stats`
  - `pg_read_all_settings`
  - `pg_stat_scan_tables`
  - `pg_monitor` (reprend les 3 précédents)

Certaines fonctionnalités nécessitent l'attribut `SUPERUSER` alors qu'il serait bon de pouvoir les effectuer sans avoir ce droit suprême. Cela concerne principalement l'administration, la sauvegarde et la supervision. Ce besoin concerne aussi des opérations normalement réservées au propriétaire de la table. Après beaucoup de discussions, les développeurs de PostgreSQL ont décidé de créer des rôles systèmes permettant d'avoir plus de droits. Plusieurs ont été créés au fil des versions.

Rôles d'administration :

Un des plus récents (PostgreSQL 17) et importants est `pg_maintain` : ce rôle autorise son détenteur à procéder à des opérations de maintenance sans être propriétaire des tables. Les opérations de maintenance couvertes sont :

- `VACUUM`, `ANALYZE`, `REINDEX`
- `REFRESH MATERIALIZED VIEW`
- `CLUSTER`, `LOCK TABLE`

Une alternative plus granulaire est de donner le droit `MAINTAIN` sur chaque table :

```
GRANT MAINTAIN ON public.matable TO pierre ;
```

(Avant la version 17, il est toujours possible d'avoir un groupe propriétaire de l'objet et d'ajouter un rôle de maintenance comme membre de ce groupe. Mais si jamais un objet est créé par quelqu'un sans transférer la propriété au groupe, le rôle de maintenance ne peut faire d'opération de maintenance sur cet objet.)

Exemple avec un `VACUUM` sur trois tables :

```
$ export PGDATABASE=b3
$ createdb b3
$ createuser maintenance
$ psql --username postgres --command 'CREATE TABLE t1 (id integer);' \
--command 'CREATE TABLE t2 (id integer);' \
--command 'CREATE TABLE t3 (id integer);'
CREATE TABLE
CREATE TABLE
CREATE TABLE
```

Le `VACUUM` ne fonctionne pas pour l'utilisateur de maintenance non propriétaire de la table :

```
$ vacuumdb --username postgres --table t1
vacuumdb: vacuuming database "b3"
$ vacuumdb --username maintenance --table t1
vacuumdb: vacuuming database "b3"
WARNING:  permission denied to vacuum "t1", skipping it
```

Et encore moins pour les tables système :

```
$ vacuumdb --username maintenance
vacuumdb: vacuuming database "b3"
WARNING:  permission denied to vacuum "pg_proc", skipping it
WARNING:  permission denied to vacuum "pg_attribute", skipping it
[...]
WARNING:  permission denied to vacuum "pg_publication", skipping it
WARNING:  permission denied to vacuum "t3", skipping it
WARNING:  permission denied to vacuum "pg_foreign_table", skipping it
[...]
WARNING:  permission denied to vacuum "pg_sequence", skipping it
WARNING:  permission denied to vacuum "t2", skipping it
WARNING:  permission denied to vacuum "pg_publication_namespace", skipping it
[...]
```

Si **maintenance** devient membre du rôle `pg_maintain`, tout fonctionne :

```
$ psql --username postgres --command 'GRANT pg_maintain TO maintenance'
GRANT ROLE

$ vacuumdb --username maintenance
vacuumdb: vacuuming database "b3"
```

Pour autant, cet utilisateur ne peut pas lire les données de la table :

```
$ psql --username maintenance --command 'SELECT * FROM t1'
ERROR:  permission denied for table t1
```

Le rôle `pg_signal_backend` donne le droit à un simple utilisateur d'exécuter les fonctions `pg_cancel_backend()` (annulation de requête) et `pg_terminate_backend()` (fin de session), sur des requêtes autres que les siennes, à l'exception des connexions des superutilisateurs.

```
postgres=# \c - u1

You are now connected to database "postgres" as user "u1".

postgres=> SELECT usename, pid FROM pg_stat_activity WHERE usename IS NOT NULL;
```

```
 usename |  pid
---------+-------
 u2      | 23194
 u1      | 23195

postgres=> SELECT pg_terminate_backend(23194);

ERROR:  must be a member of the role whose process is being terminated
        or member of pg_signal_backend

postgres=> \c - postgres

You are now connected to database "postgres" as user "postgres".

postgres=# GRANT pg_signal_backend TO u1;

GRANT ROLE

postgres=# \c - u1

You are now connected to database "postgres" as user "u1".

postgres=> SELECT pg_terminate_backend(23194);

 pg_terminate_backend
----------------------
 t

postgres=> SELECT usename, pid FROM pg_stat_activity WHERE usename IS NOT NULL;

 usename |  pid
---------+-------
 u1      | 23212
```

Le rôle `pg_checkpoint` permet à un rôle non `SUPERUSER` de lancer un `CHECKPOINT`.

Le rôle `pg_use_reserved_connections` (à partir de PostgreSQL 16) donne accès à un pool de connexions réservées défini par le paramètre `reserved connections` (0 par défaut). Cela peut servir à garantir un accès à la supervision ou à divers utilitaires sans en faire des superutilisateurs.

Rôles de supervision :

Parmi ces rôles, `pg_read_all_stats` permet de lire les tables de statistiques d'activité (vues `pg_stat_*`).

`pg_read_all_settings` permet de lire la configuration de tous les paramètres.

`pg_stat_scan_tables` permet notamment d'exécuter les procédures stockées de lecture des statistiques.

`pg_monitor` est le rôle idéal à donner à un utilisateur de supervision : il combine les trois rôles précédents.

5.4.5 Droits plus globaux 2/2

- Rôles d'accès aux fichiers
  - `pg_database_owner` (14+)
  - `pg_read_server_files`
  - `pg_write_server_files`
  - `pg_execute_server_program`
- Rôles d'accès aux données (14+)
  - `pg_read_all_data`
  - `pg_write_all_data`
- Autres
  - `pg_create_subscription` (16+)

`pg_read_server_files` permet d'autoriser la lecture de fichiers auxquels le serveur peut accéder avec la commande SQL `COPY` et toutes autres fonctions d'accès de fichiers.

`pg_write_server_files` permet la même chose en écriture. Cela sous-entend aussi les sauvegardes de fichiers côté serveur, par exemple avec `pg_basebackup` (à partir de la version 15) ou d'autres outils.

`pg_execute_server_program` autorise les utilisateurs membres à exécuter des programmes en tant que l'utilisateur qui exécute le serveur PostgreSQL au travers de la commande SQL `COPY` et de toute fonction permettant l'exécution d'un programme sur le serveur.

Ces rôles ne sont pas à octroyer à la légère : c'est une potentielle faille de sécurité. Notamment, `COPY … FROM PROGRAM…` permet d'exécuter des ordres shell sur le serveur, avec l'utilisateur système **postgres** en général.

La version 14 ajoute trois nouveaux rôles : `pg_read_all_data` permet de lire toutes les données des tables, vues et séquences, alors que `pg_write_all_data` permet de les écrire. Quant à `pg_database_owner`, il permet de se comporter comme le propriétaire des bases de données.

En version 16 apparaît `pg_create_subscription` pour gérer des souscriptions en réplication logique.

5.4.6 Héritage des droits

- Créer un rôle sans droit de connexion
- Donner les droits à ce rôle
- Placer les utilisateurs concernés comme membre de ce rôle

Plutôt que d'avoir à donner les droits sur chaque objet à chaque ajout d'un rôle, il est beaucoup plus simple d'utiliser le système d'héritage des droits.

Supposons qu'une nouvelle personne arrive dans le service de facturation. Elle doit avoir accès à toutes les tables concernant ce service. Sans utiliser l'héritage, il faudra récupérer les droits d'une autre personne du service pour retrouver la liste des droits à donner à cette nouvelle personne. De plus, si un nouvel objet est créé et que chaque personne du service doit pouvoir y accéder, il faudra ajouter l'objet et ajouter les droits pour chaque personne du service sur cet objet. C'est long et sujet à erreur. Il est préférable de créer un rôle facturation, de donner les droits sur ce rôle, puis d'ajouter chaque rôle du service facturation comme membre du rôle facturation. L'ajout et la suppression d'un objet est très simple : il suffit d'ajouter ou de retirer le droit sur le rôle facturation, et cela impactera tous les rôles membres.

Voici un exemple complet :

```
b1=# CREATE ROLE facturation;
```

```
CREATE ROLE
```

```
b1=# CREATE TABLE factures(id integer, dcreation date, libelle text,
montant numeric);
```

```
CREATE TABLE
```

```
b1=# GRANT ALL ON TABLE factures TO facturation;
```

```
GRANT
```

```
b1=# CREATE TABLE clients (id integer, nom text);
```

```
CREATE TABLE
```

```
b1=# GRANT ALL ON TABLE clients TO facturation;
```

```
GRANT
```

```
b1=# CREATE ROLE r1 LOGIN;
```

```
CREATE ROLE
```

```
b1=# GRANT facturation TO r1;
```

```
GRANT ROLE
```

```
b1=# \c b1 r1
```

```
You are now connected to database "b1" as user "r1".

b1=> SELECT * FROM factures;

 id | dcreation | libelle | montant
----+-----------+---------+---------
(0 rows)

b1=# CREATE ROLE r2 LOGIN;

CREATE ROLE

b1=# \c b1 r2

You are now connected to database "b1" as user "r2".

b1=> SELECT * FROM factures;

ERROR:  permission denied for relation factures
```

5.4.7 Changement de rôle

- Rôle par défaut
  - celui de la connexion
- Rôle emprunté :
  - après un `SET ROLE`
  - pour tout rôle dont il est membre

Par défaut, un utilisateur se connecte avec un rôle de connexion. Il obtient les droits et la configuration spécifique de ce rôle. Dans le cas où il hérite automatiquement des droits des rôles dont il est membre, il obtient aussi ces droits qui s'ajoutent aux siens. Dans le cas où il n'hérite pas automatiquement de ces droits, il peut temporairement les obtenir en utilisant la commande `SET ROLE`. Il ne peut le faire qu'avec les rôles dont il est membre.

```
b1=# CREATE ROLE r31 LOGIN;

CREATE ROLE

b1=# CREATE ROLE r32 LOGIN NOINHERIT IN ROLE r31;

CREATE ROLE

b1=# \c b1 r31

You are now connected to database "b1" as user "r31".

b1=> CREATE TABLE t1(id integer);
```

```
CREATE TABLE
b1=> INSERT INTO t1 VALUES (1), (2);
INSERT 0 2
b1=> \c b1 r32
You are now connected to database "b1" as user "r32".
b1=> SELECT * FROM t1;
ERROR:  permission denied for relation t1
b1=> SET ROLE TO r31;
SET
b1=> SELECT * FROM t1;
 id
----
  1
  2
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# ALTER ROLE r32 INHERIT;
ALTER ROLE
b1=# \c b1 r32
You are now connected to database "b1" as user "r32".
b1=> SELECT * FROM t1;
 id
----
  1
  2
b1=> \c b1 postgres
You are now connected to database "b1" as user "postgres".
b1=# REVOKE r31 FROM r32;
REVOKE ROLE
b1=# \c b1 r32
You are now connected to database "b1" as user "r32".
b1=> SELECT * FROM t1;
ERROR:  permission denied for relation t1
b1=> SET ROLE TO r31;
ERROR:  permission denied to set role "r31"
```

Le changement de rôle peut se faire uniquement au niveau de la transaction. Pour cela, il faut utiliser la clause `LOCAL`. Il peut se faire aussi sur la session, auquel cas il faut passer par la clause `SESSION`.

5.5 DROITS DE CONNEXION

- Lors d'une connexion, indication :
  - de l'hôte (socket Unix ou alias/adresse IP)
  - du nom de la base de données
  - du nom du rôle
  - du mot de passe (parfois optionnel)
- Selon les 3 premières informations
  - impose une méthode d'authentification

Lors d'une connexion, l'utilisateur fournit, explicitement ou non, plusieurs informations. PostgreSQL va choisir une méthode d'authentification en se basant sur les informations fournies et sur la configuration d'un fichier appelé `pg_hba.conf`.

5.5.1 Informations de connexion

Pour se connecter à une base, il faut :

- 4 informations
  - socket Unix ou adresse/alias IP
  - n° de port
  - nom de la base
  - nom du rôle
- Fournies explicitement
  - paramètres
  - environnement
- ou implicitement
  - environnement
  - défauts

Tous les outils fournis avec la distribution PostgreSQL (par exemple `createuser`) acceptent des options en ligne de commande pour fournir les informations en question :

- `-h` pour la socket Unix ou l'adresse/alias IP ;
- `-p` pour le numéro de port ;
- `-d` pour le nom de la base ;
- `-U` pour le nom du rôle.

Si l'utilisateur ne passe pas ces informations, plusieurs variables d'environnement sont vérifiées :

- `PGHOST` pour la socket Unix ou l'adresse/alias IP ;
- `PGPORT` pour le numéro de port ;
- `PGDATABASE` pour le nom de la base ;
- `PGUSER` pour le nom du rôle.

Au cas où ces variables ne seraient pas configurées, des valeurs par défaut sont utilisées :

- la socket Unix (`/var/run/postgresql`, parfois `/tmp`) en lieu d'un nom de machine ;
- le port 5432 ;
- la base `postgres` ou le nom de l'utilisateur PostgreSQL demandé, (suivant l'outil) ;
- le nom de l'utilisateur au niveau du système d'exploitation pour le nom du rôle.

Autrement dit, quelle que soit la situation, PostgreSQL remplacera les informations non fournies explicitement par des informations provenant des variables d'environnement, voire par des informations par défaut.

5.5.2 Configuration de l'authentification : pg_hba.conf

PostgreSQL utilise les informations de connexion pour choisir la méthode de connexion

- Fichier de configuration : `pg_hba.conf`
- Exemple :

```
local      DATABASE  USER  METHOD  [OPTIONS]
host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
hostnossl  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

- Clauses `include`, `include_if_exists`, `include_dir` (v16)
- Contrôle avec la vue `pg_hba_file_rules`

Lorsque le serveur PostgreSQL récupère une demande de connexion, il connaît le type de connexion utilisé par le client (socket Unix, connexion TCP SSL, connexion TCP simple, etc.). Il connaît aussi l'adresse IP du client (dans le cas d'une connexion via une socket TCP), le nom de la base et celui de l'utilisateur.

PostgreSQL va donc chercher les lignes correspondantes dans le tableau enregistré dans le fichier `pg_hba.conf` (HBA est l'acronyme de *Host Based Authentication*). Ce fichier est présent à côté de `postgresql.conf`, donc par exemple dans `/var/lib/pgsql/16/data` (défaut du packaging Red Hat) ou `/etc/postgresql/16/main` (défaut du packaging Debian). L'accès doit bien sûr en être le plus restreint possible.

Le contenu du fichier se présente en principe ainsi :

- 4 colonnes d'informations ;
- 1 colonne indiquant la méthode à appliquer ;
- 1 colonne optionnelle d'options.

Depuis PostgreSQL 16, il est possible d'indiquer ou plusieurs fichiers externes :

```
include pg_hba.ansible.conf
include "/chemin/fichier 2.conf"
```

(Existent aussi `include_if_exists`, qui ne tombe pas en erreur si le fichier n'existe pas, et `include_dir`, pour inclure tous les fichiers `.conf` d'un répertoire dans l'ordre de leurs noms.)

`pg_hba.conf` ne peut pas être modifié depuis PostgreSQL même, uniquement depuis le système d'exploitation. Après une modification, il faut dire explicitement à PostgreSQL de le recharger. Selon l'installation et l'OS, cet ordre peut être :

```
pg_ctl reload -D /mnt/base_pg/
systemctl reload postgresql-16
pg_ctlcluster 16 main reload
```

ou depuis PostgreSQL même :

```
SELECT pg_reload_conf() ;
```

Il y a une exception : sous Windows, le fichier est relu dès modification.

Si le fichier contient une erreur de syntaxe, il sera ignoré (et la ligne fautive apparaîtra dans les traces), sauf au (re)démarrage, où un fichier fautif bloque le démarrage !·

Exemple de trace problématique :

```
…user=,db=,app=,client= LOG:  invalid connection type "hsot"
…user=,db=,app=,client= CONTEXT:  line 121 of configuration file
 ↪  "/etc/postgresql/16/main/pg_hba.conf"
…user=,db=,app=,client= FATAL:  could not load /etc/postgresql/16/main/pg_hba.conf
```

Depuis une session utilisateur, il est possible de consulter le fichier via la vue `pg_hba_file_rules`, tel qu'il est sur le disque, et tel qu'il serait compris par PostgreSQL après un rechargement. La colonne `error` est très pratique pour repérer immédiatement les erreurs de syntaxe ou de frappe. La colonne `file_name` indique le fichier source (à partir de PostgreSQL 16).

5.5.3 Exemple de pg_hba.conf

```
# TYPE  DATABASE         USER      ADDRESS           METHOD
# accès direct par la socket
local    all             all                         peer
# accès en local via réseau (localhost IPv6 et IPv4)
host      all            all      ::1/128            scram-sha-256
host      all            all      127.0.0.0/24       scram-sha-256
# accès distants
hostssl   erp            all      192.168.0.0/16     scram-sha-256
hostssl   logistique     all      192.168.30.0/24    scram-sha-256
hostnossl test           demo     192.168.74.150/32  scram-sha-256
# vieux client
hostssl   compta         durand   192.168.10.99/32   md5
# Connexions de réplication
local   replication      all                         peer
host    replication      all      192.168.74.5/32    scram-sha-256
host    replication      all      127.0.0.1/32       scram-sha-256
host    replication      all      ::1/128            scram-sha-256
```

PostgreSQL lit le fichier dans l'ordre. La première ligne correspondant à la connexion demandée lui précise la méthode d'authentification à utiliser. Nous verrons que certaines méthodes comme `scram-sha-256` ou `md5` (les plus courantes) exigent un mot de passe, d'autres non, comme `ldap` ou `cert`. Il ne reste plus qu'à appliquer cette méthode. Si elle fonctionne, la connexion est autorisée et se poursuit.

Si elle ne fonctionne pas, quelle qu'en soit la raison, la connexion est refusée. Aucune autre ligne du fichier ne sera lue ! La raison du refus est tracée sur le serveur de manière beaucoup plus complète que pour le client :

```
2023-11-03 10:41:24 CET [1805700]: [2-1] user=attaquant,db=erp,
app=[unknown],client=::1 DETAIL:  Role "mechant_hacker" does not exist.
Connection matched file "/etc/postgresql/16/defo/pg_hba.conf"
line 110: "host    all      all      ::1/128           scram-sha-256"
```

Il est donc essentiel de bien configurer ce fichier pour avoir une protection maximale, et de faire très attention à l'ordre des entrées.

5.5.4 pg_hba.conf : colonne type

- 4 valeurs possibles
  - `local`
  - `host`
  - `hostssl` (`ssl=on` prérequis)
  - `hostnossl`

La colonne type peut contenir quatre valeurs différentes. La valeur `local` concerne les connexions via la socket Unix, donc depuis un compte sur le serveur même. Elle est généralement couplée à la méthode `peer`. En pratique, on s'en sert surtout pour les connexions administratives (de l'utiliseur système **postgres** au rôle **postgres**), pour les sauvegardes depuis le serveur même, ou pour une connexion depuis une application qui tourne sur la même machine.

Toutes les autres valeurs concernent les connexions via le réseau (socket TCP). La différence réside dans l'utilisation forcée ou non du SSL :

- `host` : SSL au choix du client ;
- `hostssl` : impérativement avec chiffrement SSL ;
- `hostnossl` : impérativement sans SSL.

Il est à noter que l'option `hostssl` n'est utilisable que si le paramètre `ssl` du fichier `postgresql.conf` est à `on` (c'est le cas par défaut sous Debian, pas forcément ailleurs).

5.5.5 pg_hba.conf : colonne database

À quelle(s) base(s) autoriser la connexion ?

- `mabase`
- `base1,base2,base3`
- `sameuser` / `samerole`
- `all`
- `@fichier.txt` (fichier avec plusieurs bases)
- `/db_[1-6]`, `/erp.*` (regex, v16+)
- `replication` (pseudo-base pour réplication physique)

La seconde colonne de `pg_hba.conf` peut recueillir le nom d'une base, le nom de plusieurs bases en les séparant par des virgules, le nom d'un fichier contenant la liste des bases ou quelques valeurs en dur. La valeur `all` indique toutes les bases.

Enfin, la valeur `sameuser` spécifie une base de données de même nom que le rôle demandé (l'utilisateur **durand** peut se connecter à la base **durand**), alors que la valeur `samerole` spécifie que le rôle demandé doit être membre du rôle portant le même nom que la base de données demandée (par exemple, si l'utilisateur **durand** est membre du groupe **compta**, il peut se connecter à la base **compta**).

Il est possible d'indiquer un fichier (préfixé de `@`) qui contiendra une liste de rôles. Il peut être à côté de `pg_hba.conf` mais on peut fournir un chemin complet.

Depuis PostgreSQL 16, il est possible d'utiliser des expressions régulières, qui doivent être précédées de `/` (voir exemple plus bas).

La valeur `replication` est utilisée pour définir les connexions de réplication physique (mais pas logique). Il n'est pas nécessaire d'avoir une base nommée **replication**, toutes les bases sont concernées par une réplication physique.

5.5.6 pg_hba.conf : colonne user

À quel(s) utilisateur(s) permettre la connexion ?

- `unrole`
- `role1,role2,role3`
- `all`
- `+groupe` (membres d'un rôle-groupe)
- `@fichier.txt` (fichier avec plusieurs utilisateurs)
- `/user_.*` (regex, v16+)

La troisième colonne peut recueillir :

- le nom d'un rôle, ou plusieurs rôles séparés par des virgules ;
- le nom d'un groupe (en le précédant d'un signe `+`) : tous les membres de ce groupe pourront se connecter avec leur rôle propre ;
- le nom d'un fichier (précédé de `@`) contenant une liste de rôles ;
- la valeur `all` qui indique tous les rôles ;
- une regex (précédée de `/`) pour indiquer plusieurs rôles (nouveauté de la version 16).

Exemple :

```
# TYPE    DATABASE      USER              ADDRESS       METHOD
hostssl   erp_hr        u_virginie,u_sharon 10.1.17.67/32 scram-sha-256
hostssl   all           +grp_admin        10.0.0.0/8    scram-sha-256
hostssl   /erp_dev[1-5] @prestataires.txt 10.1.0.0/16   scram-sha-256
hostssl   /erp.*        /u_.*             10.0.0.0/8    scram-sha-256
```

Ici, ne peuvent se connecter à la base **erp_hr** que deux utilisatrices nommées, et uniquement depuis une IP précise. Les membres du groupe **grp_admin** peuvent se connecter à toutes les bases depuis tout le réseau privé. Aux bases **erp_dev1** à **erp_dev5** peuvent se connecter les prestataires dont les rôles sont dans le fichier `prestataires.txt`, depuis un certain sous-réseau. Et tous les utilisateurs dont le rôle commence par **u_** peuvent se connecter aux bases dont le nom commence par **erp_**.

5.5.7 pg_hba.conf : colonne adresse IP

- Uniquement pour `host` / `hostssl` / `hostnossl`
- Adresse IPv4 ou IPv6 : format CIDR ou avec masque
  - `192.168.1.0/24`
  - `192.168.1.1/32`
  - `192.168.1.0  255.255.255.0`
- Nom d'hôte possible
  - mais coût recherche DNS

La colonne de l'adresse IP permet d'indiquer une adresse IP ou un sous-réseau IP. Il est donc possible de filtrer les connexions par rapport aux adresses IP, ce qui est une excellente protection. Voici deux exemples d'adresses IP au format adresse et masque de sous-réseau :

```
192.168.168.1 255.255.255.255
192.168.168.0 255.255.255.0
```

Et voici quelques exemples d'adresses IP (ou plages d'adresses) au format CIDR :

```
192.168.0.0/8        # réseau 192.168.x.y
192.168.66.0/24      # réseau 192.168.66.x
192.168.168.1/32     # 1 IP
0.0.0.0/0            # toutes les IPv4
::0/0                # toutes les IPv6
::1/128              # localhost en IPv6
fe80::7a31:c1ff:0000:0000/96  # un réseau en IPv6
```

Dans le doute, la vue `pg_hba_file_rules` indique le masque calculé.

Les alias `samehost` et `samenet` désignent le serveur (avec toutes ses adresses) et tous les réseaux auquels il appartient.

Il est possible d'utiliser un nom d'hôte ou un domaine DNS au prix d'une recherche DNS pour chaque hostname présent, pour chaque nouvelle connexion.

Rappelons que le paramètre `listen_addresses` dans `postgresql.conf` restreint les IP sur lesquelles que PostgreSQL écoute. En général on le modifie dès l'installation.

5.5.8 pg_hba.conf : colonne méthode

Quelle méthode d'authentification utiliser ?

- Interne / externe
  - et options dans la dernière colonne
- Le client peut en exiger une (v16)

La dernière colonne est généralement la méthode d'authentification, mais certaines acceptent des options dans une colonne supplémentaire. Nous verrons plus bas les différentes méthodes d'authentification, internes ou externes.

Le serveur a toujours le dernier mot pour imposer une méthode d'authentification, mais pour des raisons de sécurité le client peut aussi exiger celle qu'il attend (depuis PostgreSQL 16). Dans l'exemple suivant, le client exige la méthode `scram-sha-256`, mais le serveur est encore configuré en `md5` et la connexion échoue.

```
psql 'host=s1 dbname=postgres user=postgres require_auth=scram-sha-256'
psql: error: connection to server at "s1" (192.168.74.65), port 54325 failed:
 authentication method requirement "scram-sha-256" failed:
 server requested a hashed password
```

D'autres variantes pour `require_auth` sont :

- `require_auth=scram-sha-256,md5`
- `require_auth=!password`
- `require_auth=cert`

5.5.9 pg_hba.conf : colonne options

- Dépend de la méthode d'authentification
- Méthode externe : option `map`

Les options disponibles dépendent de la méthode d'authentification sélectionnée. Cependant, toutes les méthodes externes permettent l'utilisation de l'option `map`. Cette option a notamment pour but d'indiquer la carte de correspondance à sélectionner dans le fichier `pg_ident.conf` (voir plus bas).

5.5.10 pg_hba.conf : méthodes internes

- `trust` : dangereux !
- `reject` : pour interdire
- `password` : en clair, à éviter !
- `md5` : déconseillée
- `scram-sha-256`

La méthode `trust` est certainement la pire. À partir du moment où le rôle est reconnu, aucun mot de passe n'est demandé. Si le mot de passe est fourni malgré tout, il n'est pas vérifié. De plus, avant PostgreSQL 17, l'utilisation de la méthode `trust` n'est pas directement visible dans les traces.

N'utilisez pas la méthode `trust`.

La méthode `password` force la saisie d'un mot de passe. Cependant, ce dernier est envoyé en clair sur le réseau. Il n'est donc pas conseillé d'utiliser cette méthode, surtout sur un réseau non sécurisé. (Noter que PostgreSQL chiffrera toujours le mot de passe chiffré avec une des méthodes ci-dessous.)

La méthode `md5` est encore très utilisée mais obsolète. La saisie du mot de passe est forcée. Le mot de passe transite chiffré en `md5`. Cette méthode souffre néanmoins de certaines faiblesses décrites dans la section Mot de passe, on préférera la suivante.

La méthode `scram-sha-256` est la plus sécurisée, elle offre moins d'angles d'attaque que `md5`. Elle est à privilégier.

La méthode `reject` sert pour interdire définitivement un accès à un utilisateur, un réseau ou une base quelles que soient les lignes suivantes de `pg_hba.conf`. Par exemple, on veut que le rôle **u1** puisse se connecter à la base de données `b1` mais pas aux autres. Voici un moyen de le faire (pour une connexion via les sockets Unix) :

```
local  b1   u1  scram-sha-256
local  all  u1  reject
```

5.5.11 pg_hba.conf : méthodes externes

- Système externe : `ldap`, `radius`
- Kerberos : `gss`, `sspi`
- SSL : `cert`
- OS : `peer`, `pam`, `ident`, `bsd`

Certaines méthodes permettent d'utiliser des annuaires d'entreprise comme RADIUS, LDAP ou ActiveDirectory. Certaines méthodes sont spécifiques à un système d'exploitation.

La méthode `ldap` utilise un serveur LDAP pour authentifier l'utilisateur.

La méthode `radius` permet d'utiliser un serveur RADIUS pour authentifier l'utilisateur.

La méthode `gss` (GSSAPI) correspond au protocole du standard de l'industrie pour l'authentification sécurisée définie dans RFC 2743. PostgreSQL supporte GSSAPI avec l'authentification Kerberos suivant la RFC 1964 ce qui permet de faire du *Single Sign-On*. C'est la méthode à utiliser avec Active Directory. `sspi` (uniquement dans le monde Windows) permet d'utiliser NTLM faute d'Active Directory.

La méthode `cert` utilise un certificat SSL sur le client[13], sans mot de passe.

La méthode `ident` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Un démon fournissant le service ident est nécessaire. (Les paquets RPM la mettaient en place jusque PostgreSQL 12 inclus pour une connexion depuis `localhost`, mais `peer` est maintenant recommandé.)

La méthode `peer` permet d'associer les noms des utilisateurs du système d'exploitation aux noms des utilisateurs du système de gestion de bases de données. Ceci n'est possible qu'avec une connexion locale. Par défaut, les paquets d'installation l'utilisent pour autoriser l'utilisateur système **postgres** à se connecter à l'instance locale, sans passer par le réseau, en tant que **postgres** sans mot de passe.

[13] https://docs.postgresql.fr/current/auth-cert.html

La méthode `pam` authentifie l'utilisateur en passant par les *Pluggable Authentication Modules* (PAM) fournis par Linux et d'autres Unix. De manière similaire, la méthode `bsd` utilise le système d'authentification BSD sur OpenBSD.

5.5.12 Un (mauvais) exemple de pg_hba.conf

Un exemple:

```
TYPE    DATABASE  USER        ADDRESS         METHOD
local   all       postgres                    peer
local   web       web                         md5
local   sameuser  all                         peer
host    all       postgres    127.0.0.1/32    peer
host    all       all         127.0.0.1/32    md5
host    all       all         89.192.0.3/8    md5
hostssl recherche recherche   89.192.0.4/32   scram-sha-256
```

à ne pas suivre…

Ce fichier comporte plusieurs erreurs. Tout d'abord,

```
host    all       all         127.0.0.1/32    md5
```

autorise tous les utilisateurs, en IP, en local (127.0.0.1) à se connecter à **toutes** les bases, ce qui est en contradiction avec :

```
local   sameuser  all                         peer
```

qui, sera appliqué d'abord et restreint à la base de même nom.

Dans cette règle :

```
host    all       all         89.192.0.3/8    md5
```

le masque CIDR `/8` est incorrect, et donc, au lieu d'autoriser seulement 89.192.0.3 à se connecter, on autorise tout le réseau 89.*.

L'entrée :

```
hostssl recherche recherche   89.192.0.4/32   scram-sha-256
```

est bonne, mais inutile, car masquée par la ligne précédente : toute ligne correspondant à cette entrée correspondra aussi à la ligne précédente. Le fichier étant lu séquentiellement, cette dernière entrée ne sert à rien.

Enfin, il aurait mieux valu que tous les mots de passe utilisent le chiffrement `scram-sha-256` et non `md5`.

5.5.13 Mapping : pg_ident.conf

- Correspondance entre :
  - utilisateurs authentifiés (serveur, LDAP)
  - rôle de la base
- Fichier `pg_ident.conf` :

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
pgadmins        joe                     postgres
pgadmins        zoe                     postgres
```

- Prise en compte des modifications : rechargement
- Includes possibles
- Contrôle : `pg_ident_file_mapping` (v15+)

En utilisant une méthode d'authentification externe, locale ou à distance, le nom de l'utilisateur authentifié ne correspond pas forcément au nom du rôle avec lequel on souhaite se connecter. On peut le préciser avec `-U` mais il y a plus simple et sécurisé.

Par exemple, deux DBA ont les comptes **joe** et **zoe** sur le système d'exploitation. Ils souhaitent pouvoir se connecter en tant que rôle **postgres** sans saisie de mot de passe depuis leur compte, sans devoir faire un `sudo -iu postgres` pour lesquels on ne va pas forcément leur donner les droits.

Ceci n'est pas possible avec la configuration par défaut :

```
joe$ psql -U postgres
psql: error: FATAL:  Peer authentication failed for user "postgres"
```

Mais cela devient possible en configurant correctement les fichiers `pg_hba.conf` et `pg_ident.conf`. Il est tout d'abord nécessaire d'indiquer trois informations dans le fichier `pg_ident.conf`, une « map », les utilisateurs authentifiés (ici **joe** et **zoe**) et le rôle de connexion (ici **postgres**) :

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
pgadmins        joe                     postgres
pgadmins        zoe                     postgres
```

À partir de la version 15, il est possible de vérifier que la configuration de ce fichier est bonne. Pour cela, il faut lire la vue `pg_ident_file_mappings` (et vérifier que le champ `error` est vide).

```
SELECT * FROM pg_ident_file_mappings;

 line_number | map_name | sys_name | pg_username | error
-------------+----------+----------+-------------+-------
          43 | pgadmins | joe      | postgres    |
          44 | pgadmins | zoe      | postgres    |
```

Il faut ensuite ajouter cette *map* à la ligne de configuration d'authentification en haut du fichier `pg_hba.conf` :

```
local    all    all     peer      map=pgadmins
```

Après rechargement de la conf, la connexion de **joe** ou **zoe** à **postgres** en local se fera. Noter que la sécurité de l'accès à PostgreSQL repose à présent sur la sécurisation des utilisateurs système **joe** et **zoe** !

Tout ceci n'est valable que pour les connexions depuis le système d'exploitation même. En pratique, `pg_ident.conf` est utilisé pour se connecter directement en tant que **postgres** depuis un compte utilisateur sur le serveur, ou avec une authentification LDAP.

Depuis PostgreSQL 16, `pg_ident.conf` supporte les mêmes directives d'inclusion que `pg_hba.conf` (`include`, `include_if_exists`, `include_dir`), et le fichier d'origine d'une ligne est visible dans `pg_ident_file_mappings`.

5.6 SYNCHRONISATION DES RÔLES

- PostgreSQL externalise l'authentification uniquement
  - Le rôle, ses paramètres, ses héritages sont toujours dans PostgreSQL
- ldap2pg synchronise les rôles depuis un annuaire LDAP
  - un rôle PostgreSQL absent de l'annuaire est supprimé
  - sait synchroniser les privilèges
  - configuration avancée en YAML
  - https://labs.dalibo.com/ldap2pg

PostgreSQL sait externaliser la vérification du mot de passe ou authentifier depuis une autre méthode qu'un mot de passe. Néanmoins, PostgreSQL n'externalise jamais la configuration d'un rôle.

Au delà de son nom et de son mot de passe, un rôle possède des options (`SUPERUSER`, `LOGIN`, etc.), des héritages, des privilèges, des privilèges par défaut pour chaque base voire pour chaque schéma, des paramètres globaux ou par base. Tout cela reste dans PostgreSQL, même si l'authentification est externe.

L'outil ldap2pg permet de centraliser les rôles dans l'annuaire d'entreprise. ldap2pg est un binaire (programmé en Go) à installer à côté de PostgreSQL ou ailleurs.

ldap2pg traite une configuration avancée au format YAML décrivant toutes les recherches LDAP énumérant les rôles à provisionner dans l'annuaire. Par exemple : la liste des DBA, la liste des développeurs d'une application, etc.

À chaque recherche LDAP est associé un modèle de rôle où sont substitués des attributs LDAP fournit par l'annuaire. ldap2pg sait également configurer les paramètres propres à chaque rôle, les privilèges par défaut, et les privilèges sur les objets.

La recherche LDAP des rôles d'une instance est indépendante de l'authentification. On peut très bien rechercher dans l'annuaire la liste des DBA avec le protocole LDAP et authentifier les connexions avec Kerberos.

Page du projet : https://labs.dalibo.com/ldap2pg

5.7 TÂCHES DE MAINTENANCE

- Trois opérations essentielles
  - `VACUUM`
  - `ANALYZE`
  - `REINDEX`
- En arrière-plan : démon autovacuum (pour les deux premiers)
- Optionnellement : automatisable par cron
- Manuellement : `VACUUM ANALYZE table` (batchs, gros imports…)
- Par : propriétaire, superuser, `GRANT MAINTAIN` ou `pg_maintain`

PostgreSQL demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à conserver un système performant et agréable à utiliser.

La maintenance d'un serveur PostgreSQL revient à s'occuper de trois opérations :

- le `VACUUM`, pour éviter une fragmentation trop importante des tables ;
- l'`ANALYZE`, pour mettre à jour les statistiques sur les données contenues dans les tables ;
- le `REINDEX`, pour reconstruire les index.

Il s'agit donc de maintenir, voire d'améliorer, les performances du système. Il ne s'agit en aucun cas de s'assurer de la stabilité du système.

Généralement on se reposera sur le processus d'arrière-plan **autovacuum**, qui s'occupe des `VACUUM` et `ANALYZE` (mais pas `REINDEX`), en fonction de l'activité, et prend soin de ne pas la gêner.

Un appel explicite est parfois nécessaire, notamment au sein de batchs ou de gros imports, car l'autovacuum n'a pas forcément eu le temps de passer entre deux étapes, et les statistiques ne sont alors pas à jour : le planificateur pense que les tables sont encore vides et peut choisir un plan désastreux. On lancera donc systématiquement au moins un `ANALYZE` sur les tables modifiées après les modifications lourdes. Un `VACUUM ANALYZE` est parfois encore plus intéressant, notamment si des données ont été modifiées ou effacées, ou si les requêtes suivantes peuvent profiter d'un parcours d'index seul (*Index Only Scan*).

Il est parfaitement possible et très courant de planifier des exécutions régulières, avec cron, `pg_cron`, ou tout autre ordonnanceur, de `VACUUM` et/ou `ANALYZE` sur tout ou partie des tables, voire de `REINDEX` ou `CLUSTER` dans certains cas.

Ces opérations de maintenance peuvent être effectuées par le propriétaire de la table, ou par un superutilisateur. À partir de PostgreSQL 17, il suffit d'un utilisateur possédant le rôle `pg_maintain`, ou un utilisateur s'ayant vu octroyer un `GRANT MAINTAIN` sur la table. Cet utilisateur pourra lancer les commandes de maintenance sans pouvoir lire les données.

5.7.1 Maintenance : VACUUM

- `VACUUM nomtable ;`
  - nettoie les espaces morts pour réutilisation (& autre maintenance)
  - ne gêne pas l'activité
- Version démon :
  - `autovacuum`
- Outil shell :
  - `vacuumdb --echo`
- Suivi :
  - vue `pg_stat_progress_vacuum`

PostgreSQL ne supprime pas des tables les versions périmées des lignes après un `UPDATE` ou un `DELETE`, elles deviennent juste invisibles. La commande `VACUUM` permet de récupérer l'espace utilisé par ces lignes afin d'éviter un accroissement continuel du volume occupé sur le disque.

Une table qui subit beaucoup de mises à jour et suppressions nécessitera des nettoyages plus fréquents que les tables rarement modifiées. Le `VACUUM` « simple » (`VACUUM nomdematable ;`) marque les données expirées dans les tables et les index pour une utilisation future. Il ne tente pas de rendre au système de fichiers l'espace utilisé par les données obsolètes, sauf si l'espace est à la fin de la table et qu'un verrou exclusif de table peut être facilement obtenu. L'espace inutilisé au début ou au milieu du fichier ne provoque pas un raccourcissement du fichier et ne redonne pas d'espace mémoire au système d'exploitation. De même, l'espace d'une colonne supprimée n'est pas rendu.

Cet espace libéré n'est pas perdu : il sera disponible pour les prochaines lignes insérées et mises à jour, et la table n'aura pas besoin de grandir. (Nous verrons plus bas un ordre différent, `VACUUM FULL`, pour récupérer l'espace au besoin.)

`VACUUM` est paramétré pour être lancé sans gêne pour les utilisateurs. Il va surtout générer des lectures et écritures supplémentaires. On verra plus loin que l'autovacuum s'occupe de tout cela en tâche de fond et de manière encore moins intrusive, mais il arrive encore que l'on lance un `VACUUM` manuellement. Noter qu'un `VACUUM` s'occupe également de quelques autres opérations de maintenance qui ne seront pas détaillées ici.

L'option `VERBOSE` permet de suivre ce qui a été fait. Dans l'exemple suivant, 100 000 lignes sont nettoyées dans 541 blocs, mais 300 316 lignes ne peuvent être supprimées car une autre transaction reste susceptible de les voir.

```
VACUUM VERBOSE livraisons ;
```

```
INFO:  vacuuming "public.livraisons"
INFO:  "livraisons": removed 100000 row versions in 541 pages
INFO:  "livraisons": found 100000 removable, 300316 nonremovable
                                          row versions in 2165 out of 5406 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 6249883
There were 174 unused item pointers.
Skipped 0 pages due to buffer pins, 540 frozen pages.
0 pages are entirely empty.
CPU: user: 0.04 s, system: 0.00 s, elapsed: 0.08 s.
VACUUM
Temps : 88,990 ms
```

L'ordre `VACUUM` possède de nombreux paramètres que nous ne décrirons pas ici, car ils servent peu au quotidien (voir le module M5[14], notamment pour accélérer les opérations du `VACUUM`).

Supervision :

La vue `pg_stat_progress_vacuum` permet de suivre un `VACUUM` simple pendant son déroulement :

```
TABLE pg_stat_progress_vacuum \gx
```

```
-[ RECORD 1 ]------+--------------
pid                | 2603780
datid              | 1308955
datname            | pgbench_100
relid              | 1308962
phase              | scanning heap
heap_blks_total    | 163935
heap_blks_scanned  | 3631
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuple_bytes | 67108864
dead_tuple_bytes   | 0
num_dead_item_ids  | 0
indexes_total      | 0
indexes_processed  | 0
```

La vue `pg_stat_user_tables` contient pour chaque table la date du dernier passage d'un `VACUUM` simple (champ `last_vacuum`) celle du dernier passage automatique (`last_autovacuum`).

```
SELECT * FROM pg_stat_user_tables where relname = 'pgbench_accounts' \gx
```

```
-[ RECORD 1 ]-------+-----------------------------
relid               | 1308962
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 6
last_seq_scan       | 2024-10-18 11:03:08.493099+02
seq_tup_read        | 50000000
idx_scan            | 9954460
last_idx_scan       | 2024-10-18 19:14:30.946211+02
idx_tup_fetch       | 89954076
n_tup_ins           | 10000000
```

[14] https://dali.bo/m5_html

```
n_tup_upd           | 3318032
n_tup_del           | 0
n_tup_hot_upd       | 3127413
n_tup_newpage_upd   | 190619
n_live_tup          | 9992800
n_dead_tup          | 345265
n_mod_since_analyze | 1894832
n_ins_since_vacuum  | 0
last_vacuum         | 2024-10-18 17:58:53.305703+02
last_autovacuum     | ø
last_analyze        | 2024-10-18 11:01:20.131883+02
last_autoanalyze    | 2024-10-18 19:12:36.344106+02
vacuum_count        | 7
autovacuum_count    | 0
analyze_count       | 1
autoanalyze_count   | 1
```

Pour les passages précédents, il faudra se rabattre sur les traces (on conseille de positionner `log_autovacuum_min_duration` suffisamment bas, ou à 0).

Il est important de vérifier que les tables actives sont régulièrement nettoyées.

Outil en ligne de commande :

L'outil `vacuumdb` permet d'exécuter depuis le shell un `VACUUM` sur une ou toutes les bases. C'est l'outil idéal lorsque l'on planifie un nettoyage quotidien ou hebdomadaire.

Il sait exécuter des `VACUUM` sur plusieurs tables en parallèle, ou profiter des optimisations liées aux nettoyages en masse apparues en v16.

```
$ vacuumdb --echo --all
```

```
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « pgbench »
RESET search_path;
SELECT c.relname, ns.nspname FROM pg_catalog.pg_class c
 JOIN pg_catalog.pg_namespace ns ON c.relnamespace OPERATOR(pg_catalog.=) ns.oid
 LEFT JOIN pg_catalog.pg_class t ON c.reltoastrelid OPERATOR(pg_catalog.=) t.oid
 WHERE c.relkind OPERATOR(pg_catalog.=) ANY (array['r', 'm'])
 ORDER BY c.relpages DESC;
SELECT pg_catalog.set_config('search_path', '', false);
VACUUM (SKIP_DATABASE_STATS) public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_statistic;
…
…
VACUUM (ONLY_DATABASE_STATS);
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « postgres »
…
```

```
…
VACUUM (ONLY_DATABASE_STATS);
```

L'ordre `VACUUM` connaît beaucoup de variantes. Surtout, ne confondez pas la version simple ci-dessus avec sa variante `VACUUM FULL` ci-dessous.

5.7.2 Maintenance : VACUUM FULL

- `VACUUM FULL nomtable ;`
  - défragmente la table
  - réécrit (place nécessaire !)
  - verrou exclusif (ni lecture ni écriture !)
  - réindexe
  - utilisation exceptionnelle
  - vue `pg_stat_progress_cluster`

Un `VACUUM` simple fait rarement gagner de l'espace disque. Il faut utiliser l'option `FULL` pour ça : la commande `VACUUM FULL nomtable ;` réécrit la table en ne gardant que les données actives, et au final libère donc l'espace consommé par les lignes périmées ou les colonnes supprimées, et le rend au système d'exploitation. Les index sont réécrits au passage.

Inconvénient principal : `VACUUM FULL` acquiert un verrou exclusif sur chaque table concernée : personne ne peut plus y écrire ni même lire avant la fin de l'opération, et les sessions accédant aux tables sont mises en attente. Cela peut être long pour de grosses tables. D'autre part, le `VACUUM FULL` peut lui-même attendre la libération d'un verrou, tout en bloquant les transactions suivantes (phénomène d'empilement des verrous). Il est conseillé d'opérer par exemple ainsi :

```
SET lock_timout TO '3s';
VACUUM (FULL, VERBOSE) nomtable ;
```

Ainsi l'ordre `VACUUM FULL` sera annulé s'il n'obtient pas son verrou assez vite.

Autre inconvénient : `VACUUM FULL` écrit la nouvelle version de la table *à côté* de l'ancienne version avant d'effacer cette dernière : l'espace occupé peut donc temporairement doubler. Si vos disques sont presque pleins, vous ne pourrez donc pas faire un `VACUUM FULL` d'une grosse table pour récupérer de l'espace !

L'autovacuum ne procédera jamais à un `VACUUM FULL`, vous devrez toujours le demander explicitement. On le réservera aux périodes de maintenance, dans les cas où il est vraiment nécessaire.

En effet, il ne sert à rien de chercher à réduire au strict minimum la taille des tables par des `VACUUM FULL` répétés. Dans une base active, les espaces libres sont vite réutilisés par de nouvelles données. Le *bloat* (l'espace inutilisé d'une table) se stabilise généralement dans une proportion dépendant des débits d'insertions, suppressions et modifications dans la table.

Supervision :

La vue `pg_stat_progress_cluster` permet de suivre un `VACUUM FULL` (à partir de PostgreSQL 12). Son nom provient de la proximité avec la commande `CLUSTER` (voir plus bas).

Les `VACUUM FULL` ne sont pas tracés dans la vue `pg_stat_user_tables`.

Outil en ligne de commande :

L'outil `vacuumdb` possède une option `--full`.

5.7.3 VACUUM vs VACUUM FULL

- `VACUUM`
  - maintenance quotidienne
  - entre étapes d'un batch
  - l'autovacuum suffit généralement
- `VACUUM FULL`
  - après de grosses modifications
  - exceptionnel

Quand faut-il utiliser `VACUUM` sur une table ?

- pour des nettoyages réguliers ;
- entre les étapes d'un batch ;
- si vous constatez que l'autovacuum ne passe pas assez souvent et qu'un changement de paramétrage ne suffit pas ;
- et ce, pendant que votre base tourne.

Quand faut-il utiliser `VACUUM FULL` sur une table ?

- après des suppressions massives de données ;
- si le verrou exclusif ne gêne pas la production ;
- dans le cadre d'une maintenance exceptionnelle.

Des `VACUUM` standards et une fréquence modérée sont une meilleure approche que des `VACUUM FULL`, même non fréquents, pour maintenir des tables mises à jour fréquemment : faites confiance à l'autovacuum jusque preuve du contraire.

`VACUUM FULL` est recommandé dans les cas où vous savez que vous avez supprimé ou modifié une grande partie des lignes d'une table, et que les espaces libres ne seront pas à nouveau remplis assez vite, de façon à ce que la taille de la table soit réduite de façon conséquente.

Les deux outils peuvent se lancer à la suite. Après un `VACUUM FULL` (bloquant) sur une table, on lance souvent immédiatement un `VACUUM ANALYZE`. Cela semble inutile du point de vue des données, mais les autres opérations de maintenance impliquées peuvent améliorer les performances.

5.7.4 Maintenance : ANALYZE

- Met à jour les statistiques sur les données pour l'optimiseur de requêtes
- Géré par l'autovacuum
  - Parfois manuel : batch, `ALTER TABLE`, tables temporaires…
- Échantillonnage :
  - `default_statistics_target` (défaut 100)
  - `ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500;`
  - Attention au temps de planification !
- Progression avec `pg_stat_progress_analyze`

L'optimiseur de requêtes de PostgreSQL s'appuie sur des informations statistiques calculées à partir des données des tables. Ces statistiques sont récupérées par la commande `ANALYZE`, qui peut être invoquée seule ou comme une option de `VACUUM`. Il est important d'avoir des statistiques relativement à jour sans quoi des mauvais choix dans les plans d'exécution pourraient pénaliser les performances de la base.

L'autovacuum de PostgreSQL appelle au besoin `ANALYZE` si l'activité de la table le nécessite. C'est généralement suffisant, même s'il est fréquent de modifier le paramétrage sur de grosses tables.

Il est possible de programmer `ANALYZE` périodiquement (le dimanche, la nuit par exemple, à l'aide d'une commande `cron` par exemple), éventuellement couplé à un `VACUUM` :

```
VACUUM ANALYZE nomdematable ;
```

Il existe des cas où lancer un `ANALYZE` manuellement est nécessaire :

- en mode « batch » : l'autovacuum n'a pas forcément le temps de passer entre deux étapes, on peut être amené à intercaler un `VACUUM ANALYZE` sur des tables modifiées ;

- quand certains plans de requêtes affichent des statistiques aberrantes : la mise à jour des statistiques peut suffire (et l'on regardera ensuite dans `pg_stat_user_tables.last_autoanalyze` si l'autovacuum a tardé et s'il y a un ajustement à faire ce côté) ;
- après un `ALTER TABLE [...] ALTER COLUMN`, car les statistiques de la colonne peuvent disparaître, ou bien sûr lors de l'ajout d'une colonne pré-remplie ;
- lors de l'ajout d'un index fonctionnel : l'`ANALYZE` mène à la création d'une nouvelle entrée dans `pg_statistics` ;
- lors de l'utilisation des tables temporaires : l'autovacuum ne les voit pas.

Le paramètre `default_statistics_target` définit l'échantillonnage par défaut des statistiques pour les colonnes de chacune des tables. La valeur par défaut est de 100. Ainsi, pour chaque colonne, 30 000 lignes sont choisies au hasard, et les 100 valeurs les plus fréquentes et un histogramme à 100 bornes sont stockés dans `pg_statistics` en guise d'échantillon représentatif des données.

Des valeurs supérieures provoquent un ralentissement important d'`ANALYZE`, un accroissement de la table `pg_statistics`, et un temps de calcul des plans d'exécution plus long. On conserve généralement la valeur `100` par défaut (sauf peut-être sur certaines grosses bases aux requêtes complexes et longues, comme des entrepôts de données).

Voici la commande à utiliser si l'on veut modifier cette valeur pour une colonne précise, la valeur ainsi spécifiée prévalant sur la valeur de `default_statistics_target` :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 200 ;

ANALYZE ma_table ;
```

Sans l'`ANALYZE` explicite, la mise à jour attendrait le prochain passage de l'autovacuum.

La vue `pg_stat_user_tables` contient aussi les dates du dernier passage d'un `ANALYZE` manuel (champ `last_analyze`) ou automatique (`last_autoanalyze`). Là encore, vérifier que les tables actives sont régulièrement analysées.

La vue `pg_stat_progress_analyze` permet de suivre l'avancement des `ANALYZE` en cours (sans historisation) :

```
TABLE pg_stat_progress_analyze ;

-[ RECORD 1 ]-------------+----------------------
pid                       | 518842
datid                     | 207023
datname                   | pgbench
relid                     | 207030
phase                     | acquiring sample rows
sample_blks_total         | 30000
sample_blks_scanned       | 26862
ext_stats_total           | 0
ext_stats_computed        | 0
child_tables_total        | 0
child_tables_done         | 0
current_child_table_relid | 0
```

5.7.5 Maintenance : REINDEX

- Lancer `REINDEX` régulièrement permet
  - de gagner de l'espace disque
  - d'améliorer les performances
  - de réparer un index corrompu/invalide
- `VACUUM` ne provoque pas de réindexation
- `VACUUM FULL` réindexe
- Clause `CONCURRENTLY`
- Clause `TABLESPACE` (v14+)
- `REINDEX SYSTEM`
- `reindexdb` pour des réindexations en masse

Principe :

`REINDEX` reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. La même commande peut réindexer tous les index d'une table :

```
REINDEX INDEX nomindex ;
REINDEX (VERBOSE) TABLE nomtable ;
```

Les pages d'index qui sont devenues complètement vides sont récupérées pour être réutilisées. Il existe toujours la possibilité d'une utilisation inefficace de l'espace : même s'il ne reste qu'une clé d'index dans une page, la page reste allouée. La possibilité d'inflation n'est pas indéfinie, mais il est souvent utile de planifier une réindexation périodique pour les index fréquemment modifiés.

De plus, pour les index B-tree, un index tout juste construit est plus rapide qu'un index qui a été mis à jour plusieurs fois. En effet, dans un index nouvellement créé, les pages logiquement adjacentes sont aussi physiquement adjacentes.

La réindexation est aussi utile dans le cas d'un index corrompu. Ce cas est heureusement très rare, et souvent lié à des problèmes matériels.

Comme avec `CREATE INDEX`, il est généralement souhaitable d'ajouter la clause `CONCURRENTLY` pour réindexer sans bloquer les écritures dans la table (les lectures restent possibles) :

```
REINDEX (VERBOSE) INDEX nomindex CONCURRENTLY ;
```

Pour réindexer les tables systèmes :

```
REINDEX (VERBOSE) SYSTEM nom_de_base_courante ;
```

Ce peut être ponctuellement utile.

Depuis la version 14, il est possible de réindexer un index tout en le changeant de tablespace. Pour cela, il faut utiliser la clause `TABLESPACE` avec en argument le nom du tablespace de destination.

> L'opération `VACUUM` (sans `FULL`) nettoie les index, mais ne provoque pas de réindexation complète. Une certaine fragmentation peut alors subsister. Par contre, une réindexation est effectuée lors d'un `VACUUM FULL`.

Index invalide :

Les index « invalides » sont inutilisables et ignorés, et doivent être reconstruits. Ce statut apparaît en bas de la description de la table associée :

```
# \d+ pgbench_accounts
…
Index :
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid) INVALID
```

Un index peut devenir invalide si l'on utilisei l'option `CONCURRENTLY` des ordres `CREATE INDEX` et `REINDEX`. Cette clause permet de créer/réindexer un index sans bloquer les écritures dans la table. Cependant, si, au bout de deux passes, l'index n'est toujours pas complet, il est considéré comme invalide, et doit être soit détruit, soit reconstruit avec `REINDEX`.

reindexdb :

La commande système `reindexdb` peut être utilisée pour réindexer une table, une base ou une instance entière. Elle génère des ordres `REINDEX` (que l'on peut voir avec l'option `--echo`), sur des tables (option `--table`, à répéter au besoin), des schémas (`--schema`), des index (`--index`), les tables système (`--system`)…

De manière similaire à `vacuumdb`, l'intêrêt de `reindexdb`, par rapport à des exécutions d'un `REINDEX` global ou sur une table ou un schéma, réside dans ses possibilités d'exécution en masse et de parallélisation. L'option `--jobs=N` utilise plusieurs connexions pour autant de réindexations en parallèle. Une table est réindexée par un seul job pour des raisons techniques. Cela marche bien avec `--table` ou `--schema`, mais il faut PostgreSQL 17 pour qu'une liste d'index à reconstruire (`--index`) soit parallélisée.

S'il y a trop de jobs, il y a un risque de saturer la RAM ou le CPU (sachant que chaque indexation est déjà susceptible d'utiliser plusieurs processeurs).

Autre point fort, l'option `--all` permet de réindexer toutes les bases. À partir de PostgreSQL 17, cette dernière option peut se conjuguer avec `--table`, `--index`, etc. pour réindexer uniquement des objets de même nom dans toutes les bases.

Exemples (les sorties ont été allégées de quelques ordres techniques) :

```
$ reindexdb -d pgbench --table=pgbench_accounts --echo --verbose
```

```
SELECT pg_catalog.set_config('search_path', '', false);
RESET search_path;
[…]
REINDEX (VERBOSE) TABLE public.pgbench_accounts;
INFO:  index "pgbench_accounts_pkey" was reindexed
DÉTAIL : CPU: user: 0.08 s, system: 0.01 s, elapsed: 0.10 s

$ reindexdb -d pgbench_10 --echo \
  --index=pgbench_accounts_pkey --index=pgbench_branches_pkey

SELECT pg_catalog.set_config('search_path', '', false);
RESET search_path;
[…]
REINDEX INDEX public.pgbench_accounts_pkey;
RESET search_path;
[…]
REINDEX INDEX public.pgbench_branches_pkey;

$ reindexdb -d pgbench --system --echo

SELECT pg_catalog.set_config('search_path', '', false);
REINDEX SYSTEM pgbench;
```

5.7.6 Maintenance : CLUSTER

- `CLUSTER`
  - alternative à `VACUUM FULL`
  - tri des données de la table suivant un index
- Attention, `CLUSTER` nécessite près du double de l'espace disque utilisé pour stocker la table et ses index
- Progression avec `pg_stat_progress_cluster`

La commande `CLUSTER` provoque une réorganisation des données de la table en triant les lignes suivant l'ordre indiqué par l'index. Du fait de la réorganisation, le résultat obtenu est équivalent à un `VACUUM FULL` en terme de fragmentation. Elle verrouille tout aussi complètement la table et nécessite autant de place.

Les valeurs des données sont alors corrélées avec leur emplacement physique. L'intérêt d'une telle réorganisation peut être l'amélioration de l'effet de cache, ou l'utilisation d'un index de type spécifique (BRIN).

Attention, cette réorganisation est ponctuelle, et les données modifiées ou insérées par la suite n'en tiennent généralement pas compte. L'opération peut donc être à refaire après un certain temps.

Comme après un `VACUUM FULL`, lancer un `VACUUM ANALYZE` manuellement peut être bénéfique pour les performances.

En ligne de commande, l'outil associé `clusterdb` permet de lancer la réorganisation de tables ayant déjà fait l'objet d'une « clusterisation ».

La vue `pg_stat_progress_cluster` permet de suivre le déroulement du `CLUSTER`.

5.7.7 Maintenance : automatisation

- Automatisation des tâches de maintenance
- Cron sous Unix
- Tâches planifiées sous Windows

L'exécution des commandes `VACUUM`, `ANALYZE` et `REINDEX` peut se faire manuellement dans certains cas. Il est cependant préférable de mettre en place une exécution automatique de ces commandes. La plupart des administrateurs utilise cron sous Unix et les tâches planifiées sous Windows. pgAgent peut aussi être d'une aide précieuse pour la mise en place de ces opérations automatiques.

Peu importe l'outil. L'essentiel est que ces opérations soient réalisées et que le statut de leur exécution soit vérifié périodiquement.

La fréquence d'exécution dépend principalement de la fréquence des modifications et suppressions pour le `VACUUM` et de la fréquence des insertions, modifications et suppressions pour l'`ANALYZE`.

5.7.8 Maintenance : autovacuum

- Automatisation par cron
  - simple, voire simpliste
- Processus autovacuum
  - `VACUUM` / `ANALYZE` si nécessaire
  - Nombreux paramètres
  - Nécessite la récupération des statistiques d'activité

L'automatisation du vacuum par cron est simple à mettre en place. Cependant, elle s'exécute pour toutes les tables, sans distinction. Que la table ait été modifiée plusieurs millions de fois ou pas du tout, elle sera traitée par le script. À l'inverse, l'autovacuum est un outil qui vérifie l'état des tables et, suivant le dépassement d'une limite, déclenche ou non l'exécution d'un `VACUUM` ou d'un `ANALYZE`, voire des deux.

L'autovacuum est activé par défaut, et il est conseillé de le laisser ainsi. Son paramétrage permet d'aller assez loin si nécessaire selon la taille et l'activité des tables.

5.7.9 Maintenance : Script de REINDEX

- Automatisation par cron
- Recherche des index fragmentés
  - extension `pgstattuple`, fonction `pgstatindex()`
- Si clé primaire ou contrainte unique :
  - `REINDEX`
- Sinon :
  - `CREATE INDEX CONCURRENTLY`
- Exemple

Voici un script créé pour un client dans le but d'automatiser la réindexation uniquement pour les index le méritant. Pour cela, il vérifie les index fragmentés avec la fonction `pgstatindex()` de

l'extension `pgstattuple` (installable avec un simple `CREATE EXTENSION pgstattuple ;` dans chaque base).

Au-delà de 30 % de fragmentation (par défaut), l'index est réindexé. Pour minimiser le risque de blocage, le script utilise `CREATE INDEX CONCURRENTLY` en priorité, et `REINDEX` dans les autres cas (clés primaires et contraintes uniques).

`REINDEX` possède aussi l'option `CONCURRENTLY`. Ce script pourrait l'utiliser après avoir détecté qu'il se trouve sur une version compatible.

```
#!/bin/bash
# Script de réindexation d'une base
# ce script va récupérer la liste des index disponibles sur la base
# et réindexer l'index s'il est trop fragmenté ou invalide

# Mode debug
#set -x

# Récupération de la base maintenance
if test -z "$PGDATABASE"; then
  export PGDATABASE=postgres
fi

# quelques constantes personnalisables
TAUX_FRAGMENTATION_MAX=30
NOM_INDEX_TEMPORAIRE=index_traitement_en_cours
NB_TESTS=3
BASES=""

# Quelques requêtes
REQ_LISTEBASES="SELECT array_to_string(array(
  SELECT datname
  FROM pg_database
  WHERE datallowconn AND datname NOT IN ('postgres', 'template1')), ' ')"
REQ_LISTEINDEX="
SELECT n.nspname as \"Schéma\", tc.relname as \"Table\", ic.relname as \"Index\",
  i.indexrelid as \"IndexOid\",
  i.indisprimary OR i.indisunique as \"Contrainte\", i.indisvalid as \"Valide?\",
  round(100-(pgstatindex(n.nspname||'.'||ic.relname)).avg_leaf_density)
    as \"Fragmentation\",
  pg_get_indexdef(i.indexrelid) as \"IndexDef\"
FROM pg_index i
JOIN pg_class ic ON i.indexrelid=ic.oid
JOIN pg_class tc ON i.indrelid=tc.oid
JOIN pg_namespace n ON tc.relnamespace=n.oid
WHERE n.nspname <> 'pg_catalog'
  AND n.nspname !~ '^pg_toast'
ORDER BY ic.relname;"

# vérification de la liste des bases
if test $# -gt 1; then
  echo "Usage: $0 [nom_base]"
  exit 1
elif test $# -eq 1; then
  BASE_PRESENTE=$(psql -XAtqc \
  "SELECT count(*) FROM pg_database WHERE datname='$1'" 2>/dev/null)
```

```
  if test $BASE_PRESENTE -ne 1; then
    echo "La base de donnees $BASE n'existe pas."
    exit 2
  fi

  BASES=$1
else
  BASES=$(psql -XAtqc "$REQ_LISTEBASES" 2>/dev/null)
fi

# Pour chaque base
for BASE in $BASES
do
    # Afficher la base de données
    echo "########## $BASE ##########"

  # Vérification de la présence de la fonction pgstatindex
  FONCTION_PRESENTE=$(psql -XAtqc \
  "SELECT count(*) FROM pg_proc WHERE proname='pgstatindex'" $BASE 2>/dev/null)
  if test $FONCTION_PRESENTE -eq 0; then
    echo "La fonction pgstatindex n'existe pas."
    echo "Veuillez installer le module pgstattuple."
    exit 3
  fi

  # pour chaque index
  echo "Récupération de la liste des index (ratio cible $TAUX_FRAGMENTATION_MAX)..."
  psql -XAtF " " -c "$REQ_LISTEINDEX" $BASE | \
  while read schema table index indexoid contrainte validite fragmentation definition
  do
  #  NaN (not a number) est possible si la table est vide
  #  dans ce cas, une réindexation est rapide
    if test "$fragmentation" = "NaN"; then
      fragmentation=0
    fi

  #  afficher index, validité et fragmentation
      if test "$validite" = "t"; then
        chaine_validite="valide"
      else
        chaine_validite="invalide"
      fi
    echo "Index $index, $chaine_validite, ratio libre ${fragmentation}%"

  #  si index fragmenté ou non valide
      if test "$validite" = "f" -o $fragmentation -gt $TAUX_FRAGMENTATION_MAX; then
  #  verifier les verrous sur l'index, attendre un peu si nécessaire
        verrous=1
        tests=0
        while test $verrous -gt 0 -a $tests -le $NB_TESTS
        do
      if test $tests -gt 0; then
        echo \
    "  objet verrouillé, attente de $tests secondes avant nouvelle tentative..."
        sleep $tests
      fi
```

```
    verrous=$(psql -XAtqc \
    "SELECT count(*) FROM pg_locks WHERE relation=$indexoid" 2>/dev/null)
    tests=$(($tests + 1))
      done
    if test $verrous -gt 0; then
      echo "  objet toujours verrouillé, pas de reindexation pour $schema.$index"
      continue
    fi

#    si contrainte, reindexation simple
      if test "$contrainte" = "t"; then
        echo -n "  reindexation de la contrainte... "
      psql -Xqc "REINDEX INDEX $schema.$index;" $BASE
      if test $? -eq 0; then
        echo "OK"
      else
        echo "PROBLEME!!"
        continue
      fi
#    sinon
      else
#    renommer <ancien nom> en <index_traitement_en_cours>
          echo -n "  renommage... "
      psql -Xqc \
      "ALTER INDEX $schema.$index RENAME TO $NOM_INDEX_TEMPORAIRE;" $BASE
      if test $? -eq 0; then
        echo "OK"
      else
        echo "PROBLEME!!"
        continue
      fi
#    create index <ancien nom>
          echo -n "  création nouvel index..."
      psql -Xqc "$definition;" $BASE
#    si create OK, drop index <index_traitement_en_cours>
      if test $? -eq 0; then
        echo "OK"
            echo -n "  suppression ancien index..."
        psql -Xqc "DROP INDEX $schema.$NOM_INDEX_TEMPORAIRE;" $BASE
        if test $? -eq 0; then
          echo "OK"
        else
          echo "PROBLEME!!"
          continue
        fi
#    sinon, renommer <index_traitement_en_cours> en <ancien nom>
      else
        echo "PROBLEME!!"
            echo -n "  renommage inverse..."
        psql -Xqc \
        "ALTER INDEX $schema.$NOM_INDEX_TEMPORAIRE RENAME TO $index;" $BASE
        if test $? -eq 0; then
          echo "OK"
        else
          echo "PROBLEME!!"
          continue
```

```
            fi
          fi
        fi
      fi
    done
done
```

5.8 SÉCURITÉ

- Ce qu'un utilisateur standard peut faire
  - et ne peut pas faire
- Restreindre les droits
- Chiffrement
- Corruption de données

À l'installation de PostgreSQL, il est essentiel de s'assurer de la sécurité du serveur : sécurité au niveau des accès, au niveau des objets, ainsi qu'au niveau des données.

Ce chapitre va faire le point sur ce qu'un utilisateur peut faire par défaut et sur ce qu'il ne peut pas faire. Nous verrons ensuite comment restreindre les droits. Enfin, nous verrons les possibilités de chiffrement et de non-corruption de PostgreSQL.

5.8.1 Droits par défaut

Un utilisateur standard peut :

- `CONNECT` : accéder à toutes les bases de données
- `CREATE` :
  - créer des objets dans le schéma `public` de **toute** base de données
  - révocation fréquente
  - … mais plus en v15+ !
- `SELECT` : voir les données de ses tables
- `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` : les modifier
- `TEMP` : créer des objets temporaires
- `CREATE`, `USAGE ON LANGUAGE` : créer des fonctions
- `EXECUTE` : exécuter des fonctions définies par d'autres dans le schéma `public`

Par défaut, un utilisateur a beaucoup de droits.

Il peut accéder à toutes les bases de données. Il faut modifier le fichier `pg_hba.conf` pour éviter cela. Il est aussi possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE CONNECT ON DATABASE nom_base FROM nom_utilisateur ;
```

Avant la version 15, l'utilisateur peut créer des objets dans le schéma disponible par défaut (nommé `public`) sur chacune des bases de données où il peut se connecter, même si la base ne lui appartient pas. Il est assez courant de supprimer ce droit :

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

ou de manière globale :

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

Certains DBA suppriment même le schéma de la base. Dans tous les cas, il faut faire attention à ce que le schéma ne réapparaisse pas lors d'une restauration de sauvegarde logique, car sa création est implicite.

L'utilisateur peut créer des objets temporaires sur chacune des bases de données où il peut se connecter. Il est possible de supprimer ce droit avec l'ordre suivant :

```
REVOKE TEMP ON DATABASE nom_base FROM nom_utilisateur ;
```

L'utilisateur peut créer des fonctions, uniquement avec les langages de confiance, uniquement dans les schémas où il a le droit de créer des objets (donc dans le schéma `public` de toute base avant la V15). Il existe deux solutions :

- supprimer le droit d'utiliser un langage :

```
REVOKE USAGE ON LANGUAGE nom_langage FROM nom_utilisateur ;
```

- supprimer le droit de créer des objets dans un schéma :

```
REVOKE CREATE ON SCHEMA nom_schema FROM nom_utilisateur ;
```

L'utilisateur peut exécuter n'importe quelle fonction, y compris définie par quelqu'un d'autre, à condition que la fonction soit dans un schéma où il a accès (dont le schéma `public`, par défaut) (et donc `public`, par défaut) Il est possible d'empêcher cela en supprimant le droit d'exécution d'une fonction :

```
REVOKE EXECUTE ON FUNCTION nom_fonction FROM nom_utilisateur ;
```

5.8.2 Droits par défaut (suite)

Un utilisateur standard peut aussi :

- récupérer des informations sur l'instance
- visualiser les sources des vues et des fonctions
- Modifier des paramètres de la session :

```
SET parametre TO valeur ;
SET LOCAL parametre TO valeur ;
SHOW parametre ;

PGOPTIONS='-c param=valeur' psql
```

- Vue : `pg_settings`
- Dans `psql` : `\dconfig`

Il peut récupérer des informations sur l'instance car il a le droit de lire tous les catalogues systèmes. Par exemple, en lisant `pg_class`, il peut connaitre la liste des tables, vues, séquences, etc. En parcourant `pg_proc`, il dispose de la liste des fonctions. Il n'y a pas de contournement à cela : un utilisateur doit pouvoir accéder aux catalogues systèmes pour travailler normalement.

Il peut visualiser les sources des vues et des fonctions. Il existe des modules propriétaires de chiffrement (ou plutôt d'obfuscation) du code mais rien de natif. Le plus simple est certainement de coder les fonctions sensibles en C.

Un utilisateur peut agir sur de nombreux paramètres au sein de sa session pour modifier les valeurs par défaut du `postgresql.conf` ou ceux imposés à son rôle ou à sa base.

Un cas courant consiste à modifier la liste des schémas par défaut où chercher les tables :

```
SET search_path TO rh,admin,ventes,public ;
```

L'utilisateur peut aussi décider de s'octroyer plus de mémoire de tri :

```
SET work_mem TO '500MB' ;
```

Il est impossible d'interdire cela. Toutefois, cela permet de conserver un paramétrage par défaut prudent, tout en autorisant l'utilisation de plus de ressources quand cela s'avère nécessaire.

Les exemples suivants modifient le fuseau horaire du client, désactivent la parallélisation le temps de la session, et changent le nom de l'applicatif visible dans les outils de supervision :

```
SET timezone TO GMT ;
SET max_parallel_workers_per_gather TO 0 ;
SET application_name TO 'batch_comptabilite' ;
```

Pour une session lancée en ligne de commande, pour les outils qui utilisent la libpq, on peut fixer les paramètres à l'appel grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS="-c max_parallel_workers_per_gather=0 -c work_mem=4MB" psql < requete.sql
```

La valeur en cours est visible avec :

```
SHOW parametre ;
```

ou :

```
SELECT current_setting('parametre') ;
```

ou encore :

```
SELECT * FROM pg_settings WHERE name = 'parametre' ;
```

Sous `psql` à partir de la version 15 du client, la métacommande `\dconfig` affiche les paramètres qui n'ont pas leur valeur par défaut. Ajouter un `+` pour afficher le type et le « contexte », ce dernier permettant de savoir si le paramètre peut être changé dans la session (`user`), s'il faut recharger la configuration (`sighup`), redémarrer (`postmaster`), etc.

```
# \dconfig+
                                  Liste des paramètres de configuration à valeur
↪  personnalisée
         Paramètre          |                     Valeur                     |  Type
↪  |     Contexte      | D…
----------------------------+------------------------------------------------+----
↪  -----+-------------------+----
 application_name           | psql                                           | string
↪  | user              | ø
 client_encoding            | UTF8                                           | string
↪  | user              | ø
 cluster_name               | 17/defo                                        | string
↪  | postmaster        | ø
 config_file                | /etc/postgresql/17/defo/postgresql.conf        |
↪  string  | postmaster        | ø
 data_directory             | /var/lib/postgresql/17/defo                    |
↪  string  | postmaster        | ø
 DateStyle                  | ISO, DMY                                       | string
↪  | user              | ø
 default_text_search_config | pg_catalog.french                              |
↪  string  | user              | ø
 external_pid_file          | /var/run/postgresql/17-defo.pid                |
↪  string  | postmaster        | ø
 hba_file                   | /etc/postgresql/17/defo/pg_hba.conf            |
↪  string  | postmaster        | ø
 ident_file                 | /etc/postgresql/17/defo/pg_ident.conf          |
↪  string  | postmaster        | ø
 lc_messages                | C                                              | string
↪  | superuser         | ø
 lc_monetary                | fr_FR.UTF-8                                    | string
↪  | user              | ø
 lc_numeric                 | fr_FR.UTF-8                                    | string
↪  | user              | ø
 lc_time                    | fr_FR.UTF-8                                    | string
↪  | user              | ø
 log_autovacuum_min_duration | 0                                             |
↪  integer | sighup            | ø
```

```
 log_connections             | on                                                  | bool
↪  | superuser-backend | ø
 log_line_prefix             | %t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h  |
↪  string  | sighup            | ø
 log_replication_commands    | on                                                  | bool
↪  | superuser         | ø
 log_timezone                | Europe/Paris                                      | string
↪  | sighup            | ø
 max_prepared_transactions   | 10                                                  |
↪  integer | postmaster        | ø
 port                        | 17001                                             | integer
↪  | postmaster        | ø
 shared_preload_libraries    | pg_stat_statements                                  |
↪  string  | postmaster        | ø
 ssl                         | on                                                  | bool
↪  | sighup            | ø
 ssl_cert_file               | /etc/ssl/certs/ssl-cert-snakeoil.pem                |
↪  string  | sighup            | ø
 ssl_key_file                | /etc/ssl/private/ssl-cert-snakeoil.key              |
↪  string  | sighup            | ø
 TimeZone                    | Europe/Paris                                      | string
↪  | user              | ø
 work_mem                    | 100MB                                             | integer
↪  | user              | ø
```

L'effet d'un ordre `SET` est limité à la session en cours, et disparaît avec elle à la déconnexion, ou si l'on demande un retour à la valeur par défaut :

```
RESET parametre ;
```

Enfin, on peut n'appliquer des paramètres que le temps d'une transaction, c'est-à-dire jusqu'au prochain `COMMIT` ou `ROLLBACK` :

```
SET LOCAL work_mem TO '100MB' ;
```

De nombreux paramètres sont cependant non modifiables, ou réservés aux superutilisateurs.

5.8.3 Droits par défaut (suite)

- Un utilisateur standard ne peut pas :
  - créer une base
  - créer un rôle
  - accéder au contenu des objets créés par d'autres
  - modifier le contenu d'objets créés par d'autres

Un utilisateur standard ne peut pas créer de bases et de rôles. Il a besoin pour cela d'attributs particuliers (respectivement `CREATEDB` et `CREATEROLE`).

Il ne peut pas accéder au contenu (aux données) d'objets créés par d'autres utilisateurs. Ces derniers doivent lui donner ce droit explicitement : `GRANT USAGE ON SCHEMA secret TO utilisateur ;` pour lire un schéma, ou `GRANT SELECT ON TABLE matable TO utilisateur ;` pour lire une table.

De même, il ne peut pas modifier le contenu et la définition d'objets créés par d'autres utilisateurs. Là-aussi, ce droit doit être donné explicitement : `GRANT INSERT,DELETE,UPDATE,TRUNCATE ON TABLE matable TO util`

Il existe d'autres droits plus rares, dont :

- `GRANT TRIGGER ON TABLE ...` autorise la création de trigger ;
- `GRANT REFERENCES ON TABLE ...` autorise la création d'une clé étrangère pointant vers cette table (ce qui est interdit par défaut car cela interdit au propriétaire de supprimer ou modifier des lignes, entre autres) ;
- `GRANT USAGE ON SEQUENCE...` autorise l'utilisation d'une séquence.

Par facilité, on peut octroyer des droits en masse :

- `GRANT ALL PRIVILEGES ON TABLE matable TO utilisateur ;`
- `GRANT SELECT ON TABLE matable TO public ;`
- `GRANT SELECT ON ALL TABLES IN SCHEMA monschema ;`

5.8.4 Restreindre les droits

- Sur les connexions
  - `pg_hba.conf`
- Sur les objets
  - `GRANT` / `REVOKE`
  - `SECURITY LABEL`
- Sur les fonctions
  - `SECURITY DEFINER`
  - `LEAKPROOF`
- Sur les vues
  - `security_barrier`
  - `WITH CHECK OPTION`

Pour sécuriser plus fortement une instance, il est nécessaire de restreindre les droits des utilisateurs.

Connexions :

Cela commence par la gestion des connexions. Les droits de connexion sont généralement gérés via le fichier de configuration `pg_hba.conf`. Cette configuration a déjà été abordée dans le chapitre *Droits de connexion* de ce module de formation.

GRANT & REVOKE :

Cela passe ensuite par les droits sur les objets. On dispose pour cela des instructions `GRANT` et `REVOKE`, qui ont été expliquées dans le chapitre *Droits sur les objets* de ce module de formation.

SECURITY LABEL :

Il est possible d'aller plus loin avec l'instruction `SECURITY LABEL`. Un label de sécurité est un commentaire supplémentaire pris en compte par un module de sécurité qui disposera de la politique de sécurité. Le seul module de sécurité actuellement disponible est un module contrib pour l'intégration à SELinux, appelé sepgsql[15].

SECURITY DEFINER & LEAKPROOF :

Certains objets disposent de droits particuliers. Par exemple, les fonctions disposent des clauses `SECURITY DEFINER` et `LEAKPROOF`.

`SECURITY DEFINER` indique au système que la fonction doit s'exécuter avec les droits de son propriétaire (et non pas avec ceux de l'utilisateur l'exécutant). Cela permet d'éviter de donner des droits d'accès à certains objets.

`LEAKPROOF` permet de dire à PostgreSQL que cette fonction ne peut pas occasionner de fuites d'informations. Ainsi, le planificateur de PostgreSQL sait qu'il peut optimiser l'exécution des fonctions.

Vues avec security_barrier :

Quant aux vues, elles disposent d'une option appelée `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. Voici un exemple complet :

```
CREATE TABLE elements (id int, contenu text, prive boolean);
```

```
CREATE TABLE
```

```
INSERT INTO elements (id, contenu, prive)
VALUES (1, 'a', false), (2, 'b', false), (3, 'c super prive', true),
       (4, 'd', false), (5, 'e prive aussi', true) ;
```

```
INSERT 0 5
```

```
SELECT * FROM elements ;
```

[15] https://docs.postgresql.fr/current/sepgsql.html

```
 id |    contenu     | prive
----+----------------+-------
  1 | a              | f
  2 | b              | f
  3 | c super prive  | t
  4 | d              | f
  5 | e prive aussi  | t
```

La table `elements` contient cinq lignes, dont deux considérées comme privées. Nous allons donc créer une vue ne permettant de voir que les lignes publiques, sauf pour un utilisateur particulier :

```
CREATE OR REPLACE VIEW elements_public AS
  SELECT * FROM elements
  WHERE CASE WHEN current_user = 'guillaume' THEN TRUE
             ELSE NOT prive
        END ;
```

```
CREATE VIEW
```

Ce nouvel utilisateur ne verra donc pas les lignes privées masquées par la vue :

```
CREATE USER u1;
```

```
CREATE ROLE
```

```
GRANT SELECT ON elements_public TO u1;
```

```
GRANT
```

```
\c - u1
```

```
You are now connected to database "demo" as user "u1".
```

```
SELECT * FROM elements ;
```

```
ERROR:  permission denied for relation elements
```

```
SELECT * FROM elements_public ;
```

```
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

L'utilisateur **u1** n'a pas le droit de lire directement la table `elements` mais a le droit d'y accéder via la vue `elements_public`, uniquement pour les lignes dont le champ `prive` est à `false`.

Avec une simple fonction, **u1** peut contourner le problème. (Il devra toutefois avoir le droit d'écrire dans un schéma de la base, ce qui est par défaut possible dans `public` avant la version 15.)

```
CREATE SCHEMA schemau1;
CREATE OR REPLACE FUNCTION schemau1.abracadabra (integer, text, boolean)
RETURNS bool AS $$
BEGIN
  RAISE NOTICE '% - % - %', $1, $2, $3 ;
  RETURN true;
```

```
END$$
LANGUAGE plpgsql
COST 0.00000000000000000001 ;

CREATE SCHEMA
CREATE FUNCTION

SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive) ;

NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  3 - c super prive - t
NOTICE:  4 - d - f
NOTICE:  5 - e prive aussi - t
 id | contenu | prive
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

Les lignes secrètes ne font toujours pas partie du résultat, mais la fonction est capable d'y accéder, et de les afficher. Que s'est-il passé ? Pour comprendre, il suffit de regarder le plan d'exécution de cette requête :

```
EXPLAIN SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive) ;

                       QUERY PLAN
-------------------------------------------------------------
 Seq Scan on elements  (cost=0.00..28.75 rows=208 width=37)
   Filter: (schemau1.abracadabra(id, contenu, prive)
           AND CASE WHEN (CURRENT_USER = 'guillaume'::name) THEN true ELSE (NOT
 ↪  prive) END)
```

La fonction `abracadrabra` a un coût délibérément si faible que PostgreSQL l'exécute avant le filtre de la vue : la fonction voit toutes les lignes de la table.

L'option `security_barrier` permet d'interdire cette optimisation au planificateur :

```
\c - postgres

You are now connected to database "demo" as user "postgres".

ALTER VIEW elements_public SET (security_barrier);

ALTER VIEW

\c - u1

You are now connected to database "demo" as user "u1".

SELECT * FROM elements_public WHEREschemau1.abracadabra(id, contenu, prive);

NOTICE:  1 - a - f
NOTICE:  2 - b - f
NOTICE:  4 - d - f
 id | contenu | prive
```

```
----+---------+-------
  1 | a       | f
  2 | b       | f
  4 | d       | f
```

La fonction n'affiche plus les données privées. Son plan montre qu'elle filtre par la vue, puis appelle la fonction :

```
EXPLAIN SELECT * FROM elements_public
WHERE schemau1.abracadabra(id, contenu, prive);
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Subquery Scan on elements_public  (cost=0.00..35.00 rows=208 width=37)
   Filter: schemau1.abracadabra(elements_public.id, elements_public.contenu,
 ↪  elements_public.prive)
   ->  Seq Scan on elements  (cost=0.00..28.75 rows=625 width=37)
         Filter: CASE WHEN (CURRENT_USER = 'guillaume'::name) THEN true ELSE (NOT
 ↪  prive) END
```

Noter que `security_barrier` peut avoir un effet négatif sur les performances puisqu'il interdit de « pousser » des critères de recherche dans la vue. Voir aussi cet article, par Robert Haas[16].

Vue WITH CHECK OPTION :

Écrire à travers une vue permet de modifier les lignes même si la nouvelle valeur les lui rend inaccessibles :

```
UPDATE elements_public
SET prive = true
WHERE id = 1 ;
```

```
UPDATE 1
```

```
SELECT * FROM elements_public ;
```

```
 id | contenu | prive
----+---------+-------
  2 | b       | f
  4 | d       | f
```

Pour bloquer ce comportement, la vue peut porter l'option `WITH CHECK OPTION` :

```
CREATE OR REPLACE VIEW elements_public
WITH (security_barrier)
AS
  SELECT * FROM elements
  WHERE CASE WHEN current_user = 'guillaume'
  THEN true ELSE NOT prive END
WITH CHECK OPTION ;
```

```
\c - u1
```

```
You are now connected to database "demo" as user "u1".
```

[16] https://rhaas.blogspot.com/2012/03/security-barrier-views.html

```
UPDATE elements_public
SET prive = true
WHERE id = 2 ;
```

```
ERROR:  new row violates check option for view "elements_public"
DETAIL:  Failing row contains (prive) = (t).
```

5.8.5 Arrêter une requête ou une session

- Annuler une requête
  - `pg_cancel_backend (pid int)`
- Fermer une connexion
  - `pg_terminate_backend(pid int, timeout bigint)`
  - `kill -SIGTERM pid`, `kill -15 pid` (éviter)
- Jamais `kill -9` !!

Les fonctions `pg_cancel_backend` et `pg_terminate_backend` sont le plus souvent utilisées. Le paramètre est le numéro du processus auprès de l'OS. À partir de la version 14, `pg_terminate_backend` comprend un deuxième argument, dont la valeur par défaut est 0. Si cet argument n'est pas indiqué ou vaut 0, la fonction renvoie le booléen `true` si elle a réussi à envoyer le signal. Ce résultat n'indique donc pas la bonne terminaison du processus serveur visé. À une valeur supérieure à 0, la fonction attend que le processus visé soit arrêté. S'il ne s'est pas arrêté dans le temps indiqué par cette valeur (en millisecondes), la valeur `false` est renvoyée avec un message de niveau `WARNING`.

La première permet d'annuler une requête en cours d'exécution. Elle requiert un argument, à savoir le numéro du PID du processus `postgres` exécutant cette requête. Généralement, l'annulation est immédiate. Voici un exemple de son utilisation.

L'utilisateur, connecté au processus de PID 10901 comme l'indique la fonction `pg_backend_pid`, exécute une très grosse insertion :

```
b1=# SELECT pg_backend_pid();
```

```
 pg_backend_pid
----------------
          10901
```

```
b1=# INSERT INTO t4 SELECT i, 'Ligne '||i
FROM generate_series(2000001, 3000000) AS i;
```

Supposons qu'on veuille annuler l'exécution de cette requête. Voici comment faire à partir d'une autre connexion :

```
b1=# SELECT pg_cancel_backend(10901);

 pg_cancel_backend
-------------------
 t
```

L'utilisateur qui a lancé la requête d'insertion verra ce message apparaître :

```
ERROR:  canceling statement due to user request
```

Si la requête du `INSERT` faisait partie d'une transaction, la transaction elle-même devra se conclure par un `ROLLBACK` à cause de l'erreur. À noter cependant qu'il n'est pas possible d'annuler une transaction qui n'exécute rien à ce moment. En conséquence, `pg_cancel_backend` ne suffit pas pour parer à une session en statut `idle in transaction`.

Il est possible d'aller plus loin en supprimant la connexion d'un utilisateur. Cela se fait avec la fonction `pg_terminate_backend` qui se manie de la même manière :

```
b1=# SELECT pid, datname, usename, application_name, state
     FROM pg_stat_activity WHERE backend_type = 'client backend';

 procpid | datname |  usename   | application_name | state
---------+---------+-----------+------------------+--------
   13267 | b1      | u1         | psql             | idle
   10901 | b1      | guillaume | psql             | active

b1=# SELECT pg_terminate_backend(13267);

 pg_terminate_backend
----------------------
 t

b1=# SELECT pid, datname, usename, application_name, state
     FROM pg_stat_activity WHERE backend_type = 'client backend';

 procpid | datname |  usename   | application_name | state
---------+---------+-----------+------------------+--------
   10901 | b1      | guillaume | psql             | active
```

L'utilisateur de la session supprimée verra un message d'erreur au prochain ordre qu'il enverra. `psql` se reconnecte automatiquement mais cela n'est pas forcément le cas d'autres outils client.

```
b1=# select 1 ;

FATAL:  terminating connection due to administrator command
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.
Temps : 7,309 ms
```

Depuis la ligne de commande du serveur, un `kill <pid>` (c'est-à-dire `kill -SIGTERM` ou `kill -15`) a le même effet qu'un `SELECT pg_terminate_backend (<pid>)`. Cette méthode n'est pas recommandée car il n'y a pas de vérification que vous tuez bien un processus postgres.

N'utilisez jamais `kill -9 <pid>` (ou `kill -SIGKILL`), ou (sous Windows) `taskkill /f /pid <pid>` pour tuer une connexion : l'arrêt est alors brutal, et le processus principal n'a aucun moyen de savoir pourquoi. Pour éviter une corruption de la mémoire partagée, il va arrêter et redémarrer immédiatement tous les processus, déconnectant tous les utilisateurs au passage !

L'utilisation de `pg_terminate_backend` et `pg_cancel_backend` n'est disponible que pour les utilisateurs appartenant au même rôle que l'utilisateur à déconnecter, les utilisateurs membres du rôle `pg_signal_backend` (à partir de la 9.6) et bien sûr les superutilisateurs.

5.8.6 Chiffrements

- Connexions
  - SSL
  - avec ou sans certificats serveur et/ou client
- Données disques
  - pas en natif
  - par colonne : pgcrypto

Par défaut, les sessions ne sont pas chiffrées. Les requêtes et les données passent donc en clair sur le réseau. Il est possible de les chiffrer avec SSL, ce qui aura une conséquence négative sur les performances. Il est aussi possible d'utiliser les certificats (au niveau serveur et/ou client) pour augmenter encore la sécurité des connexions.

PostgreSQL ne chiffre pas les données sur disque. Si l'instance complète doit être chiffrée, il est conseillé d'utiliser un système de fichiers qui propose cette fonctionnalité. Attention au fait que cela ne vous protège que contre la récupération des données sur un disque non monté. Quand le disque est monté, les données sont lisibles suivant les règles d'accès d'Unix.

Néanmoins, il existe un module contrib appelé pgcrypto, permettant d'accéder à des fonctions de chiffrement et de hachage. Cela permet de protéger les informations provenant de colonnes spécifiques. Le chiffrement se fait du côté serveur, ce qui sous-entend que l'information est envoyée en clair sur le réseau. Le chiffrement SSL est donc obligatoire dans ce cas.

5.8.7 En cas de crash

- Ça arrive
- Redémarrage sans souci, en général
- Quelques statistiques d'activité sont perdues
  - `ANALYZE` (voire `VACUUM`)
- Et analyser la cause

Même si PostgreSQL est conçu pour être le plus robuste possible, un arrêt brutal est toujours possible, pour des causes externes (perte de courant, plantage du système d'exploitation, corruption de fichier sur le disque…), liées à l'utilisation de PostgreSQL (requêtes menant à une saturation mémoire sur un système mal paramétré…) ou exceptionnellement internes (extension buggée, bug de PostgreSQL).

Dans l'écrasante majorité des cas, le système de journalisation de PostgreSQL vous permettra de redémarrer sans perte de données (sinon ce qui n'a pas été committé). Le rejeu des journaux est souvent si rapide qu'un redémarrage peut passer inaperçu.

Malheureusement, les statistiques d'activités sont perdues lors d'un arrêt brutal (ou d'un arrêt en mode *immediate*, ou d'une restauration physique). Cela peut provoquer par la suite un retard dans le déclenchement de l'autovacuum et la mise à jour des statistiques. Pour éviter cela, le DBA doit penser à relancer un ordre `ANALYZE`, ainsi que, dans l'idéal, un ordre `VACUUM`.

Surtout, il faut comprendre quelle est la cause pour que le problème ne se reproduise pas.

5.8.8 Corruption de données

- `initdb --data-checksums`
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Fortement conseillé !

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Sans mécanisme de sécurité, il est donc possible de modifier un fichier sans que PostgreSQL s'en rende compte, ce qui aboutit à une corruption silencieuse.

L'apparition des sommes de contrôles (*checksums*) permet de se prémunir contre des corruptions silencieuses de données. Leur mise en place est fortement recommandée sur une nouvelle instance.

À titre d'exemple, créons une instance sans utiliser les *checksums*, et une autre qui les utilisera :

```
$ initdb --pgdata /tmp/sans_checksums/
$ initdb --pgdata /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacune des deux instances :

```
postgres=# CREATE TABLE test (name text);
```

```
CREATE TABLE
```

```
postgres=# INSERT INTO test (name) VALUES ('toto');
```

```
INSERT 0 1
```

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans *checksums* est montré en exemple).

```
postgres=# SELECT pg_relation_filepath('test');
```

```
 pg_relation_filepath
----------------------
 base/14415/16384
```

Instance arrêtée (pour ne pas être gêné par le cache), on va s'attacher à corrompre ce fichier, en remplaçant la valeur « toto » par « goto » avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/base/14415/16384
```

Enfin, on peut ensuite exécuter des requêtes sur ces deux instances.

Sans *checksums* :

```
postgres=# TABLE test;
```

```
 name
------
 goto
```

Avec *checksums* :

```
postgres=# TABLE test;
```

```
WARNING:  page verification failed, calculated checksum 29393
          but expected 24228
ERROR:  invalid page in block 0 of relation base/14415/16384
```

L'outil `pg_checksums` permet de vérifier une instance complète :

```
$ pg_checksums -D /tmp/avec_checksums
```

```
pg_checksums: error: checksum verification failed
  in file "/tmp/avec_checksums/base/14415/16384",
  block 0: calculated checksum 72D1 but block contains 5EA4
Checksum operation completed
Files scanned:  919
Blocks scanned: 3089
Bad checksums:  1
Data checksum version: 1
```

En pratique, depuis PostgreSQL 9.5 et si votre processeur supporte les instructions SSE 4.2 (ce qui est le cas de tous les processeurs Intel/AMD récents, voir dans `/proc/cpuinfo`), il n'y aura pas d'impact notable en performances. Par contre vous générerez un peu plus de journaux.

L'outil `pg_checksums` permet aussi d'activer (ou désactiver) la gestion des sommes de contrôle, instance arrêtée. (Ceci n'était pas possible avant la version 12. Il ne fallait pas oublier de les activer dès la création de l'instance.)

5.9 CONCLUSION

- PostgreSQL demande peu de travail au quotidien
- À l'installation :
  - veiller aux accès et aux droits
  - mettre la maintenance en place
- Pour le reste, surveiller :
  - les scripts automatisés
  - le contenu des journaux applicatifs
- Supervisez le serveur !

Pour aller plus loin :

- La documentation officielle est toujours une bonne lecture :
  - Planifier les tâches de maintenance[17]
  - Catalogues système[18]
- Total security in a PostgreSQL database (copie)[19]
- Managing rights in PostgreSQL, from the basics to SE PostgreSQL[20], Nicolas Thauvin, 2011

5.9.1 Questions

N'hésitez pas, c'est le moment !

[17] https://docs.postgresql.fr/current/maintenance.html
[18] https://docs.postgresql.fr/current/catalogs.html
[19] http://apc.csf.ph/jqitc/node/77
[20] https://wiki.postgresql.org/images/d/d1/Managing_rights_in_postgresql.pdf

5.10 QUIZ

https://dali.bo/f_quiz

5.11 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/f_solutions.

5.11.1 Traces maximales

But : suivre toutes les requêtes dans les traces

À titre pédagogique et pour alimenter un rapport pgBadger plus tard, toutes les requêtes vont être tracées.

Dans `postgresql.conf`, positionner ceci :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages = 'C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

Puis passer à `on` les paramètres suivants s'ils ne le sont pas déjà :

```
log_checkpoints
log_connections
log_disconnections
log_lock_waits
```

Recharger la configuration.

Laisser une fenêtre ouverte pour voir défiler le contenu des traces.

5.11.2 Méthode d'authentification

But : Gérer les rôles et les permissions

Activer la méthode d'authentification `scram-sha-256` dans `postgresql.conf` si elle n'est pas déjà en place.

Consulter les droits définis dans `pg_hba.conf` au travers de la vue système `pg_hba_file_rules`.

Dans `pg_hba.conf`, supprimer les accès avec la méthode `trust` pouvant rester après les précédents exercices. Vérifier dans la vue avant de recharger la configuration.

5.11.3 Création des bases

But : Créer des bases appartenant à un utilisateur non privilégié

Créer un utilisateur nommé **testperf** avec attribut `LOGIN`.

Créer une base **pgbench** lui appartenant.

Vérifier que la connexion ne marche pas encore depuis votre compte habituel, y compris avec `-h localhost`.

Créer un rôle **patron** avec attribut `LOGIN`, et une base **entreprise** lui appartenant.

5.11.4 Mots de passe

But : Mise en place de l'authentification par mot de passe

Créer des mots de passe pour les rôles **patron** et **testperf**.

Consulter la table `pg_authid` pour voir la version chiffrée.

Pour ouvrir les accès :

- ajuster `pg_hba.conf` pour permettre l'accès à la base **pgbench** uniquement à l'utilisateur **testperf**, permettre l'accès à la base **entreprise** à tout utilisateur, en local avec son mot de passe, en authentification `scram-sha-256` ;
- vérifier avec la vue `pg_hba_file_rules` ;
- recharger la configuration ;
- tester la connexion.

Créer un fichier `.pgpass` dans votre répertoire utilisateur (`/home/dalibo`) ou dans celui de l'utilisateur *postgres* (`/var/lib/pgsql`) pour qu'il puisse se connecter aux bases **entreprise** et **pgbench** sans entrer le mot de passe.

Le superutilisateur PostgreSQL **postgres** peut-il se connecter aux bases **entreprise** et **pgbench** ? Si non, comment lui permettre ?

Remplir la base **pgbench** et générer un peu d'activité :

```
/usr/pgsql-17/bin/pgbench -i -s1 --foreign-keys pgbench -U testperf
/usr/pgsql-17/bin/pgbench -P1 -T3 pgbench -U testperf
```

Aucun mot de passe ne doit être demandé, avec ou sans `-h localhost`.

Compléter `pg_ident.conf` pour pouvoir se connecter au rôle **postgres** depuis le compte système habituel (**dalibo**), en `local`.

Tester la connexion depuis **postgres** au rôle **postgres**, de **dalibo** à **postgres**.

5.11.5 Rôles et permissions

But : Gérer les rôles et les permissions sur les tables

Sous les utilisateurs **dalibo** comme **postgres**, créer un nouveau fichier `~/.psqlrc` contenant `\set PROMPT1 '%n@%/%R%# '` pour que l'invite indique quels sont les utilisateur et base en cours.

Ajouter à la base de données **entreprise** la table `facture (id int, objet text, creationts timestamp)`. Elle appartiendra à **patron**, administrateur de la base.

Créer un rôle **secretariat** sans droit de connexion, mais qui peut visualiser, ajouter, mettre à jour et supprimer des éléments de la table `facture`.

Créer un rôle **boulier** qui peut se connecter et appartient au rôle **secretariat**, avec un mot de passe (à ajouter au `.pgpass`).

> Vérifier la création des deux rôles.

> En tant que **boulier**, créer une table `brouillon` identique à `facture`. Dans quel schéma cela se passe-t-il ? Avec PostgreSQL 15 ou supérieur, il y a une étape supplémentaire.

> Vérifier les tables présentes et les droits `\dp`. Comment les lire ?

> À l'aide du rôle **boulier** : insérer 2 factures ayant pour objet « Vin de Bordeaux » et « Vin de Bourgogne » avec la date et l'heure courante.

> Afficher le contenu de la table `facture`.

> Mettre à jour la deuxième facture avec la date et l'heure courante.

> Supprimer la première facture.

> Retirer les droits `DELETE` sur la table `facture` au rôle **secretariat**.

> Vérifier qu'il n'est plus possible de supprimer la deuxième facture avec le rôle **boulier**.

> En tant que **patron**, créer une table **produit** contenant une colonne texte nommée `appellation` et la remplir avec des noms de boissons.

> Afficher les droits sur cette table avec `\dt` et `\dp`. Vérifier que le rôle **boulier** appartenant au rôle **secretariat** ne peut pas sélectionner les produits contenus dans la table `produit`.

> Retirer tous les droits pour le groupe **secretariat** sur la table `produit`.

> Que deviennent les droits affichés ? **boulier** peut-il lire la table ?

> Autoriser l'utilisateur **boulier** à accéder à la table `produit` en lecture.

Vérifier que **boulier** peut désormais accéder à la table `produit`.

Créer un rôle **tina** appartenant au rôle **secretariat**, avec l'attribut `LOGIN`, mais n'héritant pas des droits à la connexion. Vérifier les droits avec `\du`. Lui donner un mot de passe.

Vérifier que **tina** ne peut pas accéder à la table `facture`.

En tant que **tina**, activer le rôle **secretariat** (`SET ROLE`).

Vérifier que **tina** possède maintenant les droits du rôle **secretariat**. Sélectionner les données de la table `facture`.

5.11.6 Autorisation d'accès distant

But : Mettre en place les accès dans `pg_hba.conf`.

Autoriser tous les membres du réseau local à se connecter avec un mot de passe (autorisation en IP sans SSL) avec les utilisateurs **boulier** ou **tina**. Tester avec l'IP du serveur avant de demander aux voisins de tester.

5.11.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE

But : Effacer des données, distinguer `VACUUM` et `VACUUM FULL`.

Désactiver le démon autovacuum de l'instance.

Se connecter à la base **pgbench** en tant que **testperf**.

Grâce aux fonctions `pg_relation_size` et `pg_size_pretty`, afficher la taille de la table `pgbench_accounts`.

Copier le contenu de la table dans une nouvelle table (`pba_copie`).

Supprimer le contenu de la table `pba_copie`, à l'exception de la dernière ligne (`aid=100000`), avec un ordre `DELETE`. Quel est alors l'espace disque utilisé par cette table ?

Insérer le contenu de la table `pgbench_accounts` dans la table `pba_copie`. Quel est alors l'espace disque utilisé par la table ?

Effectuer un VACUUM simple sur `pba_copie`. Vérifier la taille de la base.

Vider à nouveau la table `pba_copie` des lignes d'`aid` inférieur à 100 000. Insérer à nouveau le contenu de la table `pgbench_accounts`. L'espace mis à disposition a-t-il été utilisé ?

Voir la taille de la base. Supprimer la table `pba_copie`. Voir l'impact sur la taille de la base.

Tout d'abord, repérer les tailles des différentes tables et le nombre de lignes de chacune.

Pour amplifier le phénomène à observer, on peut créer une session de très longue durée, laissée ouverte sans `COMMIT` ni `ROLLBACK`. Il faut qu'elle ait consulté une des tables pour que l'effet soit visible :

```
testperf@pgbench=> BEGIN ;

BEGIN
Temps : 0,608 ms

testperf@pgbench=> SELECT count(*) FROM pgbench_accounts ;

 count
--------
 100000
(1 ligne)

Temps : 26,059 ms
testperf@pgbench=> SELECT pg_sleep (10000) ;
```

Depuis un autre terminal, générer de l'activité sur la table, ici avec 10 000 transactions sur 20 clients :

```
PGOPTIONS='-c synchronous_commit=off' \
/usr/pgsql-17/bin/pgbench -U testperf -d pgbench \
--client=20 --jobs=2 -t 10000 --no-vacuum
```

(NB : La variable d'environnement `PGOPTIONS` restreint l'utilisation des journaux de transaction pour accélérer les écritures (données NON critiques ici). Le `--no-vacuum` est destiné à éviter que l'outil demande lui-même un `VACUUM`. Le test dure quelques minutes. Le relancer au besoin.)

(Optionnel) C'est l'occasion d'installer l'outil `pg_activity` depuis les dépôts du PGDG (il peut y avoir besoin du dépôt EPEL) et de le lancer en tant que **postgres** pour voir ce qui se passe dans la base.

Comparer les nouvelles tailles des tables (taille sur le disque et nombre de lignes). La table `pg_stat_user_tables` contient l'activité sur chaque table. Comment s'expliquent les évolutions ?

Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_tellers`.

Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_accounts`.

Effectuer un `VACUUM FULL VERBOSE`. Quel est l'impact sur la taille de la base ?

Créer `copie1` et `copie2` comme des copies de `pgbench_accounts`, données incluses.

Effacer le contenu de `copie1` avec `DELETE`.

Effacer le contenu de `copie2` avec `TRUNCATE`.

Quelles sont les tailles de ces deux tables après ces opérations ?

Réactiver l'autovacuum de l'instance.

Attendre quelques secondes et voir si `copie1` change de taille.

5.11.8 Statistiques

But : Savoir trouver les statistiques des données et les mettre à jour

Créer une table `copie3`, copie de `pgbench_accounts`.

Dans la vue système `pg_stats`, afficher les statistiques collectées pour la table `copie3`.

Lancer la collecte des statistiques pour cette table uniquement.

Afficher de nouveau les statistiques.

5.11.9 Réindexation

But : Réindexer

Recréer les index de la table `pgbench_accounts`.

Comment recréer tous les index de la base **pgbench** ?

Comment recréer uniquement les index des tables systèmes ?

Quelle est la différence entre la commande `REINDEX` et la séquence `DROP INDEX` + `CREATE INDEX` ?

5.11.10 Traces

But : Gérer les fichiers de traces

Quelle est la méthode de gestion des traces utilisée par défaut ?

Paramétrer le programme interne de rotation des journaux :

- modifier le fichier `postgresql.conf` pour utiliser le *logging collector* ;
- les traces doivent désormais être sauvegardés dans le répertoire `/var/lib/pgsql/traces` ;
- la rotation des journaux doit être automatisée pour générer un nouveau fichier de logs toutes les 30 minutes, quelle que soit la quantité de logs archivés ; le nom du fichier de-

vra donc comporter les minutes.
- Tester en forçant des rotations avec la fonction `pg_rotate_logfile`.
- Augmenter la trace (niveau `info`).

Comment éviter l'accumulation des fichiers ?

5.12 TRAVAUX PRATIQUES (SOLUTIONS)

Tout ce qui suit suppose :

- une installation sous Rocky Linux avec les paquets RPM du PDGD (yum.postgresql.org) ;
- une instance installée avec les options par défaut ;
- un administrateur dont le compte habituel sur la machine, non privilégié, est nommé **dalibo**.

5.12.1 Traces maximales

But : suivre toutes les requêtes dans les traces

À titre pédagogique et pour alimenter un rapport pgBadger plus tard, toutes les requêtes vont être tracées.

Dans `postgresql.conf`, positionner ceci :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages = 'C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

Éviter de faire cela en production, surtout `log_min_duration_statement = 0` ! Sur une base très active, les traces peuvent rapidement monter à plusieurs dizaines de gigaoctets !
Dans le présent TP, il faut surveiller l'espace disque pour cette raison.

Puis passer à `on` les paramètres suivants s'ils ne le sont pas déjà :

```
log_checkpoints
log_connections
log_disconnections
log_lock_waits
```

Recharger la configuration.

Laisser une fenêtre ouverte pour voir défiler le contenu des traces.

Dans `postgresql.conf` :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages='C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
```

```
SELECT pg_reload_conf() ;
```

```
 pg_reload_conf
----------------
 t
```

```
SHOW log_min_duration_statement ;
```

```
 log_min_duration_statement
----------------------------
 0
```

Laisser une fenêtre ouverte avec le fichier, par exemple avec :

```
tail -f /var/lib/pgsql/15/data/log/postgresql-Wed.log
```

La moindre requête doit y apparaître, y compris quand l'utilisateur effectue un simple `\d`.

5.12.2 Méthode d'authentification

But : Gérer les rôles et les permissions

> Activer la méthode d'authentification `scram-sha-256` dans `postgresql.conf` si elle n'est pas déjà en place.

Cette méthode existe depuis PostgreSQL 10 mais n'est le défaut que depuis PostgreSQL 14.

Dans `postgresql.conf` :

```
password_encryption = scram-sha-256
```

Ne pas oublier de recharger la configuration. Depuis psql en tant que **postgres** :

```
SELECT pg_reload_conf() ;
```

```
 pg_reload_conf
----------------
 t
```

Alternative depuis le shell :

```
 # systemctl reload postgresql-15
```

```
SHOW password_encryption ;
```

```
 password_encryption
---------------------
 scram-sha-256
```

> Consulter les droits définis dans `pg_hba.conf` au travers de la vue système `pg_hba_file_rules`.

La vue permet de voir le contenu de `pg_hba.conf` *avant* de le recharger.

```
SELECT * FROM pg_hba_file_rules ;
```

```
ln|type | database|user_name| address |       netmask        |  auth_method  | …
--+-----+---------+---------+---------+----------------------+---------------+--
80|local|{pgbench}|{all}    |         |                      | trust         |
81|local|{all}    |{all}    |         |                      | peer          |
84|host |{all}    |{all}    |127.0.0.1|255.255.255.255       | scram-sha-256 |
86|host |{all}    |{all}    |::1      |ffff:ffff:ffff:ffff:… | scram-sha-256 |
89|local|{replica…|{all}    |         |                      | peer          |
90|host |{replica…|{all}    |127.0.0.1|255.255.255.255       | scram-sha-256 |
91|host |{replica…|{all}    |::1      |ffff:ffff:ffff:ffff:… | scram-sha-256 |
```

> Dans `pg_hba.conf`, supprimer les accès avec la méthode `trust` pouvant rester après les précédents exercices. Vérifier dans la vue avant de recharger la configuration.

Dans le fichier `pg_hba.conf`, supprimer les lignes avec la méthode `trust`. La vue se met à jour immédiatement. En cas d'erreur de syntaxe dans le fichier, elle doit aussi indiquer une erreur.

Puis on recharge :

```
SELECT pg_reload_conf() ;
```

5.12.3 Création des bases

But : Créer des bases appartenant à un utilisateur non privilégié

> Créer un utilisateur nommé **testperf** avec attribut `LOGIN`.

Au choix, on peut utiliser les commandes shell ou les commandes SQL qu'elles affichent :

```
$ sudo -iu postgres
```

```
$ createuser --login --echo testperf
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE testperf NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

> Créer une base **pgbench** lui appartenant.

```
$ createdb --echo pgbench --owner testperf
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE pgbench OWNER testperf;
```

> Vérifier que la connexion ne marche pas encore depuis votre compte habituel, y compris avec `-h localhost`.

La connexion ne peut se faire car dans `pg_hba.conf` ne figure, en `local`, que l'accès `peer` pour un utilisateur nommé **postgres** (nom système comme nom du rôle.) Elle ne fonctionnera pas pour des utilisateurs nommés différemment.

Corriger cela impliquerait de modifier `pg_ident.conf`.

```
$ psql -U testperf pgbench
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432"
↪  failed: FATAL:  Peer authentication failed for user "testperf"
```

Si l'on ajoute `-h localhost`, l'accès est contrôlé via une ligne `host`, et exige un mot de passe qui n'existe pas encore :

```
$ psql -U testperf pgbench -h localhost
Password for user testperf:
```

Nous créerons ces mots de passe plus bas.

> Créer un rôle **patron** avec attribut `LOGIN`, et une base **entreprise** lui appartenant.

```
$ createuser --login patron
$ createdb --owner patron entreprise
```

Ce qui est équivalent à :

```
CREATE ROLE patron LOGIN;
CREATE DATABASE entreprise OWNER patron;
```

Noter que, là encore, c'est le superutilisateur **postgres** qui crée la base et affecte les droits à **patron**. Celui-ci n'a pas le droit de créer des bases.

5.12.4 Mots de passe

But : Mise en place de l'authentification par mot de passe

> Créer des mots de passe pour les rôles **patron** et **testperf**.

Déclarer le mot de passe se fait facilement depuis `psql` (en tant que superutilisateur **postgres**) :

```
postgres=# \password testperf

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :

postgres=# \password patron

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
```

Un outil courant pour générer des mots de passe aléatoires longs est `pwgen` :

```
$ pwgen 20 1
jahT0eeRov2aiQuae1iM
```

Si `pwgen` n'est pas présent sur le système, un bon mot de passe peut être généré ainsi :

```
$ echo "faitespasserunchatsurleclavier" | md5sum
b0cdc36ff6c986b3930bfc269f37f3f2
```

Pour l'exercice, il est possible de donner le même mot de passe à tous les utilisateurs (ce que personne ne fait en production, bien sûr).

> Consulter la table `pg_authid` pour voir la version chiffrée.

Noter que, même identiques, les mots de passe n'ont pas la même signature.

```
SELECT * FROM pg_authid WHERE rolname IN ('testperf','patron') \gx

-[ RECORD 1 ]--+-------------------------------------------------------------
oid            | 25097
rolname        | patron
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
rolpassword    | SCRAM-SHA-256$4096:a0IE9MKlZRTYd9FlXxDX0g==$wT0rQtaolI2gpP...
rolvaliduntil  |
-[ RECORD 2 ]--+-------------------------------------------------------------
oid            | 25096
rolname        | testperf
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
```

```
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
rolpassword    | SCRAM-SHA-256$4096:XNd9Ndrb6ljGAVyTek3sig==$ofeTaBumh2p6WA...
rolvaliduntil  |
```

> Pour ouvrir les accès :
>
> - ajuster `pg_hba.conf` pour permettre l'accès à la base **pgbench** uniquement à l'utilisateur **testperf**, permettre l'accès à la base **entreprise** à tout utilisateur, en local avec son mot de passe, en authentification `scram-sha-256` ;
> - vérifier avec la vue `pg_hba_file_rules` ;
> - recharger la configuration ;
> - tester la connexion.

Ajouter ceci dans `pg_hba.conf`, **en tête** (nous verrons que c'est une erreur) :

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   pgbench         testperf                                scram-sha-256
local   entreprise      all                                     scram-sha-256
```

Recharger la configuration et tenter une connexion depuis un compte utilisateur normal : cela doit fonctionner en entrant le mot de passe.

```
$ sudo -iu postgres psql -c 'SELECT pg_reload_conf()';
$ psql -U testperf -d pgbench
Mot de passe pour l'utilisateur testperf :
psql (15.1)
Type "help" for help.

pgbench=>
```

patron doit aussi pouvoir se connecter :

```
$ psql -U patron -d entreprise
```

> Créer un fichier `.pgpass` dans votre répertoire utilisateur (`/home/dalibo`) ou dans celui de l'utilisateur *postgres* (`/var/lib/pgsql`) pour qu'il puisse se connecter aux bases **entreprise** et **pgbench** sans entrer le mot de passe.

Le fichier doit contenir le mot de passe en clair sous cette forme :

```
localhost:5432:pgbench:testperf:b0cdc36ff6c986b3930bfc269f37f3f2
localhost:5432:entreprise:patron:b0cdc36ff6c986b3930bfc269f37f3f2
```

NB : la mention `localhost` dans ce fichier couvre aussi bien les accès via `::1` ou `127.0.0.1` (lignes `host` de `pg_hba.conf`) que les accès via la socket unix (lignes `local`).

Si le mot de passe est le même pour tous les utilisateurs créés par la suite, le nom d'utilisateur **patron** peut même être remplacé par `*`.

Si la connexion échoue, vérifier que le fichier est en mode 600 :

```
WARNING: password file "/home/dalibo/.pgpass" has group or world access; permissions
 ↪  should be u=rw (0600) or less
```

```
$ chmod u=rw,go= ~/.pgpass
```

La connexion doit à présent se faire sans mot de passe.

> Le superutilisateur PostgreSQL **postgres** peut-il se connecter aux bases **entreprise** et **pgbench** ? Si non, comment lui permettre ?

Même depuis l'utilisatur système **postgres**, la connexion aux deux bases que nous venons de créer échoue :

```
$ sudo -iu postgres  psql -d entreprise -U postgres
Mot de passe pour l'utilisateur postgres :
psql: fe_sendauth: no password supplied
```

La cause est dans les traces :

```
FATAL:  password authentication failed for user "postgres"
DETAIL:  User "postgres" has no password assigned.
        Connection matched pg_hba.conf line 81:
        "local   entreprise      all             scram-sha-256"
```

L'échec est donc normal : la ligne de `pg_hba.conf` qui permet un accès inconditionnel à toute base depuis le compte système **postgres** est à présent la troisième. Pour corriger cela sans créer de mot de passe, la remplacer par cette **toute première ligne** de `pg_hba.conf` :

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             postgres                                peer
```

Et recharger la configuration.

Depuis l'utilisateur système `postgres`, il ne doit plus y avoir de mot de passe demandé à la connexion :

```
$ psql -d entreprise
```

> Remplir la base **pgbench** et générer un peu d'activité :
>
> ```
> /usr/pgsql-17/bin/pgbench -i -s1 --foreign-keys pgbench -U testperf
> /usr/pgsql-17/bin/pgbench -P1 -T3 pgbench -U testperf
> ```
>
> Aucun mot de passe ne doit être demandé, avec ou sans `-h localhost`.

```
$ sudo -iu postgres

$ createuser --login --echo testperf
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE testperf NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;

$ createdb --echo pgbench
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE pgbench OWNER testperf;
```

Initialisation de la base (23 Mo) :

```
$ sudo -iu postgres
$ /usr/pgsql-17/bin/pgbench -i --foreign-keys -d pgbench -U testperf -h localhost
```

Génération de données :

```
/usr/pgsql-17/bin/pgbench -P1 -T3 pgbench -U testperf -h localhost
```

Le `pg_hba.conf` doit ressembler à celui-ci :

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

local   all             postgres                                peer
local   pgbench         testperf                                scram-sha-256
local   entreprise      all                                     scram-sha-256

host    all             all             127.0.0.1/32            scram-sha-256
host    all             all             ::1/128                 scram-sha-256

local   replication     all                                     peer
host    replication     all             127.0.0.1/32            scram-sha-256
host    replication     all             ::1/128                 scram-sha-256
```

L'accès sans `-h localhost` fonctionne grâce aux lignes `local`. L'accès via `-h localhost` fonctionne grâce à l'une des lignes `host`. On avait vu qu'il ne manquait que le mot de passe. Évidemment cela ne marche pas depuis une adresse extérieure.

Les 3 dernières lignes sont réservées à des connexions de réplication.

> Compléter `pg_ident.conf` pour pouvoir se connecter au rôle **postgres** depuis le compte système habituel (**dalibo**), en `local`.

Remplacer la première ligne de `pg_hba.conf` par :

```
local   all             postgres                                peer map=admins
```

Dans `pg_ident.conf`, ajouter :

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
admins           postgres                postgres
admins           dalibo                  postgres
```

On recharge :

```
$ sudo systemctl reload postgresql-15
```

Toute erreur peut empêcher la connexion au rôle **postgres** !

> Tester la connexion depuis **postgres** au rôle **postgres**, de **dalibo** à **postgres**.

Ces connexions doivent fonctionner sans mot de passe :

```
postgres$ psql
postgres$ psql -d entreprise
```

```
dalibo$ psql -U postgres
dalibo$ psql -U postgres -d entreprise
```

5.12.5 Rôles et permissions

But : Gérer les rôles et les permissions sur les tables

> Sous les utilisateurs **dalibo** comme **postgres**, créer un nouveau fichier `~/.psqlrc` contenant `\set PROMPT1 '%n@%/%R%# '` pour que l'invite indique quels sont les utilisateur et base en cours.

```
$ psql -U testperf -d pgbench
psql (15.1)
Type "help" for help.

testperf@pgbench=>
```

Noter que l'affichage de l'invite est légèrement différente selon que le type d'utilisateur : superutilisateur ou un utilisateur « normal ».

> Ajouter à la base de données **entreprise** la table `facture (id int, objet text, creationts timestamp)`. Elle appartiendra à **patron**, administrateur de la base.

Se connecter avec l'utilisateur **patron** (administrateur de la base **entreprise**) :

```
$ psql -U patron entreprise
```

Créer la table `facture` :

```
patron@entreprise=> CREATE TABLE facture (id int, objet text, creationts timestamp);
```

Noter que la table appartient à celui qui la crée :

```
patron@entreprise=> \d

           Liste des relations
 Schéma |   Nom    | Type  | Propriétaire
--------+----------+-------+--------------
 public | facture | table | patron
```

Création d'un utilisateur et d'un groupe

> Créer un rôle **secretariat** sans droit de connexion, mais qui peut visualiser, ajouter, mettre à jour et supprimer des éléments de la table `facture`.

Il faut le faire avec le rôle **postgres**, car **patron** n'a pas les droits :

```
patron@entreprise=# \c - postgres
You are now connected to database "entreprise" as user "postgres".

postgres@entreprise=# CREATE ROLE secretariat;
postgres@entreprise=# GRANT SELECT, INSERT, UPDATE, DELETE ON facture TO secretariat;
```

> Créer un rôle **boulier** qui peut se connecter et appartient au rôle **secretariat**, avec un mot de passe (à ajouter au `.pgpass`).

```
postgres@entreprise=# CREATE ROLE boulier LOGIN IN ROLE SECRETARIAT;
postgres@entreprise=# \password boulier

Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
```

> Vérifier la création des deux rôles.

```
postgres@entreprise=# \du
                     Liste des rôles
 Nom du rôle |      Attributs                        |   Membre de
-------------+---------------------------------------+---------------
 boulier     |                                       | {secretariat}
 dupont      |                                       | {}
 patron      |                                       | {}
 postgres    | Superutilisateur, Créer un rôle, ... | {}
 secretariat | Ne peut pas se connecter              | {}
 testperf    |                                       | {}
```

> En tant que **boulier**, créer une table `brouillon` identique à `facture`. Dans quel schéma cela se passe-t-il ? Avec PostgreSQL 15 ou supérieur, il y a une étape supplémentaire.

La connexion doit se faire sans problème avec le mot de passe.

```
$ psql -U boulier -d entreprise
```

Une astuce à connaître pour créer une table vide de même structure qu'une autre est :

```
boulier@entreprise=> CREATE TABLE brouillon (LIKE facture INCLUDING ALL) ;
```

Cet ordre fonctionnera directement jusque PostgreSQL 14 compris, car la table est créée implicitement dans le schéma **public**. À partir de PostgreSQL 15, par défaut, seul le propriétaire de la base (**patron**) peut écrire dans le schéma **public** :

```
ERROR:  permission denied for schema public
LINE 1: CREATE table brouillon (like facture);
```

Il doit donc donner ce droit à **boulier** :

```
patron@entreprise=> GRANT CREATE ON schema public TO boulier ;
patron@entreprise=> \dn+ public

                                    List of schemas
 Name  |      Owner      |           Access privileges            | Descript…
-------+-----------------+----------------------------------------+----------
public |pg_database_owner| pg_database_owner=UC/pg_database_owner+| standard
       |                 | =U/pg_database_owner                  +|  public
       |                 | boulier=C/pg_database_owner            |   schema

boulier@entreprise=> CREATE TABLE brouillon (LIKE facture INCLUDING ALL) ;
```

> Vérifier les tables présentes et les droits `\dp`. Comment les lire ?

La nouvelle table appartient bien à **boulier** :

```
boulier@entreprise=> \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 public | brouillon | table | boulier
 public | facture   | table | patron
(2 rows)

boulier@entreprise=> \dp

                                Access privileges
 Schema |   Name    | Type  |   Access privileges     | …
--------+-----------+-------+-------------------------+---
 public | brouillon | table |                         | …
 public | facture   | table | patron=arwdDxt/patron  +| …
        |           |       | secretariat=arwd/patron | …
```

Sans affectation explicite de droits, les droits par défauts ne figurent pas : par exemple, `brouillon` pourra être lu et modifié par son propriétaire, **boulier**.

patron a tous les droits sur la table `facture` (il possède la table).

On retrouve les droits donnés plus haut au rôle **secretariat** : insertion (`a` pour *append*), lecture (`r` pour *read*), modification (`w` pour *write*) et suppression (`d` pour *delete*).

On ne voit pas explicitement les droits de **boulier** (membre de **secretariat**) sur `facture`.

> À l'aide du rôle **boulier** : insérer 2 factures ayant pour objet « Vin de Bordeaux » et « Vin de Bourgogne » avec la date et l'heure courante.

```
boulier@entreprise=> INSERT INTO facture VALUES
    (1, 'Vin de Bordeaux', now()),
    (2, 'Vin de Bourgogne', now());
```

> Afficher le contenu de la table `facture`.

```
boulier@entreprise=> SELECT * FROM facture;
```

```
 id |      objet       |         creationts
----+------------------+----------------------------
  1 | Vin de Bordeaux  | 2019-07-16 17:50:28.942862
  2 | Vin de Bourgogne | 2019-07-16 17:50:28.942862
```

> Mettre à jour la deuxième facture avec la date et l'heure courante.

```
boulier@entreprise=> UPDATE facture SET creationts = now() WHERE id = 2 ;
```

```
UPDATE 1
```

> Supprimer la première facture.

```
boulier@entreprise=> DELETE FROM facture WHERE id = 1 ;
```

```
DELETE 1
```

Modification des permissions

> Retirer les droits `DELETE` sur la table `facture` au rôle **secretariat**.

```
boulier@entreprise=> \c - patron
```

```
Vous êtes maintenant connecté à la base de données « entreprise »
en tant qu'utilisateur « patron ».
```

```
patron@entreprise=> REVOKE DELETE ON facture FROM secretariat;
```

```
REVOKE
```

> Vérifier qu'il n'est plus possible de supprimer la deuxième facture avec le rôle **boulier**.

```
patron@entreprise=> \c - boulier
```

```
Vous êtes maintenant connecté à la base de données « entreprise »
en tant qu'utilisateur « boulier ».
```

```
boulier@entreprise=> DELETE FROM facture WHERE id = 2;
```

```
ERROR:  permission denied for table facture
```

> En tant que **patron**, créer une table **produit** contenant une colonne texte nommée `appellation` et la remplir avec des noms de boissons.

```
boulier@entreprise=> \c - patron
```

```
Vous êtes maintenant connecté à la base de données « entreprise »
en tant qu'utilisateur « patron ».
```

```
patron@entreprise=> CREATE TABLE produit (appellation text) ;
```

```
CREATE TABLE
```

```
patron@entreprise=> INSERT INTO produit VALUES
('Gewurtzraminer vendanges tardives'), ('Cognac'), ('Eau plate'),
('Eau gazeuse'), ('Jus de groseille') ;
```

```
INSERT 0 5
```

> Afficher les droits sur cette table avec `\dt` et `\dp`. Vérifier que le rôle **boulier** appartenant au rôle **secretariat** ne peut pas sélectionner les produits contenus dans la table `produit`.

On voit bien que `produit` appartient à **patron** et que **secretariat** n'a à priori aucun droit dessus.

```
patron@entreprise=> \dt

          List of relations
 Schema |   Name     | Type  |  Owner
--------+-----------+-------+--------------
 public | brouillon | table | boulier
 public | facture   | table | patron
 public | produit   | table | patron

patron@entreprise=> \dp

                                Access privileges
 Schema |   Name     | Type  |   Access privileges     | …
--------+-----------+-------+-------------------------+----
 public | brouillon | table |                         |
 public | facture   | table | patron=arwdDxt/patron +|
        |           |       | secretariat=arw/patron |
 public | produit   | table |                         |
```

En conséquence, **boulier** ne peut lire la table :

```
patron@entreprise=> \c - boulier
Vous êtes maintenant connecté à la base de données « entreprise »
en tant qu'utilisateur « boulier ».
boulier@entreprise=> SELECT * FROM produit;
ERROR:  permission denied for table produit
```

> Retirer tous les droits pour le groupe **secretariat** sur la table `produit`.

```
patron@entreprise=> REVOKE ALL ON produit from secretariat;
```

> Que deviennent les droits affichés ? **boulier** peut-il lire la table ?

secretariat n'avait pourtant aucun droit, mais l'affichage a changé et énumère à présent explicitement les droits présents :

```
patron@entreprise=> \dp

                                Access privileges
 Schema |   Name     | Type  |   Access privileges     | …
--------+-----------+-------+-------------------------+----
 public | brouillon | table |                         |
 public | facture   | table | patron=arwdDxt/patron +|
        |           |       | secretariat=arw/patron |
 public | produit   | table | patron=arwdDxt/patron   |
```

> Autoriser l'utilisateur **boulier** à accéder à la table `produit` en lecture.

```
patron@entreprise=> GRANT SELECT ON produit TO boulier ;
```

```
GRANT
```

> Vérifier que **boulier** peut désormais accéder à la table `produit`.

```
boulier@entreprise=> SELECT * FROM produit ;
```

```
            appellation
-----------------------------------
 Gewurtzraminer vendanges tardives
 Cognac
 Eau plate
 Eau gazeuse
 Jus de groseille
(5 rows)
```

Héritage des droits au login

> Créer un rôle **tina** appartenant au rôle **secretariat**, avec l'attribut `LOGIN`, mais n'héritant pas des droits à la connexion. Vérifier les droits avec `\du`. Lui donner un mot de passe.

La clause `NOINHERIT` évite qu'un rôle hérite immédiatement des droits des autres rôles :

```
postgres@entreprise=> CREATE ROLE tina LOGIN NOINHERIT ;
```

```
CREATE ROLE
```

```
postgres@entreprise=> GRANT secretariat TO tina;
```

```
postgres@entreprise=# \du
```

```
                                    List of roles
  Role name  |      Attributes                      |   Member of
-------------+--------------------------------------+---------------
 …
 tina        | No inheritance                       | {secretariat}
 …
```

```
postgres@entreprise=# \password tina
```

Tester la connexion en tant que **tina**.

> Vérifier que **tina** ne peut pas accéder à la table `facture`.

```
tina@entreprise=> SELECT * FROM facture;
```

```
ERROR:  permission denied for table facture
```

> En tant que **tina**, activer le rôle **secretariat** (`SET ROLE`).

```
tina@entreprise=> SET ROLE secretariat;
```

> Vérifier que **tina** possède maintenant les droits du rôle **secretariat**. Sélectionner les données de la table `facture`.

L'utilisateur **tina** possède maintenant les droits du rôle **secretariat** :

```
tina@entreprise=> SELECT * FROM facture;
 id |      objet       |         creationts
----+------------------+----------------------------
  2 | Vin de Bourgogne | 2019-07-16 17:50:53.725971
```

5.12.6 Autorisation d'accès distant

But : Mettre en place les accès dans `pg_hba.conf`.

> Autoriser tous les membres du réseau local à se connecter avec un mot de passe (autorisation en IP sans SSL) avec les utilisateurs **boulier** ou **tina**. Tester avec l'IP du serveur avant de demander aux voisins de tester.

Pour tester, repérer l'adresse IP du serveur avec `ip a`, par exemple `192.168.28.1`, avec un réseau local en `192.168.28.*`.

Ensuite, lors des appels à `psql`, utiliser `-h 192.168.28.1` au lieu de la connexion locale ou de **localhost** :

```
$ psql -h 192.168.123.180 -d entreprise -U tina
```

Ajouter les lignes suivantes dans le fichier `pg_hba.conf` :

```
host        entreprise    tina,boulier    192.168.28.0/24  scram-sha-256
```

Il ne faut pas oublier d'ouvrir PostgreSQL aux connexions extérieures dans `postgresql.conf` :

```
listen_addresses = '*'
```

Cette modification néces¨site un rédémarrage

Plus prudemment, on peut juste ajouter l'adresse publique de l'instance PostgreSQL :

```
listen_addresses = 'localhost,192.168.28.1'
```

Il y a peut-être un *firewall* à désactiver :

```
$ sudo systemctl status firewalld
$ sudo systemctl stop firewalld
```

5.12.7 VACUUM, VACUUM FULL, DELETE, TRUNCATE

But : Effacer des données, distinguer `VACUUM` et `VACUUM FULL`.

Pré-requis

> Désactiver le démon autovacuum de l'instance.

Dans le fichier `postgresql.conf`, désactiver le démon autovacuum en modifiant le paramètre suivant :

```
autovacuum = off
```

Ne jamais faire cela en production !

On recharge la configuration :

```
$ psql -c 'SELECT pg_reload_conf()'
```

On vérifie que le paramètre a bien été modifié :

```
postgres@postgres=# show autovacuum ;

 autovacuum
------------
 off
```

Nettoyage avec VACUUM

> Se connecter à la base **pgbench** en tant que **testperf**.

```
$ psql -U testperf -d pgbench
```

> Grâce aux fonctions `pg_relation_size` et `pg_size_pretty`, afficher la taille de la table `pgbench_accounts`.

`\d+` affiche les tailles des tables, mais il existe des fonctions plus ciblées.

Pour visualiser la taille de la table, il suffit d'utiliser la fonction `pg_relation_size`. Comme l'affichage a parfois trop de chiffres pour être facilement lisible, on utilise `pg_size_pretty`.

Il est facile de retrouver facilement une fonction en effectuant une recherche par mot clé dans `psql`, notamment pour retrouver ses paramètres. Exemple :

```
postgres=# \df *pretty*

Liste des fonctions
-[ RECORD 1 ]------------------+---------------
Schéma                         | pg_catalog
Nom                            | pg_size_pretty
Type de données du résultat    | text
Type de données des paramètres | bigint
Type                           | normal
```

Cela donne au final :

```
testperf@pgbench=> SELECT pg_relation_size('pgbench_accounts');

  pg_relation_size
 ------------------
         13434880

testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pgbench_accounts'));

  pg_size_pretty
 ----------------
  13 MB
```

> Copier le contenu de la table dans une nouvelle table (`pba_copie`).

```
testperf@pgbench=> CREATE table pba_copie AS SELECT * FROM pgbench_accounts;

SELECT 100000
```

> Supprimer le contenu de la table `pba_copie`, à l'exception de la dernière ligne (`aid=100000`), avec un ordre `DELETE`. Quel est alors l'espace disque utilisé par cette table ?

```
testperf@pgbench=> DELETE FROM pba_copie WHERE aid <100000;

DELETE 99999
```

Il ne reste qu'une ligne, mais l'espace disque est toujours utilisé :

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));

 pg_size_pretty
----------------
 13 MB
```

Noter que même si l'autovacuum n'était pas désactivé, il n'aurait pas réduit l'espace occupé par la table car il reste la ligne à la fin de celle-ci. De plus, il n'aurait pas eu forcément le temps de passer sur la table entre les deux ordres précédents.

> Insérer le contenu de la table `pgbench_accounts` dans la table `pba_copie`. Quel est alors l'espace disque utilisé par la table ?

```
testperf@pgbench=> INSERT into pba_copie SELECT * FROM pgbench_accounts;
```

```
INSERT 0 100000
```

L'espace disque utilisé a doublé :

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
```

```
 pg_size_pretty
----------------
 26 MB
```

Les nouvelles données se sont ajoutées à la fin de la table. Elles n'ont pas pris la place des données effacées précédemment.

> Effectuer un VACUUM simple sur `pba_copie`. Vérifier la taille de la base.

La commande vacuum « nettoie » mais ne libère pas d'espace disque :

```
testperf@pgbench=> VACUUM pba_copie;
```

```
VACUUM
```

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
```

```
 pg_size_pretty
----------------
 26 MB
```

> Vider à nouveau la table `pba_copie` des lignes d'`aid` inférieur à 100 000. Insérer à nouveau le contenu de la table `pgbench_accounts`. L'espace mis à disposition a-t-il été utilisé ?

```
testperf@pgbench=> DELETE FROM pba_copie WHERE aid <100000;
```

```
DELETE 99999
```

```
testperf@pgbench=> INSERT into pba_copie SELECT * FROM pgbench_accounts;
```

```
INSERT 0 100000
```

```
testperf@pgbench=> SELECT pg_size_pretty(pg_relation_size('pba_copie'));
```

```
 pg_size_pretty
----------------
 26 MB
```

Cette fois, la table n'a pas augmenté de taille. PostgreSQL a pu réutiliser la place des lignes effacées que `VACUUM` a marqué comme disponibles.

> Voir la taille de la base. Supprimer la table `pba_copie`. Voir l'impact sur la taille de la base.

Nous verrons plus tard comment récupérer de l'espace. Pour le moment, on se contente de supprimer la table.

```
postgres@pgbench=# SELECT pg_size_pretty(pg_database_size ('pgbench')) ;

 pg_size_pretty
----------------
 49 MB

postgres@pgbench=# DROP TABLE pba_copie ;

DROP TABLE

postgres@pgbench=# SELECT pg_size_pretty(pg_database_size ('pgbench')) ;

 pg_size_pretty
----------------
 23 MB
```

Supprimer une table rend immédiatement l'espace disque au système.

VACUUM avec les requêtes de pgbench

> Tout d'abord, repérer les tailles des différentes tables et le nombre de lignes de chacune.

```
postgres@pgbench=#d+

                                Liste des relations
 Schéma |          Nom          | Type  | Propriétaire |   Taille   | Description
--------+-----------------------+-------+--------------+------------+-------------
 public | pgbench_accounts      | table | testperf     | 13 MB      |
 public | pgbench_branches      | table | testperf     | 40 kB      |
 public | pgbench_history       | table | testperf     | 0 bytes    |
 public | pgbench_tellers       | table | testperf     | 40 kB      |

testperf@pgbench=> SELECT count(*) FROM pgbench_accounts;

 count
--------
 100000

testperf@pgbench=> SELECT count(*) FROM pgbench_tellers;

 count
-------
    10

testperf@pgbench=> SELECT count(*) FROM pgbench_branches;

 count
-------
     1

testperf@pgbench=> SELECT count(*) FROM pgbench_history;
```

```
 count
--------
     0
```

(Le contenu de cette dernière table dépend de l'historique de la base.)

> Pour amplifier le phénomène à observer, on peut créer une session de très longue durée, laissée ouverte sans `COMMIT` ni `ROLLBACK`. Il faut qu'elle ait consulté une des tables pour que l'effet soit visible :
>
> ```
> testperf@pgbench=> BEGIN ;
>
> BEGIN
> Temps : 0,608 ms
>
> testperf@pgbench=> SELECT count(*) FROM pgbench_accounts ;
>
>  count
> --------
>  100000
> (1 ligne)
>
> Temps : 26,059 ms
> testperf@pgbench=> SELECT pg_sleep (10000) ;
> ```

> Depuis un autre terminal, générer de l'activité sur la table, ici avec 10 000 transactions sur 20 clients :
>
> ```
> PGOPTIONS='-c synchronous_commit=off' \
> /usr/pgsql-17/bin/pgbench -U testperf -d pgbench \
> --client=20 --jobs=2 -t 10000 --no-vacuum
> ```
>
> (NB : La variable d'environnement `PGOPTIONS` restreint l'utilisation des journaux de transaction pour accélérer les écritures (données NON critiques ici). Le `--no-vacuum` est destiné à éviter que l'outil demande lui-même un `VACUUM`. Le test dure quelques minutes. Le relancer au besoin.)

Après quelques minutes, `pgbench` affichera le nombre de transactions par seconde, bien sûr très dépendant de la machine :

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 2
number of transactions per client: 10000
number of transactions actually processed: 200000/200000
latency average = 58.882 ms
tps = 339.663189 (including connections establishing)
tps = 339.664978 (excluding connections establishing)
```

> (Optionnel) C'est l'occasion d'installer l'outil `pg_activity` depuis les dépôts du PGDG (il peut y avoir besoin du dépôt EPEL) et de le lancer en tant que **postgres** pour voir ce qui se passe dans la base.

Pour pg_activity :

```
$ sudo yum install epel-release
$ sudo yum install pg_activity
```

Il se lance ainsi :

```
$ sudo -iu postgres pg_activity
```

Le premier écran affiche les sessions en cours, le deuxième celles en attente de libération d'un verrou, le troisième celles qui en bloquent d'autres.

Noter que la session restée délibérément ouverte n'est pas bloquante.

> Comparer les nouvelles tailles des tables (taille sur le disque et nombre de lignes). La table `pg_stat_user_tables` contient l'activité sur chaque table. Comment s'expliquent les évolutions ?

La volumétrie des tables a explosé :

```
testperf@pgbench=> \d+

 Schéma |       Nom        | Type  | Propriétaire | Taille  | Description
--------+------------------+-------+--------------+---------+-------------
 public | pgbench_accounts | table | testperf     | 39 MB   |
 public | pgbench_branches | table | testperf     | 7112 kB |
 public | pgbench_history  | table | testperf     | 10 MB   |
 public | pgbench_tellers  | table | testperf     | 8728 kB |
```

On constate que le nombre de lignes reste le même malgré l'activité, sauf pour la table d'historique :

```
testperf@pgbench=> SELECT count(*) FROM pgbench_accounts;

 count
--------
 100000

testperf@pgbench=> SELECT count(*) FROM pgbench_tellers;

 count
-------
    10

testperf@pgbench=> SELECT count(*) FROM pgbench_branches;

 count
-------
     1

testperf@pgbench=> SELECT count(*) FROM pgbench_history;
```

```
 count
--------
 200000
```

Ce dernier chiffre dépend de l'activité réelle et du nombre de requêtes.

Les statistiques d'activité de la table sont dans `pg_stat_user_tables`. Pour `pgbench_accounts`, la plus grosse table, on y trouve ceci :

```
testperf@pgbench=> SELECT * FROM pg_stat_user_tables ORDER BY relname \gx
```

```
-[ RECORD 1 ]-------+------------------------------
relid               | 17487
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 6
seq_tup_read        | 300000
idx_scan            | 600000
idx_tup_fetch       | 600000
n_tup_ins           | 100000
n_tup_upd           | 200000
n_tup_del           | 0
n_tup_hot_upd       | 1120
n_live_tup          | 100000
n_dead_tup          | 200000
n_mod_since_analyze | 300000
last_vacuum         | 2021-09-04 18:51:31.889825+02
last_autovacuum     | ¤
last_analyze        | 2021-09-04 18:51:31.927611+02
last_autoanalyze    | ¤
vacuum_count        | 1
autovacuum_count    | 0
analyze_count       | 1
autoanalyze_count   | 0
```

Le champ `n_tup_upd` montre qu'il y a eu 200 000 mises à jour après l'insertion initiale de 100 000 lignes (champ `n_tup_ins`). Il y a toujours 100 000 lignes visibles (`n_live_tup`).

Le `VACUUM` a été demandé explicitement à la création (`last_vacuum`) mais n'est pas passé depuis.

La `VACUUM` étant inhibé, il est normal que les lignes mortes se soient accumulées (`n_dead_tup`) : il y en a 200 000, ce sont les anciennes versions des lignes modifiées.

Pour la table `pgbench_history` :

```
-[ RECORD 3 ]-------+------------------------------
relid               | 17481
schemaname          | public
relname             | pgbench_history
seq_scan            | 4
seq_tup_read        | 200000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 200000
n_tup_upd           | 0
n_tup_del           | 0
```

```
n_tup_hot_upd       | 0
n_live_tup          | 200000
n_dead_tup          | 0
...
```

La table `pgbench_history` a subi 200 000 insertions et contient à présent 200 000 lignes : il est normal qu'elle ait grossi de 0 à 10 Mo.

Pour la table `pgbench_tellers` :

```
-[ RECORD 4 ]-------+-----------------------------
...
relname             | pgbench_tellers
seq_scan            | 20383
seq_tup_read        | 117437
idx_scan            | 379620
idx_tup_fetch       | 379620
n_tup_ins           | 10
n_tup_upd           | 200000
...
n_live_tup          | 10
n_dead_tup          | 199979
n_mod_since_analyze | 200010
...
```

Elle ne contient toujours que 10 lignes visibles (`n_live_up`), mais 199 979 lignes « mortes » (`n_dead_tup`).

Même s'il n'a gêné aucune opération du point de vue de l'utilisateur, le verrou posé par la session en attente est visible dans la table des verrous `pg_locks` :

```
postgres@pgbench=# SELECT * FROM pg_locks
  WHERE relation = (SELECT relid FROM pg_stat_user_tables
  WHERE relname = 'pgbench_accounts' ) ;

-[ RECORD 1 ]------+----------------
locktype           | relation
database           | 16729
relation           | 17487
page               | ¤
tuple              | ¤
virtualxid         | ¤
transactionid      | ¤
classid            | ¤
objid              | ¤
objsubid           | ¤
virtualtransaction | 1/37748
pid                | 22581
mode               | AccessShareLock
granted            | t
fastpath           | t
waitstart          | 2021-09-04 19:01:27.824567+02
```

Nettoyage avec VACUUM FULL

> Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_tellers`.

```
postgres@pgbench=# VACUUM FULL VERBOSE pgbench_tellers ;
```

```
INFO:  vacuuming "public.pgbench_tellers"
INFO:  "pgbench_tellers": found 200000 removable, 10 nonremovable row versions in
 ↪  1082 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.03 s.
VACUUM
```

Un `\d+` indique que la taille de la table est bien retombée à 8 ko (1 bloc), ce qui suffit pour 10 lignes.

> Exécuter un `VACUUM FULL VERBOSE` sur `pgbench_accounts`.

Si celui-ci reste bloqué, il faudra sans doute arrêter la transaction restée ouverte plus haut.

```
postgres@pgbench=# VACUUM FULL VERBOSE pgbench_accounts ;
```

```
INFO:  vacuuming "public.pgbench_accounts"
INFO:  "pgbench_accounts": found 200000 removable,100000 nonremovable row versions
       in 4925 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.09 s, system: 0.06 s, elapsed: 0.17 s.
VACUUM
Durée : 16411,719 ms (00:16,412)
```

Soit : 100 000 lignes conservées, 200 000 mortes supprimées dans 4925 blocs (39 Mo).

> Effectuer un `VACUUM FULL VERBOSE`. Quel est l'impact sur la taille de la base ?

Même les tables système seront nettoyées :

```
postgres@pgbench=> VACUUM FULL VERBOSE ;
```

```
INFO:  vacuuming "pg_catalog.pg_statistic"
INFO:  "pg_statistic": found 11 removable, 433 nonremovable row versions in 20 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
...
INFO:  vacuuming "public.pgbench_branches"
INFO:  "pgbench_branches": found 200000 removable, 1 nonremovable row versions in
 ↪  885 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.03 s.
INFO:  vacuuming "public.pgbench_history"
INFO:  "pgbench_history": found 0 removable, 200000 nonremovable row versions in
 ↪  1281 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.11 s, system: 0.02 s, elapsed: 0.13 s.
INFO:  vacuuming "public.pgbench_tellers"
INFO:  "pgbench_tellers": found 0 removable, 10 nonremovable row versions in 1 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
```

```
INFO:  vacuuming "public.pgbench_accounts"
INFO:  "pgbench_accounts": found 0 removable, 100000 nonremovable row versions in
 ↪  1640 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.03 s, system: 0.01 s, elapsed: 0.05 s.
VACUUM
```

Seule `pgbench_branches` était encore à nettoyer (1 ligne conservée).

La taille de la base retombe à 32 Mo selon `\l+`. Elle faisait au départ 22 Mo, et 10 Mo ont été ajoutés dans `pgbench_history`.

Truncate ou Delete ?

> Créer `copie1` et `copie2` comme des copies de `pgbench_accounts`, données incluses.

```
postgres@pgbench=# CREATE TABLE copie1 AS SELECT * FROM pgbench_accounts ;
```

```
SELECT 100000
```

```
postgres@pgbench=# CREATE TABLE copie2 AS SELECT * FROM pgbench_accounts ;
```

```
SELECT 100000
```

> Effacer le contenu de `copie1` avec `DELETE`.

```
postgres@pgbench=# DELETE FROM copie1 ;
```

```
DELETE 100000
```

> Effacer le contenu de `copie2` avec `TRUNCATE`.

```
postgres@pgbench=# TRUNCATE copie2 ;
```

```
TRUNCATE TABLE
```

> Quelles sont les tailles de ces deux tables après ces opérations ?

```
postgres@pgbench=# \d+
```

```
                           Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Taille  | Description
--------+------------------+-------+--------------+---------+-------------
 public | copie1           | table | postgres     | 13 MB   |
 public | copie2           | table | postgres     | 0 bytes |
...
```

Pour une purge complète, `TRUNCATE` est à préférer : il vide la table et rend l'espace au système. `DELETE` efface les lignes mais l'espace n'est pas encore rendu.

5.12.8 Réactivation de l'autovacuum

> Réactiver l'autovacuum de l'instance.

Dans `postgresql.conf` :

```
autovacuum = on
```

```
postgres@pgbench=# SELECT pg_reload_conf() ;

 pg_reload_conf
----------------
 t

postgres@pgbench=# SHOW autovacuum;

 autovacuum
------------
 on
```

> Attendre quelques secondes et voir si `copie1` change de taille.

Après quelques instants, la taille de `copie1` (qui avait été vidée plus tôt avec `DELETE`) va redescendre à quelques kilooctets.

Le passage de l'autovacuum en arrière-plan est tracé dans `last_autovacuum` :

```
postgres@pgbench=# SELECT * FROM pg_stat_user_tables WHERE relname ='copie1' \gx

-[ RECORD 1 ]-------+-----------------------------
relid               | 18920
schemaname          | public
relname             | copie1
seq_scan            | 1
seq_tup_read        | 100000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 100000
n_tup_upd           | 0
n_tup_del           | 100000
n_tup_hot_upd       | 0
n_live_tup          | 0
n_dead_tup          | 0
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     | 2019-07-17 14:04:21.238296+01
last_analyze        |
last_autoanalyze    | 2019-07-17 14:04:21.240525+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 1
```

5.12.9 Statistiques des données

But : Savoir trouver les statistiques des données et les mettre à jour

> Créer une table `copie3`, copie de `pgbench_accounts`.

```
CREATE TABLE copie3 AS SELECT * FROM pgbench_accounts ;
```

```
SELECT 100000
```

> Dans la vue système `pg_stats`, afficher les statistiques collectées pour la table `copie3`.

```
postgres@pgbench=# SELECT * FROM pg_stats WHERE tablename = 'copie3' ;
```

```
(0 ligne)
```

L'autovacuum n'est pas passé, les statistiques ne sont pas encore présentes. Noter que, même activé, il n'aurait pas forcément eu le temps de passer entre les deux ordres précédents.

> Lancer la collecte des statistiques pour cette table uniquement.

La collecte se déclenche avec la commande `ANALYZE` :

```
postgres@pgbench=# ANALYZE VERBOSE copie3 ;
```

```
INFO:  analyzing "public.copie3"
INFO:  "copie3": scanned 1640 of 1640 pages,
       containing 100000 live rows and 0 dead rows;
       30000 rows in sample, 100000 estimated total rows
ANALYZE
```

30 000 lignes font partie de l'échantillonnage.

> Afficher de nouveau les statistiques.

```
SELECT * FROM pg_stats WHERE tablename = 'copie3' ;
```

Cette fois, la vue `pg_stats` renvoie des informations, colonne par colonne.

Le champ `aid` est la clé primaire, ses valeurs sont toutes différentes. L'histogramme des valeurs compte 100 valeurs qui délimite autant de *buckets*. Ils sont là régulièrement répartis, ce qui indique une répartition homogène des valeurs.

```
SELECT * FROM pg_stats WHERE tablename = 'copie3' ;
```

```
-[ RECORD 1 ]-------------------------------------------------
schemaname             | public
tablename              | copie3
```

```
attname                 | aid
inherited               | f
null_frac               | 0
avg_width               | 4
n_distinct              | -1
most_common_vals        |
most_common_freqs       |
histogram_bounds        | {2,1021,2095,3098,4058,5047,6120,
7113,8058,9075,10092,11090,12061,13064,14053,15091,16106,
17195,18234,19203,20204,21165,22183,23156,24162,25156,26192,
27113,28159,29193,30258,31260,32274,33316,34346,35350,36281,
37183,38158,39077,40007,41070,42084,43063,44064,45101,46089,
47131,48189,49082,50100,51157,52113,53009,54033,55120,56114,
57066,58121,59111,60122,61088,62151,63217,64195,65168,66103,
67088,68126,69100,70057,71104,72105,73092,73994,75007,76067,
77092,78141,79180,80165,81100,82085,83094,84107,85200,86242,
87246,88293,89288,90286,91210,92197,93172,94084,95070,96086,
97067,98031,99032,99998}
correlation             | 1
most_common_elems       |
most_common_elem_freqs  |
elem_count_histogram    |
```

Autre exemple, le champ `bid` : on voit qu'il ne possède qu'une seule valeur.

```
-[ RECORD 2 ]-------------------------------------------------
schemaname              | public
tablename               | copie3
attname                 | bid
inherited               | f
null_frac               | 0
avg_width               | 4
n_distinct              | 1
most_common_vals        | {1}
most_common_freqs       | {1}
histogram_bounds        |
correlation             | 1
most_common_elems       |
most_common_elem_freqs  |
elem_count_histogram    |
```

De même, on pourra vérifier que le champ `filler` a une taille moyenne de 85 octets, ou voir la répartition des valeurs du champ `abalance`.

5.12.10 Réindexation

But : Réindexer

> Recréer les index de la table `pgbench_accounts`.

La réindexation d'une table se fait de la manière suivante :

```
postgres@pgbench=# REINDEX (VERBOSE) TABLE pgbench_accounts ;
```

```
INFO:  index "pgbench_accounts_pkey" was reindexed
DÉTAIL : CPU: user: 0.05 s, system: 0.00 s, elapsed: 0.08 s
REINDEX
```

> Comment recréer tous les index de la base **pgbench** ?

```
postgres@pgbench=# REINDEX (VERBOSE) DATABASE pgbench ;
```

```
...
INFO:  index "pg_shseclabel_object_index" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  index "pg_toast_3592_index" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  table "pg_catalog.pg_shseclabel" was reindexed
INFO:  index "pgbench_branches_pkey" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  table "public.pgbench_branches" was reindexed
INFO:  index "pgbench_tellers_pkey" was reindexed
DÉTAIL : CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  table "public.pgbench_tellers" was reindexed
INFO:  index "pgbench_accounts_pkey" was reindexed
DÉTAIL : CPU: user: 0.07 s, system: 0.01 s, elapsed: 0.12 s
INFO:  table "public.pgbench_accounts" was reindexed
REINDEX
```

> Comment recréer uniquement les index des tables systèmes ?

Pour réindexer uniquement les tables systèmes :

```
postgres@pgbench=# REINDEX SYSTEM pgbench ;
```

> Quelle est la différence entre la commande `REINDEX` et la séquence `DROP INDEX` + `CREATE INDEX` ?

`REINDEX` est similaire à une suppression et à une nouvelle création de l'index. Cependant, les conditions de verrouillage sont différentes :

- `REINDEX` verrouille les écritures mais pas les lectures de la table mère de l'index. Il prend aussi un verrou exclusif sur l'index en cours de traitement, ce qui bloque les lectures qui tentent d'utiliser l'index.
- Au contraire, `DROP INDEX` crée temporairement un verrou exclusif sur la table parent, bloquant ainsi écritures et lectures. Le `CREATE INDEX` qui suit verrouille les écritures mais pas les lectures ; comme l'index n'existe pas, aucune lecture ne peut être tentée, signifiant qu'il n'y a aucun blocage et que les lectures sont probablement forcées de réaliser des parcours séquentiels complets.

5.12.11 Traces

But : Gérer les fichiers de traces

> Quelle est la méthode de gestion des traces utilisée par défaut ?

Par défaut, le mode de journalisation est **stderr** :

```
postgres@pgbench=# SHOW log_destination ;

 log_destination
-----------------
 stderr
```

> Paramétrer le programme interne de rotation des journaux :
>
> - modifier le fichier `postgresql.conf` pour utiliser le *logging collector* ;
> - les traces doivent désormais être sauvegardés dans le répertoire `/var/lib/pgsql/traces` ;
> - la rotation des journaux doit être automatisée pour générer un nouveau fichier de logs toutes les 30 minutes, quelle que soit la quantité de logs archivés ; le nom du fichier devra donc comporter les minutes.
> - Tester en forçant des rotations avec la fonction `pg_rotate_logfile`.
> - Augmenter la trace (niveau `info`).

Sur Red Hat/CentOS/Rocky Linux, le collecteur des traces (*logging collector*) est activé par défaut dans `postgresql.conf` (mais ce ne sera pas le cas sur un environnement Debian ou avec une installation compilée, et il faudra redémarrer pour l'activer) :

```
logging_collector = on
```

On crée le répertoire, où **postgres** doit pouvoir écrire :

```
$ sudo mkdir -m700 /var/lib/pgsql/traces
$ sudo chown postgres:  /var/lib/pgsql/traces
```

Puis paramétrer le comportement du récupérateur :

```
log_directory = '/var/lib/pgsql/traces'
log_filename = 'postgresql-%Y-%m-%d_%H-%M.log'
log_rotation_age = 30min
log_rotation_size = 0
log_min_messages = info
```

Recharger la configuration et voir ce qui se passe dans `/var/lib/pgsql/traces` :

```
$ sudo systemctl reload postgresql-12
$ sudo watch -n 5 ls -lh /var/lib/pgsql/traces
```

Dans une autre fenêtre, générer un peu d'activité, avec `pgbench` ou tout simplement avec :

```
postgres@pgbench=# SELECT 1 ;
postgres@pgbench=# \watch 1
```

Les fichiers générés doivent porter un nom ressemblant à `postgresql-2019-08-02_16-55.log`.

Pour forcer le passage à un nouveau fichier de traces :

```
postgres@pgbench=# SELECT pg_rotate_logfile() ;
```

> Comment éviter l'accumulation des fichiers ?

- La première méthode consiste à avoir un `log_filename` cyclique. C'est le cas par défaut sur Red Hat/CentOS/Rocky Linux avec `postgresql-%a`, qui reprend les jours de la semaine. Noter qu'il n'est pas forcément garanti qu'un `postgresql-%H-%M.log` planifié toutes les 5 minutes écrase les fichiers de la journée précédente. En pratique, on descend rarement en-dessous de l'heure.
- Utiliser l'utilitaire `logrotate`, fourni avec toute distribution Linux, dont le travail est de gérer rotation, compression et purge. Il est activé par défaut sur Debian.
- Enfin, on peut rediriger les traces vers un système externe.

6/ PostgreSQL : Politique de sauvegarde

Photo de l'incendie du datacenter OVHcloud à Strasbourg du 10 mars 2021 fournie gracieusement par l'ITBR67[1].

Cet incendie a provoqué de nombreux arrêts et pertes de données dans toute la France et ailleurs[2].

[1] https://itbr67.fr/
[2] https://fr.wikipedia.org/wiki/Incendie_du_centre_de_donn%C3%A9es_d%27OVHcloud_%C3%A0_Strasbourg

6.1 INTRODUCTION

- Le pire peut arriver
- Politique de sauvegarde

6.1.1 Au menu

- Objectifs
- Approche
- Points d'attention

6.2 DÉFINIR UNE POLITIQUE DE SAUVEGARDE

- Pourquoi établir une politique ?
- Que sauvegarder ?
- À quelle fréquence sauvegarder les données ?
- Quels supports ?
- Quels outils ?
- Vérifier la restauration des sauvegardes

Afin d'assurer la sécurité des données, il est nécessaire de faire des sauvegardes régulières.

Ces sauvegardes vont servir, en cas de problème, à restaurer les bases de données dans un état le plus proche possible du moment où le problème est survenu.

Cependant, le jour où une restauration sera nécessaire, il est possible que la personne qui a mis en place les sauvegardes ne soit pas présente. C'est pour cela qu'il est essentiel d'écrire et de maintenir un document qui indique la mise en place de la sauvegarde et qui détaille comment restaurer une sauvegarde.

En effet, suivant les besoins, les outils pour sauvegarder, le contenu de la sauvegarde, sa fréquence ne seront pas les mêmes.

Par exemple, il n'est pas toujours nécessaire de tout sauvegarder. Une base de données peut contenir des données de travail, temporaires et/ou faciles à reconstruire, stockées dans des tables standards. Il est également possible d'avoir une base dédiée pour stocker ce genre d'objets. Pour diminuer le temps de sauvegarde (et du coup de restauration), il est possible de sauvegarder partiellement son serveur pour ne conserver que les données importantes.

La fréquence peut aussi varier. Un utilisateur peut disposer d'un serveur PostgreSQL pour un entrepôt de données, serveur qu'il n'alimente qu'une fois par semaine. Dans ce cas, il est inutile de sauvegarder tous les jours. Une sauvegarde après chaque alimentation (donc chaque semaine) est suffisante. En fait, il faut déterminer la fréquence de sauvegarde des données selon :

- le volume de données à sauvegarder et/ou restaurer ;
- la criticité des données ;
- la quantité de données qu'il est « acceptable » de perdre en cas de problème.

Le support de sauvegarde est lui aussi très important. Il est possible de sauvegarder les données sur un disque réseau (à travers SMB/CIFS ou NFS), sur des disques locaux dédiés, sur des bandes ou tout autre support adapté. Dans tous les cas, il est fortement déconseillé de stocker les sauvegardes sur les disques utilisés par la base de données.

Ce document doit aussi indiquer comment effectuer la restauration. Si la sauvegarde est composée de plusieurs fichiers, l'ordre de restauration des fichiers peut être essentiel. De plus, savoir où se trouvent les sauvegardes permet de gagner un temps important, qui évitera une immobilisation trop longue.

De même, vérifier la restauration des sauvegardes de façon régulière est une précaution très utile.

6.2.1 Objectifs

- Sécuriser les données
- Mettre à jour le moteur de données
- Dupliquer une base de données de production
- Archiver les données

L'objectif essentiel de la sauvegarde est la sécurisation des données. Autrement dit, l'utilisateur cherche à se protéger d'une panne matérielle ou d'une erreur humaine (un utilisateur qui supprimerait des données essentielles). La sauvegarde permet de restaurer les données perdues. Mais ce n'est pas le seul objectif d'une sauvegarde.

Une sauvegarde peut aussi servir à dupliquer une base de données sur un serveur de développement, de test ou de préproduction. Elle permet aussi d'archiver des tables. Cela se voit par exemple dans le cadre des tables partitionnées, où l'archivage de la table la plus ancienne permet ensuite sa suppression de la base pour gagner en espace disque.

Un autre cas d'utilisation de la sauvegarde est la mise à jour majeure de versions PostgreSQL. Il s'agit de la solution historique de mise à jour (export/import). Historique, mais pas obsolète.

6.2.2 Différentes approches

- Sauvegarde à chaud en SQL (ou logique)
- Sauvegarde physique des fichiers à froid
- Sauvegarde à chaud des fichiers + journaux
  - niveau baie
  - `pg_basebackup`
- Sauvegarde physique & PITR

À ces différents objectifs vont correspondre différentes approches de la sauvegarde.

La sauvegarde logique permet de créer un fichier texte de commandes SQL ou un fichier binaire contenant le schéma et les données de la base de données, à chaud (sans arrêt de l'activité de la base). C'est une méthode très sûre, mais souvent trop longue.

La sauvegarde au niveau système de fichiers permet de conserver une image cohérente de l'intégralité des répertoires de données. Pour garantir cette cohérence, elle n'est simplement possible qu'à froid (base arrêtée), ce qui est rarement possible de nos jours.

La sauvegarde à chaud des fichiers est cependant possible avec quelques précautions, notamment l'archivage des journaux. PostgreSQL fournit un outil avec quelques limitations mais très simple pour cela : `pg_basebackup`.

Une autre alternative est le snapshot de baie (avec les mêmes précautions, ou avec des outils comme Veeam ou Commvault…), ce qui est très intéressant avec les grosses bases de données.

Les méthodes précédentes ne donnent qu'une image à un moment donné de la base. Les sauvegardes PITR sont une évolution de la sauvegarde physique, où la restauration peut se faire à n'importe quel moment du passé.

Suivant les prérequis et les limitations de chaque méthode, il est fort possible qu'une seule de ces solutions soit utilisable. Par exemple :

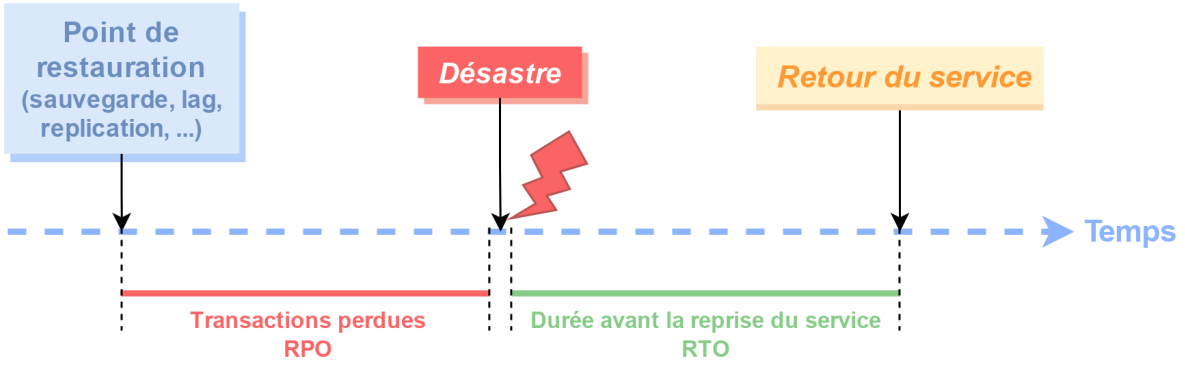
- si le serveur ne peut pas être arrêté, la sauvegarde à froid est exclue d'office ;
- si la base de données est très volumineuse, la sauvegarde logique devient très longue ;
- si l'espace disque est limité et que l'instance génère beaucoup de journaux de transactions, la sauvegarde PITR sera difficile à mettre en place.

Rien n'interdit d'utiliser plusieurs méthodes à la fois pour différents besoins.

6.2.3 RTO/RPO

La politique de sauvegarde découle du :

- **RPO** (*Recovery Point Objective*) : Perte de Données Maximale Admissible
  - faible ou importante ?
- **RTO** (*Recovery Time Objective*) : Durée Maximale d'Interruption Admissible
  - courte ou longue ?



Le RPO et RTO sont deux concepts déterminants dans le choix des politiques de sauvegardes.

La **RPO** (ou PDMA[3]) est la perte de données maximale admissible, ou quantité de données que l'on peut tolérer de perdre lors d'un sinistre majeur, souvent exprimée en heures ou minutes.

Pour un système mis à jour épisodiquement ou avec des données non critiques, ou facilement récupérables, le RPO peut être important (par exemple une journée). Peuvent alors s'envisager des solutions comme :

- les sauvegardes logiques (dump) ;
- les sauvegardes des fichiers à froid.

Dans beaucoup de cas, la perte de données admissible est très faible (heures, quelques minutes), voire nulle. Il faudra s'orienter vers des solutions de type :

- sauvegarde à chaud ;
- sauvegarde d'instantané à un point donnée dans le temps (PITR) ;
- réplication asynchrone, voire synchrone.

La **RTO** (ou DMIA[4]) est la durée maximale d'interruption du service.

Dans beaucoup de cas, les utilisateurs peuvent tolérer une indisponibilité de plusieurs heures, voire jours. La durée de reprise du service n'est alors pas critique, on peut utiliser des solutions simples comme :

- la restauration des fichiers ;
- la restauration d'une sauvegarde logique (dump).

Si elle est plus courte, le service doit très vite remonter. Cela nécessite des procédures avec un minimum d'acteurs et de manipulation :

- réplication ;
- solutions HA (Haute Disponibilité).

Plus le besoin en RTO/RPO sera court, plus les solutions seront complexes à mettre en œuvre — et chères. Inversement, pour des données non critiques, un RTO/RPO long permet d'utiliser des solutions simples et peu coûteuses.

[3] https://fr.wikipedia.org/wiki/Perte_de_donn%C3%A9es_maximale_admissible
[4] https://fr.wikipedia.org/wiki/Dur%C3%A9e_maximale_d%27interruption_admissible#RTO

6.2.4 Industrialisation

- Évaluer les coûts humains et matériels
- Intégrer les méthodes de sauvegardes avec le reste du SI
  - sauvegarde sur bande centrale
  - supervision
  - plan de continuité et de reprise d'activité

Les moyens nécessaires pour la mise en place, le maintien et l'intégration de la sauvegarde dans le SI ont un coût financier qui apporte une contrainte supplémentaire sur la politique de sauvegarde.

Du point de vue matériel, il faut disposer principalement d'un volume de stockage qui peut devenir conséquent. Cela dépend de la volumétrie à sauvegarder, il faut considérer les besoins suivants :

- Stocker plusieurs sauvegardes. Même avec une rétention d'une sauvegarde, il faut pouvoir stocker la suivante durant sa création : on ne doit purger les anciennes sauvegardes une fois qu'on est sûr que la sauvegarde s'est correctement déroulée.
- Avoir suffisamment de place pour restaurer sans avoir besoin de supprimer la base ou l'instance en production. Un tel espace de travail est également intéressant pour réaliser des restaurations partielles. Cet espace peut être mutualisé. On peut utiliser également le serveur de préproduction s'il dispose de la place suffisante.

Avec une rétention d'une sauvegarde unique, il est bon de prévoir 3 fois la taille de la base ou de l'instance. Pour une faible volumétrie, cela ne pose pas de problèmes, mais quand la volumétrie devient de l'ordre du téraoctet, les coûts augmentent significativement.

L'autre poste de coût est la mise en place de la sauvegarde. Une équipe de DBA peut tout à fait décider de créer ses propres scripts de sauvegarde et restauration, pour diverses raisons, notamment :

- maîtrise complète de la sauvegarde, maintien plus aisé du code ;
- intégration avec les moyens de sauvegardes communs au SI (bandes, externalisation…) ;
- adaptation au PRA/PCA plus fine.

Enfin, le dernier poste de coût est la maintenance, à la fois des scripts et par le test régulier de la restauration.

6.2.5 Documentation

- Documenter les éléments clés de la politique :
  - perte de données
  - rétention
  - durée de restauration
- Documenter les processus de sauvegarde et restauration
- Imposer des révisions régulières des procédures

Comme pour n'importe quelle procédure, il est impératif de documenter la politique de sauvegarde, les procédures de sauvegarde et de restauration ainsi que les scripts.

Au strict minimum, la documentation doit permettre à un DBA non familier de l'environnement de comprendre la sauvegarde, retrouver les fichiers et restaurer les données le cas échéant, le plus rapidement possible et sans laisser de doute. En effet, en cas d'avarie nécessitant une restauration, le service aux utilisateurs finaux est généralement coupé, ce qui génère un climat de pression propice aux erreurs qui ne fait qu'empirer la situation.

L'idéal est de réviser la documentation régulièrement en accompagnant ces révisions de tests de restauration : avoir un ordre de grandeur de la durée d'une restauration est primordial. On demandera toujours au DBA qui restaure une base ou une instance combien de temps cela va prendre.

6.2.6 Règle 3-2-1

- 3 exemplaires des données
- 2 sur différents médias
- 1 hors site (et hors ligne)
- Un RAID n'est pas une sauvegarde !
- Le *cloud* n'est pas une solution magique !

L'un des points les plus importants à prendre en compte est l'endroit où sont stockés les fichiers des sauvegardes. Laisser les sauvegardes sur la même machine n'est pas suffisant : si une défaillance matérielle se produisait, les sauvegardes pourraient être perdues en même temps que l'instance sauvegardée, rendant ainsi la restauration impossible.

Il est conseillé de suivre au moins la règle 3-2-1. Les données elles-mêmes sont le premier exemplaire.

Les deux copies doivent se trouver sur des supports physiques différents (et de préférence sur un autre serveur) pour parer à la destruction du support original (notamment une perte de disques durs). La première copie peut être à proximité pour faciliter une restauration.

Des disques en RAID ne sont pas une sauvegarde ! Ils peuvent parer à la défaillance d'un disque, pas à une fausse manipulation (`rm -rf /` ou `TRUNCATE` malheureux). La perte de la carte contrôleur peut entraîner la perte de toute la grappe.
Un conseil courant est de choisir des disques de séries différentes pour éviter des défaillances simultanées.

Le troisième exemplaire doit se trouver à un autre endroit, pour parer aux scénarios les plus catastrophiques (cambriolage, incendie…). Selon la criticité, le délai nécessaire pour remonter rapidement un système fonctionnel, et le budget, ce troisième exemplaire peut être une copie manuelle sur un disque externe stocké dans un coffre, ou une infrastructure répliquée complète avec sa copie des sauvegardes à l'autre bout de la ville, voire du pays.

Pour limiter la consommation d'espace disque des copies multiples, les durées de rétention peuvent différer. La dernière sauvegarde en date peut résider sur la machine, les cinq dernières sur un serveur distant, et une autre sur des bandes déposées dans un site sécurisé tous les mois.

Stocker vos données dans le *cloud* n'est pas une solution miracle. Un datacenter peut brûler entièrement[5]. Dans la formule choisie, il faut donc bien vérifier que le fournisseur sauvegarde les données sur un autre site. Ce n'est pas forcément suffisant : en 2024, Google Cloud a effacé par erreur tout le cloud privé (deux zones) de l'assureur australien UniSuper[6], qui n'a pu remonter son infrastructure que grâce à des sauvegardes *hors* de ce cloud.

N'hésitez donc pas à faire vous-même une copie de vos données, sur un autre site ou dans un autre cloud.

Il ne faut pas non-plus sous-estimer le risque d'une attaque (piratage, malveillance ou *ransomware*…), qui s'en prendra aussi à toute sauvegarde accessible en ligne. En 2023 l'hébergeur danois CloudNordic a perdu toutes les données de ses clients à cause d'un ransomware[7].

Une copie physique hors ligne est donc chaudement recommandée pour les données les plus critiques.

[5] https://dali.bo/20210310-ovh-incendie-strasbourg
[6] https://youtu.be/3GOAUyipnM4
[7] https://dcmag.fr/cloudnordic-le-datacenter-qui-a-perdu-toutes-les-donnees-de-ses-clients-lors-dune-attaque-par-ransomware/

6.2.7 Fichiers de configuration

- Sauvegarder les fichiers de configuration
- Et vos scripts
  - paramétrage
  - sauvegarde
  - maintenance

La sauvegarde ne concerne pas uniquement les données. Il est également fortement conseillé de sauvegarder les fichiers de configuration du serveur et les scripts d'administration. Ces scripts sont notamment ceux de maintenance pour diverses opérations (techniques ou fonctionnelles), ou ceux de sauvegarde eux-même.

Le paramétrage est parfois géré par l'outil d'industrialisation (Ansible, Docker Compose…), par un outil externe (Patroni…), ou versionné quelque part. L'idéal est copier les fichiers avec les sauvegardes. On peut parfois inclure ces scripts dans une sauvegarde au niveau système, vu que ce sont de simples fichiers.

Les principaux fichiers de PostgreSQL à prendre en compte sont : `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`. Ils sont parfois dans le PGDATA avec les données, parfois pas. Ils ont des clauses d'inclusion pour utiliser d'autres fichiers. Cette liste n'est en aucun cas exhaustive.

Il s'agit donc de recenser l'ensemble des fichiers et scripts nécessaires si l'on désirait recréer le serveur depuis zéro. Il faut prévoir le pire des cas, où l'infrastructure a disparu aussi.

6.2.8 Tester la restauration

- De nombreuses catastrophes auraient pu être évitées avec un test
- Validation de la procédure
- Estimation de la durée

Même si les sauvegardes se déroulent correctement, il est indispensable de tester la restauration, et de vérifier qu'elle se déroule sans erreur. Une erreur de copie lors de l'externalisation peut, par exemple, rendre la sauvegarde inutilisable.

> *Just that backup tapes are seen to move, or backup scripts are run for a lengthy period of time, should not be construed as verifying that data backups are properly being performed.*
>
> Que l'on voit bouger les bandes de sauvegardes, ou que les scripts de sauvegarde fonctionnent pendant une longue période, ne doit pas être interprété comme une validation que les sauvegardes sont faites.
>
> (NASA.gov, *Lessons learned* #1781, https://llis.nasa.gov/lesson/1781)

Le test de restauration permet de vérifier l'ensemble de la procédure :

- ensemble des objets sauvegardés ;
- intégrité de la copie ;
- liste et ordre des commandes à passer pour une restauration complète.

La rejouer régulièrement vous évitera de découvrir la procédure dans l'urgence, le stress, voire la panique, alors que vous serez harcelé par de nombreux utilisateurs ou clients bloqués. Ce stress peut vous faire faire des erreurs.

Le test permet aussi de connaître la durée de restauration, une information toujours utile. Si elle est trop importante, il faudra peut-être revoir ou optimiser la méthode de sauvegarde.

Pour se faire la main, restaurer régulièrement les bases de test ou de préproduction à partir des sauvegardes de la production est une bonne idée. Il est conseillé que développeurs et testeurs aient des données aussi proches que possible de la production à disposition.

Nous rencontrons régulièrement en clientèle des scripts de sauvegarde qui ne fonctionnent pas, et jamais testés. Vous trouverez sur Internet de nombreuses histoires de catastrophes qui auraient été évitées par un simple test. Entre mille autres :

- une base disparaît à l'arrêt et ses sauvegardes sont vides[8] : erreur de manipulation, script ne vérifiant pas qu'il a bien fait sa tâche, monitoring défaillant ;
- perte de 6 heures de données chez Gitlab en 2017[9] : procédures de sauvegarde et de réplication incomplètes, complexes et peu claires, outils mal choisis ou mal connus, sauvegardes vides et jamais testées, distraction d'un opérateur — Gitlab a le mérite d'avoir tout soigneusement documenté[10].

Voir aussi :

- *Sauvegarder c'est bien, restaurer c'est mieux* (Blog Dalibo, 2024)[11]

[8] http://www.pgdba.org/post/restoring_data/#a-database-horror-story
[9] https://about.gitlab.com/2017/02/01/gitlab-dot-com-database-incident/
[10] https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/
[11] https://blog.dalibo.com/2024/03/29/world_backup_day_restauration.html

6.3 CONCLUSION

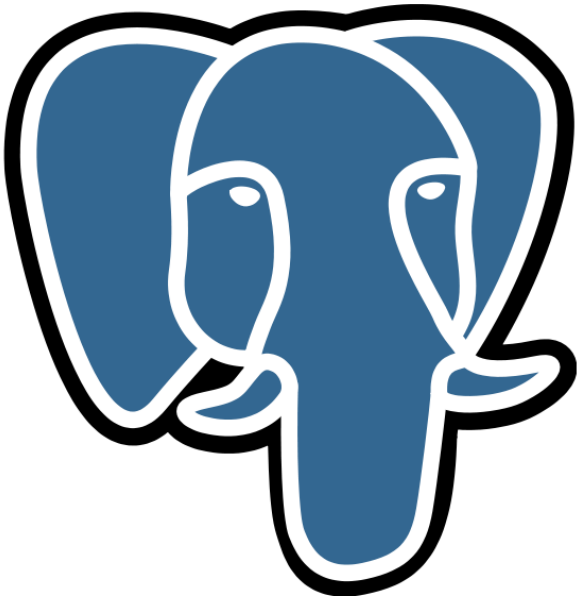
- Les techniques de sauvegarde de PostgreSQL sont :
  - complémentaires
  - automatisables
- La maîtrise de ces techniques est indispensable pour assurer un service fiable.
- **Testez vos sauvegardes !**

L'écosystème de PostgreSQL offre tout le nécessaire pour effectuer des sauvegardes fiables. Le plan de sauvegarde doit être fait sérieusement, et les sauvegardes testées. Cela a un coût, mais un désastre détruisant toutes vos données sera incommensurablement plus ruineux.

6.4 QUIZ

https://dali.bo/i0_quiz

7/ PostgreSQL : Sauvegarde et restauration



7.1 INTRODUCTION

Les sauvegardes sont essentielles pour la sécurisation des données :

- PostgreSQL propose différentes solutions
  - de sauvegarde à froid ou à chaud, mais cohérentes
  - des méthodes de restauration partielle ou complète

La mise en place d'une solution de sauvegarde est une des opérations les plus importantes après avoir installé un serveur PostgreSQL. En effet, nul n'est à l'abri d'un bogue logiciel, d'une panne matérielle, voire d'une erreur humaine.

Cette opération est néanmoins plus complexe qu'une sauvegarde standard car elle doit pouvoir s'adapter aux besoins des utilisateurs. Quand le serveur ne peut jamais être arrêté, la sauvegarde à froid des fichiers ne peut convenir. Il faudra passer dans ce cas par un outil qui pourra sauvegarder les données alors que les utilisateurs travaillent et qui devra respecter les contraintes ACID pour fournir une sauvegarde cohérente des données.

PostgreSQL va donc proposer des méthodes de sauvegardes à froid (autrement dit serveur arrêté) comme à chaud, mais de toute façon cohérentes. Les sauvegardes pourront être partielles ou complètes, suivant le besoin des utilisateurs.

La méthode de sauvegarde dictera l'outil de restauration. Suivant l'outil, il fonctionnera à froid ou à chaud, et permettra même dans certains cas de faire une restauration partielle.

7.1.1 Au menu

- Sauvegardes logiques
  - et restauration
- Sauvegarde physique à froid des fichiers
- Sauvegardes à chaud

7.2 SAUVEGARDES LOGIQUES

7.2.1 Sauvegardes logiques : caractéristiques

- À chaud
- Cohérente
- Locale ou à distance
- 2 outils
  - `pg_dump`
  - `pg_dumpall`
- Pas d'impact sur les utilisateurs
  - sauf certaines opérations DDL
- Jamais inclus :
  - tables systèmes
  - fichiers de configuration

La sauvegarde logique nécessite que le serveur soit en cours d'exécution. Un outil se connecte à la base et récupère la déclaration des différents objets ainsi que les données des tables.

La technique alors utilisée permet de s'assurer de la cohérence des données : lors de la sauvegarde, l'outil ne voit pas les modifications faites par les autres utilisateurs. Pour cela, quand il se connecte à la base à sauvegarder, il commence une transaction pour que sa vision des enregistrements de l'ensemble des tables soit cohérente. Cela empêche le recyclage des enregistrements par `VACUUM` pour les enregistrements dont il pourrait avoir besoin. Par conséquent, que la sauvegarde dure 10 minutes ou 10 heures, le résultat correspondra au contenu de la base telle qu'elle était au début de la transaction.

Des verrous sont placés sur chaque table, mais leur niveau est très faible (*Access Share*). Il visent juste à éviter la suppression des tables pendant la sauvegarde, ou la modification de leur structure. Les opérations habituelles sont toutes permises en lecture ou écriture, sauf quand elles réclament un verrou très invasif, comme `TRUNCATE`, `VACUUM FULL` ou certains `LOCK TABLE`. Les verrous ne sont relâchés qu'à la fin de la sauvegarde.

Par ailleurs, pour assurer une vision cohérente de la base durant toute la durée de son export, cette transaction de longue durée est de type *REPEATABLE READ*, et non de type *READ COMMITED*, celui utilisé par défaut.

Il existe deux outils pour la sauvegarde logique dans la distribution officielle de PostgreSQL :

- `pg_dump`, pour sauvegarder uniquement des bases de l'instance, complètement ou partiellement, avec de nombreuses options et formats ;
- `pg_dumpall` pour sauvegarder toutes les définitions et données des bases en un seul script SQL, ainsi que les objets globaux (rôles, tablespaces).

Pour la restauration :

- `psql` exécute les ordres SQL contenus dans des *dumps* (sauvegardes) au format texte ;
- `pg_restore` traite uniquement les *dumps* au format binaire, et produit le SQL qui permet de restaurer les données.

Il est important de bien comprendre que ces outils n'échappent pas au fonctionnement client-serveur de PostgreSQL. Ils « dialoguent » avec l'instance PostgreSQL uniquement en SQL, aussi bien pour la sauvegarde que la restauration.

Comme ce type d'outil n'a besoin que d'une connexion standard à la base de données, il peut se connecter en local comme à distance. Cela implique qu'il doive aussi respecter les autorisations de connexion configurées dans le fichier `pg_hba.conf`.

L'export ne concerne que les données des utilisateurs : les tables systèmes ne sont jamais concernées, il n'y a pas de risque de les écraser lors d'un import. En effet, les schémas systèmes `pg_catalog` et `information_schema` et leurs objets sont gérés uniquement par PostgreSQL. Vous n'êtes d'ailleurs pas censé modifier leur contenu, ni y ajouter ou y effacer quoi que ce soit !

La configuration du serveur (fichiers `postgresql.conf`, `pg_hba.conf`…) n'est jamais incluse et doit être sauvegardée à part. Un export logique ne concerne que des données et structures.

Les extensions ne posent pas de problème non plus : la sauvegarde contiendra une mention de l'extension, et les *données* des éventuelles tables gérées par cette extension. Mais à la restauration, il faudra que les binaires de l'extension soient installés sur le système cible.

7.2.2 pg_dump

- Sauvegarde une base de données :

```
pg_dump nombase > nombase.dump
```

- Sauvegarde complète ou partielle

`pg_dump` est l'outil le plus utilisé pour sauvegarder une base de données PostgreSQL. Une sauvegarde peut se faire de façon très simple. Par exemple :

```
$ pg_dump b1 > b1.dump
```

sauvegardera la base **b1** de l'instance locale sur le port 5432 dans un fichier `b1.dump`.

Sous Windows avec le Powershell, préférer la syntaxe `pg_dump -f b1.dump b1`, car une redirection avec `>` peut corrompre la sauvegarde.

Mais `pg_dump` permet d'aller bien plus loin que la sauvegarde d'une base de données complète. Il existe pour cela de nombreuses options en ligne de commande.

7.2.3 pg_dump : formats de sauvegarde

Format	Dump	Restore
plain (SQL)	`pg_dump -Fp` ou `pg_dumpall`	`psql`
tar	`pg_dump -Ft`	`pg_restore`
custom	`pg_dump -Fc`	`pg_restore`
directory	`pg_dump -Fd`	`pg_restore`

Un élément important est le format des données extraites. Selon l'outil de sauvegarde utilisé et les options de la commande, l'outil de restauration diffère. Le tableau indique les outils compatibles selon le format choisi.

`pg_dump` accepte d'enregistrer la sauvegarde suivant quatre formats :

- le format SQL, soit un fichier texte unique pour toute la base, non compressé ;

- le format tar, un fichier binaire, non compressé, comprenant un index des objets ;
- le format « personnalisé » (*custom*), un fichier binaire, compressé, avec un index des objets ;
- le format « répertoire » (*directory*), arborescence de fichiers binaires généralement compressés, comprenant aussi un index des objets.

Pour choisir le format, il faut utiliser l'option `--format` (ou `-F`) et le faire suivre par le nom ou le caractère indiquant le format sélectionné :

- `plain` ou `p` pour un fichier SQL (texte) ;
- `tar` ou `t` pour un fichier tar ;
- `custom` ou `c` pour un fichier « personnalisé » ;
- `directory` ou `d` pour le répertoire.

Le format `plain` est lisible directement par psql. Les autres nécessitent de passer par `pg_restore` pour restaurer tout ou partie de la sauvegarde.

Le fichier SQL (`plain`) est naturellement lisible par n'importe quel éditeur texte. Le fichier texte est divisé en plusieurs parties :

- configuration de certaines variables ;
- création des objets de la base : schémas, tables, vues, procédures stockées, etc., à l'exception des index, contraintes et triggers ;
- ajout des données aux tables (ordres `COPY` par défaut) ;
- ajout des index, contraintes et triggers ;
- définition des droits d'accès aux objets ;
- rafraîchissement des vues matérialisées.

Les index figurent vers la fin pour des raisons de performance : il est plus rapide de créer un index à partir des données finales que de le mettre à jour en permanence pendant l'ajout des données.

Les contraintes et le rafraîchissement des vues matérialisées sont aussi à la fin parce qu'il faut que les données soient déjà restaurées dans leur ensemble.

Les triggers ne devant pas être déclenchés pendant la restauration, ils sont aussi restaurés vers la fin.

Les propriétaires et droits sont restaurés sur chacun des objets après création.

Voici un exemple de sauvegarde d'une base de 2 Go pour chaque format :

```
$ time pg_dump -Fp b1 > b1.Fp.dump
real  0m33.523s
user  0m10.990s
sys   0m1.625s

$ time pg_dump -Ft b1 > b1.Ft.dump
real  0m37.478s
user  0m10.387s
sys   0m2.285s

$ time pg_dump -Fc b1 > b1.Fc.dump
real  0m41.070s
```

```
user  0m34.368s
sys   0m0.791s

$ time pg_dump -Fd -f b1.Fd.dump b1
real  0m38.085s
user  0m30.674s
sys   0m0.650s
```

La sauvegarde la plus longue est la sauvegarde au format personnalisé (`custom`) car elle est compressée.

La sauvegarde au format répertoire se trouve entre la sauvegarde au format personnalisée et la sauvegarde au format `tar` : elle est aussi compressée mais sur des fichiers plus petits.

En terme de taille, la comparaison donne :

```
$ du -sh b1.F?.dump
116M    b1.Fc.dump
116M    b1.Fd.dump
379M    b1.Fp.dump
379M    b1.Ft.dump
```

Le format compressé est évidemment le plus petit. Le format texte et le format tar[1] sont les plus lourds à cause du manque de compression. Le format `tar` est même généralement un peu plus lourd que le format texte à cause de l'entête des fichiers tar. Évidemment, le taux de compression dépendra beaucoup des données.

7.2.4 pg_dump : choisir le format de sauvegarde

- Format `plain` (SQL)
  - restaurations partielles très difficiles (ou manuelles)
- Parallélisation de la sauvegarde :
  - uniquement format `directory`
- Préférer les formats binaires

```
pg_dump -Fc
pg_dump -Fd
```

- Ajouter les objets globaux (rôles…)

```
pg_dumpall -g
```

[1] https://fr.wikipedia.org/wiki/Tar_(informatique)

Il convient de bien appréhender les limites de chaque outil de dump et des formats.

Tout d'abord, le format `tar` est à éviter. Il n'apporte aucune plus-value par rapport au format `custom`.

Ensuite, même si c'est le plus portable (et le seul disponible avec `pg_dumpall`), le format `plain` rend les restaurations partielles difficiles car il faut extraire manuellement le SQL d'un fichier texte souvent très volumineux. Ce peut être ponctuellement pratique cependant, mais `pg_restore` régénère facilement un fichier SQL complet à partir d'une sauvegarde binaire.

Certaines informations (notamment les commandes `ALTER DATABASE … SET` pour modifier un paramètre pour une base précise) ne sont pas incluses dans le format `plain`, à moins de penser à rajouter `--create` (pour les ordres de création).
Par contre, elles sont incluses dans les entêtes des formats `custom` ou `directory`, où un `pg_restore --create` saura les retrouver.

On privilégiera donc les formats `custom` et `directory` pour plus de flexibilité à la restauration.

Le format `directory` a l'énorme intérêt de permettre la sauvegarde en parallèle de `pg_dump --jobs`, avec de gros gains de temps de sauvegarde (ou de migration) à la clé.

Enfin, l'outil `pg_dumpall`, initialement prévu pour les montées de versions majeures, permet en fait de sauvegarder les objets globaux d'une instance.

Ainsi, pour avoir la sauvegarde la plus complète possible d'une instance, il faut combiner `pg_dumpall -g` (pour la définition des objets globaux), et `pg_dump` (pour sauvegarder les bases de données une par une au format `custom` ou `directory`).

7.2.5 pg_dump : compression

- `-z/--compress` : compression par zlib
  - de 0 à 9 (défaut : 6)
- (v16) mieux : `zstd`, `lz4`

```
pg_dump -Z1  # gzip
# v16+
pg_dump -Z gzip:3
pg_dump -Z lz4:'level=12'
pg_dump -Z zstd:'level=22,long'
```

La compression permet de réduire énormément la volumétrie d'une sauvegarde logique par rapport à la taille physique des fichiers de données.

Par défaut, `pg_dump -Fc` ou `-Fd` utilise le niveau de compression par défaut de la zlib, à priori le meilleur compromis entre compression et vitesse, correspondant à la valeur 6. `-Z1` comprimera peu mais rapidement, et `-Z9` sera nettement plus lent mais compressera au maximum. Seuls des tests permettent de déterminer le niveau acceptable pour un cas d'utilisation particulier.

Depuis PostgreSQL 16, des algorithmes de compression plus modernes sont disponibles : `zstd` et `lz4` sont généralement préférables. Une nouvelle syntaxe de l'option `-Z` permet de choisir l'algorithme, le niveau de compression (1 à 22 pour `zstd`, 1 à 12 pour `lz4`), éventuellement d'autres paramètres propres à l'algorithme (comme le mode `long` de `zstd`, plus consommateur en RAM). `lz4` a la réputation d'être très rapide mais bien moins bon en compression. Cependant, le choix du bon algorithme doit se faire en testant sur vos données réelles sur votre machine avec vos contraintes en CPU, RAM, place disque et temps.

Le format `plain` (pur texte) accepte aussi l'option `-Z`, ce qui permet d'obtenir un export texte compressé. Cependant, cela ne remplace pas complètement un format `custom`, plus souple.

Même les nouveaux algorithmes sont par défaut monothread. Si l'on veut paralléliser la compression sur plusieurs processeurs, nous verrons plus bas qu'il faut soit paralléliser `pg_dump` avec `--jobs`, soit compresser la sortie *plain* avec un autre outil.

7.2.6 pg_dump : fichier ou sortie standard

- `-f` : fichier/répertoire où stocker la sauvegarde
- sinon : sortie standard

Par défaut, et en dehors du format répertoire, toutes les données d'une sauvegarde sont renvoyées sur la sortie standard de `pg_dump`. Il faut donc utiliser une redirection pour renvoyer dans un fichier.

Cependant, il est aussi possible d'utiliser l'option `-f` pour spécifier le fichier (ou répertoire pour le format `directory`) de la sauvegarde. L'utilisation de cette option est conseillée car elle permet à `pg_restore` de trouver plus efficacement les objets à restaurer dans le cadre d'une restauration partielle.

7.2.7 pg_dump : structure ou données

- `--schema-only` / `-s` : uniquement la structure
- `--data-only` / `-a` : uniquement les données ou :
- `--section`
  - `pre-data` : définition des objets (hors contraintes et index)
  - `data` : les données
  - `post-data` : définition des contraintes et index

Il est possible de ne sauvegarder que la structure avec l'option `--schema-only` (ou `-s`). De cette façon, seules les requêtes de création d'objets seront générées. Cette sauvegarde est généralement très rapide. Cela permet de créer un serveur de tests très facilement. Dans ce cas, le format `plain` suffit, pour obtenir une suite de commandes SQL.

Il est aussi possible de ne sauvegarder que les données pour les réinjecter dans une base préalablement créée avec l'option `--data-only` (ou `-a`).

Il est possible de sauvegarder plus finement, par section. En fait, un fichier de sauvegarde complet est composé de trois sections :

- la définition des objets ;
- les données ;

- la définition des contraintes et index.

Il est parfois intéressant de sauvegarder par section plutôt que de sauvegarder schéma et données séparément, car cela permet d'avoir la partie contrainte et index dans un script à part. Cela peut aider à optimiser le temps de certaines restaurations complexes.

7.2.8 pg_dump : sélection d'objets

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf celui-là
- `-t <table>` : uniquement cette relation (sans dépendances !)
- `-T <table>` : toutes les tables sauf celle-là
- En option
  - possibilité d'en mettre plusieurs
  - exclure les données : `--exclude-table-data=<table>`
  - avoir une erreur si l'objet est inconnu : `--strict-names`
- `--filter` & fichier avec `include` / `exclude` des objets (v17)
- Attention : pas de dépendance automatique si sélection !

Sélection objet par objet :

En dehors de la distinction structure/données, il est possible de demander de ne sauvegarder qu'un objet. Les seuls objets sélectionnables au niveau de `pg_dump` sont les tables et les schémas d'une base.

- L'option `--schema` (ou `-n`) permet de sauvegarder seulement le schéma cité après alors que l'option `--exclude-schema` (ou `-N`) permet de sauvegarder tous les schémas sauf celui cité après l'option.
- Le même système existe pour les tables avec les options `--table` / `-t` et `--exclude-table` / `-T`.

Ni la stucture ni les données des tables ou schémas concernés ne figureront dans la sauvegarde.

Ignorer le contenu de plusieurs tables ou schémas est possible en répétant l'option ou avec des patterns[2] (motifs). Par exemple, `-T '(log|temp)_'` excluera toutes les tables dont le nom commence par `log_` ou `temp_`.

[2] https://docs.postgresql.fr/current/app-psql.html#APP-PSQL-PATTERNS

Les dépendances ne sont pas gérées. Si vous demandez à sauvegarder une vue avec `pg_dump -t unevue`, la sauvegarde ne contiendra **pas** les définitions des objets nécessaires à la construction de cette vue. Toute sauvegarde partielle possède ce risque d'incohérence.

Par défaut, si certains objets sont trouvés et d'autres non, `pg_dump` ne dit rien, et l'opération se termine avec succès. Ajouter l'option `--strict-names` permet de s'assurer d'être averti avec une erreur sur le fait que `pg_dump` n'a pas sauvegardé tous les objets souhaités. En voici un exemple (`t1` existe, `t20` n'existe pas) :

```
$ pg_dump -t t1 -t t20 -f postgres.dump postgres
$ echo $?
0
$ pg_dump -t t1 -t t20 --strict-names -f postgres.dump postgres
pg_dump: no matching tables were found for pattern "t20"
$ echo $?
1
```

Exclure les données :

Pour sauvegarder les données de la base, sauf celles d'une table donnée, tout en conservant la structure de celle-ci, utiliser l'option `--exclude-table-data=nomtable`. Elle peut servir pour ignorer le contenu d'une table de trace ou d'une table de travail, par exemple. Ignorer le contenu de plusieurs tables est possible en répétant l'option ou avec des patterns[3] (motifs) : `--exclude-table-data="(log|temp)_*"`

Fichiers de filtrage :

À partir de PostgreSQL 17, une option plus puissante apparaît : `--filter`. Elle reprend sous une autre forme le mécanisme d'inclusion/exclusion en ligne de commande.

`--filter` est utilisable aussi à la restauration avec `pg_restore`.

Voici un exemple de fichier de filtrage, nommé ici `filter1.txt` :

```
# ceci est un commentaire

include table t1?
exclude table t12
```

L'appel se fait ainsi :

```
pg_dump --filter filter1.txt b1
```

[3] https://docs.postgresql.fr/current/app-psql.html#APP-PSQL-PATTERNS

Seront ainsi sauvegardées toutes les tables dont le nom correspond à l'expression `t1?` à l'exception de la table `t12`.

Le premier champ indique si l'on exclue ou inclue des objets. Si une ligne `include` est présente, les objets *non* inclus ne seront pas sauvés. S'il n'y a pas de ligne `include`, tous les objets seront par défaut sauvegardés, sauf exclusion explicite.

L'objet en seconde colonne indique le type d'objet et vaut, au choix :

- `schema`, `table` ;
- `table_data` (pour les données de table, exclusion uniquement) ;
- `extension`, `foreign_data` (tables étrangères, inclusion uniquement) ;
- `table_and_children`, `table_data_and_children` (variantes des options `table` et `table_data` pour les tables partitionnées)

Le troisième champ est un motif pour la sélection des objets.

Il peut y avoir plusieurs fichiers de filtrage. `--filter -` lit les filtres depuis l'entrée standard (`STDIN`).

Des exemples de ces options sont aussi disponibles dans la documentation officielle[4].

7.2.9 pg_dump : parallélisation

- `--jobs <nombre_de_threads>`
- Format directory (`-Fd`) uniquement
- Granularité : la table
  - optimal quand la volumétrie est bien répartie

Par défaut, `pg_dump` n'utilise qu'une seule connexion à la base de données pour sauvegarder la définition des objets et les données. Cependant, une fois que la première étape de récupération de la définition des objets est réalisée, l'étape de sauvegarde des données peut être parallélisée pour profiter des nombreux processeurs disponibles sur un serveur. La compression est en effet souvent le facteur limitant d'une sauvegarde.

Cette option n'est compatible qu'avec le format de sortie `directory` (option `-Fd`).

[4] https://docs.postgresql.fr/current/app-pgdump.html#PG-DUMP-EXAMPLES

La parallélisation de requêtes ne s'applique hélas pas aux ordres `COPY` utilisés par `pg_dump`, qui concernent la table à sauvegarder entière. Mais `pg_dump` peut lancer plusieurs sessions simultanément avec l'option `--jobs` (`-j`). Elle permet de préciser le nombre de connexions vers la base de données, et aussi de connexions (Unix) ou *threads* (Windows). Cela permet d'améliorer considérablement la vitesse de sauvegarde.

Une limite est qu'il faut pouvoir réellement paralléliser. Par exemple, si une table occupe 15 Go sur une base de données de 20 Go, il y a peu de chance que la parallélisation de `pg_dump` change fondamentalement la durée de sauvegarde. Par contre, la parallélisation profitera du partitionnement de cette table.

7.2.10 pg_dump : objets binaires

- Deux types dans PostgreSQL :
  - bytea
  - Large Objects
- Option `--large-objects`
  - si filtrage (`-n` / `-N` / `-t` / `-T` / `--filter`)
- Option `--no-blobs`
  - pour ne pas sauvegarder les Large Objects
- Option `bytea_output`
  - `escape`
  - `hex`

Il existe deux types d'objets binaires dans PostgreSQL : les bytea[5] et les Large Objects[6].

bytea :

`bytea` est un type de données tout à fait classique, stocké dans les tables utilisateur.

Il est possible d'indiquer le format de sortie des données binaires, grâce au paramètre `bytea_output` qui se trouve dans le fichier `postgresql.conf`. Le défaut est `hex`. Le format `hex` utilise deux octets pour enregistrer un octet binaire de la base alors que le format `escape` utilise un nombre variable

[5] https://dali.bo/s9_html#bytea
[6] https://dali.bo/s9_html#large-object

d'octets. Dans le cas de données ASCII, ce format n'utilisera qu'un octet. Dans les autres cas, il en utilisera quatre pour afficher textuellement la valeur octale de la donnée (un caractère d'échappement suivi des trois caractères du codage octal). La taille de la sauvegarde s'en ressent, sa durée de création aussi (surtout en activant la compression).

Si vous restaurez une sauvegarde d'une base antérieure à la 9.0, notez que la valeur par défaut était différente (`escape`).

Large Objects :

Les Larges Objects ne sont pas des types de données habituels. L'utilisation en est découragée.

Ces objets sont stockés dans une table système appelé `pg_largeobjects`, et non pas dans les tables utilisateurs. Du coup, en cas d'utilisation des options `-n` / `-N` et/ou `-t` / `-T`, la table système contenant les Large Objects sera généralement exclue. Pour être sûr que les Large Objects soient inclus, il faut en plus ajouter l'option `--large-objects` (`-b`).

Il arrive que des applications oublient de purger les *Large Objects* inutilisés (un des défauts de ce type est que la purge n'est pas automatique). `pg_dump` peut alors durer bien plus longtemps, et consommer beaucoup de mémoire.

7.2.11 pg_dump : extensions

- `--extension` (`-e`)
  - uniquement si sélection/exclusion (`-n` / `-N` et/ou `-t` / `-T`)
- `--exclude-extension` (v17)

Par défaut, les extensions sont sauvegardées. Rappelons que la sauvegarde contiendra généralement une simple mention de l'extension (`CREATE EXTENSION`). Toutefois, l'extension peut indiquer que ses propres tables système doivent être sauvegardées.

Cependant, dans le cas où les options de sélection/exclusion de schéma (`-n` / `-N`) ou de tables (`-t` / `-T`) sont utilisées, les extensions ne sont pas sauvegardées. Or, elles pourraient être nécessaires pour les schémas et tables sélectionnées. L'option `-e` permet de forcer la sauvegarde des extensions précisées.

À l'inverse, depuis PostgreSQL 17, il est possible de ne pas sauvegarder des extensions d'une base :

```
# exclure toute les extensions dont le nom commence par 'pg'
pg_dump -U postgres -Fc --exclude-extension=pg*
```

7.2.12 pg_dump : options diverses

- `--create` (`-C`) : recréer la base
  - y compris paramétrage utilisateur sur base (format `plain`)
  - inutile dans les autres formats
- `--no-owner`, `--no-privileges`
- `--no-tablespaces`
- `--inserts` / `--rows-per-insert` : `INSERT` au lieu de `COPY`
- `--on-conflict-do-nothing`
- divers paramètres pour les tables partitionnées
- `-v` : progression

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques.

–create :

`--create` (ou `-C`) n'a d'intérêt qu'en format texte (`-Fp`), pour ajouter les instructions de création de base. (Avec les autres formats, il vaut mieux utiliser l'option équivalente lors de la restauration avec `pg_restore`. Le nom de la base est dans la sauvegarde.)

```
CREATE DATABASE b1 WITH TEMPLATE = template0
                   ENCODING = 'UTF8' LC_COLLATE = 'C' LC_CTYPE = 'C';
ALTER DATABASE b1 OWNER TO postgres;
```

et éventuellement, s'il y a un paramétrage propre à la base :

```
ALTER DATABASE b1 SET work_mem TO '100MB' ;
ALTER ROLE chef IN DATABASE b1 SET work_mem TO '1GB';
```

À partir de la version 11, ces dernières informations sont incluses d'office aux formats `custom` ou `directory`, où `pg_restore --create` peut les retrouver.
Jusqu'en version 10, elles ne se retrouvaient que dans `pg_dumpall` (sans `-g`), ce qui n'était pas pratique quand on ne restaurait qu'une base.
Dans tous les cas, il faut bien vérifier que votre procédure de restauration reprend ce paramétrage.

On peut rajouter `--clean` pour ajouter un `DROP DATABASE` préalable dans le script généré (option dangereuse !), et même `--clean --if-exists` pour éviter une erreur si la base cible n'existait en fait pas.

Masquer des droits et autres propriétés :

`--no-owner`, `--no-privileges`, `--no-comments` et `--no-tablespaces` permettent de ne pas récupérer respectivement le propriétaire, les droits, le commentaire et le tablespace de l'objet dans la sauvegarde s'ils posent problème.

Ordres INSERT au lieu de COPY :

Par défaut, `pg_dump` génère des commandes `COPY`, qui sont bien plus rapides que les `INSERT`. Cependant, notamment pour pouvoir restaurer plus facilement la sauvegarde sur un autre moteur de bases de données, il est possible d'utiliser des `INSERT` au lieu des `COPY`. Il faut forcer ce comportement avec l'option `--inserts`.

S'il s'agit de réimporter ces données dans PostgreSQL, un intérêt des `INSERT` est la flexibilité pour rejeter des lignes : l'option `--on-conflict-do-nothing` permet d'éviter des messages d'erreur si un `INSERT` tente d'insérer une ligne violant une contrainte existante. Très souvent, ce sera pour éviter des problèmes de doublons (même clé primaire) dans une table déjà partiellement chargée, ou des soucis de cohérence avec d'autres données (non respect de clés étrangères).

Par contre, l'inconvénient des `INSERT` ligne à ligne est la lenteur, par rapport à un `COPY` qui traite tout en bloc (même avec des astuces comme `synchronous_commit=off`). Pour diminuer cet inconvénient, il est possible d'insérer les lignes par paquets avec l'option `--rows-per-insert` : si un `INSERT` échoue, seulement ce nombre de lignes sera annulé.

(Avec PostgreSQL 17, `--inserts` devient un peu moins utile : la nouvelle syntaxe `COPY … WITH (ON_ERROR ignore)` permet de ne pas arrêter l'insertion, par exemple si un champ est trop grand. Par contre, une violation de contrainte, comme un doublon, continuera de mener à un rejet par `COPY` de toutes les lignes).

Tables partitionnées :

Autres options :

Il existe des options pour gérer l'export des tables partitionnées, que nous ne détaillerons pas ici : `--load-via-partition-root`, et depuis la version 16 : `--table-and-children`, `--exclude-table-and-children` et `--exclude-table-data-and-children`.

Enfin, l'option `-v` (ou `--verbose`) permet de voir la progression de la sauvegarde.

7.2.13 pg_dumpall

- Outil dédié à la sauvegarde d'une instance complète
  - objets globaux (utilisateurs, tablespaces…)
  - toutes les bases de données
  - sauf `--exclude-database`
- Format texte (SQL) uniquement (`--f nomfichier.dmp`)

`pg_dumpall` sauvegarde toute l'instance par défaut. Cet outil possède peu d'options de sélection. Il ne permet ni compression ni parallélisation.

`pg_dumpall` peut toutefois être intéressant ponctuellement. Il permet notamment la sauvegarde de la définition des objets globaux, c'est-à-dire les utilisateurs et les tablespaces, entre autres. Il est donc complémentaire de `pg_dump`, qui ne sauvegarde que la structure et les données d'une base de données précise, sans rôle ni tablespace.

S'il y a plusieurs bases et peu de volumétrie, reconstruire l'instance, rôles compris, est très simple avec `pg_dumpall` car il suffit de rejouer un seul fichier de sauvegarde, ou d'utiliser un *pipe* :

```
pg_dumpall -h ancienserveur | psql -h nouveauserveur
```

Contrairement à `pg_dump`, `pg_dumpall` ne dispose que d'un format en sortie : des ordres SQL en texte. La sauvegarde est envoyée sur la sortie standard, ou le fichier précisé avec l'option `--file` (`-f`).

En général, on effectuera la sauvegarde logique d'une instance avec un appel à `pg_dumpall -g` pour les objets globaux, et des appels à `pg_dump` pour chaque base, pour profiter de la parallélisation et de la compression des données.

Si l'on veut presque tout sauvegarder avec `pg_dumpall`, il existe une option `--exclude-database` pour ignorer une des bases, option qui peut être répétée ou contenir des motifs.

`pg_dumpall` a quelques options communes avec `pg_dump`, notamment les options de connexion, `--schema-only`, `--no-owner` ou `--no-subscriptions`. D'autres options communes, comme `--inserts`, `--rows-per-insert`, sont rarement utilisées avec `pg_dumpall`. À partir de PostgreSQL 17, l'option `--filter` peut indiquer un fichier de filtrage, qui ne peut servir qu'à exclure des objets de type `database`.

7.2.14 pg_dumpall : sélection des objets

- `-g` : tous les objets globaux (conseillé)
- `-r` : uniquement les rôles
- `-t` : uniquement les tablespaces
- `--no-role-passwords` : sans les mots de passe
  - permet de ne pas être superutilisateur

`pg_dumpall` étant créé pour sauvegarder l'instance complète, il ne dispose quasiment pas d'options de sélection d'objets. Le plus important est la sauvegarde en SQL :

- des rôles uniquement (`--roles-only` / `-r`) ;
- des tablespaces uniquement (`--tablespaces-only` / `-t`) ;
- de tous les objets globaux (`--globals-only` / `-g`), dont les rôles et tablespaces.

Par exemple, la sauvegarde des rôles renvoie ce script :

```
$ pg_dumpall -r
--
-- PostgreSQL database cluster dump
--

SET default_transaction_read_only = off;

SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;

--
-- Roles
--

CREATE ROLE admin;
ALTER ROLE admin WITH SUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN
                  NOREPLICATION NOBYPASSRLS;
CREATE ROLE dupont;
ALTER ROLE dupont WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                  NOREPLICATION NOBYPASSRLS
                  PASSWORD 'md5505548e69dafa281a5d676fe0dc7dc43';
CREATE ROLE durant;
ALTER ROLE durant WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                  NOREPLICATION NOBYPASSRLS
                  PASSWORD 'md56100ff994522dbc6e493faf0ee1b4f41';
CREATE ROLE martin;
ALTER ROLE martin WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN
                  NOREPLICATION NOBYPASSRLS
```

```
                 PASSWORD 'md5d27a5199d9be183ccf9368199e2b1119';
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
                 REPLICATION BYPASSRLS;
CREATE ROLE utilisateur;
ALTER ROLE utilisateur WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOLOGIN
                 NOREPLICATION NOBYPASSRLS;

--
-- User Configurations
--

--
-- User Config "u1"
--

ALTER ROLE u1 SET maintenance_work_mem TO '256MB';

--
-- Role memberships
--

GRANT admin TO dupont GRANTED BY postgres;
GRANT admin TO durant GRANTED BY postgres;
GRANT utilisateur TO martin GRANTED BY postgres;


--
-- PostgreSQL database cluster dump complete
--
```

On remarque que le mot de passe est sauvegardé sous forme de *hash*.

La sauvegarde des rôles se fait en lisant le catalogue système `pg_authid`. Seuls les superutilisateurs ont accès à ce catalogue système car il contient les mots de passe des utilisateurs. `pg_dumpall` peut s'utiliser sans rôle superutilisateur, avec l'option `--no-role-passwords`. Celle-ci a pour effet de ne pas sauvegarder les mots de passe. Dans ce cas, `pg_dumpall` va lire le catalogue système `pg_roles` qui est accessible par tout le monde.

7.2.15 pg_dump/pg_dumpall : options de connexions

- `-h` / `$PGHOST` (serveur ou socket Unix)
- `-p` / `$PGPORT` (5432)
- `-U` / `$PGUSER` (utilisateur du système)
- `-W` / `$PGPASSWORD`
  - ou `.pgpass`

Les commandes `pg_dump` et `pg_dumpall` se connectent au serveur PostgreSQL comme n'importe quel autre outil (psql, pgAdmin, etc.). Ils disposent donc des options habituelles pour se connecter :

- `-h` ou `--host` pour indiquer l'alias ou l'adresse IP du serveur ;
- `-p` ou `--port` pour préciser le numéro de port ;
- `-U` ou `--username` pour spécifier l'utilisateur ;
- `-W` ne permet pas de saisir le mot de passe en ligne de commande, mais force `pg_dump` à le demander (en interactif donc, et qu'il soit vérifié ou non, ceci dépendant de la méthode d'authentification).

En général, une sauvegarde automatique est effectuée sur le serveur directement par l'utilisateur système `postgres` (connexion par `peer` sans mot de passe), ou à distance avec le mot de passe stocké dans un fichier `.pgpass`.

Il est possible d'indiquer plusieurs hôtes et ports. C'est utile pour lancer une sauvegarde depuis une instance en réplication, ou à défaut depuis le primaire. L'hôte sélectionné est le premier qui répond au paquet de démarrage. Si l'authentication ne passe pas, la connexion sera en erreur. Il est aussi possible de préciser si la connexion doit se faire sur un serveur en lecture/écriture ou en lecture seule. Par exemple on effectuera une sauvegarde depuis le premier serveur disponible ainsi :

```
pg_dumpall -h secondaire,primaire -p 5432,5433 -U postgres -f sauvegarde.sql
```

Si la connexion nécessite un mot de passe, ce dernier sera réclamé lors de la connexion. Il faut donc faire attention avec `pg_dumpall`, qui se connecte à chaque base de données, une par une.

Il est préférable d'utiliser un fichier `.pgpass` qui indique les mots de passe de connexion. Ce fichier est créé à la racine du répertoire personnel de l'utilisateur qui exécute la sauvegarde. Il contient les informations suivantes :

```
hote:port:base:utilisateur:mot de passe
```

Ce fichier est sécurisé, dans le sens où seul l'utilisateur doit avoir le droit de lire et écrire ce fichier (c'est-à-dire des droits `600`). L'outil vérifie cela avant d'accepter d'utiliser les informations qui s'y trouvent.

7.2.16 pg_dump/pg_dumpall : privilèges nécessaires

- Les outils se comportent comme des clients pour PostgreSQL
- Préférer un rôle superutilisateur
- Sinon :
  - connexion à autoriser
  - le rôle doit pouvoir lire tous les objets à exporter (`pg_read_all_data`)

Même si ce n'est pas obligatoire, il est recommandé d'utiliser un rôle de connexion disposant des droits de superutilisateur pour la sauvegarde et la restauration. En effet, pour sauvegarder, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf`, être propriétaire ou avoir le privilège `CONNECT` ;
- voir le contenu des différents schémas : être propriétaire ou avoir le privilège `USAGE` sur le schéma ;
- lire le contenu des tables : être propriétaire ou avoir le privilège `SELECT` sur la table.

Pour restaurer, il faut pouvoir :

- se connecter à la base de données : autorisation dans `pg_hba.conf`, être propriétaire ou avoir le privilège `CONNECT` ;
- optionnellement, pouvoir créer la base de données cible et pouvoir s'y connecter (option `-C` de `pg_restore`)
- pouvoir créer des schémas : être propriétaire de la base de données ou avoir le privilège `CREATE` sur celle-ci ;
- pouvoir créer des objets dans les schémas : être propriétaire du schéma ou avoir le privilège `CREATE` sur celui-ci ;
- pouvoir écrire dans les tablespaces cibles : être propriétaire du tablespace ou avoir le privilège `CREATE` sur celui-ci ;
- avoir la capacité de donner ou retirer des privilèges : faire partie des rôles bénéficiant d'ACL dans le `dump`.

Le nombre de ces privilèges explique pourquoi il n'est parfois possible de restaurer qu'avec un superutilisateur.

7.2.17 pg_dump/pg_dumpall : compresser la sortie

- Compresser 1 fichier
  - utile avec formats `plain`, `tar`, `custom`

  `pg_dump | bzip2`
- Outils multithreads de compression, bien plus rapides :
  - `pbzip2`, `pigz`
  - `xz`, `zstd`...

À part pour le format `directory`, il est possible d'envoyer la sortie standard à un autre outil. Pour les formats non compressés (`tar` et `plain`), cela permet d'abord de compresser avec l'outil de son choix.

Il y a un intérêt même avec le format `custom` : celui d'utiliser des outils de compression plus performants que la zlib, comme `bzip2` ou `lzma` (compression plus forte au prix d'une exécution plus longue) ou `pigz`[7], `pbzip2`[8], ou encore `xz`, `zstd`... beaucoup plus rapides grâce à l'utilisation de plusieurs threads. On met ainsi à profit les nombreux processeurs des machines récentes, au prix d'un très léger surcoût en taille. Ces outils sont présents dans les distributions habituelles.

Au format `custom`, il faut penser à désactiver la compression par défaut, comme dans cet exemple avec `pigz` :

```
$ pg_dump -Fc -Z0 -v foobar | pigz > sauvegarde.dump.gzip
```

On peut aussi utiliser n'importe quel autre outil Unix. Par exemple, pour répartir sur plusieurs fichiers :

```
$ pg_dump | split
```

Nous verrons plus loin que la sortie de `pg_dump` peut même être fournie directement à `pg_restore` ou `psql`, ce qui est fort utile dans certains cas.

[7] https://www.zlib.net/pigz/
[8] http://compression.ca/pbzip2/

7.3 RESTAURATION D'UNE SAUVEGARDE LOGIQUE

7.3.1 Quel outil pour restaurer une sauvegarde logique ?

- `psql`
  - restauration de SQL (option `-Fp`) :
- `pg_restore`
  - restauration binaire (`-Ft` / `-Fc` / `-Fd`)

`pg_dump` permet de réaliser deux types de sauvegarde : une sauvegarde texte (via le format `plain`) et une sauvegarde binaire (via les formats tar, personnalisé et répertoire).

Chaque type de sauvegarde aura son outil :

- `psql` pour les sauvegardes textes ;
- `pg_restore` pour les sauvegardes binaires.

7.3.2 psql

- Client standard PostgreSQL
- Capable d'exécuter des requêtes
  - donc de restaurer une sauvegarde texte (*plain*)
- Très limité dans les options de restauration
  - ni sélection ni parallélisation

`psql` est la console interactive de PostgreSQL. Elle permet de se connecter à une base de données et

d'y exécuter des requêtes, soit une par une, soit un script complet. Or, la sauvegarde texte de `pg_dump` et de `pg_dumpall` fournit un script SQL. Ce dernier est donc exécutable via `psql`.

C'est parfois très pratique, mais il faut savoir que `psql` applique les ordres reçus à la suite, donc sans aucune parallélisation (sauf à lancer plusieurs sessions différentes sur des parties du script). Il ne permet pas de choisir les objets restaurés.

7.3.3 psql : options pour une restauration

- `-f`
  - pour indiquer le fichier contenant la sauvegarde
  - sans `-f` : lit l'entrée standard
- `-1` (`--single-transaction`)
  - pour tout restaurer en une seule transaction
- `-e`
  - pour afficher les ordres SQL exécutés
- `ON_ERROR_ROLLBACK` / `ON_ERROR_STOP`

`psql` étant le client standard de PostgreSQL, le dump au format `plain` se trouve être un script SQL qui peut également contenir des commandes psql, comme `\connect` pour se connecter à une base de données (ce que fait `pg_dumpall` pour changer de base de données).

On bénéficie alors de toutes les options de `psql`, les plus utiles étant celles relatives au contrôle de l'aspect transactionnel de l'exécution.

On peut restaurer avec `psql` de plusieurs manières :

- envoyer le script sur l'entrée standard de psql :

```
cat b1.dump | psql b1
```

- utiliser l'option en ligne de commande -f :

```
psql -f b1.dump b1
```

- utiliser la méta-commande `\i` :

```
b1 =# \i b1.dump
```

Dans les deux premiers cas, la restauration peut se faire à distance alors que dans le dernier cas, le fichier de la sauvegarde doit se trouver sur le serveur de bases de données.

Le script est exécuté comme tout autre script SQL. Comme il n'y a pas d'instruction `BEGIN` au début, l'échec d'une requête ne va pas empêcher l'exécution de la suite du script, ce qui va généralement apporter un flot d'erreurs. De plus `psql` fonctionne par défaut en autocommit : après une erreur, les requêtes précédentes sont déjà validées. La base de données sera donc dans un état à moitié modifié, ce qui peut poser un problème s'il ne s'agissait pas d'une base vierge.

Il est donc souvent conseillé d'utiliser l'option en ligne de commande `-1` pour que le script complet soit exécuté dans une seule transaction. Dans ce cas, si une requête échoue, aucune modification n'aura réellement lieu sur la base, et il sera possible de relancer la restauration après correction du problème.

Enfin, il est à noter qu'une restauration partielle de la sauvegarde est assez complexe à faire. Deux solutions existent, parfois pénibles :

- modifier le script SQL dans un éditeur de texte, ce qui peut être impossible si ce fichier est suffisamment gros ;
- utiliser des outils tels que `grep` et/ou `sed` pour extraire les portions voulues, ce qui peut facilement devenir long et complexe.

Deux variables psql peuvent être modifiées, ce qui permet d'affiner le comportement de psql lors de l'exécution du script :

ON_ERROR_ROLLBACK :

Par défaut il est à `off`, et toute erreur dans **une transaction** entraîne le `ROLLBACK` de toute la transaction. Les commandes suivantes échouent toutes. Activer `ON_ERROR_ROLLBACK` permet de n'annuler que la commande en erreur. psql effectue des *savepoints* avant chaque ordre, et y retourne en cas d'erreur, avant de continuer le script, toujours dans la même transaction.

Cette option n'est donc **pas** destinée à tout arrêter en cas de problème, au contraire. Mais elle peut être utile pour passer outre à une erreur quand on utilise `-1` pour enrober le script dans une transaction.

`ON_ERROR_ROLLBACK` peut valoir `interactive` (ne s'arrêter dans le script qu'en mode interactif, c'est-à-dire quand c'est une commande `\i` qui est lancée) ou `on` dans quel cas il est actif en permanence.

ON_ERROR_STOP :

Par défaut, dans **un script**, une erreur n'arrête pas le déroulement du script. On se retrouve donc souvent avec un ordre en erreur, et beaucoup de mal pour le retrouver, puisqu'il est noyé dans la masse des messages. Quand `ON_ERROR_STOP` est positionné à `on`, le script est interrompu dès qu'une erreur est détectée.

C'est l'option à privilégier quand on veut arrêter un script au moindre problème. Si `-1` est utilisé, et que `ON_ERROR_ROLLBACK` est resté à `off`, le script entier est bien sûr annulé, et on évite les nombreux messages de type :

```
ERROR: current transaction is aborted,
commands ignored until end of transaction block
```

après la première requête en erreur.

Les variables psql peuvent être modifiées :

- par édition du `.psqlrc` (à déconseiller, cela va modifier le comportement de psql pour toute personne utilisant le compte) :

```
cat .psqlrc
\set ON_ERROR_ROLLBACK interactive
```

- en option de ligne de commande de psql :

```
psql --set=ON_ERROR_ROLLBACK='on'
psql -v ON_ERROR_ROLLBACK='on'
```

- de façon interactive dans psql:

```
psql>\set ON_ERROR_ROLLBACK on
```

7.3.4 pg_restore

- Restaure uniquement les sauvegardes au format binaire
  - format autodétecté (`-F` inutile)
- Nombreuses options très intéressantes
- Restaure une base de données
  - complètement ou partiellement

`pg_restore` est un outil capable de restaurer les sauvegardes au format binaire, quel qu'en soit le format. Il offre de nombreuses options très intéressantes, la plus essentielle étant de permettre une restauration partielle de façon aisée. L'exemple typique d'utilisation de `pg_restore` est le suivant :

```
pg_restore -d b1 b1.dump
```

La base de données où la sauvegarde va être restaurée est indiquée avec l'option `-d` et le nom du fichier de sauvegarde est le dernier argument dans la ligne de commande.

7.3.5 pg_restore : indiquer la base de données

- `-d` : base de données de connexion
- `-C` (`--create`) :
  - connexion (`-d`) et `CREATE DATABASE`
  - connexion à la nouvelle base et exécute le SQL
  - aussi : `--clean --if-exists`

Avec `pg_restore`, il est indispensable de fournir le nom de la base de données de connexion avec l'option `-d`. Le fichier à restaurer s'indique en dernier argument sur la ligne de commande.

L'option `--create` (ou `C`) permet de créer la base de données cible. Dans ce cas l'option `-d` doit indiquer une autre base déjà existante (**postgres** souvent), afin que `pg_restore` se connecte pour exécuter l'ordre `CREATE DATABASE`. Après cela, il se connecte à la base nouvellement créée pour exécuter les ordres SQL de restauration. Pour vérifier cela, on peut lancer la commande sans l'option `-d`. En observant le code SQL renvoyé on remarque un `\connect` :

```
$ pg_restore -C b1.dump

--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

--
-- Name: b1; Type: DATABASE; Schema: -; Owner: postgres
--

CREATE DATABASE b1 WITH TEMPLATE = template0 ENCODING = 'UTF8'
                        LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';


ALTER DATABASE b1 OWNER TO postgres;

\connect b1

SET statement_timeout = 0;
-- Suite du dump…
```

Noter que le propriétaire de la base est conservé.

Avec l'option `--create`, il n'est pas possible de restaurer directement dans une base avec un nom différent de la base de données d'origine, car le nom est stocké dans la sauvegarde. Il faut créer la nouvelle base auparavant.

`--create` peut être associé à `--clean --if-exists` pour détruire une base préexistante, et sans message d'erreur si elle n'existe en fait pas. Il vaut mieux être sûr d'être sur le bon serveur.

7.3.6 pg_restore : fichiers en entrée/sortie

- Entrée : Fichier à restaurer en dernier argument de la ligne de commande
- Sortie :
  - `-l` : liste
  - `-f` : fichier SQL
  - `-f -` pour sortie standard
- Attention à ne pas écraser la sauvegarde !

L'option `-f` envoie le SQL généré dans un script, qui sera donc du SQL parfaitement lisible :

```
$ pg_restore base.dump -f script_restauration.sql
```

`-f` n'indique **pas** le fichier de sauvegarde, mais bien la sortie de `pg_restore` quand on ne restaure pas vers une base. N'écrasez pas votre sauvegarde !

Avec `-f -`, le SQL transmis au serveur est affiché sur la sortie standard, ce qui est très pratique pour voir ce qui va être restauré, et par exemple valider les options d'une restauration partielle, récupérer des définitions d'index ou de table, voire « piper » le contenu vers un autre outil.

`pg_restore` exige soit `-d`, soit `-f` : soit on restaure dans une base, soit on génère un fichier SQL. (Avant la version 12, `-f` était le défaut si ni `-d` ni `-f` n'étaient précisés ; il était alors conseillé de rediriger la sortie standard plutôt que d'utiliser `-f` pour éviter toute ambiguïté.)

Pour obtenir le journal d'activité complet d'une restauration, il suffit classiquement de rediriger la sortie :

```
$ pg_restore -d cible --verbose base.dump > restauration.log 2>&1
```

7.3.7 pg_restore : structure ou données

- `--schema-only` : uniquement la structure
- `--data-only` : uniquement les données

ou :

- `--section`
  - `pre-data`
  - `data`
  - `post-data`

Comme pour `pg_dump`, il est possible de ne restaurer que la structure, ou que les données.

Il est possible de restaurer une base section par section. En fait, un fichier de sauvegarde complet est composé de trois sections : la définition des objets, les données, la définition des contraintes et index. Il peut être intéressant de restaurer par section pour optimiser la durée de certaines restaurations complexes.

C'est encore plus intéressant dans les cas un peu délicats (modification des fichiers, imports partiels). On peut alors traiter séparément chaque étape. Par exemple, si l'on veut modifier le SQL (modifier des noms de champs, renommer des index…) tout en tenant à compresser ou paralléliser la sauvegarde pour des raisons de volume :

```
$ mkdir data.dump
$ pg_dump -d source --section=pre-data -f predata.sql
$ pg_dump -d source --section=data -Fd --jobs=8 -f data.dump
$ pg_dump -d source --section=post-data -f postdata.sql
```

Après modification, on réimporte :

```
$ psql -d cible < predata.sql
$ pg_restore -d cible --jobs=8  data.dump
$ psql -d cible < postdata.sql
```

Le script issu de `--section=pre-data` (ci-dessous, allégé des commentaires) contient les `CREATE TABLE`, les contraintes de colonne, les attributions de droits mais aussi les fonctions, les extensions, etc. :

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
```

```
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

CREATE EXTENSION IF NOT EXISTS plperl WITH SCHEMA pg_catalog;
COMMENT ON EXTENSION plperl IS 'PL/Perl procedural language';

CREATE EXTENSION IF NOT EXISTS pg_stat_statements WITH SCHEMA public;
COMMENT ON EXTENSION pg_stat_statements
 IS 'track execution statistics of all SQL statements executed';

CREATE FUNCTION public.impair(i integer) RETURNS boolean
    LANGUAGE sql IMMUTABLE
    AS $$
select mod(i,2)=1 ;
$$;

ALTER FUNCTION public.impair(i integer) OWNER TO postgres;

SET default_tablespace = '';
SET default_table_access_method = heap;

CREATE TABLE public.fils (
    i integer,
    CONSTRAINT impair_ck CHECK ((public.impair(i) IS TRUE)),
    CONSTRAINT nonzero_ck CHECK ((i > 0))
);
ALTER TABLE public.fils OWNER TO postgres;

CREATE TABLE public.pere (
    i integer NOT NULL
);

ALTER TABLE public.pere OWNER TO postgres;
```

La partie `--section=data`, compressée ou non, ne contient que des ordres `COPY` :

```
# lecture du fichier data.dump sur la sortie standard (-)
$ pg_restore -f - data.dump

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

COPY public.fils (i) FROM stdin;
1
2
```

```
…
\.
COPY public.pere (i) FROM stdin;
1
2
…
\.
```

Quant au résultat de `--section=pre-data`, il regroupe notamment les contraintes de clés primaire, de clés étrangères, et les créations d'index. Il est nettement plus rapide de charger la table avant de poser contraintes et index que l'inverse.

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
SET default_tablespace = '';

ALTER TABLE ONLY public.pere
    ADD CONSTRAINT pere_pkey PRIMARY KEY (i);

CREATE UNIQUE INDEX fils_i_idx ON public.fils USING btree (i);

ALTER TABLE ONLY public.fils
    ADD CONSTRAINT fk FOREIGN KEY (i) REFERENCES public.pere(i);
```

7.3.8 pg_restore : sélection d'objets

- `-n <schema>` : uniquement ce schéma
- `-N <schema>` : tous les schémas sauf ce schéma
- `-t <table>` : cette relation
- `-T <trigger>` : ce trigger
- `-I <index>` : cet index
- `-P <fonction>` : cette fonction
- En option
  - possibilité d'en mettre plusieurs
  - `--strict-names`, pour avoir une erreur si l'objet est inconnu
- `--filter` & fichier avec `include` / `exclude` des objets (v17)

`pg_restore` fournit quelques options supplémentaires pour sélectionner les objets à restaurer. Les options `--schema` / `-n`, `--exclude-schema` / `-N` et `--table` / `-t` ont la même signification que pour `pg_dump`. Par contre, `-T` a une signification différente entre les deux outils : `-T` précise un trigger dans `pg_restore`.

Il existe en plus les options `--index` / `-I` et `--function` / `-P` pour restaurer respectivement un index et une routine stockée spécifique.

Là aussi, il est possible de mettre plusieurs fois les options pour restaurer plusieurs objets de même type ou de type différent.

Par défaut, si le nom de l'objet est inconnu, `pg_restore` ne dit rien, et l'opération se termine avec succès. Ajouter l'option `--strict-names` permet de s'assurer d'être averti avec une erreur sur le fait que `pg_restore` n'a pas restauré l'objet souhaité. En voici un exemple :

```
$ pg_restore -t t2 -d postgres pouet.dump
$ echo $?
0
$ pg_restore -t t2 --strict-names -d postgres pouet.dump
pg_restore: [archiver] table "t2" not found
$ echo $?
1
```

Depuis PostgreSQL 17, `pg_restore` accepte des fichiers de filtrage, via l'option `--filter`, similaire à l'option de `pg_dump` vue plus haut. Les types d'objets que l'on peut inclure ou exclure sont : `schema`, `table`, et, en inclusion uniquement : `index`, `function` `trigger`.

7.3.9 pg_restore : sélection avancée

- `-l` : récupération de la liste des objets
- `-L <liste_objets>` : restauration uniquement des objets listés dans ce fichier

`pg_restore` fournit un autre moyen avancé pour sélectionner les objets.

L'option `-l` (`--list`) permet de connaître la liste des actions que réalisera `pg_restore` avec un fichier particulier. Par exemple :

```
$ pg_restore -l b1.dump
;
; Archive created at 2020-09-16 15:44:35 CET
;     dbname: b1
;     TOC Entries: 15
;     Compression: -1
;     Dump Version: 1.14-0
```

```
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 13.0
;     Dumped by pg_dump version: 13.0
;
;
; Selected TOC Entries:
;
200; 1255 24625 FUNCTION public f1() postgres
201; 1255 24626 PROCEDURE public p1() postgres
197; 1259 24630 TABLE public t2 postgres
199; 1259 24637 MATERIALIZED VIEW public mv1 postgres
196; 1259 24627 TABLE public t1 postgres
198; 1259 24633 VIEW public v1 postgres
3902; 0 24627 TABLE DATA public t1 postgres
3903; 0 24630 TABLE DATA public t2 postgres
3778; 2606 24642 CONSTRAINT public t2 t2_pkey postgres
3776; 1259 24643 INDEX public t1_c1_idx postgres
3904; 0 24637 MATERIALIZED VIEW DATA public mv1 postgres
```

Toutes les lignes qui commencent avec un point-virgule sont des commentaires. Le reste indique les objets à créer : la fonction `f1`, la procédure stockée `p1`, les tables `t1` et `t2`, la vue `v1`, la clé primaire sur `t2` et l'index sur `t1`. Il indique aussi les données à restaurer avec des lignes du type `TABLE DATA`. Donc, dans cette sauvegarde, il y a les données pour les tables `t1` et `t2`. Enfin, il y a le rafraîchissement des données de la vue matérialisée `mv1`.

Il est possible de stocker cette information dans un fichier, de modifier le fichier pour qu'il ne contienne que les objets que l'on souhaite restaurer, et de demander à `pg_restore`, avec l'option `-L` (`--use-list`), de ne prendre en compte que les actions contenues dans le fichier. Voici un exemple complet :

```
$ pg_restore -l b1.dump > liste_actions

$ cat liste_actions | \
  grep -v "f1" | \
  grep -v "TABLE DATA public t2" | \
  grep -v "INDEX public t1_c1_idx" \
  > liste_actions_modifiee

$ createdb b1_new

$ pg_restore -L liste_actions_modifiee -d b1_new -v b1.dump
pg_restore: connecting to database for restore
pg_restore: creating PROCEDURE "public.p1()"
pg_restore: creating TABLE "public.t2"
pg_restore: creating MATERIALIZED VIEW "public.mv1"
pg_restore: creating TABLE "public.t1"
pg_restore: creating VIEW "public.v1"
pg_restore: processing data for table "public.t1"
pg_restore: creating CONSTRAINT "public.t2 t2_pkey"
pg_restore: creating MATERIALIZED VIEW DATA "public.mv1"
```

L'option `-v` de `pg_restore` permet de visualiser sa progression dans la restauration. On remarque

bien que la fonction `f1` ne fait pas partie des objets restaurés, tout comme l'index sur `t1` et les données de la table `t2`.

Répétons qu'il est à la charge de l'utilisateur de fournir une liste cohérente en terme de dépendances. Par exemple, sélectionner seulement l'entrée `TABLE DATA` alors que la table n'existe pas dans la base de données cible provoquera une erreur.

7.3.10 pg_restore : parallélisation

- `--jobs <nombre_de_threads>` / `-jN`
  - formats `custom` ou `directory`

Historiquement, `pg_restore` n'utilise qu'une seule connexion à la base de données pour y exécuter en série toutes les requêtes nécessaires pour restaurer la base. Cependant, une fois que la première étape de création des objets est réalisée, l'étape de copie des données et celle de création des index peuvent être parallélisées pour profiter des nombreux processeurs disponibles sur un serveur. L'option `--jobs` / `-j` permet de préciser le nombre de connexions réalisées vers la base de données. Chaque connexion est gérée dans `pg_restore` par un processus sous Unix et par un thread sous Windows.

Cela permet d'améliorer considérablement la vitesse de restauration, en partie parce que les index peuvent être calculés en parallèle. Par exemple, dans un test, une restauration d'une base de 150 Go prenait 5 h avec une seule connexion, mais seulement 3 h avec plusieurs connexions.

Il est à noter que, même si PostgreSQL supporte la parallélisation de certains types de requêtes, cela ne concerne pas la commande `COPY` de `pg_restore`. Il est donc possible qu'il n'y ait pas de gain si une table contient l'essentiel de la volumétrie : la table ne sera importée que par une seule connexion.

Le format `plain` (texte) n'est pas compatible avec cette option.

7.3.11 pg_restore : options diverses

- `-O` / `--no-owner` : ignorer le propriétaire
- `-x` / `--no-privileges` : ignorer les droits
- `--no-comments` : ignorer les commentaires
- `--no-tablespaces` : ignorer les tablespaces
- `-1` / `--single-transaction` : tout restaurer en une seule transaction
  - ou `--transaction-size = N` pour regrouper les ordres (v17)
- `-c` / `--clean` `--if-exists` : détruire un objet avant de le restaurer
- `-v` / `--verbose`

Il reste quelques options plus anecdotiques mais qui peuvent se révéler très pratiques.

`--no-owner`, `--no-privileges`, `--no-comments` et `--no-tablespaces` permettent de ne pas restaurer, respectivement, le propriétaire, les droits, le commentaire et le tablespace des objets.

Regroupement des transactions :

L'option `--single-transaction` / `-1` permet d'exécuter `pg_restore` dans une seule transaction. Le premier intérêt est bien sûr de garantir l'atomicité de l'import. Un autre intérêt est de supprimer l'essentiel des synchronisations sur disque à chaque ordre. (Sur ce dernier point, une alternative est de paramétrer `PGOPTIONS='-c synchronous_commit=off` avant l'appel à `pg_restore`.)

Malheureusement, `--single-transaction` est incompatible avec le parallélisme (`--jobs`) car on ne peut pas avoir plusieurs sessions qui partagent la même transaction. Un autre problème est le maintien d'un grand nombre de verrous pendant toute la durée de la restauration. Avec des milliers de tables, la mémoire dédiée aux verrous peut ne pas suffire.

À partir de PostgreSQL 17, l'option `--transaction-size` est voisine : il s'agit de regrouper les ordres par paquets (par exemple 100 pour `--transaction-size=100`) et d'intercaler des `BEGIN` et `COMMIT`. Il n'y a plus d'atomicité, mais le nombre de synchronisations est réduit sans poser trop de verrous. Ce peut être intéressant avec de très nombreux objets. De plus, cette option est compatible avec le parallélisme.

Suppression préalable :

L'option `--clean` / `-c` permet d'exécuter des `DROP` des objets avant de les restaurer. Ce qui évite les conflits à la restauration de tables par exemple : l'ancienne est détruite avant de restaurer la nouvelle. On ajoutera souvent `--if-exists` pour éviter un message d'erreur si l'objet n'existe en fait pas.

7.4 AUTRES CONSIDÉRATIONS SUR LA SAUVEGARDE LOGIQUE

- Versions des outils & version du serveur
- Script de sauvegarde
- Sauvegarder sans passer par un fichier
- Statistiques et maintenance après import
- Durée d'exécution d'une sauvegarde
- Taille d'une sauvegarde

La sauvegarde logique est très simple à mettre en place. Mais certaines considérations sont à prendre en compte lors du choix de cette solution : comment gérer les statistiques, quelle est la durée d'exécution d'une sauvegarde, quelle est la taille d'une sauvegarde, etc.

7.4.1 Versions des outils clients et version de l'instance

- `pg_dump` :
  - reconnaît les versions de PG antérieures
  - serveur 9.2 minimum si `pg_dump` récent (v15+)
- `pg_restore`
  - minimum la version du `pg_dump` utilisé
  - si possible celle du serveur cible
- Pas de problème entre OS différents

Il est fréquent de générer une sauvegarde sur une version de PostgreSQL pour l'importer sur un serveur de version différente, en général plus récente. Une différence de version mineure n'est d'habitude pas un problème, ce sont les versions majeures (9.6, 12, 17…) qui importent.

Il faut bien distinguer les versions des instances source et cible, et les versions des outils `pg_dump` et `pg_restore` de celles des instances. Attention, il peut y avoir plusieurs versions des outils sur un serveur ou un client. La version sur le poste où l'on sauvegarde peut différer de la version du serveur. Or, leurs formats de sauvegarde `custom` ou `directory` diffèrent. C'est une bonne idée de préciser les chemins complets des binaires `pg_dump` et `pg_restore`, qui contiennent en général le numéro de version.

`pg_dump` sait sauvegarder depuis une instance de version antérieure à la sienne Il refusera de tenter une sauvegarde d'une instance de version postérieure. Donc, pour sauvegarder une base PostgreSQL 14, utilisez un `pg_dump` de version 14, 15 ou supérieure.

PostgreSQL 15 et supérieurs ne savent plus sauvegarder des serveurs antérieurs à PostgreSQL 9.2 (publié en 2012). Au besoin, il faudra générer la sauvegarde avec un client `pg_dump` ou `pg_dumpall` en version 14 ou moins, ou les outils du serveur. La sauvegarde générée sera compatible avec les versions récentes (sauf incompatibilité à un autre niveau, comme une fonctionnalité abandonnée).

`pg_restore` sait lire les sauvegardes des versions antérieures à la sienne. Il peut restaurer vers une instance de version supérieure à la sienne, même s'il vaut mieux utiliser le `pg_restore` de la même version que l'instance cible. La restauration vers une version antérieure à l'outil a de bonnes chances d'échouer en raison d'une évolution de la syntaxe.

En dernier recours, le format `plain` (SQL pur) est toujours lisible par `psql`, et `pg_restore -f dump.sql` permet toujours d'en regénérer un depuis une sauvegarde `plain` ou `directory`. Il peut même être envoyé directement sans fichier intermédiaire, par exemple ainsi :

```
pg_restore -f dump.sql | psql -h serveurcible -d basecible
```

S'il y a une nouveauté ou une régression que l'instance cible ne sait pas interpréter, il est possible de modifier ce SQL. Dans beaucoup de cas, il suffira d'adapter le SQL dans les parties générées par `--section=pre-data` et `--section=post-data`, et de charger directement les données avec `pg_restore --section=data`.

Enfin, puisqu'il s'agit de sauvegardes logiques, des différences de système d'exploitation ne devraient pas poser de problème de compatibilité supplémentaire.

7.4.2 Script de sauvegarde idéal

- Objets globaux :

```
pg_dumpall -g
```

- Chaque base :

```
pg_dump -Fc
pg_dump -Fd
```

- Bien tester !

`pg_dumpall` n'est intéressant que pour récupérer les objets globaux. Le fait qu'il ne supporte pas les formats binaires entraîne que sa sauvegarde n'est utilisable que dans un cas : la restauration de toute

une instance. C'est donc une sauvegarde très spécialisée, ce qui ne conviendra pas à la majorité des cas.

Le mieux est donc d'utiliser `pg_dumpall` avec l'option `-g`, puis d'utiliser `pg_dump` pour sauvegarder chaque base dans un format binaire.

- Avant la version 11, les paramètres sur les bases (`ALTER DATABASE xxx SET param=valeur;`) ne seront pas du tout dans les sauvegardes : `pg_dumpall -g` n'exporte pas les définitions des bases (voir `pg_dump --create` plus haut).
- Les paramètres sur les rôles dans les bases figureront dans l'export de `pg_dumpall -g` ainsi :

```
ALTER role xxx IN DATABASE xxx SET param=valeur;
```

mais les bases n'existeront pas forcément au moment où l'`ALTER ROLE` sera exécuté ! Il faudra donc penser à les restaurer à la main…

Voici un exemple de script minimaliste :

```
#!/bin/sh
# Script de sauvegarde pour PostgreSQL

REQ="SELECT datname FROM pg_database WHERE datallowconn ORDER BY datname"

pg_dumpall -g > globals.dump
psql -XAtc "$REQ" postgres | while read base
do
  pg_dump -Fc $base > ${base}.dump
done
```

Évidemment, il ne conviendra pas à tous les cas, mais donne une idée de ce qu'il est possible de faire. (Voir plus bas `pg_back` pour un outil plus complet.)

Exemple de script de sauvegarde adapté pour un serveur Windows :

```
@echo off

SET PGPASSWORD=super_password
SET PATH=%PATH%;C:\Progra~1\PostgreSQL\11\bin\

pg_dumpall -g -U postgres postgres > c:\pg-globals.sql

for /F %%v in ('psql -XAt -U postgres -d cave
             -c "SELECT datname FROM pg_database WHERE NOT datistemplate"') do (
    echo "dump %%v"
    pg_dump -U postgres -Fc %%v > c:\pg-%%v.dump
```

```
)
pause
```

Autre exemple plus complet de script de sauvegarde totale de toutes les bases, avec une période de rétention :

```
#!/bin/sh
#-----------------------------------------------------------------
#
# Script used to perform a full backup of all databases from a
# PostgreSQL Cluster. The pg_dump use the custom format is done
# into one file per database. There's also a backup of all global
# objects using pg_dumpall -g.
#
# Backup are preserved following the given retention days (default
# to 7 days).
#
# This script should be run daily as a postgres user cron job:
#
# 0 23 * * * /path/to/pg_fullbackup.sh >/tmp/fullbackup.log 2>&1
#
#-----------------------------------------------------------------

# Backup user who must run this script, most of the time it should be postgres
BKUPUSER=postgres
# Number of days you want to preserve your backup
RETENTION_DAYS=7
# Where the backup files should be saved
#BKUPDIR=/var/lib/pgsql/backup
BKUPDIR=/var/lib/postgresql/backup_bases
# Prefix used to prefix the name of all backup files
PREFIX=bkup
# Optional hostname of remote server
# Leave empty to use unix socket
HOSTNAME=""
# Optional user to log in to the remote server
USERNAME=""

WHO=`whoami`
if [ "${WHO}" != "${BKUPUSER}" ]; then
        echo "FATAL: you must run this script as ${BKUPUSER} user."
        exit 1;
fi

# Testing backup directory
if [ ! -e "${BKUPDIR}" ]; then
        mkdir -p "${BKUPDIR}"
fi
echo "Begining backup at "`date`

# Set the query to list the available databases
REQ="SELECT datname FROM pg_database WHERE datistemplate = 'f'
     AND datallowconn ORDER BY datname"

# Set the date variable to be used in the backup file names
DATE=$(date +%Y-%m-%d_%Hh%M)
```

```
# Define the additional pg program options
PG_OPTION=""
if [ $HOSTNAME != "" ]; then
        PG_OPTION="${PG_OPTION} -h $HOSTNAME"
fi;
if [ $USERNAME != "" ]; then
        PG_OPTION="${PG_OPTION} -U $USERNAME"
fi;

# Dump PostgreSQL Cluster global objects
echo "Dumping global object into ${BKUPDIR}/${PREFIX}_globals_${DATE}.dump"
pg_dumpall ${PG_OPTION} -g > "${BKUPDIR}/${PREFIX}_globals_${DATE}.dump"
if [ $? -ne 0 ]; then
        echo "FATAL: Can't dump global objects with pg_dumpall."
        exit 2;
fi

# Extract the list of database
psql ${PG_OPTION} -Atc "$REQ" postgres | while read base

# Dump content of all databases
do
  echo "Dumping database $base into ${BKUPDIR}/${PREFIX}_${base}_${DATE}.dump"
  pg_dump ${PG_OPTION} -Fc $base > "${BKUPDIR}/${PREFIX}_${base}_${DATE}.dump"
  if [ $? -ne 0 ]; then
          echo "FATAL: Can't dump database ${base} with pg_dump."
          exit 3;
  fi
done
if [ $? -ne 0 ]; then
        echo "FATAL: Can't list databases with psql query."
        exit 4;
fi

# Starting deletion of obsolete backup files
if [ ${RETENTION_DAYS} -gt 0 ]; then
  echo "Removing obsolete backup files older than ${RETENTION_DAYS} day(s)."
  find ${BKUPDIR}/ -maxdepth 1 -name "${PREFIX}_*" -mtime ${RETENTION_DAYS} \
                   -exec rm -rf '{}' ';'
fi

echo "Backup ending at "`date`

exit 0
```

7.4.3 pg_back - Présentation

- https://github.com/orgrim/pg_back
- Type de sauvegardes : **logiques** (`pg_dump`)
- Langage : **bash** (v1) / **go** (v2)
- Licence : **BSD** (libre)
- Type de stockage : **local** + export cloud
- Planification : **crontab**
- **Unix/Linux** (v1 & 2) / **Windows (v2)**
- Compression : via pg_dump
- Versions compatibles : **toutes**
- Rétention : **durée**

pg_back[9] a été écrit par Nicolas Thauvin, alors consultant de Dalibo, également auteur original de pitrery[10].

Ce programme assez complet vise à gérer le plus simplement possible des sauvegardes logiques en utilisant `pg_dump` et `pg_dumpall`.

La version 1 est en bash, directement utilisable et éprouvée, mais ne sera plus maintenue à terme. La version 2, parue en 2021, a été réécrite en go. Le binaire est directement utilisable, et permet notamment une configuration différente par base, une meilleure gestion des paramètres de connexion à PostgreSQL et le support de Windows. Pour les instances avant la 11, le script pg_dumpacl[11] est intégré (v2) ou supporté (v1) pour sauvegarder le paramétrage au niveau des bases.

Les sauvegardes sont bien sûr aux formats gérés par `pg_dump` : SQL, custom, par répertoire, compressées ou non…

Le premier intérêt de l'outil est la gestion de la rétention. Les anciennes sauvegardes sont automatiquement purgées, et l'on peut en conserver un nombre minimum.

Il est possible de chiffrer les sauvegardes avec une phrase de passe (chiffrage AGE), de générer des sommes de contrôle, et d'exporter vers Azure, Google Cloud, Amazon S3 ou Backblaze B2, ou n'importe quel serveur distant accessible avec `ssh` en SFTP.

`pg_back` ne propose pas d'option pour restaurer les données : il faut utiliser ceux de PostgreSQL (`pg_restore`, `psql`).

[9] https://github.com/orgrim/pg_back
[10] https://dalibo.github.io/pitrery/
[11] https://github.com/dalibo/pg_dumpacl

7.4.4 Sauvegarde et restauration sans fichier intermédiaire

- `pg_dump -Fp | psql`
- `pg_dump -Ft | pg_restore`
- `pg_dump -Fc | pg_restore`
- Utilisation des options `-h`, `-p`, `-d`
- Attention à la gestion des erreurs !

La duplication d'une base ne demande pas forcément de passer par un fichier intermédiaire. Il est possible de fournir la sortie de `pg_dump` (format `plain` implicite) à `psql` ou `pg_restore`. Par exemple :

```
$ createdb nouvelleb1
$ pg_dump -Fp b1 | psql nouvelleb1
```

Ces deux commandes permettent de dupliquer `b1` dans `nouvelleb1`.

L'utilisation des options `-h`, `-p`, `-d` permet de sauvegarder et restaurer une base sur des instances différentes, qu'elles soient locales ou à distance.

Le gain de temps est appréciable : l'import peut commencer avant la fin de l'export. Comme toujours avec `pg_dump`, les données restaurées correspondent à celles qui existaient au début de la sauvegarde. On épargne aussi la place nécessaire au stockage du backup. Par contre on ne peut pas paralléliser l'export.

Avec `-Fp`, le flux circule en pur texte : il est possible d'intercaler des appels à des outils comme `awk` ou `sed` pour effectuer certaines opérations à la volée (renommage…).

Cette version peut être intéressante :

```
$ pg_dump -Fc  | pg_restore
```

car elle permet de profiter des options propres au format `custom` à commencer par la compression. Il n'y a alors pas besoin d'intercaler des commandes comme `gzip` / `gunzip` pour alléger la charge réseau.

Le point délicat est la gestion des erreurs puisqu'il y a deux processus à surveiller : par exemple dans un script `bash`, on testera le contenu de `${PIPESTATUS[@]}` (et pas seulement `$?`) pour vérifier que tout s'est bien déroulé, et l'on ajoutera éventuellement `set -o pipefail` en début de script.

7.4.5 Statistiques et maintenance après import

- Statistiques non sauvegardées
  - `ANALYZE` impérativement après une restauration !
- Pour les performances :
  - `VACUUM` (ou `VACUUM ANALYZE`)
- À plus long terme :
  - `VACUUM FREEZE`

Les statistiques sur les données, utilisées par le planificateur pour sélectionner le meilleur plan d'exécution possible, ne font pas partie de la sauvegarde. Il faut donc exécuter un `ANALYZE` après avoir restauré une sauvegarde. Sans cela, les premières requêtes pourraient s'exécuter très mal du fait de statistiques non à jour.

Lancer un `VACUUM` sur toutes les tables restaurées est également conseillé. S'il n'y a pas besoin de les défragmenter, certaines opérations de maintenance effectuées par un `VACUUM` ont un impact sur les performances. En premier lieu, les *hint bits* (« bits d'indice ») de chaque enregistrement seront mis à jour au plus tard à la relecture suivante, et généreront de nombreuses écritures. Un `VACUUM` explicite forcera ces écritures, si possible avant la mise en production de la base. Il créera aussi la *visibility map* des tables, ce qui autorisera les *Index Only Scans*, une optimisation extrêmement puissante, sans laquelle certaines requêtes seront beaucoup plus lentes.

L'`autovacuum` se chargera bien sûr progressivement de tout cela. Il est cependant par défaut bridé pour ne pas gêner les opérations, et pas toujours assez réactif. (De plus, jusqu'en PostgreSQL 12 compris, les insertions ne provoquaient que l'`ANALYZE`, pas le `VACUUM`, qui pouvait donc tarder sur les tables un peu statiques.)

Il est donc préférable de lancer un `VACUUM ANALYZE` manuel à la fin de la restauration, afin de procéder immédiatement au passage des statistiques et aux opérations de maintenance.

Il est possible de séparer les deux étapes. L'`ANALYZE` est impératif et rapide ; le `VACUUM` est beaucoup plus lent mais peut avoir lieu durant la production si le temps presse.

À plus long terme, dans le cas d'un gros import ou d'une restauration de base, existe un autre danger : le `VACUUM FREEZE`. Les numéros de transaction étant cycliques, l'autovacuum les « nettoie » des

tables quand il sont assez vieux. Les lignes ayant été importées en même temps, cela peut générer l'écriture de gros volumes de manière assez soudaine. Pour prévenir cela, lancez un `VACUUM FREEZE` dans une période calme quelques temps après l'import.

7.4.6 Durée d'exécution

- Difficile à chiffrer
- Dépend de l'activité sur le serveur
- Option `-v`
- Suivre les `COPY`
  - vue `pg_stat_progress_copy` (v14+)
- Suivre l'indexation
  - vue `pg_stat_progress_create_index`

La durée d'exécution d'une sauvegarde et d'une restauration est difficile à estimer. Cela dépend beaucoup de l'activité présente sur le serveur, de la volumétrie, du matériel, etc.

L'option `-v` de `pg_dump` et de `pg_restore` permet de suivre les opérations, action par action (donc la création des objets, mais aussi leur remplissage). Cependant, sans connaître la base sauvegardée ou restaurée, il est difficile de prédire le temps restant pour la fin de l'opération. Nous savons seulement qu'il a fini de traiter tel objet.

Depuis la version 14, il est possible de suivre individuellement les opérations de copie des données, si ces dernières passent par l'instruction `COPY` (qui est le défaut, mais forcer `INSERT` est possible). La vue dédiée `pg_stat_progress_copy` affiche des informations sur le `COPY` en cours. Là encore, il est difficile d'en tirer beaucoup d'informations sans bien connaître la base en cours de traitement. Cependant, cela permet de savoir si une table a bientôt fini d'être traitée par le `COPY` en cours. Pour les tables volumineuses, c'est intéressant.

La partie finale de création des index peut être suivie dans la vue système `pg_stat_progress_create_index`.

7.4.7 Taille d'une sauvegarde logique

- Difficile à évaluer
- Contenu des index non sauvegardé
  - donc sauvegarde plus petite
- Objets binaires :
  - entre 2 et 4 fois plus gros
  - donc sauvegarde plus grosse

Il est très difficile de corréler la taille d'une base avec celle de sa sauvegarde.

Le contenu des index n'est pas sauvegardé. Donc, sur une base contenant 10 Go de tables et 10 Go d'index, avoir une sauvegarde de 10 Go ne serait pas étonnant. Le contenu des index est généré lors de la restauration.

Par contre, les données des tables prennent généralement plus de place. Un objet binaire est stocké sous la forme d'un texte, soit de l'octal (donc 4 octets), soit de l'hexadécimal (2 octets). Donc un objet binaire prend 2 à 4 fois plus de place dans la sauvegarde que dans la base. Mais même un entier ne va pas avoir la même occupation disque. La valeur 10 ne prend que 2 octets dans la sauvegarde, alors qu'il en prend quatre dans la base. Et la valeur 1 000 000 prend 7 octets dans la sauvegarde alors qu'il en prend toujours 4 dans la base.

Tout ceci permet de comprendre que la taille d'une sauvegarde n'a pas tellement de lien avec la taille de la base. Il est par contre plus intéressant de comparer la taille de la sauvegarde de la veille avec celle du jour. Tout gros changement peut être annonciateur d'un changement de volumétrie de la base, changement voulu ou non.

7.4.8 Avantages de la sauvegarde logique

- Simple et rapide
- Sans interruption de service
- Indépendante de la version de PostgreSQL
- Granularité de sélection à l'objet
- Taille réduite
- Ne conserve pas la fragmentation des tables et des index
- Éventuellement depuis un serveur secondaire

La sauvegarde logique ne nécessite aucune configuration particulière de l'instance, hormis l'autorisation de la connexion du client effectuant l'opération. La sauvegarde se fait sans interruption de service. Par contre, l'instance doit être disponible, ce qui n'est pas un problème dans la majorité des cas.

Elle est indépendante de la version du serveur PostgreSQL, source et cible. Elle ne contient que des ordres SQL nécessaires à la création des objets à l'identique et permet donc de s'abstraire du format de stockage sur le serveur. De ce fait, la fragmentation des tables et des index disparaît à la restauration.

Une restauration de sauvegarde logique est d'ailleurs la méthode officielle de montée de version majeure. Même s'il existe d'autres méthodes de migration de version majeure, elle reste le moyen le plus sûr parce que le plus éprouvé.

Une sauvegarde logique ne contenant que les données utiles, sa taille est généralement beaucoup plus faible que la base de données source, sans parler de la compression qui peut encore réduire l'occupation sur disque. Par exemple, seuls les ordres DDL permettant de créer les index sont stockés, pas leur contenu. Ils sont alors créés de zéro à la restauration.

Les outils permettent de sélectionner très finement les objets sur lesquels on travaille, à la sauvegarde comme à la restauration.

Il est assez courant d'effectuer une sauvegarde logique à partir d'un serveur secondaire (réplica de la production), pour ne pas charger les disques du primaire ; quitte à mettre la réplication du secondaire en pause le temps de la sauvegarde.

7.4.9 Inconvénients de la sauvegarde logique

- Durée : dépendante des données et de l'activité
- Restauration : uniquement au démarrage de l'export
- Efficace si < 200 Go
- Plusieurs outils pour sauvegarder une instance complète
- `ANALYZE`, `VACUUM ANALYZE`, `VACUUM FREEZE` après import

L'un des principaux inconvénients d'une sauvegarde et restauration porte sur la durée d'exécution. Elle est proportionnelle à la taille de la base de données, ou à la taille des objets choisis pour une sauvegarde partielle.

En conséquence, il est généralement nécessaire de réduire le niveau de compression pour les formats `custom` et `directory` afin de gagner du temps. Avec des disques mécaniques en RAID 10, il est généralement nécessaire d'utiliser d'autres méthodes de sauvegarde lorsque la volumétrie dépasse 200 Go.

Le second inconvénient majeur de la sauvegarde logique est l'absence de granularité temporelle. Une « photo » des données est prise au démarrage de la sauvegarde, et on ne peut restaurer qu'à cet instant,

quelle que soit la durée d'exécution de l'opération. Il faut cependant se rappeler que cela garantit la cohérence du contenu de la sauvegarde d'un point de vue transactionnel.

Comme les objets et leur contenu sont effectivement recréés à partir d'ordres SQL lors de la restauration, on se débarrasse de la fragmentation des tables et index, mais on perd aussi les statistiques de l'optimiseur, ainsi que certaines méta-données des tables. Il est donc nécessaire de lancer ces opérations de maintenance après l'import.

7.5 SAUVEGARDE PHYSIQUE À FROID DES FICHIERS

7.5.1 Sauvegarde physique à froid : principe

- Instance arrêtée : sauvegarde cohérente
- Ne pas oublier : journaux, tablespaces, configuration !
- Outils : système, aucun spécifique à PostgreSQL
  - `cp`, `tar`...
  - souvent : `rsync` en 2 étapes : à chaud puis à froid
  - snapshots SAN/LVM (attention à la cohérence)

Toutes les données d'une instance PostgreSQL se trouvent dans des fichiers. Donc sauvegarder les fichiers permet de sauvegarder une instance. Cependant, cela ne peut pas se faire aussi simplement que ça. Lorsque PostgreSQL est en cours d'exécution, il modifie certains fichiers du fait de l'activité des utilisateurs ou des processus (interne ou non) de maintenances diverses. Sauvegarder les fichiers de la base sans plus de manipulation ne peut donc se faire qu'à froid. Il faut donc arrêter PostgreSQL pour disposer d'une sauvegarde cohérente si la sauvegarde se fait au niveau du système de fichiers.

Il est cependant essentiel d'être attentif aux données qui ne se trouvent pas directement dans le répertoire des données (`PGDATA`), notamment :

- le répertoire des journaux de transactions (`pg_wal` ou `pg_xlog`), qui est souvent placé dans un autre système de fichiers pour gagner en performances : sans lui, la sauvegarde ne sera pas utilisable ;
- les répertoires des tablespaces s'il y en a, sinon une partie des données manquera ;
- les fichiers de configuration (sous `/etc` sous Debian, notamment), y compris ceux des outils annexes à PostgreSQL.

Voici un exemple de sauvegarde (Cent OS 7) :

```
$ systemctl stop postgresql-12
$ tar cvfj data.tar.bz2 /var/lib/pgsql/12/data
$ systemctl start postgresql-12
```

Le gros avantage de cette sauvegarde se trouve dans le fait que vous pouvez utiliser tout outil de sauvegarde de fichier : `cp`, `scp`, `tar`, `ftp`, `rsync`, etc. et tout outil de sauvegarde utilisant lui-même ces outils.

Comme la sauvegarde doit être effectuée avec l'instance arrêtée, la durée de l'arrêt est dépendante du volume de données à sauvegarder. On peut optimiser les choses en réduisant le temps d'interruption avec l'utilisation de snapshots au niveau système de fichier ou avec `rsync`.

Pour utiliser des snapshots, il faut soit disposer d'un SAN offrant cette possibilité ou bien utiliser la fonctionnalité équivalente de LVM voire du système de fichier. Dans ce cas, la procédure est la suivante :

- arrêt de l'instance PostgreSQL ;
- création des snapshots de l'ensemble des systèmes de fichiers ;
- démarrage de l'instance ;
- sauvegarde des fichiers à partir des snapshots ;
- destruction des snapshots.

Si on n'a pas la possibilité d'utiliser des snapshots, on peut utiliser `rsync` de cette manière :

- `rsync` de l'ensemble des fichiers de l'instance PostgreSQL en fonctionnement pour obtenir une première copie (incohérente) ;
- arrêt de l'instance PostgreSQL ;
- `rsync` de l'ensemble des fichiers de l'instance pour ne transférer que les différences ;
- redémarrage de l'instance PostgreSQL.

7.5.2 Avantages des sauvegardes à froid

- Simple
- Rapide à la sauvegarde
- Rapide à la restauration
- Beaucoup d'outils disponibles

L'avantage de ce type de sauvegarde est sa rapidité. Cela se voit essentiellement à la restauration où les fichiers ont seulement besoin d'être créés. Les index ne sont pas recalculés par exemple, ce qui est certainement le plus long dans la restauration d'une sauvegarde logique.

7.5.3 Inconvénients des sauvegardes à froid

- Arrêt de la production
- Sauvegarde de l'instance complète (donc aucune granularité)
- Restauration de l'instance complète
- Conservation de la fragmentation
- Impossible de changer d'architecture
  - Réindexation si changement OS

Il existe aussi de nombreux inconvénients à cette méthode.

Le plus important est certainement le fait qu'il faut arrêter la production. L'instance PostgreSQL doit être arrêtée pour que la sauvegarde puisse être effectuée.

Il ne sera pas possible de réaliser une sauvegarde ou une restauration partielle, il n'y a pas de granularité. C'est forcément l'intégralité de l'instance qui sera prise en compte.

Étant donné que les fichiers sont sauvegardés, toute la fragmentation des tables et des index est conservée.

De plus, la structure interne des fichiers implique l'architecture où cette sauvegarde sera restaurée. Donc une telle sauvegarde impose de conserver un serveur 32 bits pour la restauration si la sauvegarde a été effectuée sur un serveur 32 bits. De même, l'architecture *little endian/big endian* doit être respectée.

De plus, des différences entre deux systèmes, et même entre deux versions d'une même distribution, peuvent mener à devoir réindexer toute l'instance[12].

Tous ces inconvénients ne sont pas présents pour la sauvegarde logique. Cependant, cette sauvegarde a aussi ses propres inconvénients, comme une lenteur importante à la restauration.

7.5.4 Diminuer l'immobilisation

- Utilisation de rsync
- Une fois avant l'arrêt
- Une fois après

[12] https://blog-postgresql.verite.pro/2018/08/30/glibc-upgrade.html

Il est possible de diminuer l'immobilisation d'une sauvegarde de fichiers en utilisant la commande `rsync`.

`rsync` permet de synchroniser des fichiers entre deux répertoires, en local ou à distance. Il va comparer les fichiers pour ne transférer que ceux qui ont été modifiés. Il est donc possible d'exécuter `rsync` avec PostgreSQL en cours d'exécution pour récupérer un maximum de données, puis d'arrêter PostgreSQL, de relancer `rsync` pour ne récupérer que les données modifiées entre temps, et enfin de relancer PostgreSQL. Notez que l'utilisation des options `--delete` et `--checksum` est *fortement conseillée* lors de la deuxième passe, pour rendre la copie totalement fiable.

Voici un exemple de ce cas d'utilisation :

```
$ rsync -a /var/lib/postgresql/ /var/lib/postgresql2
$ /etc/init.d/postgresql stop
$ rsync -a --delete --checksum /var/lib/postgresql /var/lib/postgresql2
$ /etc/init.d/postgresql start
```

7.6 SAUVEGARDE À CHAUD

Snapshot, ou les outils de PostgreSQL ?

7.6.1 Sauvegarde à chaud des fichiers par snapshot de partition : principe

- Avec certains systèmes de fichiers
- Avec LVM
- Avec la majorité des SAN
- Attention : cohérence entre partitions

Certains systèmes de fichiers (principalement les systèmes de fichiers ZFS et BTRFS) ainsi que la majorité des SAN sont capables de faire une sauvegarde d'un système de fichiers en instantané. En fait, ils figent les blocs utiles à la sauvegarde. S'il est nécessaire de modifier un bloc figé, ils utilisent un autre bloc pour stocker la nouvelle valeur. Cela revient un peu au fonctionnement de PostgreSQL dans ses fichiers.

L'avantage est de pouvoir sauvegarder instantanément un système de fichiers. L'inconvénient est que cela ne peut survenir que sur un seul système de fichiers : impossible dans ce cas de déplacer les journaux de transactions sur un autre système de fichiers pour gagner en performance ou d'utiliser des tablespaces pour gagner en performance et faciliter la gestion de la volumétrie des disques. De plus, comme PostgreSQL n'est pas arrêté au moment de la sauvegarde, au démarrage de PostgreSQL sur la sauvegarde restaurée, ce dernier devra rejouer les journaux de transactions.

Une baie SAN assez haut de gamme pourra disposer d'une fonctionnalité de snapshot cohérent sur plusieurs volumes (« LUN »), ce qui permettra, si elle est bien paramétrée, de réaliser un snapshot de tous les systèmes de fichiers composant la base de façon cohérente.

Néanmoins, cela reste une méthode de sauvegarde très appréciable quand on veut qu'elle ait le moins d'impact possible sur les utilisateurs.

7.6.2 Sauvegarde à chaud des fichiers avec PostgreSQL

- `pg_basebackup`
  - sauvegarde ponctuelle
- PITR : *Point In Time Recovery*
  - restauration à la transaction près
  - technique avancée, complexe à mettre en place et à maintenir
  - pas de coupure de service
  - outils dédiés (pgBackRest, barman)
  - et archivage des journaux de transaction
  - outils conseillés : pgBackRest, barman

Pour une sauvegarde physique à chaud ponctuelle, on peut utiliser `pg_basebackup`[13], fourni avec PostgreSQL. Cet outil se base sur le protocole de réplication par flux (*streaming*) et les slots de réplications pour créer une copie des répertoires de données depuis une instance en cours de fonctionnement. Son maniement est très simple. Il existe en plus un outil de vérification de la sauvegarde (`pg_verifybackup`). La sauvegarde sera l'image de l'instance au moment de la fin de sauvegarde. À partir de PostgreSQL 17, l'outil sait même faire des sauvegardes incrémentales.

La sauvegarde la plus intéressante est la sauvegarde PITR (*Point In Time Recovery*) Elle consiste à réaliser les sauvegardes des fichiers sans arrêter l'instance (à chaud), sans perte ou presque de données, et avec une possible restauration à un point précis dans le temps. On peut donc restaurer l'instance à la seconde précédant un sinistre ou une fausse manipulation. Les autres méthodes de sauvegarde ne sont qu'une image à un moment précis du passé, souvent la nuit précédente.

La sauvegarde PITR est cependant une technique avancée, qui nécessite la compréhension de concepts comme l'archivage des journaux de transaction, non abordés ici. Il est fortement conseillé de la mettre en place à l'aide d'outils dédiés et éprouvés comme pgBackRest[14] ou barman[15].

En raison de la complexité de ces méthodes, y compris à la restauration, on testera d'autant plus soigneusement les procédures.

[13] https://dali.bo/i2_html#copie-physique-%C3%A0-chaud-ponctuelle-avec-pg_basebackup
[14] https://pgbackrest.org/
[15] https://www.pgbarman.org

7.7 RECOMMANDATIONS GÉNÉRALES

- Prendre le temps de bien choisir sa méthode
- Ne pas oublier les fichiers de configuration
- Bien la tester
- Bien tester la restauration
- Et tester régulièrement !

On ne répétera jamais assez qu'il faut régulièrement tester ses procédures de restauration. Il ne faut pas hésiter à automatiser la restauration. Une petite machine avec assez de disque peut suffire.

7.7.1 Matrice des options de sauvegarde

	Simplicité	Coupure	Restauration	Fragmentation
copie à froid	facile	longue	rapide	conservée
snapshot FS	facile	aucune	rapide	conservée
pg_dump	facile	aucune	lente	perdue
rsync + copie à froid	moyen	courte	rapide	conservée
PITR	difficile	aucune	rapide	conservée

Ce tableau indique les points importants de chaque type de sauvegarde. Il permet de faciliter un choix entre les différentes méthodes.

7.8 CONCLUSION

- Plusieurs solutions pour la sauvegarde et la restauration
- Sauvegarde/Restauration complète ou partielle
- Toutes cohérentes
- La plupart à chaud
- Méthode de sauvegarde avancée : PITR

PostgreSQL propose plusieurs méthodes de sauvegardes et de restaurations. Elles ont chacune leurs avantages et leurs inconvénients. Cependant, elles couvrent à peu près tous les besoins : sauvegardes et restaurations complètes ou partielles, sauvegardes cohérentes, sauvegardes à chaud comme à froid.

La méthode de sauvegarde avancée dite *Point In Time Recovery*, ainsi que les dernières sophistications du moteur en matière de réplication par streaming, permettent d'offrir les meilleures garanties afin de minimiser les pertes de données, tout en évitant toute coupure de service liée à la sauvegarde. Il s'agit de techniques avancées, beaucoup plus complexes à mettre en place et à maintenir que les méthodes évoquées précédemment.

7.8.1 Questions

N'hésitez pas, c'est le moment !

7.9 QUIZ

https://dali.bo/i1_quiz

7.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/i1_solutions.

Ce TP suppose que vous avez déjà quelques bases et rôles dans votre instance. Créez au moins ceci, qui existe peut-être déjà suite à d'autres TP :

```
createuser testperf
createuser patron
createdb entreprise --owner patron

createdb pgbench --owner testperf
/usr/pgsql-17/bin/pgbench -i -s1  pgbench --foreign-keys
```

7.10.1 Sauvegardes logiques

But : Sauvegarder des bases

Créer un répertoire `backups` dans le répertoire `home` de **postgres** pour y déposer les fichiers d'export.

À l'aide de `pg_dumpall`, sauvegarder toutes les bases de données de l'instance PostgreSQL dans le fichier `~postgres/backups/base_all.sql`.

Quelle est la taille de la sauvegarde ?

Avec quel outil consulter le contenu du fichier de sauvegarde ?

Ouvrir le fichier de sauvegarde. Quelles en sont les principales parties ?

Recommencer la sauvegarde en compressant la sauvegarde avec `gzip`. Quelle en est la taille ?

Qu'affiche un `pg_dump -d entreprise` ?

À l'aide de `pg_dump`, sauvegarder la base de données **pgbench** au format *custom* dans le fichier `~postgres/backups/base_pgbench.dump`. Quelle est sa taille ?

Avec `pg_restore` et son option `-l`, consulter le contenu de `~postgres/backups/base_pgbench.dump`.

Exporter uniquement les objets globaux de l'instance (rôles, définitions de tablespaces) à l'aide de `pg_dumpall` dans le fichier `~postgres/backups/base_globals.sql`.

Consulter le contenu de ce fichier.

Sauvegarder la table `pgbench_accounts` dans le fichier `~postgres/backups/table_pgbench_accounts.sql` au format *plain text*. Que contient ce fichier ?

7.10.2 Restaurations logiques

But : Restaurer des sauvegardes

Que se passe-t-il si l'on exécute ceci ?

```
pg_restore ~postgres/backups/base_pgbench.dump -f -
```

Créer une nouvelle base **pgbench2** appartenant au rôle **testperf**.

Ajuster `.pgpass` et `pg_hba.conf` pour que **testperf** puisse se connecter à toute base par son mot de passe.

Restaurer le contenu de la base de donnée **pgbench** dans **pgbench2** en utilisant le fichier de sauvegarde `base_pgbench.dump`. Toutes les tables sont-elles restaurées à l'identique ?

Puis renommer la base de données **pgbench** en **pgbench_old** et la base de données **pgbench2** en **pgbench**.

Dans la base **pgbench**, détruire la table `pgbench_history`.

À partir de la sauvegarde de la base **pgbench** au format *custom* faite précédemment, restaurer uniquement la table `pgbench_history` dans la nouvelle base **pgbench**, avec `pg_restore -t`. La table est-elle complète ?

Rechercher les morceaux manquants dans la section `post-data` de la sauvegarde.

Dans une fenêtre, lancer une instance `pgbench` sur la base **pgbench**

```
/usr/pgsql-14/bin/pgbench -U testperf -T 600 pgbench
```

Pour dupliquer la base **pbench** :

- Créer une nouvelle base **pgbench3** appartenant à **testperf**.
- Avec `pg_dump`, copier les données de la base **pgbench** dans cette nouvelle base **pgbench3** sans passer par un fichier de sauvegarde, avec un pipe.

7.10.3 Sauvegarde et restauration partielle

But : Choisir les données à restaurer

Il faut restaurer dans une base **pgbench4** tout le contenu de la sauvegarde de la base **pgbench**, sauf les *données* de la table `pgbench_accounts` ni les tables `copie*`. La définition de la table `pgbench_accounts` et toutes les contraintes s'y rapportant doivent être restaurées. Pour cela :

- utiliser les options `-l` de `pg_restore` pour créer le fichier des objets à charger ;
- adapter ce fichier en fonction des critères ci-dessus ;
- utiliser ce fichier grâce à l'option `-L` de `pg_restore`.

7.10.4 (Optionnel) Sauvegarde et restauration par parties avec modification

But : Manipuler les données à restaurer

Effectuer une sauvegarde de **pgbench_old**, puis une restauration vers une nouvelle base **test**, lors de laquelle on ne rechargera que les tables `pgbench_*` en les renommant en `test_*`. Pour cela :

- créer plusieurs fichiers (format *plain*) grâce aux options `--section=` de `pg_dump` ;
- modifier les > fichiers obtenus ;
- les charger ;

- vérifier que les tables sont bien là avec les nouveaux noms.

Recommencer avec une sauvegarde des données en format *custom* en scriptant les modifications à la volée.

7.10.5 (Optionnel) Sauvegardes d'objets isolés

But : Sauvegarder des objets séparément

On veut sauvegarder les tables toutes les tables `pgbench_*` dans des fichiers séparés.

On veut sauvegarder les tables toutes les tables `pgbench_*` dans des fichiers séparés :

- Créer un script shell nommé `sauvegarde_tables.sh`
- il utilisera `psql` pour lister les tables concernées depuis la vue `pg_tables` ou `\d+`, dans la base **pgbench** ;
- pour simplifier la sortie : utiliser les options `-A`, `-t`, `-F` de `psql` ;
- enfin, boucler sur ces tables pour lancer un `pg_dump`, au format *plain text*, sur chaque table, vers des fichiers nommés `table_<nom_table>.sql`.

Quel est le problème de principe de ces sauvegardes séparées sur une base active ?

7.11 TRAVAUX PRATIQUES (SOLUTIONS)

Ce TP suppose que vous avez déjà quelques bases et rôles dans votre instance. Créez au moins ceci, qui existe peut-être déjà suite à d'autres TP :

```
createuser testperf
createuser patron
createdb entreprise --owner patron

createdb pgbench --owner testperf
/usr/pgsql-17/bin/pgbench -i -s1  pgbench --foreign-keys
```

7.11.1 Sauvegardes logiques

Répertoire cible

> Créer un répertoire `backups` dans le répertoire `home` de **postgres** pour y déposer les fichiers d'export.

En tant qu'utilisateur **postgres** :

```
$ mkdir ~postgres/backups
```

Sauvegarde logique de toutes les bases

> À l'aide de `pg_dumpall`, sauvegarder toutes les bases de données de l'instance PostgreSQL dans le fichier `~postgres/backups/base_all.sql`.

En tant qu'utilisateur **postgres**, puis exécuter la commande suivante :

```
$ pg_dumpall > ~postgres/backups/base_all.sql
```

> Quelle est la taille de la sauvegarde ?

```
$ ls -lh ~postgres/backups/
total 37M
-rw-r--r-- 1 postgres postgres 37M 1 août 17:17 base_all.sql
```

> Avec quel outil consulter le contenu du fichier de sauvegarde ?

Ne jamais ouvrir un fichier potentiellement volumineux avec un éditeur comme `vi` car il pourrait alors consommer des gigaoctets de mémoire. Sous Unix, `less` convient en général :

```
$ less ~postgres/backups/base_all.sql
```

> Ouvrir le fichier de sauvegarde. Quelles en sont les principales parties ?

Le fichier contient en clair des ordres SQL :

- d'abord du paramétrage destiné à la restauration :

```
SET default_transaction_read_only = off;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
```

- la déclaration des rôles avec des mots de passe hachés, et les affectations des groupes :

```
CREATE ROLE boulier;
ALTER ROLE boulier WITH NOSUPERUSER INHERIT
NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS
PASSWORD 'SCRAM-SHA-256$4096:ksvTos26fH+0qUov4zssLQ==$un1…';
…
GRANT secretariat TO boulier GRANTED BY postgres;
GRANT secretariat TO tina GRANTED BY postgres;
```

- des ordres de création des bases :

```
CREATE DATABASE entreprise WITH TEMPLATE = template0 ENCODING = 'UTF8'
LOCALE = 'en_GB.UTF-8';

ALTER DATABASE entreprise OWNER TO patron;
```

- les ordres pour se connecter à chaque base et y déclarer des objets :

```
\connect entreprise
…
CREATE TABLE public.brouillon (
    id integer,
    objet text,
    creationts timestamp without time zone
);
ALTER TABLE public.brouillon OWNER TO boulier;
…
```

- les données elles-mêmes, sous formes d'ordres `COPY` :

```
COPY public.produit (appellation) FROM stdin;
Gewurtzraminer vendanges tardives
Cognac
Eau plate
Eau gazeuse
Jus de framboise
\.
```

- ainsi que divers ordres comme des contraintes ou des créations d'index :

```
ALTER TABLE ONLY public.pgbench_accounts
    ADD CONSTRAINT pgbench_accounts_pkey PRIMARY KEY (aid);
```

> Recommencer la sauvegarde en compressant la sauvegarde avec `gzip`. Quelle en est la taille ?

```
$ pg_dumpall | gzip > ~postgres/backups/base_all.sql.gz
```

```
$ ls -lh ~postgres/backups/base_all.sql.gz
-rw-r--r--. 1 postgres postgres 6,0M  1 août  17:39
 ↪  /var/lib/pgsql/backups/base_all.sql.gz
```

Le taux de compression de 80 % n'est pas inhabituel, mais il dépend bien entendu énormément des données.

Sauvegarde logique d'une base

> Qu'affiche un `pg_dump -d entreprise` ?

Le résultat est un script, en clair, de création des objets de la base **entreprise**, sans ordre de création de la base ni d'utilisateur.

Notamment, les `GRANT` suppose la pré-existence des rôles adéquats.

> À l'aide de `pg_dump`, sauvegarder la base de données **pgbench** au format *custom* dans le fichier `~postgres/backups/base_pgbench.dump`. Quelle est sa taille ?

En tant qu'utilisateur **postgres**, puis exécuter la commande suivante :

```
$ pg_dump -Fc -f ~postgres/backups/base_pgbench.dump pgbench
```

La taille de 6 Mo environ indique que cette base contenait l'essentiel de la volumétrie de l'instance, et que la sauvegarde *custom* est bien compressée.

> Avec `pg_restore` et son option `-l`, consulter le contenu de `~postgres/backups/base_pgbench.dump`.

```
$ pg_restore -l ~postgres/backups/base_pgbench.dump
;
; Archive created at 2021-11-22 16:36:31 CET
;     dbname: pgbench
;     TOC Entries: 27
;     Compression: -1
;     Dump Version: 1.14-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 15.2
;     Dumped by pg_dump version: 15.2
;
;
; Selected TOC Entries:
;
200; 1259 18920 TABLE public copie1 postgres
201; 1259 18923 TABLE public copie2 postgres
202; 1259 18928 TABLE public copie3 postgres
```

```
198; 1259 18873 TABLE public pgbench_accounts testperf
199; 1259 18876 TABLE public pgbench_branches testperf
196; 1259 18867 TABLE public pgbench_history testperf
197; 1259 18870 TABLE public pgbench_tellers testperf
3712; 0 18920 TABLE DATA public copie1 postgres
3713; 0 18923 TABLE DATA public copie2 postgres
3714; 0 18928 TABLE DATA public copie3 postgres
3710; 0 18873 TABLE DATA public pgbench_accounts testperf
3711; 0 18876 TABLE DATA public pgbench_branches testperf
3708; 0 18867 TABLE DATA public pgbench_history testperf
3709; 0 18870 TABLE DATA public pgbench_tellers testperf
3579; 2606 18888 CONSTRAINT public pgbench_accounts pgbench_accounts_pkey testperf
3581; 2606 18884 CONSTRAINT public pgbench_branches pgbench_branches_pkey testperf
3577; 2606 18886 CONSTRAINT public pgbench_tellers pgbench_tellers_pkey testperf
3586; 2606 18894 FK CONSTRAINT public pgbench_accounts pgbench_accounts_bid_fkey…
3584; 2606 18909 FK CONSTRAINT public pgbench_history pgbench_history_aid_fkey…
3582; 2606 18899 FK CONSTRAINT public pgbench_history pgbench_history_bid_fkey…
3583; 2606 18904 FK CONSTRAINT public pgbench_history pgbench_history_tid_fkey…
3585; 2606 18889 FK CONSTRAINT public pgbench_tellers pgbench_tellers_bid_fkey…
```

Export des objets globaux

> Exporter uniquement les objets globaux de l'instance (rôles, définitions de tablespaces) à l'aide de `pg_dumpall` dans le fichier `~postgres/backups/base_globals.sql`.

```
$ pg_dumpall -g > ~postgres/backups/base_globals.sql
```

> Consulter le contenu de ce fichier.

Le contenu du fichier est très réduit et ne reprend que les premiers éléments du fichier créé plus haut :

```
CREATE ROLE patron;
…
CREATE ROLE testperf;
…
```

Sauvegarde logique de tables

> Sauvegarder la table `pgbench_accounts` dans le fichier `~postgres/backups/table_pgbench_accounts.sql` au format *plain text*. Que contient ce fichier ?

```
$ pg_dump -t pgbench_accounts pgbench > ~postgres/backups/table_pgbench_accounts.sql
```

Le contenu se consulte avec :

```
less  ~postgres/backups/table_pgbench_accounts.sql
```

On y trouvera entre autre les ordres de création :

```
CREATE TABLE public.pgbench_accounts (
    aid integer NOT NULL,
    bid integer,
    abalance integer,
```

```
    filler character(84)
)
WITH (fillfactor='100');
```

l'affectation du propriétaire :

```
ALTER TABLE public.pgbench_accounts OWNER TO testperf;
```

les données :

```
COPY public.pgbench_accounts (aid, bid, abalance, filler) FROM stdin;
35      1       0
38      1       -11388
18      1       7416
7       1       -1124
...
```

Tout à la fin, les contraintes :

```
ALTER TABLE ONLY public.pgbench_accounts
    ADD CONSTRAINT pgbench_accounts_pkey PRIMARY KEY (aid);

ALTER TABLE ONLY public.pgbench_accounts
    ADD CONSTRAINT pgbench_accounts_bid_fkey FOREIGN KEY (bid)
    REFERENCES public.pgbench_branches(bid);
```

7.11.2 Restaurations logiques

Restauration d'une base

> Que se passe-t-il si l'on exécute ceci ?
>
> `pg_restore ~postgres/backups/base_pgbench.dump -f -`

La sortie affiche les ordres qui seront envoyés lors de la restauration. C'est très pratique pour consulter le contenu d'une sauvegarde compressée avant de l'exécuter.

À partir de la version 12, `pg_restore` exige d'avoir soit un fichier (`-f`) soit une base (`-d`) en destination. `-f -` envoie le résultat sur la sortie standard. Pour envoyer vers un fichier, demander `-f dump.sql`, et l'on obtiendra le SQL contenu dans la sauvegarde.

Pour une simple consultation du contenu :

```
$ pg_restore ~postgres/backups/base_pgbench.dump -f - |less
```

> Créer une nouvelle base **pgbench2** appartenant au rôle **testperf**.

En tant que **postgres** :

```
$ createdb --owner testperf pgbench2
```

> Ajuster `.pgpass` et `pg_hba.conf` pour que **testperf** puisse se connecter à toute base par son mot de passe.

La connexion se fait par exemple en ajoutant le mot de passe dans le `.pgpass` si ce n'est déjà fait :

```
localhost:5432:*:testperf:860e74ea6eba6fdee4574c54aadf4f98
```

ainsi que ceci en tête du `pg_hba.conf` (ne pas oublier de recharger la configuration) :

```
local   all             testperf                                scram-sha-256
```

En tant qu'utilisateur **postgres** :

```
$ psql -U testperf -d pgbench2 -c "SELECT 'test de connexion' "
```

> Restaurer le contenu de la base de donnée **pgbench** dans **pgbench2** en utilisant le fichier de sauvegarde `base_pgbench.dump`. Toutes les tables sont-elles restaurées à l'identique ?

```
$ pg_restore -v -U testperf -d pgbench2 ~postgres/backups/base_pgbench.dump
```

L'option `-v` est optionnelle. Elle permet d'avoir le détail suivant :

```
pg_restore: connecting to database for restore
pg_restore: creating TABLE "public.copie1"
pg_restore: [archiver (db)] Error while PROCESSING TOC:
pg_restore: [archiver (db)] Error from TOC entry 200;1259 18920
                            TABLE copie1 postgres
pg_restore: [archiver (db)] could not execute query:
    ERROR:  must be member of role "postgres"
    Command was: ALTER TABLE public.copie1 OWNER TO postgres;
…
pg_restore: creating TABLE "public.pgbench_accounts"
pg_restore: creating TABLE "public.pgbench_branches"
pg_restore: creating TABLE "public.pgbench_history"
pg_restore: creating TABLE "public.pgbench_tellers"
…
pg_restore: processing data for table "public.pgbench_history"
pg_restore: processing data for table "public.pgbench_tellers"
pg_restore: creating CONSTRAINT "public.pgbench_accounts pgbench_accounts_pkey"
pg_restore: creating FK CONSTRAINT "public.pgbench_accounts
                                    pgbench_accounts_bid_fkey"
…
```

Un simple `\d+` sur les bases **pgbench** et **pgbench2** montre deux différences :

- la taille des tables n'est pas forcément la même (bien que les données soient identiques) car l'organisation sur le disque est plus compacte sur une base qui n'a pas de vécu ;
- les propriétaires des tables `copie*` ne sont pas les mêmes : en effet, on voit ci-dessus que les ordres `ALTER TABLE` échouent : **testperf** n'a pas les droits suffisants pour changer le propriétaire des tables. L'option `--no-owner` permet d'éviter ces messages.

> Puis renommer la base de données **pgbench** en **pgbench_old** et la base de données **pgbench2** en **pgbench**.

En tant qu'utilisateur **postgres** connecté à la base de même nom :

```
postgres@postgres=# ALTER DATABASE pgbench RENAME TO pgbench_old;
postgres@postgres=# ALTER DATABASE pgbench2 RENAME TO pgbench;
```

Les ordres peuvent échouer si une connexion est toujours en cours à la base.

```
postgres@postgres=# \l
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    | Access privileges
-----------+----------+----------+------------+------------+---------------------
entreprise | patron   | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
pgbench    | testperf | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
pgbench_old| postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
postgres   | postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|
template0  | postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|=c/postgres         +
           |          |          |           |           |postgres=CTc/postgres
template1  | postgres | UTF8     |en_GB.UTF-8|en_GB.UTF-8|=c/postgres         +
           |          |          |           |           |postgres=CTc/postgres
```

Restauration d'une table

> Dans la base **pgbench**, détruire la table `pgbench_history`.

```
$ psql -U testperf -d pgbench

testperf@pgbench=> \d pgbench_history
                        Table « public.pgbench_history »
 Colonne |            Type             | Collationnement | NULL-able | Par défaut
---------+-----------------------------+-----------------+-----------+------------
 tid     | integer                     |                 |           |
 bid     | integer                     |                 |           |
 aid     | integer                     |                 |           |
 delta   | integer                     |                 |           |
 mtime   | timestamp without time zone |                 |           |
 filler  | character(22)               |                 |           |
Contraintes de clés étrangères :
    "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
    "pgbench_history_bid_fkey" FOREIGN KEY (bid) REFERENCES pgbench_branches(bid)
    "pgbench_history_tid_fkey" FOREIGN KEY (tid) REFERENCES pgbench_tellers(tid)

testperf@pgbench=> DROP TABLE pgbench_history ;
DROP TABLE
```

> À partir de la sauvegarde de la base **pgbench** au format *custom* faite précédemment, restaurer uniquement la table `pgbench_history` dans la nouvelle base **pgbench**, avec `pg_restore -t`. La table est-elle complète ?

```
$ pg_restore -d pgbench -U testperf -t pgbench_history \
                                      ~postgres/backups/base_pgbench.dump
```

Les données sont bien là mais les contraintes n'ont *pas* été restaurées : en effet, `pg_restore -t` ne considère que les tables proprement dites.

```
postgres@pgbench=# \d pgbench_history
                        Table « public.pgbench_history »
 Colonne |            Type             | Collationnement | NULL-able | Par défaut
---------+-----------------------------+-----------------+-----------+------------
 tid     | integer                     |                 |           |
 bid     | integer                     |                 |           |
 aid     | integer                     |                 |           |
 delta   | integer                     |                 |           |
 mtime   | timestamp without time zone |                 |           |
 filler  | character(22)               |                 |           |
```

> Rechercher les morceaux manquants dans la section `post-data` de la sauvegarde.

La manière la plus propre de récupérer une table depuis une sauvegarde est la technique avec `-L` décrite plus loin.

Une autre possibilité est d'avoir précédemment écrit un script de sauvegarde table à table comme ci-dessus. `pg_dump -t`, lui, sauvegarde les contraintes des tables.

Une autre possibilité est de retrouver les ordres manquants dans la sauvegarde. Ils sont dans la section `post-data` :

```
$ pg_restore --section=post-data  ~postgres/backups/base_pgbench.dump -f -
```

On a donc une version en SQL de la fin d'une sauvegarde, et on peut y piocher les trois ordres `ALTER TABLE ... ADD CONSTRAINT ...` concernant la table.

Migration de données

> Dans une fenêtre, lancer une instance `pgbench` sur la base **pgbench**
>
> `/usr/pgsql-14/bin/pgbench -U testperf -T 600 pgbench`

Cet ordre va générer de l'activité sur la base pendant la sauvegarde suivante.

> Pour dupliquer la base **pbench** :
>
> - Créer une nouvelle base **pgbench3** appartenant à **testperf**.
> - Avec `pg_dump`, copier les données de la base **pgbench** dans cette nouvelle base **pgbench3** sans passer par un fichier de sauvegarde, avec un pipe.

```
$ createdb -O testperf  pgbench3
$ pg_dump -Fp  -U testperf pgbench | psql -U testperf -d pgbench3
```

Cet ordre ne doit pas générer d'erreur bien que les données soient modifiées pendant le backup. L'état des données restaurées sera celui au moment du début du `pg_dump`.

Le format *custom* est possible aussi, mais il utilise par défaut une compression ici inutile :

```
$ pg_dump -Fc -Z0  -U testperf pgbench | pg_restore -U testperf -d pgbench3
```

7.11.3 Sauvegarde et restauration partielle

> Il faut restaurer dans une base **pgbench4** tout le contenu de la sauvegarde de la base **pgbench**, sauf les *données* de la table `pgbench_accounts` ni les tables `copie*`. La définition de la table `pgbench_accounts` et toutes les contraintes s'y rapportant doivent être restaurées. Pour cela :
>
> - utiliser les options `-l` de `pg_restore` pour créer le fichier des objets à charger ;
> - adapter ce fichier en fonction des critères ci-dessus ;
> - utiliser ce fichier grâce à l'option `-L` de `pg_restore`.

On procède en deux étapes :

- lister le contenu de l'archive dans un fichier :

```
$ pg_restore -l ~postgres/backups/base_pgbench.dump > /tmp/contenu_archive
```

- dans `/tmp/contenu_archive`, supprimer ou commenter la ligne `TABLE DATA` de la table `pgbench_accounts` et tout ce qui peut concerner des tables `copie*` :

```
;200; 1259 18920 TABLE public copie1 postgres
;201; 1259 18923 TABLE public copie2 postgres
;202; 1259 18928 TABLE public copie3 postgres
...
;3712; 0 18920 TABLE DATA public copie1 postgres
;3713; 0 18923 TABLE DATA public copie2 postgres
;3714; 0 18928 TABLE DATA public copie3 postgres
;3710; 0 18873 TABLE DATA public pgbench_accounts testperf
```

- Créer la base de données **pgbench4** appartenant à **testperf**, puis restaurer en utilisant ce fichier :

```
$ createdb -O testperf  pgbench4
$ pg_restore -U testperf -d pgbench4 -L /tmp/contenu_archive \
                                        ~postgres/backups/base_pgbench.dump
```

S'il y a une contrainte vers la table `pgbench_accounts`, il est alors normal qu'elle échoue. Il y aura simplement un message d'erreur. Le plus propre aurait été de l'exclure d'entrée dans le fichier plus haut.

7.11.4 (Optionnel) Sauvegarde et restauration par parties avec modification

> Effectuer une sauvegarde de **pgbench_old**, puis une restauration vers une nouvelle base **test**, lors de laquelle on ne rechargera que les tables `pgbench_*` en les renommant en `test_*`. Pour cela :
>
> - créer plusieurs fichiers (format *plain*) grâce aux options `--section=` de `pg_dump` ;
> - modifier les > fichiers obtenus ;

> - les charger ;
> - vérifier que les tables sont bien là avec les nouveaux noms.

On peut séparer la sauvegarde en 3 fichiers texte :

```
$ pg_dump -Fp -U testperf -d pgbench_old -t 'pgbench_*' --section=pre-data \
                                    -f ~postgres/backups/base_pgbench_old-1.sql
$ pg_dump -Fp -U testperf -d pgbench_old -t 'pgbench_*' --section=data \
                                    -f ~postgres/backups/base_pgbench_old-2.sql
$ pg_dump -Fp -U testperf -d pgbench_old -t 'pgbench_*' --section=post-data \
                                    -f ~postgres/backups/base_pgbench_old-3.sql
```

Le premier contient les déclarations d'objets. Supprimer les tables non voulues s'il y en a. Renommer les tables `pgbench_*` en `test_*`.

Procéder de même dans le troisième fichier, qui contient les contraintes et les index.

Le second contient les données elles-mêmes. Modifier les noms de table ici aussi. Heureusement le fichier est encore assez peu volumineux pour permettre une modification manuelle.

Noter que ces fichiers ne forment une sauvegarde cohérente que s'il n'y a pas eu de modification pendant leur création.

Restaurer les fichiers l'un après l'autre :

```
$ createdb -O testperf test

$ psql -U testperf -d test < ~postgres/backups/base_pgbench_old-1.sql
...
$ psql -U testperf -d test < ~postgres/backups/base_pgbench_old-2.sql
...
COPY 100000
COPY 1
COPY 38051
COPY 10

$ psql -U testperf -d test < ~postgres/backups/base_pgbench_old-3.sql
...
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

Vérifier le résultat :

```
$ psql -d test -c '\d+'
                           List of relations
 Schema |      Name       | Type  |  Owner   |     Size     | Description
--------+-----------------+-------+----------+--------------+-------------
 public | test_accounts   | table | testperf | 13 MB        |
 public | test_branches   | table | testperf | 8192 bytes   |
 public | test_history    | table | testperf | 1968 kB      |
 public | test_tellers    | table | testperf | 8192 bytes   |
```

> Recommencer avec une sauvegarde des données en format *custom* en scriptant les modifications à la volée.

Jusque là il n'y a pas une grande différence avec une restauration depuis un fichier de sauvegarde unique. Les sections sont plus intéressantes s'il y a plus de données et qu'il n'est pas réaliste de les stocker sur le disque dans un fichier texte. Le deuxième fichier sera donc au format *custom* ou *tar* :

```
$ pg_dump -Fc -U testperf -d pgbench_old -t 'pgbench_*' --section=data \
                                -f ~postgres/backups/base_pgbench_old-2.dump
```

Pour le restaurer, il faudra modifier les ordres `COPY` à la volée en sortie de `pg_restore`, sortie qui est réinjectée ensuite dans la cible avec `psql`. Le script ressemble par exemple à ceci :

```
$ pg_restore ~postgres/backups/base_pgbench_old-2.dump -f - | \
    awk '/^COPY/ {gsub("COPY public.pgbench_", "COPY public.test_") ; print }
         ! /^COPY/ {print} ' | \
    psql -U testperf -d test
```

La technique est utilisable pour toute manipulation des données par script. Pour éviter tout fichier intermédiaire, on peut même appliquer la technique à la sortie d'un `pg_dump -Fp --section=data`.

7.11.5 (Optionnel) Sauvegardes d'objets isolés

> On veut sauvegarder les tables toutes les tables `pgbench_*` dans des fichiers séparés :
>
> - Créer un script shell nommé `sauvegarde_tables.sh`
> - il utilisera `psql` pour lister les tables concernées depuis la vue `pg_tables` ou `\d+`, dans la base **pgbench** ;
> - pour simplifier la sortie : utiliser les options `-A`, `-t`, `-F` de `psql` ;
> - enfin, boucler sur ces tables pour lancer un `pg_dump`, au format *plain text*, sur chaque table, vers des fichiers nommés `table_<nom_table>.sql`.

En tant qu'utilisateur système **postgres**, créer le script avec un éditeur de texte comme `vi` :

```
$ vi ~postgres/backups/sauvegarde_tables.sh
```

Le script peut ressemble à ceci :

```
#!/bin/bash
set -xue
export PGDATABASE=pgbench
export PGUSER=postgres
export PGPORT=5432
export PGHOST=/var/run/postgresql
for t in $(psql -XAt -F'.' -c "SELECT schemaname, tablename
                               FROM pg_tables
                               WHERE schemaname = 'public'
                               AND tablename LIKE 'pgbench_%' " )
```

```
do
  pg_dump -t "$t" pgbench > ~postgres/backups/table_$t.sql
done
```

Les options de `psql` sont importantes pour avoir une sortie propre :

- `-X` court-circuite un éventuel `~/.psqlrc`, qui peut contenir beaucoup de choses pouvant polluer l'affichage ;
- `-c` permet de fournir la commande SQL ;
- `-A` simplifie l'affichate *unaligned*) ;
- `-t` réduit l'affichage aux lignes (*tuples*) ;
- `-F` permet de préciser un séparateur, ci-dessus le point entre les deux champs.

Une variante utilise `\dt` et analyse la sortie avec `awk` :

```
#!/bin/bash
set -xue  # pour débogage et arrêt immédiat en cas d'erreur
LIST_TABLES=$(psql -XAt -c '\dt public.pgbench_*' -d pgbench | \
              awk -F'|' '{print $2}')
for t in $LIST_TABLES ; do
  echo "Sauvegarde $t..."
  pg_dump -t $t pgbench > ~postgres/backups/table_${t}.sql
done
```

NB : En production, il faudra bien sûr une gestion d'erreur.

Enfin, il faut donner les droits d'exécution au script :

```
$ chmod +x ~postgres/backups/sauvegarde_tables.sh
```

Et on teste :

```
$ ~postgres/backups/sauvegarde_tables.sh
```

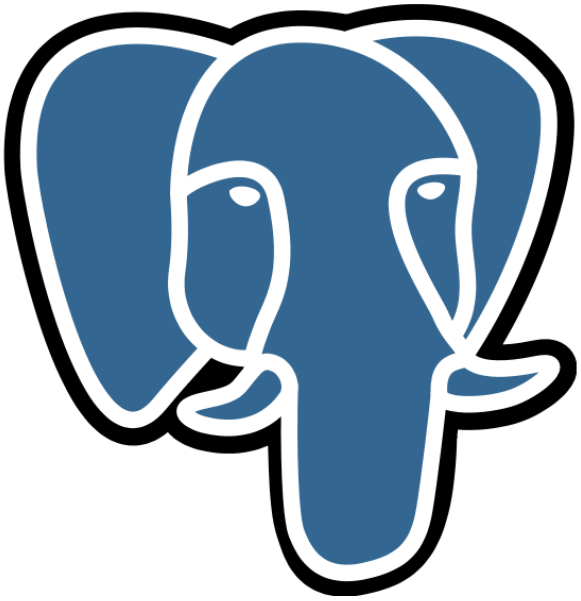
> Quel est le problème de principe de ces sauvegardes séparées sur une base active ?

Ce genre de script possède un problème fondamental : les sauvegardes sont faites séparément et la cohérence entre les tables exportées n'est plus du tout garantie !

Cela peut réserver de très mauvaises surprises à la restauration. `pg_dump` pare à cela en utilisant une transaction en *repeatable read*.

Pour des exports d'objets isolés, il est donc conseillé d'utiliser plutôt un même ordre `pg_dump` avec plusieurs paramètres `-t`. Noter que `pg_dump` ne peut garantir l'intégrité des données par rapport aux objets non exportés de la base.

8/ Architecture & fichiers de PostgreSQL



8.1 AU MENU

- Rappels sur l'installation
- Les processus
- Les fichiers

Le présent module vise à donner un premier aperçu global du fonctionnement interne de PostgreSQL.

Après quelques rappels sur l'installation, nous verrons essentiellement les processus et les fichiers utilisés.

8.2 RAPPELS SUR L'INSTALLATION

- Plusieurs possibilités
  - paquets Linux précompilés
  - outils externes d'installation
  - code source
- Chacun ses avantages et inconvénients
  - Dalibo recommande fortement les paquets précompilés

Nous recommandons très fortement l'utilisation des paquets Linux précompilés. Dans certains cas, il ne sera pas possible de faire autrement que de passer par des outils externes, comme l'installeur d'EntrepriseDB sous Windows.

8.2.1 Paquets précompilés

- Paquets Debian ou Red Hat suivant la distribution utilisée
- Préférence forte pour ceux de la communauté
- Installation du paquet
  - installation des binaires
  - création de l'utilisateur postgres
  - initialisation d'une instance (Debian seulement)
  - lancement du serveur (Debian seulement)
- (Red Hat) Script de création de l'instance

Debian et Red Hat fournissent des paquets précompilés adaptés à leur distribution. Dalibo recommande d'installer les paquets de la communauté, ces derniers étant bien plus à jour que ceux des distributions.

L'installation d'un paquet provoque la création d'un utilisateur système nommé postgres et l'installation des binaires. Suivant les distributions, l'emplacement des binaires change. Habituellement, tout est placé dans `/usr/pgsql-<version majeure>` pour les distributions Red Hat et dans `/usr/lib/postgresql/<version majeure>` pour les distributions Debian.

Dans le cas d'une distribution Debian, une instance est immédiatement créée dans `/var/lib/postgresql/<version` Elle est ensuite démarrée.

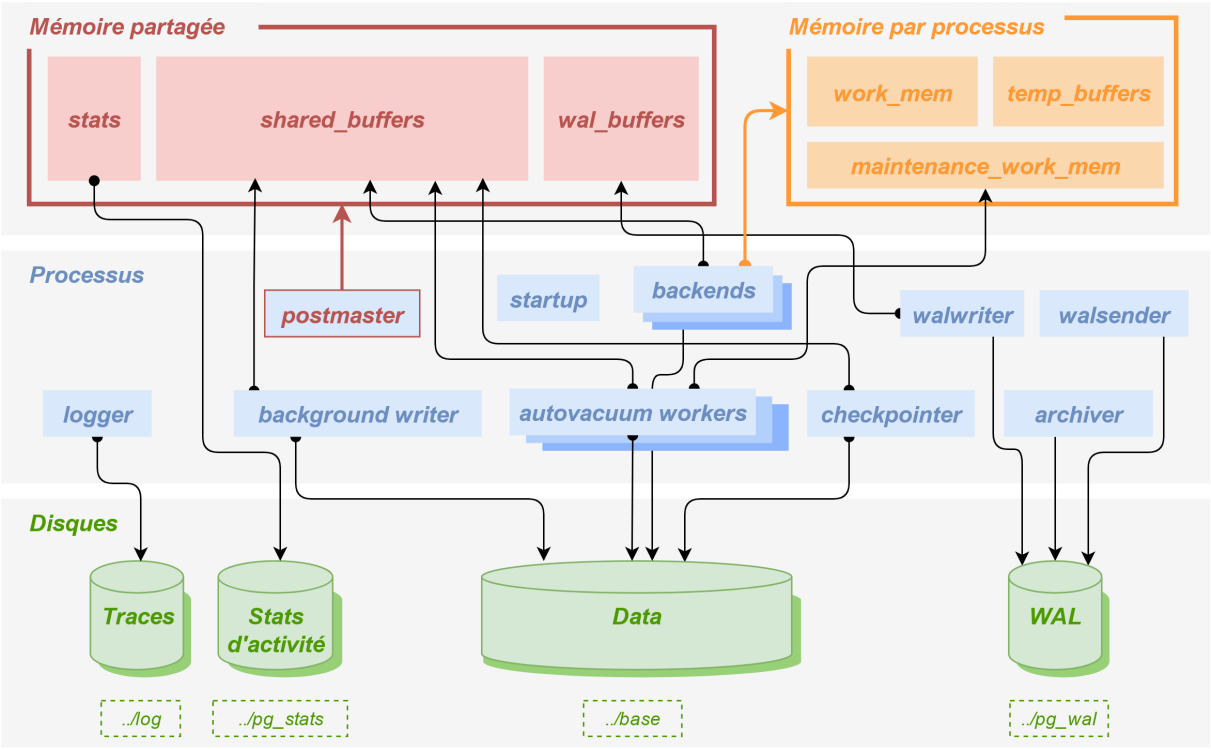
Dans le cas d'une distribution Red Hat, aucune instance n'est créée automatiquement. Il faudra utiliser un script (dont le nom dépend de la version de la distribution) pour créer l'instance, puis nous pourrons utiliser le script de démarrage pour lancer le serveur.

8.2.2 Installons PostgreSQL

- Prenons un moment pour
  - installer PostgreSQL
  - créer une instance
  - démarrer l'instance
- Pas de configuration spécifique pour l'instant

L'annexe ci-dessous décrit l'installation de PostgreSQL sans configuration particulière pour suivre le reste de la formation.

8.3 PROCESSUS DE POSTGRESQL



8.3.1 Introduction

- PostgreSQL est :
  - multiprocessus (et non multithread)
  - à mémoire partagée
  - client-serveur

L'architecture PostgreSQL est une architecture multiprocessus et non multithread.

Cela signifie que chaque processus de PostgreSQL s'exécute dans un contexte mémoire isolé, et que la communication entre ces processus repose sur des mécanismes systèmes inter-processus : sémaphores, zones de mémoire partagée, sockets. Ceci s'oppose à l'architecture multithread, où l'ensemble du moteur s'exécute dans un seul processus, avec plusieurs threads (contextes) d'exécution, où tout est partagé par défaut.

Le principal avantage de cette architecture multiprocessus est la stabilité : un processus, en cas de problème, ne corrompt que sa mémoire (ou la mémoire partagée), le plantage d'un processus n'affecte

pas directement les autres. Son principal défaut est une allocation statique des ressources de mémoire partagée : elles ne sont pas redimensionnables à chaud.

Tous les processus de PostgreSQL accèdent à une zone de « mémoire partagée ». Cette zone contient les informations devant être partagées entre les clients, comme un cache de données, ou des informations sur l'état de chaque session par exemple.

PostgreSQL utilise une architecture client-serveur. Nous ne nous connectons à PostgreSQL qu'à travers d'un protocole bien défini, nous n'accédons jamais aux fichiers de données.

8.3.2 Processus d'arrière-plan

```
# ps f -e --format=pid,command | grep -E "postgres|postmaster"
  96122 /usr/pgsql-17/bin/postmaster -D /var/lib/pgsql/15/data/
  96123  \_ postgres: logger
  96125  \_ postgres: checkpointer
  96126  \_ postgres: background writer
  96127  \_ postgres: walwriter
  96128  \_ postgres: autovacuum launcher
  96131  \_ postgres: logical replication launcher
```

(sous Rocky Linux 8)

Nous constatons que plusieurs processus sont présents dès le démarrage de PostgreSQL. Nous allons les détailler.

NB : sur Debian, le postmaster est nommé *postgres* comme ses processus fils ; sous Windows les noms des processus sont par défaut moins verbeux.

8.3.3 Processus d'arrière-plan (suite)

- Les processus présents au démarrage :
  - Un processus père : `postmaster`
  - `background writer`
  - `checkpointer`
  - `walwriter`
  - `autovacuum launcher`
  - `stats collector` (avant v15)
  - `logical replication launcher`
- et d'autres selon la configuration et le moment :
  - dont les *background workers* : parallélisation, extensions…

Le `postmaster` est responsable de la supervision des autres processus, ainsi que de la prise en compte des connexions entrantes.

Le `background writer` et le `checkpointer` s'occupent d'effectuer les écritures en arrière plan, évitant ainsi aux sessions des utilisateurs de le faire.

Le `walwriter` écrit le journal de transactions de façon anticipée, afin de limiter le travail de l'opération `COMMIT`.

L'`autovacuum launcher` pilote les opérations d'« autovacuum ».

Avant la version 15, le `stats collector` collecte les statistiques d'activité du serveur. À partir de la version 15, ces informations sont conservées en mémoire jusqu'à l'arrêt du serveur où elles sont stockées sur disque jusqu'au prochain démarrage.

Le `logical replication launcher` est un processus dédié à la réplication logique.

Des processus supplémentaires peuvent apparaître, comme un `walsender` dans le cas où la base est le serveur primaire du cluster de réplication, un `logger` si PostgreSQL doit gérer lui-même les fichiers de traces (par défaut sous Red Hat, mais pas sous Debian), ou un `archiver` si l'instance est paramétrée pour générer des archives de ses journaux de transactions.

Ces différents processus seront étudiées en détail dans d'autres modules de formation.

Aucun de ces processus ne traite de requête pour le compte des utilisateurs. Ce sont des processus d'arrière-plan effectuant des tâches de maintenance.

Il existe aussi les *background workers* (processus d'arrière-plan), lancés par PostgreSQL, mais aussi par des extensions tierces. Par exemple, la parallélisation des requêtes se base sur la création tem-

poraire de *background workers* épaulant le processus principal de la requête. La réplication logique utilise des *background workers* à plus longue durée de vie. De nombreuses extensions en utilisent pour des raisons très diverses. Le paramètre `max_worker_processes` régule le nombre de ces *workers*. Ne descendez pas en-dessous du défaut (8). Il faudra même parfois monter plus haut.

8.4 PROCESSUS PAR CLIENT (CLIENT BACKEND)

- Pour chaque client, nous avons un processus :
  - créé à la connexion
  - dédié au client…
  - …et qui dialogue avec lui
  - détruit à la déconnexion
- Un processus gère une requête
  - peut être aidé par d'autres processus
- Le nombre de processus est régi par les paramètres :
  - `max_connections` (défaut : 100) - connexions réservées
  - compromis nombre requêtes actives/nombre cœurs/complexité/mémoire

Pour chaque nouvelle session à l'instance, le processus `postmaster` crée un processus fils qui s'occupe de gérer cette session. Il y a donc un processus dédié à chaque connexion cliente, et ce processus est détruit à fin de cette connexion.

Ce processus dit *backend* reçoit les ordres SQL, les interprète, exécute les requêtes, trie les données, et enfin retourne les résultats. Dans certains cas, il peut demander le lancement d'autres processus pour l'aider dans l'exécution d'une requête en lecture seule (parallélisme).

Le dialogue entre le client et ce processus respecte un protocole réseau bien défini. Le client n'a jamais accès aux données par un autre moyen que par ce protocole.

Le nombre maximum de connexions à l'instance simultanées, actives ou non, est limité par le paramètre `max_connections`. Le défaut est 100.

Certaines connexions sont réservées. Les administrateurs doivent pouvoir se connecter à l'instance même si la limite est atteinte. Quelques connexions sont donc réservées aux superutilisateurs (paramètre `superuser_reserved_connections`, à 3 par défaut). On peut aussi octroyer le rôle `pg_use_reserved_connections` à certains utilisateurs pour leur garantir l'accès à un nombre de connexions réservées, à définir avec le paramètre `reserved_connections` (vide par défaut), cela à partir de PostreSQL 16.

Attention : modifier un de ces paramètres impose un redémarrage complet de l'instance, puisqu'ils ont un impact sur la taille de la mémoire partagée entre les processus PostgreSQL. Il faut donc réfléchir au bon dimensionnement avant la mise en production.

La valeur 100 pour `max_connections` est généralement suffisante. Il peut être intéressant de la diminuer pour se permettre de monter `work_mem` et autoriser plus de mémoire de tri. Il est possible de

monter `max_connections` pour qu'un plus grand nombre de clients puisse se connecter en même temps.

Il s'agit aussi d'arbitrer entre le nombre de requêtes à exécuter à un instant t, le nombre de CPU disponibles, la complexité des requêtes, et le nombre de processus que peut gérer l'OS. Ce dimensionnement est encore compliqué par le parallélisme et la limitation de la bande passante des disques.

Intercaler un « pooler » entre les clients et l'instance peut se justifier dans certains cas :

- connexions/déconnexions très fréquentes (la connexion a un coût) ;
- centaines, voire milliers, de connexions généralement inactives.

Le plus réputé est PgBouncer, mais il est aussi souvent inclus dans des serveurs d'application (par exemple Tomcat).

8.5 GESTION DE LA MÉMOIRE

Structure de la mémoire sous PostgreSQL

- Zone de mémoire partagée :
  - *shared buffers* surtout
  - …
- Zone de chaque processus
  - tris en mémoire (`work_mem`)
  - …

La gestion de la mémoire dans PostgreSQL mérite un module de formation à lui tout seul.

Pour le moment, bornons-nous à la séparer en deux parties : la mémoire partagée et celle attribuée à chacun des nombreux processus.

La mémoire partagée stocke principalement le cache des données de PostgreSQL (*shared buffers*, paramètre `shared_buffers`), et d'autres zones plus petites : cache des journaux de transactions, données de sessions, les verrous, etc.

La mémoire propre à chaque processus sert notamment aux tris en mémoire (définie en premier lieu par le paramètre `work_mem`), au cache de tables temporaires, etc.

8.6 FICHIERS

- Une instance est composée de fichiers :
  - Répertoire de données
  - Fichiers de configuration
  - Fichier PID
  - Tablespaces
  - Statistiques
  - Fichiers de trace

Une instance est composée des éléments suivants :

Le répertoire de données :

Il contient les fichiers obligatoires au bon fonctionnement de l'instance : fichiers de données, journaux de transaction….

Les fichiers de configuration :

Selon la distribution, ils sont stockés dans le répertoire de données (Red Hat et dérivés comme CentOS ou Rocky Linux), ou dans `/etc/postgresql` (Debian et dérivés).

Un fichier PID :

Il permet de savoir si une instance est démarrée ou non, et donc à empêcher un second jeu de processus d'y accéder.

Le paramètre `external_pid_file` permet d'indiquer un emplacement où PostgreSQL créera un second fichier de PID, généralement à l'extérieur de son répertoire de données.

Des tablespaces :

Ils sont totalement optionnels. Ce sont des espaces de stockage supplémentaires, stockés habituellement dans d'autres systèmes de fichiers.

Le fichier de statistiques d'exécution :

Généralement dans `pg_stat_tmp/`.

Les fichiers de trace :

Typiquement, des fichiers avec une variante du nom `postgresql.log`, souvent datés. Ils sont par défaut dans le répertoire de l'instance, sous `log/`. Sur Debian, ils sont redirigés vers la sortie d'erreur du système. Ils peuvent être redirigés vers un autre mécanisme du système d'exploitation (syslog sous Unix, journal des événements sous Windows),

8.6.1 Répertoire de données

```
postgres$ ls $PGDATA
base               pg_ident.conf  pg_stat      pg_xact
current_logfiles   pg_logical     pg_stat_tmp  postgresql.auto.conf
global             pg_multixact   pg_subtrans  postgresql.conf
log                pg_notify      pg_tblspc    postmaster.opts
pg_commit_ts       pg_replslot    pg_twophase  postmaster.pid
pg_dynshmem        pg_serial      PG_VERSION
pg_hba.conf        pg_snapshots   pg_wal
```

- Une seule instance PostgreSQL doit y accéder !

Le répertoire de données est souvent appelé PGDATA, du nom de la variable d'environnement que l'on peut faire pointer vers lui pour simplifier l'utilisation de nombreux utilitaires PostgreSQL. Il est possible aussi de le connaître, une fois connecté à une base de l'instance, en interrogeant le paramètre `data_directory`.

```
SHOW data_directory;

      data_directory
---------------------------
 /var/lib/pgsql/15/data
```

Ce répertoire ne doit être utilisé que par une seule instance (processus) à la fois ! PostgreSQL vérifie au démarrage qu'aucune autre instance du même serveur n'utilise les fichiers indiqués, mais cette protection n'est pas absolue, notamment avec des accès depuis des systèmes différents (ou depuis un conteneur comme docker).
Faites donc bien attention de ne lancer PostgreSQL qu'une seule fois sur un répertoire de données.

Il est recommandé de ne jamais créer ce répertoire PGDATA à la racine d'un point de montage, quel que soit le système d'exploitation et le système de fichiers utilisé. Si un point de montage est dédié à l'utilisation de PostgreSQL, positionnez-le toujours dans un sous-répertoire, voire deux niveaux en dessous du point de montage. (par exemple `<point de montage>/<version majeure>/<nom instance>`).

Voir à ce propos le chapitre *Use of Secondary File Systems* dans la documentation officielle : https://www.postgresql.org/docs/current/creating-cluster.html.

Vous pouvez trouver une description de tous les fichiers et répertoires dans la documentation officielle[1].

[1] https://www.postgresql.org/docs/current/static/storage-file-layout.html

8.6.2 Fichiers de configuration

- `postgresql.conf` (+ fichiers inclus)
- `postgresql.auto.conf`
- `pg_hba.conf` (+ fichiers inclus (v16))
- `pg_ident.conf` (idem)

Les fichiers de configuration sont de simples fichiers textes. Habituellement, ce sont les suivants.

`postgresql.conf` contient une liste de paramètres, sous la forme `paramètre=valeur`. Tous les paramètres sont modifiables (et présents) dans ce fichier. Selon la configuration, il peut inclure d'autres fichiers, mais ce n'est pas le cas par défaut. Sous Debian, il est sous `/etc`.

`postgresql.auto.conf` stocke les paramètres de configuration fixés en utilisant la commande `ALTER SYSTEM`. Il surcharge donc `postgresql.conf`. Comme pour `postgresql.conf`, sa modification impose de recharger la configuration ou redémarrer l'instance. Il est techniquement possible, mais déconseillé, de le modifier à la main. `postgresql.auto.conf` est toujours dans le répertoire des données, même si `postgresql.conf` est ailleurs.

`pg_hba.conf` contient les règles d'authentification à la base selon leur identité, la base, la provenance, etc.

`pg_ident.conf` est plus rarement utilisé. Il complète `pg_hba.conf`, par exemple pour rapprocher des utilisateurs système ou propres à PostgreSQL.

Ces deux derniers fichiers peuvent eux-mêmes inclure d'autres fichiers (depuis PostgreSQL 16). Leur utilisation est décrite dans notre première formation[2].

8.6.3 Autres fichiers dans PGDATA

- `PG_VERSION` : fichier contenant la version majeure de l'instance
- `postmaster.pid`
  - nombreuses informations sur le processus père
  - fichier externe possible, paramètre `external_pid_file`
- `postmaster.opts`

[2] https://dali.bo/f_html

`PG_VERSION` est un fichier. Il contient en texte lisible la version majeure devant être utilisée pour accéder au répertoire (par exemple `15`). On trouve ces fichiers `PG_VERSION` à de nombreux endroits de l'arborescence de PostgreSQL, par exemple dans chaque répertoire de base du répertoire `PGDATA/base/` ou à la racine de chaque tablespace.

Le fichier `postmaster.pid` est créé au démarrage de PostgreSQL. PostgreSQL y indique le PID du processus père sur la première ligne, l'emplacement du répertoire des données sur la deuxième ligne et la date et l'heure du lancement de postmaster sur la troisième ligne ainsi que beaucoup d'autres informations. Par exemple :

```
~$ cat /var/lib/postgresql/15/data/postmaster.pid
7771
/var/lib/postgresql/15/data
1503584802
5432
/tmp
localhost
  5432001  54919263
ready

$ ps -HFC postgres
UID  PID    SZ     RSS PSR STIME TIME   CMD
pos 7771 0 42486 16536   3 16:26 00:00  /usr/local/pgsql/bin/postgres
                                          -D /var/lib/postgresql/15/data
pos 7773 0 42486  4656   0 16:26 00:00  postgres: checkpointer
pos 7774 0 42486  5044   1 16:26 00:00  postgres: background writer
pos 7775 0 42486  8224   1 16:26 00:00  postgres: walwriter
pos 7776 0 42850  5640   1 16:26 00:00  postgres: autovacuum launcher
pos 7777 0 42559  3684   0 16:26 00:00  postgres: logical replication launcher

$ ipcs -p |grep  7771
54919263   postgres   7771       10640

$ ipcs  | grep 54919263
0x0052e2c1 54919263   postgres   600        56         6
```

Le processus père de cette instance PostgreSQL a comme PID le 7771. Ce processus a bien réclamé une sémaphore d'identifiant 54919263. Cette sémaphore correspond à des segments de mémoire partagée pour un total de 56 octets. Le répertoire de données se trouve bien dans `/var/lib/postgresql/15/data`.

Le fichier `postmaster.pid` est supprimé lors de l'arrêt de PostgreSQL. Cependant, ce n'est pas le cas après un arrêt brutal. Dans ce genre de cas, PostgreSQL détecte le fichier et indique qu'il va malgré tout essayer de se lancer s'il ne trouve pas de processus en cours d'exécution avec ce PID. Un fichier supplémentaire peut être créé ailleurs grâce au paramètre `external_pid_file`, c'est notamment le défaut sous Debian :

```
external_pid_file = '/var/run/postgresql/15-main.pid'
```

Par contre, ce fichier ne contient que le PID du processus père.

Quant au fichier `postmaster.opts`, il contient les arguments en ligne de commande correspondant au dernier lancement de PostgreSQL. Il n'est jamais supprimé. Par exemple :

```
$ cat $PGDATA/postmaster.opts
/usr/local/pgsql/bin/postgres "-D" "/var/lib/postgresql/15/data"
```

8.6.4 Fichiers de données

- `base/` : contient les fichiers de données
  - un sous-répertoire par base de données
  - `pgsql_tmp` : fichiers temporaires
- `global/` : contient les objets globaux à toute l'instance

`base/` contient les fichiers de données (tables, index, vues matérialisées, séquences). Il contient un sous-répertoire par base, le nom du répertoire étant l'OID de la base dans `pg_database`. Dans ces répertoires, nous trouvons un ou plusieurs fichiers par objet à stocker. Ils sont nommés ainsi :

- Le nom de base du fichier correspond à l'attribut `relfilenode` de l'objet stocké, dans la table `pg_class` (une table, un index…). Il peut changer dans la vie de l'objet (par exemple lors d'un `VACUUM FULL`, un `TRUNCATE`…)
- Si le nom est suffixé par un « . » suivi d'un chiffre, il s'agit d'un fichier d'extension de l'objet : un objet est découpé en fichiers de 1 Go maximum.
- Si le nom est suffixé par `_fsm`, il s'agit du fichier stockant la *Free Space Map* (liste des blocs réutilisables).
- Si le nom est suffixé par `_vm`, il s'agit du fichier stockant la *Visibility Map* (liste des blocs intégralement visibles, et donc ne nécessitant pas de traitement par `VACUUM`).

Un fichier `base/1247/14356.1` est donc le second segment de l'objet ayant comme `relfilenode` 14356 dans le catalogue `pg_class`, dans la base d'OID 1247 dans la table `pg_database`.

Savoir identifier cette correspondance ne sert que dans des cas de récupération de base très endommagée. Vous n'aurez jamais, durant une exploitation normale, besoin d'obtenir cette correspondance. Si, par exemple, vous avez besoin de connaître la taille de la table `test` dans une base, il vous suffit d'exécuter la fonction `pg_table_size()`. En voici un exemple complet :

```
CREATE TABLE test (id integer);
INSERT INTO test SELECT generate_series(1, 5000000);
SELECT pg_table_size('test');

 pg_table_size
---------------
     181305344
```

Depuis la ligne de commande, il existe un utilitaire nommé `oid2name`, dont le but est de faire la liaison entre le nom de fichier et le nom de l'objet PostgreSQL. Il a besoin de se connecter à la base :

```
$ pwd
/var/lib/pgsql/15/data/base/16388

$ /usr/pgsql-17/bin/oid2name -f 16477 -d employes
From database "employes":
  Filenode         Table Name
----------------------------
     16477  employes_big_pkey
```

Le répertoire `base` peut aussi contenir un répertoire `pgsql_tmp`. Ce répertoire contient des fichiers temporaires utilisés pour stocker les résultats d'un tri ou d'un hachage. À partir de la version 12, il est possible de connaître facilement le contenu de ce répertoire en utilisant la fonction `pg_ls_tmpdir()`, ce qui peut permettre de suivre leur consommation.

Si nous demandons au sein d'une première session :

```
SELECT * FROM generate_series(1,1e9) ORDER BY random() LIMIT 1 ;
```

alors nous pourrons suivre les fichiers temporaires depuis une autre session :

```
SELECT * FROM pg_ls_tmpdir() ;
```

```
       name         |    size    |      modification
--------------------+------------+------------------------
 pgsql_tmp12851.16 | 1073741824 | 2020-09-02 15:43:27+02
 pgsql_tmp12851.11 | 1073741824 | 2020-09-02 15:42:32+02
 pgsql_tmp12851.7  | 1073741824 | 2020-09-02 15:41:49+02
 pgsql_tmp12851.5  | 1073741824 | 2020-09-02 15:41:29+02
 pgsql_tmp12851.9  | 1073741824 | 2020-09-02 15:42:11+02
 pgsql_tmp12851.0  | 1073741824 | 2020-09-02 15:40:36+02
 pgsql_tmp12851.14 | 1073741824 | 2020-09-02 15:43:06+02
 pgsql_tmp12851.4  | 1073741824 | 2020-09-02 15:41:19+02
 pgsql_tmp12851.13 | 1073741824 | 2020-09-02 15:42:54+02
 pgsql_tmp12851.3  | 1073741824 | 2020-09-02 15:41:09+02
 pgsql_tmp12851.1  | 1073741824 | 2020-09-02 15:40:47+02
 pgsql_tmp12851.15 | 1073741824 | 2020-09-02 15:43:17+02
 pgsql_tmp12851.2  | 1073741824 | 2020-09-02 15:40:58+02
 pgsql_tmp12851.8  | 1073741824 | 2020-09-02 15:42:00+02
 pgsql_tmp12851.12 | 1073741824 | 2020-09-02 15:42:43+02
 pgsql_tmp12851.10 | 1073741824 | 2020-09-02 15:42:21+02
 pgsql_tmp12851.6  | 1073741824 | 2020-09-02 15:41:39+02
 pgsql_tmp12851.17 |  546168976 | 2020-09-02 15:43:32+02
```

Le répertoire `global/` contient notamment les objets globaux à toute une instance, comme la table des bases de données, celle des rôles ou celle des tablespaces ainsi que leurs index.

8.6.5 Fichiers liés aux transactions

- `pg_wal/` : journaux de transactions
  - sous-répertoire `archive_status`
  - nom : *timeline*, journal, segment
  - ex : `00000002 00000142 000000FF`
  - créés, initialisés à zéro, recyclés
- `pg_xact/` : état des transactions
- mais aussi : `pg_commit_ts/`, `pg_multixact/`, `pg_serial/` `pg_snapshots/`, `pg_subtrans/`, `pg_twophase/`
- **Ces fichiers sont vitaux !**

Le répertoire `pg_wal` contient les journaux de transactions. Ces journaux garantissent la durabilité des données dans la base, en traçant toute modification devant être effectuée **AVANT** de l'effectuer réellement en base.

Les fichiers contenus dans `pg_wal` ne doivent **jamais** être effacés manuellement. Ces fichiers sont cruciaux au bon fonctionnement de la base. PostgreSQL gère leur création et suppression. S'ils sont toujours présents, c'est que PostgreSQL en a besoin.

Par défaut, les fichiers des journaux font tous 16 Mo. Ils ont des noms sur 24 caractères, comme par exemple :

```
$ ls -l
total 2359320
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001420000007C
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001420000007D
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001420000007E
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001420000007F
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000014300000000
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000014300000001
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000014300000002
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000014300000003
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001430000001D
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001430000001E
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001430000001F
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000014300000020
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000021
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000022
```

```
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000023
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000024
drwx------ 2 postgres postgres    16384 Mar 26 16:28 archive_status
```

La première partie d'un nom de fichier (ici `00000002`) correspond à la *timeline* (« ligne de temps »), qui ne s'incrémente que lors d'une restauration de sauvegarde ou une bascule entre serveurs primaire et secondaire. La deuxième partie (ici `00000142` puis `00000143`) correspond au numéro de journal à proprement parler, soit un ensemble de fichiers représentant 4 Go. La dernière partie correspond au numéro du segment au sein de ce journal. Selon la taille du segment fixée à l'initialisation, il peut aller de `00000000` à `000000FF` (256 segments de 16 Mo, configuration par défaut, soit 4 Go), à `00000FFF` (4096 segments de 1 Mo), ou à `0000007F` (128 segments de 32 Mo, exemple ci-dessus), etc. Une fois ce maximum atteint, le numéro de journal au centre est incrémenté et les numéros de segments reprennent à zéro.

L'ordre d'écriture des journaux est numérique (en hexadécimal), et leur archivage doit suivre cet ordre. Il ne faut pas se fier à la date des fichiers pour le tri : pour des raisons de performances, PostgreSQL recycle généralement les fichiers en les renommant. Dans l'exemple ci-dessus, le dernier journal écrit est `000000020000014300000020` et non `000000020000014300000024`.

Toujours pour des raisons de performance liées aux métadata (voir cette explication d'Andres Freund[3]), les nouveaux journaux sont initialisés par des zéros avant d'être utilisés. Recyclage et initialisation peuvent être désactivés en passant les paramètres `wal_recycle` et `wal_init_zero` à `off`, ce qui n'est conseillé que sur certains systèmes de fichiers comme ZFS ou btrfs.

Dans le cadre d'un archivage PITR et/ou d'une réplication par *log shipping*, le sous-répertoire `pg_wal/archive_status` indique l'état des journaux dans le contexte de l'archivage. Les fichiers `.ready` indiquent les journaux restant à archiver (normalement peu nombreux), les `.done` ceux déjà archivés. Il est possible de connaître facilement le contenu de ce répertoire en utilisant la fonction `pg_ls_archive_statusdir()` :

```
SELECT * FROM pg_ls_archive_statusdir() ORDER BY 1 ;
```

```
              name                | size |      modification
----------------------------------+------+------------------------
 000000010000000000000067.done    |    0 | 2020-09-02 15:52:57+02
 000000010000000000000068.done    |    0 | 2020-09-02 15:52:57+02
 000000010000000000000069.done    |    0 | 2020-09-02 15:52:58+02
 00000001000000000000006A.ready   |    0 | 2020-09-02 15:53:53+02
 00000001000000000000006B.ready   |    0 | 2020-09-02 15:53:53+02
 00000001000000000000006C.ready   |    0 | 2020-09-02 15:53:54+02
 00000001000000000000006D.ready   |    0 | 2020-09-02 15:53:54+02
 00000001000000000000006E.ready   |    0 | 2020-09-02 15:53:54+02
 00000001000000000000006F.ready   |    0 | 2020-09-02 15:53:54+02
 000000010000000000000070.ready   |    0 | 2020-09-02 15:53:55+02
 000000010000000000000071.ready   |    0 | 2020-09-02 15:53:55+02
```

Le répertoire `pg_xact` contient l'état de toutes les transactions passées ou présentes sur la base (validées, annulées, en sous-transaction ou en cours), comme nous le détaillerons dans le module « Mécanique du moteur transactionnel ».

[3] https://www.postgresql.org/message-id/4xaez4hh3cwkclr77tnkdba6wisueu32vza44iugozai5hklyb%405epf6zpo66aa

Les fichiers contenus dans le répertoire `pg_xact` ne doivent **jamais** être effacés. Ils sont cruciaux au bon fonctionnement de la base.

D'autres répertoires contiennent des fichiers essentiels à la gestion des transactions :

- `pg_commit_ts` contient l'horodatage de la validation de chaque transaction ;
- `pg_multixact` est utilisé dans l'implémentation des verrous partagés (`SELECT … FOR SHARE`) ;
- `pg_serial` est utilisé dans l'implémentation de SSI (`Serializable Snapshot Isolation`) ;
- `pg_snapshots` est utilisé pour stocker les snapshots exportés de transactions ;
- `pg_subtrans` stocke l'imbrication des transactions lors de sous-transactions (les `SAVEPOINTS`) ;
- `pg_twophase` est utilisé pour l'implémentation du *Two-Phase Commit*, aussi appelé `transaction préparée`, `2PC`, ou transaction `XA` dans le monde Java par exemple.

La version 10 a été l'occasion du changement de nom de quelques répertoires pour des raisons de cohérence et pour réduire les risques de fausses manipulations. Jusqu'en 9.6, `pg_wal` s'appelait `pg_xlog`, `pg_xact` s'appelait `pg_clog`.
Les fonctions et outils ont été renommés en conséquence :

- dans les noms de fonctions et d'outils, `xlog` a été remplacé par `wal` (par exemple `pg_switch_xlog` est devenue `pg_switch_wal`) ;
- toujours dans les fonctions, `location` a été remplacé par `lsn`.

8.6.6 Fichiers liés à la réplication

- `pg_logical/`
- `pg_repslot/`

`pg_logical` contient des informations sur la réplication logique.

`pg_replslot` contient des informations sur les slots de réplications, qui sont un moyen de fiabiliser la réplication physique ou logique.

Sans réplication en place, ces répertoires sont quasi-vides. Là encore, il ne faut pas toucher à leur contenu.

8.6.7 Répertoire des tablespaces

- `pg_tblspc/` : tablespaces
  - si vraiment nécessaires
  - liens symboliques ou points de jonction
  - totalement optionnels

Dans PGDATA, le sous-répertoire `pg_tblspc` contient les *tablespaces*, c'est-à-dire des espaces de stockage.

8.6.7.1 Sous Linux

Sous Linux, ce sont des liens symboliques vers un simple répertoire extérieur à PGDATA. Chaque lien symbolique a comme nom l'OID du tablespace (table système `pg_tablespace`). PostgreSQL y crée un répertoire lié aux versions de PostgreSQL et du catalogue, et y place les fichiers de données.

```
postgres=# \db+
                           Liste des tablespaces
    Nom     | Propriétaire |      Emplacement      | … | Taille  |…
------------+--------------+-----------------------+---+---------+-
 froid      | postgres     | /mnt/hdd/pg           |   | 3576 kB |
 pg_default | postgres     |                       |   | 6536 MB |
 pg_global  | postgres     |                       |   | 587 kB  |
```

```
sudo ls -R /mnt/hdd/pg
```

```
/mnt/hdd/pg:
PG_15_202209061

/mnt/hdd/pg/PG_15_202209061:
5

/mnt/hdd/pg/PG_15_202209061/5:
142532  142532_fsm  142532_vm
```

8.6.7.2 Sous Windows

Sous Windows, les liens sont à proprement parler des *Reparse Points* (ou *Junction Points*) :

```
postgres=# \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
```

```
 pg_default | postgres     |
 pg_global  | postgres     |
 tbl1       | postgres     | T:\TBL1
```

```
PS P:\PGDATA13> dir 'pg_tblspc/*' | ?{$_.LinkType} | select FullName,LinkType,Target

FullName                     LinkType Target
--------                     -------- ------
P:\PGDATA13\pg_tblspc\105921 Junction {T:\TBL1}
```

Par défaut, `pg_tblspc/` est vide. N'existent alors que les tablespaces `pg_global` (sous-répertoire `global/` des objets globaux à l'instance) et `pg_default` (soit `base/`).

> La création d'autres tablespaces est totalement optionnelle.

Leur utilité et leur gestion seront abordés plus loin.

8.6.8 Fichiers des statistiques d'activité

> Statistiques d'activité :
>
> - `stats collector` (≤v14) & extensions
> - `pg_stat_tmp/` : temporaires
> - `pg_stat/` : définitif

`pg_stat_tmp` est, jusqu'en version 15, le répertoire par défaut de stockage des statistiques d'activité de PostgreSQL, comme les entrées-sorties ou les opérations de modifications sur les tables. Ces fichiers pouvant générer une grande quantité d'entrées-sorties, l'emplacement du répertoire peut être modifié avec le paramètre `stats_temp_directory`. Par exemple, Debian place ce paramètre par défaut en `tmpfs` :

```
-- jusque v14
SHOW stats_temp_directory;

           stats_temp_directory
----------------------------------------
 /var/run/postgresql/14-main.pg_stat_tmp
```

À l'arrêt, les fichiers sont copiés dans le répertoire `pg_stat/`.

PostgreSQL gérant ces statistiques en mémoire partagée à partir de la version 15, le collecteur n'existe plus. Mais les deux répertoires sont encore utilisés par des extensions comme `pg_stat_statements` ou `pg_stat_kcache`.

8.6.9 Autres répertoires

- `pg_dynshmem/`
- `pg_notify/`

`pg_dynshmem` est utilisé par les extensions utilisant de la mémoire partagée dynamique.

`pg_notify` est utilisé par le mécanisme de gestion de notification de PostgreSQL (`LISTEN` et `NOTIFY`) qui permet de passer des messages de notification entre sessions.

8.6.10 Les fichiers de traces (journaux)

- Fichiers texte traçant l'activité
- Très paramétrables
- Gestion des fichiers soit :
  - par PostgreSQL
  - délégués au système d'exploitation (*syslog*, *eventlog*)

Le paramétrage des journaux est très fin. Leur configuration est le sujet est évoquée dans notre première formation[4].

Si `logging_collector` est activé, c'est-à-dire que PostgreSQL collecte lui-même ses traces, l'emplacement de ces journaux se paramètre grâce aux paramètres `log_directory`, le répertoire où les stocker, et `log_filename`, le nom de fichier à utiliser, ce nom pouvant utiliser des échappements comme `%d` pour le jour de la date, par exemple. Les droits attribués au fichier sont précisés par le paramètre `log_file_mode`.

Un exemple pour `log_filename` avec date et heure serait :

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

La liste des échappements pour le paramètre `log_filename` est disponible dans la page de manuel de la fonction `strftime` sur la plupart des plateformes de type UNIX.

[4] https://dali.bo/h1_html

8.7 RÉSUMÉ

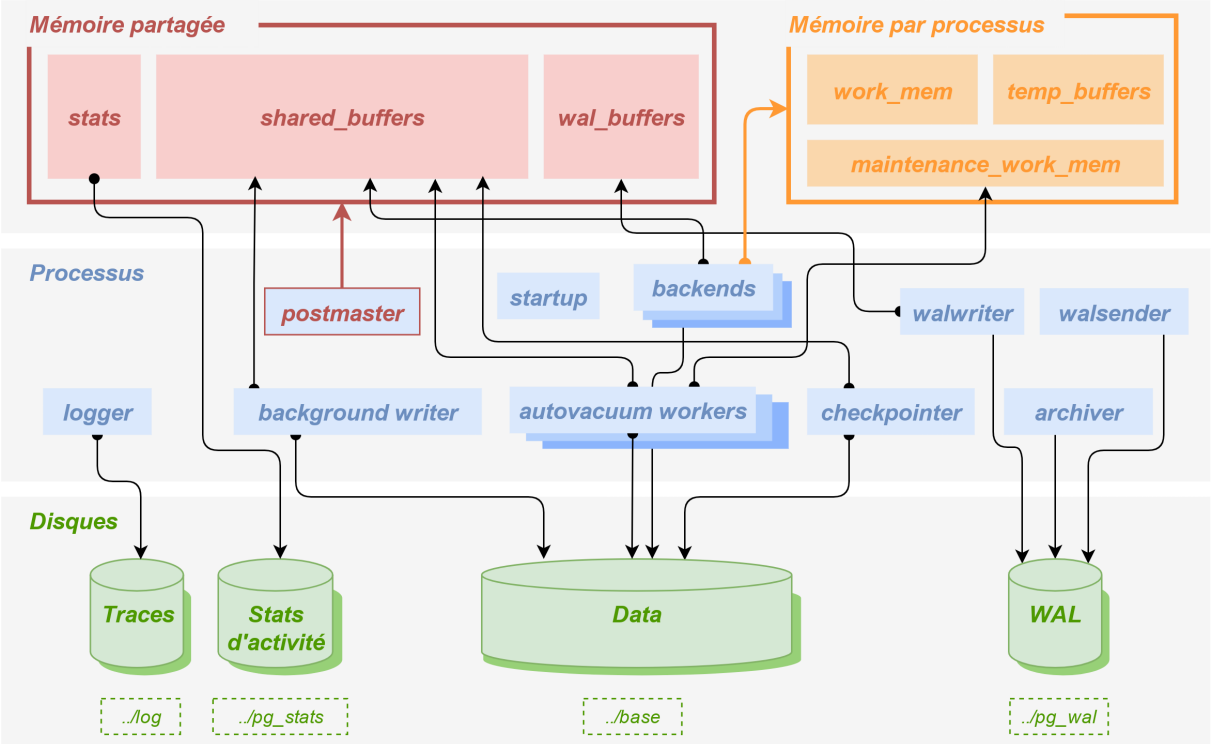


Figure 8/ .1: Architecture de PostgreSQL

Ce schéma ne soit à présent plus avoir aucun secret pour vous.

8.8 CONCLUSION

- PostgreSQL est complexe, avec de nombreux composants
- Une bonne compréhension de cette architecture est la clé d'une bonne administration.
- Pour aller (beaucoup) plus loin :

8.8.1 Questions

N'hésitez pas, c'est le moment !

8.9 QUIZ

https://dali.bo/m1_quiz

8.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m1_solutions.

8.10.1 Processus

But : voir quelques processus de PostgreSQL

Si ce n'est pas déjà fait, démarrer l'instance PostgreSQL.

Lister les processus du serveur PostgreSQL. Qu'observe-t-on ?

Se connecter à l'instance PostgreSQL.

Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

Créer une nouvelle base de données nommée **b0**.

Se connecter à la base de données **b0** et créer une table `t1` avec une colonne `id` de type `integer`.

Insérer 10 millions de lignes dans la table `t1` avec :

```
INSERT INTO t1 SELECT generate_series(1, 10000000) ;
```

Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

Configurer la valeur du paramètre `max_connections` à `15`.

Redémarrer l'instance PostgreSQL.

Vérifier que la modification de la valeur du paramètre `max_connections` a été prise en compte.

Se connecter 15 fois à l'instance PostgreSQL sans fermer les sessions, par exemple en lançant plusieurs fois :

```
psql -c 'SELECT pg_sleep(1000)' &
```

Se connecter une seizième fois à l'instance PostgreSQL. Qu'observe-t-on ?

Configurer la valeur du paramètre `max_connections` à sa valeur initiale.

8.10.2 Fichiers

But : voir les fichiers de PostgreSQL

Aller dans le répertoire des données de l'instance PostgreSQL. Lister les fichiers.

Aller dans le répertoire `base`. Lister les fichiers.

À quelle base est lié chaque répertoire présent dans le répertoire `base` ? (Voir l'oid de la base dans `pg_database` ou l'utilitaire en ligne de commande `oid2name`)

Créer une nouvelle base de données nommée **b1**. Qu'observe-t-on dans le répertoire `base` ?

Se connecter à la base de données **b1**. Créer une table `t1` avec une colonne `id` de type `integer`.

Récupérer le chemin vers le fichier correspondant à la table `t1` (il existe une fonction `pg_relation_filepath`).

Regarder la taille du fichier correspondant à la table `t1`. Pourquoi est-il vide ?

Insérer une ligne dans la table `t1`. Quelle taille fait le fichier de la table `t1` ?

> Insérer 500 lignes dans la table `t1` avec `generate_series`. Quelle taille fait le fichier de la table `t1` ?

> Pourquoi cette taille pour simplement 501 entiers de 4 octets chacun ?

8.11 TRAVAUX PRATIQUES (SOLUTIONS)

8.11.1 Processus

> Si ce n'est pas déjà fait, démarrer l'instance PostgreSQL.

Sous Rocky Linux, CentOS ou Red Hat en tant qu'utilisateur **root** :

```
# systemctl start postgresql-15
```

> Lister les processus du serveur PostgreSQL. Qu'observe-t-on ?

En tant qu'utilisateur **postgres** :

```
$ ps -o pid,cmd fx
    PID CMD
  27886 -bash
  28666  \_ ps -o pid,cmd fx
  27814 /usr/pgsql-17/bin/postmaster -D /var/lib/pgsql/15/data/
  27815  \_ postgres: logger
  27816  \_ postgres: checkpointer
  27817  \_ postgres: background writer
  27819  \_ postgres: walwriter
  27820  \_ postgres: autovacuum launcher
  27821  \_ postgres: logical replication launcher
```

> Se connecter à l'instance PostgreSQL.

```
$ psql postgres
psql (15.0)
Type "help" for help.

postgres=#
```

> Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

```
$ ps -o pid,cmd fx
    PID CMD
  28746 -bash
  28779  \_ psql
  27886 -bash
  28781  \_ ps -o pid,cmd fx
  27814 /usr/pgsql-17/bin/postmaster -D /var/lib/pgsql/15/data/
  27815  \_ postgres: logger
  27816  \_ postgres: checkpointer
  27817  \_ postgres: background writer
  27819  \_ postgres: walwriter
  27820  \_ postgres: autovacuum launcher
  27821  \_ postgres: logical replication launcher
  28780  \_ postgres: postgres postgres [local] idle
```

Il y a un nouveau processus (ici PID 28780) qui va gérer l'exécution des requêtes du client psql.

> Créer une nouvelle base de données nommée **b0**.

Depuis le shell, en tant que **postgres** :

```
$ createdb b0
```

Alternativement, depuis la session déjà ouverte dans `psql` :

```
CREATE DATABASE b0;
```

> Se connecter à la base de données **b0** et créer une table `t1` avec une colonne `id` de type `integer`.

Pour se connecter depuis le shell :

```
psql b0
```

ou depuis la session `psql` :

```
\c b0
```

Création de la table :

```
CREATE TABLE t1 (id integer);
```

> Insérer 10 millions de lignes dans la table `t1` avec :
>
> ```
> INSERT INTO t1 SELECT generate_series(1, 10000000) ;
> ```

```
INSERT INTO t1 SELECT generate_series(1, 10000000);
```

```
INSERT 0 10000000
```

> Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

```
$ ps -o pid,cmd fx
    PID CMD
  28746 -bash
  28779  \_ psql
  27886 -bash
  28781  \_ ps -o pid,cmd fx
  27814 /usr/pgsql-17/bin/postmaster -D /var/lib/pgsql/15/data/
  27815  \_ postgres: logger
  27816  \_ postgres: checkpointer
  27817  \_ postgres: background writer
  27819  \_ postgres: walwriter
  27820  \_ postgres: autovacuum launcher
  27821  \_ postgres: logical replication launcher
  28780  \_ postgres: postgres postgres [local] INSERT
```

Le processus serveur exécute l'`INSERT`, ce qui se voit au niveau du nom du processus. Seul est affiché le dernier ordre SQL (*ie* le mot `INSERT` et non pas la requête complète).

> Configurer la valeur du paramètre `max_connections` à `15`.

La première méthode est d'ouvrir le fichier de configuration `postgresql.conf` et de modifier la valeur du paramètre `max_connections` :

```
max_connections = 15
```

La seconde méthode se fait directement depuis psql en tant que superutilisateur :

```
ALTER SYSTEM SET max_connections TO 15 ;
```

Ce dernier ordre fera apparaître une ligne dans le fichier `/var/lib/pgsql/15/data/postgresql.auto.conf`.

Dans les deux cas, la prise en compte n'est pas automatique. Pour ce paramètre, il faut redémarrer l'instance PostgreSQL.

> Redémarrer l'instance PostgreSQL.

En tant qu'utilisateur **root** :

```
# systemctl restart postgresql-15
```

> Vérifier que la modification de la valeur du paramètre `max_connections` a été prise en compte.

```
postgres=# SHOW max_connections ;

 max_connections
-----------------
 15
```

> Se connecter 15 fois à l'instance PostgreSQL sans fermer les sessions, par exemple en lançant plusieurs fois :
>
> ```
> psql -c 'SELECT pg_sleep(1000)' &
> ```

Il est possible de le faire manuellement ou de l'automatiser avec ce petit script shell :

```
$ for i in $(seq 1 15); do psql -c "SELECT pg_sleep(1000);" postgres & done
[1] 998
[2] 999
...
[15] 1012
```

> Se connecter une seizième fois à l'instance PostgreSQL. Qu'observe-t-on ?

```
$ psql postgres
psql: FATAL:  sorry, too many clients already
```

Il est impossible de se connecter une fois que le nombre de connexions a atteint sa limite configurée avec `max_connections`. Il faut donc attendre que les utilisateurs se déconnectent pour accéder de nouveau au serveur.

> Configurer la valeur du paramètre `max_connections` à sa valeur initiale.

Si le fichier de configuration `postgresql.conf` avait été modifié, restaurer la valeur du paramètre `max_connections` à 100.

Si `ALTER SYSTEM` a été utilisée, il faudrait pouvoir entrer dans psql :

```
ALTER SYSTEM RESET max_connections ;
```

Mais cela ne fonctionne pas si toutes les connexions réservées à l'utilisateur ont été consommées (paramètre `superuser_reserved_connections`, par défaut à 3). Dans ce cas, il n'y a plus qu'à aller dans le fichier de configuration `/var/lib/pgsql/15/data/postgresql.auto.conf`, pour supprimer la ligne avec le paramètre `max_connections` avant de redémarrer l'instance PostgreSQL.

Il est déconseillé de modifier `postgresql.auto.conf` à la main, mais pour le TP, nous nous permettons quelques libertés.

Toutefois si l'instance est démarrée et qu'il est encore possible de s'y connecter, le plus propre est ceci :

```
ALTER SYSTEM RESET max_connections ;
```

Puis redémarrer PostgreSQL : toutes les connexions en cours vont être coupées.

```
# systemctl restart postgresql-15
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
[...]
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
connection to server was lost
```

Il est à présent possible de se reconnecter. Vérifier que cela a été pris en compte :

```
postgres=# SHOW max_connections ;

 max_connections
-----------------
 100
```

8.11.2 Fichiers

> Aller dans le répertoire des données de l'instance PostgreSQL. Lister les fichiers.

En tant qu'utilisateur système **postgres** :

```
 echo $PGDATA
/var/lib/pgsql/15/data

$ cd $PGDATA

$ ls -al
total 68
drwx------. 7 postgres postgres    59 Nov  4 09:55 base
-rw-------. 1 postgres postgres    30 Nov  4 10:38 current_logfiles
drwx------. 2 postgres postgres  4096 Nov  4 10:38 global
drwx------. 2 postgres postgres    58 Nov  4 07:58 log
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_commit_ts
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_dynshmem
-rw-------. 1 postgres postgres  4658 Nov  4 09:50 pg_hba.conf
-rw-------. 1 postgres postgres  1685 Nov  3 14:16 pg_ident.conf
drwx------. 4 postgres postgres    68 Nov  4 10:38 pg_logical
drwx------. 4 postgres postgres    36 Nov  3 14:11 pg_multixact
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_notify
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_replslot
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_serial
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_snapshots
drwx------. 2 postgres postgres     6 Nov  4 10:38 pg_stat
drwx------. 2 postgres postgres    35 Nov  4 10:38 pg_stat_tmp
drwx------. 2 postgres postgres    18 Nov  3 14:11 pg_subtrans
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_tblspc
drwx------. 2 postgres postgres     6 Nov  3 14:11 pg_twophase
-rw-------. 1 postgres postgres     3 Nov  3 14:11 PG_VERSION
drwx------. 3 postgres postgres    92 Nov  4 09:55 pg_wal
drwx------. 2 postgres postgres    18 Nov  3 14:11 pg_xact
-rw-------. 1 postgres postgres    88 Nov  3 14:11 postgresql.auto.conf
-rw-------. 1 postgres postgres 29475 Nov  4 09:36 postgresql.conf
-rw-------. 1 postgres postgres    58 Nov  4 10:38 postmaster.opts
-rw-------. 1 postgres postgres   104 Nov  4 10:38 postmaster.pid
```

> Aller dans le répertoire `base`. Lister les fichiers.

```
$ cd base

$ ls -al
total 60
drwx------  8 postgres postgres   78 Nov  4 16:21 .
drwx------ 20 postgres postgres 4096 Nov  4 15:33 ..
drwx------. 2 postgres postgres 8192 Nov  4 10:38 1
drwx------. 2 postgres postgres 8192 Nov  4 09:50 16404
drwx------. 2 postgres postgres 8192 Nov  3 14:11 4
drwx------. 2 postgres postgres 8192 Nov  4 10:39 5
drwx------  2 postgres postgres    6 Nov  3 15:58 pgsql_tmp
```

> À quelle base est lié chaque répertoire présent dans le répertoire `base` ? (Voir l'oid de la base dans `pg_database` ou l'utilitaire en ligne de commande `oid2name`)

Chaque répertoire correspond à une base de données. Le numéro indiqué est un identifiant système (OID). Il existe deux moyens pour récupérer cette information :

- directement dans le catalogue système `pg_database` :

```
$ psql postgres
```

```
psql (15.0)
Type "help" for help.
```

```
postgres=# SELECT oid, datname FROM pg_database ORDER BY oid::text;
```

```
  oid  |  datname
-------+-----------
     1 | template1
 16404 | b0
     4 | template0
     5 | postgres
```

- avec l'outil `oid2name` (à installer au besoin via le paquet `postgresql15-contrib`) :

```
$ /usr/pgsql-17/bin/oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  16404            b0  pg_default
      5      postgres  pg_default
      4     template0  pg_default
      1     template1  pg_default
```

Donc ici, le répertoire `1` correspond à la base `template1`, et le répertoire `5` à la base `postgres` (ces nombres peuvent changer suivant l'installation).

> Créer une nouvelle base de données nommée **b1**. Qu'observe-t-on dans le répertoire `base` ?

```
$ createdb b1
```

```
$ ls -al
total 60
drwx------  8 postgres postgres   78 Nov  4 16:21 .
drwx------ 20 postgres postgres 4096 Nov  4 15:33 ..
drwx------. 2 postgres postgres 8192 Nov  4 10:38 1
drwx------. 2 postgres postgres 8192 Nov  4 09:50 16404
drwx------. 2 postgres postgres 8192 Nov  4 09:55 16405
drwx------. 2 postgres postgres 8192 Nov  3 14:11 4
drwx------. 2 postgres postgres 8192 Nov  4 10:39 5
drwx------  2 postgres postgres    6 Nov  3 15:58 pgsql_tmp
```

Un nouveau sous-répertoire est apparu, nommé `16405`. Il correspond bien à la base `b1` d'après `oid2name`.

> Se connecter à la base de données **b1**. Créer une table `t1` avec une colonne `id` de type `integer`.

```
$ psql b1
psql (15.0)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);

CREATE TABLE
```

> Récupérer le chemin vers le fichier correspondant à la table `t1` (il existe une fonction `pg_relation_filepath`).

La fonction a pour définition :

```
b1=# \df pg_relation_filepath
                                  List of functions
   Schema   |         Name         | Result data type | Argument data types | Type
------------+----------------------+------------------+---------------------+------
 pg_catalog | pg_relation_filepath | text             | regclass            | func
```

L'argument `regclass` peut être l'OID de la table, ou son nom.

L'emplacement du fichier sur le disque est donc :

```
b1=# SELECT current_setting('data_directory') || '/' || pg_relation_filepath('t1')
     AS chemin;

                 chemin
----------------------------------------
 /var/lib/pgsql/15/data/base/16405/16406
```

> Regarder la taille du fichier correspondant à la table `t1`. Pourquoi est-il vide ?

```
$ ls -l /var/lib/pgsql/15/data/base/16393/16398
-rw-------. 1 postgres postgres 0 Nov  4 10:42
 ↪  /var/lib/pgsql/15/data/base/16405/16406
```

La table vient d'être créée. Aucune donnée n'a encore été ajoutée. Les métadonnées se trouvent dans d'autres tables (des catalogues systèmes). Donc il est logique que le fichier soit vide.

> Insérer une ligne dans la table `t1`. Quelle taille fait le fichier de la table `t1` ?

```
b1=# INSERT INTO t1 VALUES (1);
INSERT 0 1

$ ls -l /var/lib/pgsql/15/data/base/16393/16398
-rw-------. 1 postgres postgres 8192 Nov  4 12:40
 ↪  /var/lib/pgsql/15/data/base/16405/16406
```

Il fait à présent 8 ko. En fait, PostgreSQL travaille par blocs de 8 ko. Si une ligne ne peut être placée dans un espace libre d'un bloc existant, un bloc entier est ajouté à la table.

Vous pouvez consulter le fichier avec la commande `hexdump -x <nom du fichier>` (faites un `CHECKPOINT` avant pour être sûr qu'il soit écrit sur le disque).

> Insérer 500 lignes dans la table `t1` avec `generate_series`. Quelle taille fait le fichier de la table `t1` ?

```
b1=# INSERT INTO t1 SELECT generate_series(1, 500);
INSERT 0 500
```

```
$ ls -l /var/lib/pgsql/15/data/base/16393/16398
-rw-------. 1 postgres postgres 24576 Nov  4 12:41
 ↪  /var/lib/pgsql/15/data/base/16405/16406
```

Le fichier fait maintenant 24 ko, soit 3 blocs de 8 ko.

> Pourquoi cette taille pour simplement 501 entiers de 4 octets chacun ?

Nous avons enregistré 501 entiers dans la table. Un entier de type `int4` prend 4 octets. Donc nous avons 2004 octets de données utilisateurs. Et pourtant, nous arrivons à un fichier de 24 ko.

En fait, PostgreSQL enregistre aussi dans chaque bloc des informations systèmes en plus des données utilisateurs. Chaque bloc contient un en-tête, des pointeurs, et l'ensemble des lignes du bloc. Chaque ligne contient les colonnes utilisateurs mais aussi des colonnes système. La requête suivante permet d'en savoir plus sur les colonnes présentes dans la table :

```
b1=# SELECT CASE WHEN attnum<0 THEN 'systeme' ELSE 'utilisateur' END AS type,
     attname, attnum, typname, typlen,
     sum(typlen) OVER (PARTITION BY attnum<0) AS longueur_tot
     FROM pg_attribute a
     JOIN pg_type t ON t.oid=a.atttypid
     WHERE attrelid ='t1'::regclass
     ORDER BY attnum;
```

```
    type     | attname   | attnum | typname | typlen | longueur_tot
-------------+----------+--------+---------+--------+--------------
 systeme     | tableoid |     -6 | oid     |      4 |           26
 systeme     | cmax     |     -5 | cid     |      4 |           26
 systeme     | xmax     |     -4 | xid     |      4 |           26
 systeme     | cmin     |     -3 | cid     |      4 |           26
 systeme     | xmin     |     -2 | xid     |      4 |           26
 systeme     | ctid     |     -1 | tid     |      6 |           26
 utilisateur | id       |      1 | int4    |      4 |            4
```

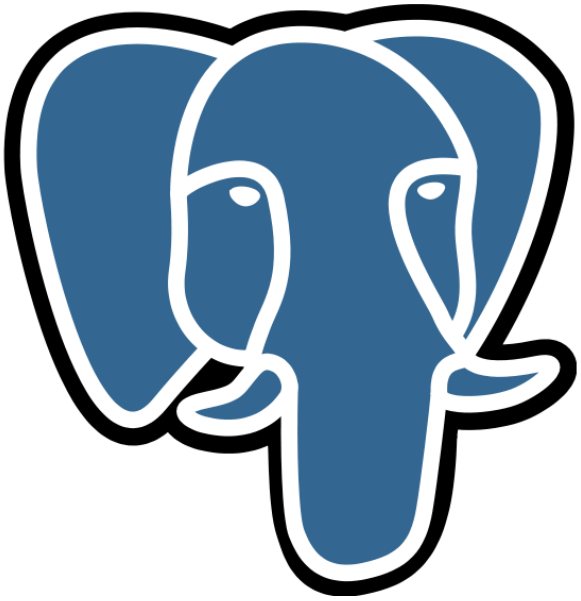
Vous pouvez voir ces colonnes système en les appelant explicitement :

```
SELECT cmin, cmax, xmin, xmax, ctid, *
FROM t1 ;
```

L'en-tête de chaque ligne pèse 26 octets dans le cas général. Dans notre cas très particulier avec une seule petite colonne, c'est très défavorable mais ce n'est généralement pas le cas.

Avec 501 lignes de 26+4 octets, nous obtenons 15 ko. Chaque bloc possède quelques informations de maintenance : nous dépassons alors 16 ko, ce qui explique pourquoi nous en sommes à 24 ko (3 blocs).

9/ Configuration de PostgreSQL



9.1 AU MENU

- Les paramètres en lecture seule
- Les différents fichiers de configuration
  - survol du contenu
- Quelques paramétrages importants :
  - tablespaces
  - connexions
  - statistiques
  - optimiseur

9.2 PARAMÈTRES EN LECTURE SEULE

- Options de compilation ou lors d'`initdb`
- Quasiment jamais modifiés
  - risque d'incompatabilité des fichiers, avec les outils
- Tailles de bloc ou de fichier
  - `block_size` : 8 ko
  - `wal_block_size` : 8 ko
  - `segment_size` : 1 Go
  - `wal_segment_size` : 16 Mo (option `--wal-segsize` d'`initdb` en v11)

Ces paramètres sont en lecture seule, mais peuvent être consultés par la commande `SHOW`, ou en interrogeant la vue `pg_settings`. Il est possible aussi d'obtenir l'information via la commande `pg_controldata`.

- `block_size` est la taille d'un bloc de données de la base, par défaut 8192 octets ;
- `wal_block_size` est la taille d'un bloc de journal, par défaut 8192 octets ;
- `segment_size` est la taille maximum d'un fichier de données, par défaut 1 Go ;
- `wal_segment_size` est la taille d'un fichier de journal de transactions (WAL), par défaut 16 Mo.

Ces paramètres sont tous fixés à la compilation, sauf `wal_segment_size` à partir de la version 11 : `initdb` accepte alors l'option `--wal-segsize` et l'on peut monter la taille des journaux de transactions à 1 Go. Cela n'a d'intérêt que pour des instances générant énormément de journaux.

Recompiler avec une taille de bloc de 32 ko s'est déjà vu sur de très grosses installations (comme le rapporte par exemple Christophe Pettus (San Francisco, 2023)[1]) avec un `shared_buffers` énorme, mais cette configuration est très peu testée, nous la déconseillons dans le cas général.

Un moteur compilé avec des options non standard ne pourra pas ouvrir des fichiers n'ayant pas les mêmes valeurs pour ces options.

[1] https://thebuild.com/blog/2023/02/08/xtreme-postgresql/

Des tailles non standard vous exposent à rencontrer des problèmes avec des outils s'attendant à des blocs de 8 ko. (Remontez alors le bug.)

9.3 FICHIERS DE CONFIGURATION

- `postgresql.conf` (+ fichiers inclus)
- `postgresql.auto.conf`
- `pg_hba.conf` (+ fichiers inclus (v16))
- `pg_ident.conf` (idem)

Les fichiers de configuration sont habituellement les 4 suivants :

- `postgresql.conf` : il contient une liste de paramètres, sous la forme `paramètre=valeur` ;
- `pg_hba.conf` : il contient les règles d'authentification à la base.
- `pg_ident.conf` : il complète `pg_hba.conf`, quand nous déciderons de nous reposer sur un mécanisme d'authentification extérieur à la base (identification par le système ou par un annuaire par exemple) ;
- `postgresql.auto.conf` : il stocke les paramètres de configuration fixés en utilisant la commande `ALTER SYSTEM` et surcharge donc `postgresql.conf`.

9.4 POSTGRESQL.CONF

Fichier principal de configuration :

- Emplacement :
  - défaut/Red Hat & dérivés : répertoires des données (`/var/lib/…`)
  - Debian : `/etc/postgresql/<version>/<nom>/postgresql.conf`
- Format `clé = valeur`
- Sections, commentaires (redémarrage !)

C'est le fichier le plus important. Il contient le paramétrage de l'instance. PostgreSQL le cherche au démarrage dans le PGDATA. Par défaut, dans les versions compilées, ou depuis les paquets sur Red Hat, CentOS ou Rocky Linux, il sera dans le répertoire principal avec les données (`/var/lib/pgsql/15/data/postgresql.conf` par exemple). Debian le place dans `/etc` (`/etc/postgresql/15/main/postgresql.conf` pour l'instance par défaut).

Dans le doute, il est possible de consulter la valeur du paramètre `config_file`, ici dans la configuration par défaut sur Rocky Linux :

```
# SHOW config_file;

               config_file
---------------------------------------------
 /var/lib/postgresql/15/data/postgresql.conf
```

Ce fichier contient un paramètre par ligne, sous le format :

```
clé = valeur
```

Les commentaires commencent par « # » (croisillon) et les chaînes de caractères doivent être encadrées de « ' » (*single quote*). Par exemple :

```
data_directory = '/var/lib/postgresql/15/main'
listen_addresses = 'localhost'
port = 5432
shared_buffers = 128MB
```

Les valeurs de ce fichier ne seront pas forcément les valeurs actives !

Nous allons en effet voir que l'on peut les surcharger.

9.4.1 Surcharge des paramètres de postgresql.conf

- Inclusion externe : `include`, `include_if_exists`
- Surcharge dans cet ordre :
  - `ALTER SYSTEM SET …` (renseigne `postgresql.auto.conf`)
  - paramètres de `pg_ctl`
  - `ALTER DATABASE | ROLE … SET paramètre = …`
  - `SET` / `SET LOCAL`
- Consulter :
  - `SHOW`
  - `pg_settings`
  - `pg_file_settings`

Le paramétrage ne dépend pas seulement du contenu de `postgresql.conf`.

Nous pouvons inclure d'autres fichiers depuis `postgresql.conf` grâce à l'une de ces directives :

```
include = 'nom_fichier'
include_if_exists = 'nom_fichier'
include_dir = 'répertoire'       # contient des fichiers .conf
```

Le ou les fichiers indiqués sont alors inclus à l'endroit où la directive est positionnée. Avec `include`, si le fichier n'existe pas, une erreur `FATAL` est levée ; au contraire la directive `include_if_exists` permet de ne pas s'arrêter si le fichier n'existe pas. Ces directives permettent notamment des ajustements de configuration propres à plusieurs machines d'un ensemble primaire/secondaires dont le `postgresql.conf` de base est identique, ou de gérer la configuration hors de `postgresql.conf`.

Si des paramètres sont répétés dans `postgresql.conf`, éventuellement suite à des inclusions, la dernière occurrence écrase les précédentes. Si un paramètre est absent, la valeur par défaut s'applique.

Ces paramètres peuvent être surchargés par le fichier `postgresql.auto.conf`, qui contient le résultat des commandes de ce type :

```
ALTER SYSTEM SET paramètre = valeur ;
```

Ce fichier est principalement utilisé par les administrateurs et les outils qui n'ont pas accès au système de fichiers du serveur. (À partir de PostgreSQL 17, l'utilisation de `ALTER SYSTEM` peut être interdite, en passant le paramètre `allow_alter_system` à `off`. Le but est d'éviter des erreurs de manipulation quand la configuration est gérée par un outil extérieur, comme Ansible ou Patroni. À noter qu'un superutilisateur malveillant saura tout de même contourner cette limite.)

Si des options sont passées directement en arguments à `pg_ctl` (situation rare), elles seront prises en compte en priorité par rapport à celles de ces fichiers de configuration.

Il est possible de surcharger les options modifiables à chaud par utilisateur, par base, et par combinaison « utilisateur+base », avec par exemple :

```
ALTER ROLE nagios SET log_min_duration_statement TO '1min';
ALTER DATABASE dwh SET work_mem TO '1GB';
ALTER ROLE patron IN DATABASE dwh SET work_mem TO '2GB';
```

Ces surcharges sont visibles dans la table `pg_db_role_setting` ou via la commande `\drds` de `psql`.

Ensuite, un utilisateur peut changer à volonté les valeurs de beaucoup de paramètres au sein d'une session :

```
SET parametre = valeur ;
```

ou même juste au sein d'une transaction :

```
SET LOCAL parametre = valeur ;
```

Au final, l'ordre des surcharges est le suivant :

```
paramètre par défaut
  -> postgresql.conf
    -> ALTER SYSTEM SET (postgresql.auto.conf)
      -> option de pg_ctl / postmaster
        -> paramètre par base
          -> paramètre par rôle
            -> paramètre base+rôle
              -> paramètre dans la chaîne de connexion
                -> paramètre de session (SET)
                  -> paramètre de transaction (SET LOCAL)
```

À titre d'exemple, voici ce qu'il se passe en appliquant la requête suivante :

```
SET client_min_messages = debug2;
```

La meilleure source d'information sur les valeurs actives est la vue `pg_settings` :

```
SELECT name,source,context,setting,boot_val,reset_val
FROM pg_settings
WHERE name IN ('client_min_messages', 'log_checkpoints', 'wal_segment_size');
```

```
        name          |  source  | context  | setting  | boot_val | reset_val
----------------------+----------+----------+----------+----------+-----------
 client_min_messages | default  | user     | debug2   | notice   | notice
 log_checkpoints     | default  | sighup   | off      | off      | off
 wal_segment_size    | override | internal | 16777216 | 16777216 | 16777216
```

Nous constatons par exemple que, dans la session ayant effectué la requête, la valeur du paramètre `client_min_messages` a été modifiée à la valeur `debug2`. Nous pouvons aussi voir le contexte dans lequel le paramètre est modifiable : le `client_min_messages` est modifiable par l'utilisateur dans

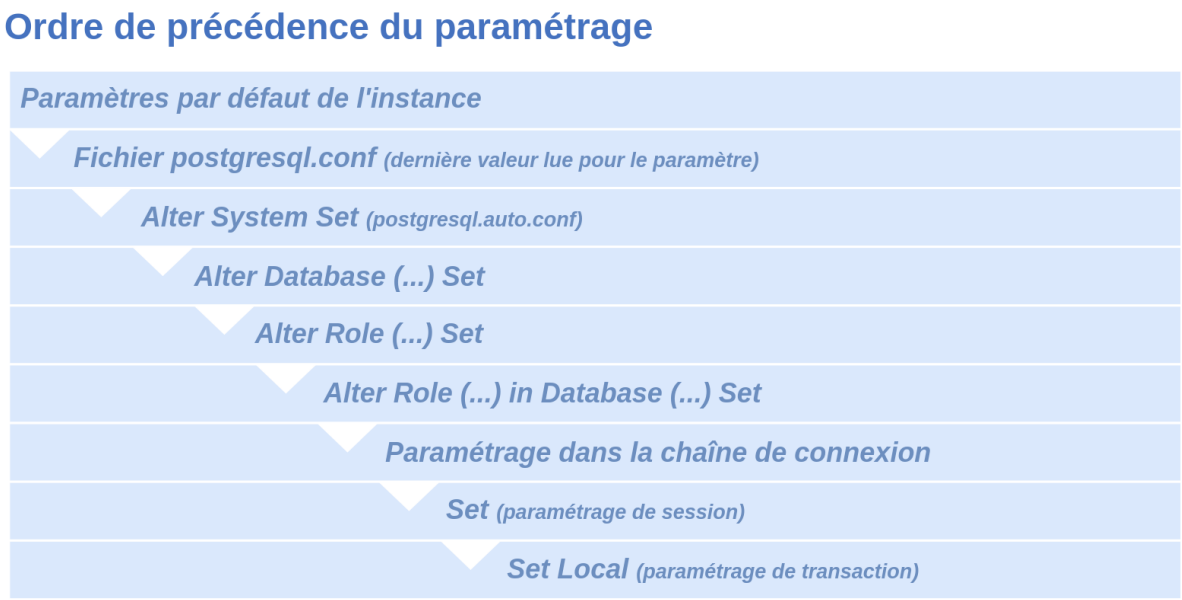
sa session. Le `log_checkpoints` seulement par `sighup`, c'est-à-dire par un `pg_ctl reload`, et le `wal_segment_size` n'est pas modifiable après l'initialisation de l'instance.

De nombreuses autres colonnes sont disponibles dans `pg_settings`, comme une description détaillée du paramètre, l'unité de la valeur, ou le fichier et la ligne d'où provient le paramètre. Le champ `pending_restart` indique si un paramètre a été modifié mais attend encore un redémarrage pour être appliqué.

Il existe aussi une vue `pg_file_settings`, qui indique la configuration présente dans les fichiers de configuration (mais pas forcément active !). Elle peut être utile lorsque la configuration est répartie dans plusieurs fichiers. Par exemple, suite à un `ALTER SYSTEM`, les paramètres sont ajoutés dans `postgresql.auto.conf` mais un rechargement de la configuration n'est pas forcément suffisant pour qu'ils soient pris en compte :

```
ALTER SYSTEM SET work_mem TO '16MB' ;
ALTER SYSTEM SET max_connections TO 200 ;

SELECT pg_reload_conf() ;

 pg_reload_conf
----------------
 t

SELECT * FROM pg_file_settings
WHERE name IN ('work_mem','max_connections')
ORDER BY name ;

-[ RECORD 1 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/15/data/postgresql.conf
sourceline | 64
seqno      | 2
name       | max_connections
setting    | 100
applied    | f
error      |
-[ RECORD 2 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/15/data/postgresql.auto.conf
sourceline | 4
seqno      | 17
name       | max_connections
setting    | 200
applied    | f
error      | setting could not be applied
-[ RECORD 3 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/15/data/postgresql.auto.conf
sourceline | 3
seqno      | 16
name       | work_mem
setting    | 16MB
applied    | t
error      |
```

9.4.2 Précédence des paramètres



PostgreSQL offre une certaine granularité dans sa configuration, ainsi certains paramètres peuvent être surchargés par rapport au fichier `postgresql.conf`. Il est utile de connaître l'ordre de précédence. Par exemple, un utilisateur peut spécifier un paramètre dans sa session avec l'ordre `SET`, celui-ci sera prioritaire par rapport à la configuration présente dans le fichier `postgresql.conf`.

9.4.3 Survol de postgresql.conf

- Emplacement de fichiers
- Connections & authentification
- Ressources (hors journaux de transactions)
- Journaux de transactions
- Réplication
- Optimisation de requête
- Traces
- Statistiques d'activité
- Autovacuum
- Paramétrage client par défaut
- Verrous
- Compatibilité

`postgresql.conf` contient environ 300 paramètres. Il est séparé en plusieurs sections, dont les plus importantes figurent ci-dessous. Il n'est pas question de les détailler toutes.

La plupart des paramètres ne sont jamais modifiés. Les défauts sont souvent satisfaisants pour une petite installation. Les plus importants sont supposés acquis (au besoin, voir la formation DBA1[2]).

Les principales sections sont :

Connections and authentication

S'y trouveront les classiques `listen_addresses`, `port`, `max_connections`, `password_encryption`, ainsi que les paramétrages TCP (*keepalive*) et SSL.

Resource usage (except WAL)

Cette partie fixe des limites à certaines consommations de ressources.

Sont normalement déjà bien connus `shared_buffers`, `work_mem` et `maintenance_work_mem` (qui seront couverts extensivement plus loin).

On rencontre ici aussi le paramétrage du `VACUUM` (pas directement de l'autovacuum !), du *background writer*, du parallélisme dans les requêtes.

Write-Ahead Log

Les journaux de transaction sont gérés ici. Cette partie sera également détaillée dans un autre module.

Tout est prévu pour faciliter la mise en place d'une réplication sans avoir besoin de modifier cette partie sur le primaire.

Dans la partie *Archiving*, l'archivage des journaux peut être activé pour une sauvegarde PITR ou une réplication par *log shipping*.

Depuis la version 12, tous les paramètres de restauration (qui peuvent servir à la réplication) figurent aussi dans les sections *Archive Recovery* et *Recovery Target*. Auparavant, ils figuraient dans un fichier `recovery.conf` séparé.

Replication

Cette partie fournit le nécessaire pour alimenter un secondaire en réplication par *streaming*, physique ou logique.

Ici encore, depuis la version 12, l'essentiel du paramétrage nécessaire à un secondaire physique ou logique est intégré dans ce fichier.

Query tuning

Les paramètres qui peuvent influencer l'optimiseur sont à définir dans cette partie, notamment `seq_page_cost` et `random_page_cost` en fonction des disques, et éventuellement le parallélisme, le niveau de finesse des statistiques, le JIT…

Reporting and logging

[2] https://dali.bo/dba1_html

Si le paramétrage par défaut des traces ne convient pas, le modifier ici. Il faudra généralement augmenter leur verbosité. Quelques paramètres `log_*` figurent dans d'autres sections.

Autovacuum

L'autovacuum fonctionne généralement convenablement, et des ajustements se font généralement table par table. Il arrive cependant que certains paramètres doivent être modifiés globalement.

Client connection defaults

Cette partie un peu fourre-tout définit le paramétrage au niveau d'un client : langue, fuseau horaire, extensions à précharger, tablespaces par défaut…

Lock management

Les paramètres de cette section sont rarement modifiés.

9.5 PG_HBA.CONF ET PG_IDENT.CONF

- Authentification multiple :
  - utilisateur / base / source de connexion
- Fichiers :
  - `pg_hba.conf` (*Host Based Authentication*)
  - `pg_ident.conf` : si mécanisme externe d'authentification
  - paramètres : `hba_file` et `ident_file`

L'authentification est paramétrée au moyen du fichier `pg_hba.conf`. Dans ce fichier, pour une tentative de connexion à une base donnée, pour un utilisateur donné, pour un transport (IP, IPV6, Socket Unix, SSL ou non), et pour une source donnée, ce fichier permet de spécifier le mécanisme d'authentification attendu.

Si le mécanisme d'authentification s'appuie sur un système externe (LDAP, Kerberos, Radius…), des tables de correspondances entre utilisateur de la base et utilisateur demandant la connexion peuvent être spécifiées dans `pg_ident.conf`.

Ces noms de fichiers ne sont que les noms par défaut. Ils peuvent tout à fait être remplacés en spécifiant de nouvelles valeurs de `hba_file` et `ident_file` dans `postgresql.conf` (les installations Red Hat et Debian utilisent là aussi des emplacements différents, comme pour `postgresql.conf`).

Leur utilisation est décrite dans notre première formation[3].

[3] https://dali.bo/f_html

9.6 TABLESPACES

- Espace de stockage physique d'objets
  - et non logique !
- Simple répertoire (**hors de PGDATA**) + lien symbolique
- Pour :
  - répartir I/O et volumétrie
  - quotas (par le FS, mais pas en natif)
  - tri sur disque séparé
- Utilisation selon des droits

Par défaut, PostgreSQL se charge du placement des objets sur le disque, dans son répertoire des données, mais il est possible de créer des répertoires de stockage supplémentaires, nommés *tablespaces*.

9.6.0.1 Utilité des tablespaces

Un *tablespace*, vu de PostgreSQL, est un espace de stockage des objets (tables et index principalement). Son rôle est purement physique, il n'a pas à être utilisé pour une séparation *logique* des tables (c'est le rôle des bases et des schémas), encore moins pour gérer des droits.

Pour le système d'exploitation, il s'agit juste d'un répertoire, déclaré ainsi :

```
CREATE TABLESPACE ssd LOCATION '/mnt/ssd/pg';
```

L'idée est de séparer physiquement les objets suivant leur utilisation. Les cas d'utilisation des tablespaces dans PostgreSQL sont :

- l'ajout d'un disque après saturation de la partition du PGDATA sans possibilité de l'étendre au niveau du système (par LVM ou dans la baie de stockage, par exemple) ;
- la répartition des entrées-sorties… si le SAN ou la virtualisation permet encore d'agir à ce niveau ;
- et notamment la séparation des index et des tables, pour répartir les écritures ;
- le déport des fichiers temporaires vers un tablespace dédié, pour la performance ou éviter qu'ils saturent le PGDATA ;
- la séparation entre données froides et chaudes sur des disques de performances différentes, ou encore des index et des tables ;
- les quotas : PostgreSQL ne disposant pas d'un système de quotas, dédier une partition entière d'une taille précise à un tablespace est un contournement ; une transaction voulant étendre un fichier sera alors annulée avec l'erreur `cannot extend file`.

Sans un réel besoin physique, il n'y a pas besoin de créer des tablespaces, et de complexifier l'administration.

Un tablespace n'est pas adapté à une séparation logique des objets. Si vous tenez à distinguer les fichiers de chaque base sans besoin physique, rappelez-vous que PostgreSQL crée déjà un sous-répertoire par base de données dans `PGDATA/base/`.

PostgreSQL ne connaît pas de notion de tablespace en lecture seule, ni de tablespace transportable entre deux bases ou deux instances.

9.6.0.2 Emplacement des tablespaces

Il y a quelques pièges à éviter à la définition d'un tablespace :

Pour des raisons de sécurité et de fiabilité, le répertoire choisi **ne doit pas** être à la racine d'un point de montage. (Cela vaut aussi pour les répertoires PGDATA ou `pg_wal`).

Positionnez toujours les données dans un sous-répertoire, par exemple dans `/mnt/ssd/pg` plutôt que directement dans le point de montage `/mnt/ssd`. (Voir *Utilisation de systèmes de fichiers secondaires*[4] dans la documentation officielle, ou le bug à l'origine de ce conseil[5].)

Surtout, le tablespace doit **impérativement être placé hors de PGDATA**. Certains outils poseraient problème sinon.

Si ce conseil n'est pas suivi, PostgreSQL crée le tablespace mais renvoie un avertissement :

```
WARNING:  tablespace location should not be inside the data directory
CREATE TABLESPACE
```

Il est aussi déconseillé de mettre le numéro de version de PostgreSQL dans le chemin du tablespace. PostgreSQL le gère déjà à l'intérieur du tablespace. Cela évite des incohérences dans les noms des chemins si vous migrez plus tard avec `pg_upgrade`.

[4] https://doc.postgresql.fr/current/creating-cluster.html#CREATING-CLUSTER-MOUNT-POINTS
[5] https://bugzilla.redhat.com/show_bug.cgi?id=1247477#c1

9.6.1 Tablespaces : mise en place

```
-- déclaration
CREATE TABLESPACE ssd LOCATION '/mnt/ssd/pg';
-- droit pour un utilisateur
GRANT CREATE ON TABLESPACE ssd TO un_utilisateur ;
-- pour toute une base
CREATE DATABASE nomdb TABLESPACE ssd;
ALTER  DATABASE nomdb SET default_tablespace TO ssd ;
-- pour une table
CREATE TABLE une_table (…) TABLESPACE ssd ;
ALTER  TABLE une_table SET TABLESPACE ssd ;  -- verrou !
-- pour un index (pas automatique)
ALTER INDEX une_table_i_idx SET TABLESPACE ssd ;
```

Le répertoire du tablespace doit exister et les accès ouverts et restreints à l'utilisateur système sous lequel tourne l'instance (en général **postgres** sous Linux, **Network Service** sous Windows) :

```
# /mnt/ssd/ doit exister (point de montage d'un SSD par exemple)
 # chown postgres:postgres /mnt/ssd/pg
 # chmod 700 /mnt/ssd/pg
```

Les ordres SQL plus haut permettent de :

- créer un tablespace simplement en indiquant son emplacement dans le système de fichiers du serveur ;
- créer une base de données dont le tablespace par défaut sera celui indiqué ;
- modifier le tablespace par défaut d'une base ;
- donner le droit de créer des tables dans un tablespace à un utilisateur (c'est nécessaire avant de l'utiliser) ;
- créer une table dans un tablespace ;
- déplacer une table dans un tablespace ;
- déplacer un index dans un tablespace.

Quelques choses à savoir :

- La table ou l'index est totalement verrouillé le temps du déplacement.
- Par défaut, les nouveaux index ne sont **pas** créés automatiquement dans le même tablespace que la table, mais en fonction de `default_tablespace`.
- Les index existants ne « suivent » pas automatiquement une table déplacée, il faut les déplacer séparément.
- Depuis PostgreSQL 14, il est possible de préciser un tablespace de réindexation lors d'une réindexation (`REINDEX … TABLESPACE …`).

Les tablespaces des tables sont visibles dans la vue système `pg_tables`, dans `\d+` sous psql, et dans `pg_indexes` pour les index :

```
SELECT schemaname, indexname, tablespace
FROM   pg_indexes
WHERE  tablename = 'ma_table';
```

```
 schemaname |  indexname    | tablespace
------------+---------------+------------
 public     | matable_idx   | chaud
 public     | matable_pkey  |
```

9.6.2 Tablespaces : configuration

```
CREATE TABLESPACE  ssd      LOCATION '/mnt/data_ssd/' ;
CREATE TABLESPACE  ssd_tri1 LOCATION '/mnt/temp1' ;
CREATE TABLESPACE  ssd_tri2 LOCATION '/mnt/temp2' ;
GRANT CREATE ON TABLESPACE  ssd  TO dupont ;
GRANT CREATE ON TABLESPACE  ssd_tri1,ssd_tri2  TO dupont ;
```

- `default_tablespace`

```
default_tablespace = ssd   # postgresql.conf

ALTER DATABASE/ROLE nomdb SET default_tablespace = ssd ;
```

- `temp_tablespaces` :
  - tri & tables temporaires, en alternance
  - protéger le PGDATA

```
ALTER ROLE etl SET temp_tablespaces = ssd_tri1,ssd_tri2;
```

9.6.2.1 Tablespaces de données

Le paramètre `default_tablespace` permet d'utiliser un autre tablespace que celui par défaut dans PGDATA. En plus du `postgresql.conf`, il peut être défini au niveau rôle, base, ou le temps d'une session :

```
ALTER DATABASE erp_prod     SET default_tablespace TO ssd ;   -- base
ALTER DATABASE erp_archives SET default_tablespace TO froid ; -- base
ALTER ROLE etl              SET default_tablespace TO ssd ; -- niveau rôle
ALTER ROLE audit IN DATABASE erp_prod SET default_tablespace TO froid ; -- niveau
 ↪  rôle dans une base
SET default_tablespace TO ssd ; -- session
```

9.6.2.2 Tablespaces de tri

Les opérations de tri et les tables temporaires peuvent être déplacées vers un ou plusieurs tablespaces dédiés grâce au paramètre `temp_tablespaces`. Le premier intérêt est de dédier aux tris une partition rapide (SSD, disque local…). Un autre est de ne plus risquer de saturer la partition du PGDATA en cas de fichiers temporaires énormes dans `base/pgsql_tmp/`.

Ne jamais utiliser de RAM disque (comme `tmpfs`) pour des tablespaces de tri : la mémoire de la machine ne doit servir qu'aux applications et outils, au cache de l'OS, et aux tris en RAM. Favorisez ces derniers en jouant sur `work_mem`.
En cas de redémarrage, ce tablespace ne serait d'ailleurs plus utilisable. Un RAM disque est encore plus dangereux pour les tablespaces de données, bien sûr.

Il faudra ouvrir les droits aux utilisateurs ainsi :

```
GRANT CREATE ON TABLESPACE ssd_tri1 TO dupont ;
```

Plusieurs tablespaces temporaires peuvent être paramétrés. Noter que la déclaration se fait sans guillemet. Chaque transaction en choisira un de façon aléatoire à la création d'un objet temporaire, puis utilisera alternativement les autres pour chaque nouveau fichier. C'est un bon moyen de lisser la charge :

- lors d'une requête, un gros fichier de tri peut s'étaler sur plusieurs tablespaces de tri, car il se découpe au besoin en fichiers de 1 Go, voire moins ;
- les différentes tables temporaires d'une même session se répartiront dans les différents tablespaces temporaires ; par contre les différents fichiers d'une grosse table temporaire resteront ensemble dans le même tablespace (une table n'a qu'un tablespace), et ses index avec elle par défaut.

Si un des tablespaces temporaires sature, la requête tombe en erreur immédiatement : PostgreSQL ne regarde pas si autre tablespace temporaire a de la place libre.
Il vaut donc mieux regrouper les espaces disponibles dans un même système de fichiers, et n'avoir qu'un grand tablespace temporaire.

9.6.3 Tablespaces : performance

- Temps d'accès
  - `seq_page_cost` (1)
  - `random_page_cost` (4)
- Opérations simultanées sur le disque
  - `effective_io_concurrency` (1)
  - `maintenance_io_concurrency` (10)

```
ALTER TABLESPACE ssd SET ( random_page_cost = 1 );
ALTER TABLESPACE ssd SET ( effective_io_concurrency    = 500,
                           maintenance_io_concurrency = 500 ) ;
```

Dans le cas de disques de performances différentes, il faut adapter les paramètres concernés aux caractéristiques du tablespace si la valeur par défaut ne convient pas. Ce sont des paramètres classiques qui ne seront pas décrits en détail ici :

- `seq_page_cost` (coût d'accès à un bloc pendant un parcours, défaut 1) ;
- `random_page_cost` (coût d'accès à un bloc isolé, défaut 4) ;
- `effective_io_concurrency` (nombre d'I/O simultanées, défaut 1) et `maintenance_io_concurrency` (idem, pour une opération de maintenance, défaut 10).

Notamment :

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[6], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. Ces valeurs doivent être réduites sur un système très chargé. Une valeur excessive mène au gaspillage de CPU. Le défaut d `effective_io_concurrency` est seulement de 1, et la valeur maximale est 1000.

(Avant la version 13, le principe était le même, mais la valeur exacte de ce paramètre devait être 2 à 5 fois plus basse comme le précise la formule des notes de version de la version 13[7].)

`maintenance_io_concurrency` est similaire à `effective_io_concurrency`, mais pour les opérations de maintenance. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Il faut penser à le monter aussi si on adapte `effective_io_concurrency`.

[6] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[7] https://docs.postgresql.fr/13/release.html

Par exemple, un système paramétré pour des disques classiques aura comme paramètres par défaut :

```
random_page_cost = 4
effective_io_concurrency = 1
maintenance_io_concurrency = 10
```

et un tablespace sur un SSD les surchargera ainsi :

```
ALTER TABLESPACE ssd SET ( random_page_cost = 1 );
ALTER TABLESPACE ssd SET ( effective_io_concurrency   = 500,
                           maintenance_io_concurrency = 500 ) ;
```

9.7 GESTION DES CONNEXIONS

- L'accès à la base se fait par un protocole réseau clairement défini :
  - sockets TCP (IPV4 ou IPV6)
  - sockets Unix (Unix uniquement)
- Les demandes de connexion sont gérées par le *postmaster*.
- Paramètres : `port`, `listen_adresses`, `unix_socket_directories`, `unix_socket_group` et `unix_socket_permissions`

Le processus *postmaster* est en écoute sur les différentes sockets déclarées dans la configuration. Cette déclaration se fait au moyen des paramètres suivants :

- `port` : le port TCP. Il sera aussi utilisé dans le nom du fichier socket Unix (par exemple : `/tmp/.s.PGSQL.5432` ou `/var/run/postgresql/.s.PGSQL.5432` selon les distributions) ;
- `listen_adresses` : la liste des adresses IP du serveur auxquelles s'attacher ;
- `unix_socket_directories` : le répertoire où sera stocké la socket Unix ;
- `unix_socket_group` : le groupe (système) autorisé à accéder à la socket Unix ;
- `unix_socket_permissions` : les droits d'accès à la socket Unix.

Les connexions par socket Unix ne sont possibles sous Windows qu'à partir de la version 13.

9.7.1 TCP

- Paramètres de keepalive TCP
  - `tcp_keepalives_idle`
  - `tcp_keepalives_interval`
  - `tcp_keepalives_count`
- Paramètre de vérification de connexion
  - `client_connection_check_interval` (v14)

Il faut bien faire la distinction entre session TCP et session de PostgreSQL. Si une session TCP sert

de support à une requête particulièrement longue, laquelle ne renvoie pas de données pendant plusieurs minutes, alors le firewall peut considérer la session inactive, même si le statut du backend dans `pg_stat_activity` est `active`.

Il est possible de préciser les propriétés *keepalive* des sockets TCP, pour peu que le système d'exploitation les gère. Le keepalive est un mécanisme de maintien et de vérification des sessions TCP, par l'envoi régulier de messages de vérification sur une session TCP inactive. `tcp_keepalives_idle` est le temps en secondes d'inactivité d'une session TCP avant l'envoi d'un message de keepalive. `tcp_keepalives_interval` est le temps entre un keepalive et le suivant, en cas de non-réponse. `tcp_keepalives_count` est le nombre maximum de paquets sans réponse accepté avant que la session ne soit déclarée comme morte.

Les valeurs par défaut (0) reviennent à utiliser les valeurs par défaut du système d'exploitation.

Le mécanisme de keepalive a deux intérêts :

- il permet de détecter les clients déconnectés même si ceux-ci ne notifient pas la déconnexion (plantage du système d'exploitation, fermeture de la session par un firewall…) ;
- il permet de maintenir une session active au travers de firewalls, qui seraient fermées sinon : la plupart des firewalls ferment une session inactive après 5 minutes, alors que la norme TCP prévoit plusieurs jours.

Un autre cas peut survenir. Parfois, un client lance une requête. Cette requête met du temps à s'exécuter et le client quitte la session avant de récupérer les résultats. Dans ce cas, le serveur continue à exécuter la requête et ne se rendra compte de l'absence du client qu'au moment de renvoyer les premiers résultats. Depuis la version 14, il est possible d'autoriser la vérification de la connexion pendant l'exécution d'une requête. Il faut pour cela définir la durée d'intervalle entre deux vérifications avec le paramètre `client_connection_check_interval`. Par défaut, cette option est désactivée et sa valeur est de 0.

9.7.2 SSL

- Paramètres SSL
  - `ssl`, `ssl_ciphers`, `ssl_renegotiation_limit`

Il existe des options pour activer SSL et le paramétrer. `ssl` vaut `on` ou `off`, `ssl_ciphers` est la liste des algorithmes de chiffrement autorisés, et `ssl_renegotiation_limit` le volume maximum de données échangées sur une session avant renégociation entre le client et le serveur. Le paramétrage SSL impose aussi la présence d'un certificat. Pour plus de détails, consultez la documentation officielle[8].

[8] https://docs.postgresql.fr/current/ssl-tcp.html

9.8 STATISTIQUES SUR L'ACTIVITÉ

- (Ne pas confondre avec statistiques sur les données !)
- Statistiques consultables par des vues systèmes
- Paramètres :
  - `track_activities`, `track_activity_query_size`
  - `track_counts`, `track_io_timing` et `track_functions`
  - `update_process_title`
  - `stats_temp_directory` (< v15)

Les différents processus de PostgreSQL collectent des statistiques d'activité qui ont pour but de mesurer l'activité de la base. Notamment :

- combien de fois cette table a-t-elle été parcourue séquentiellement ?
- combien de blocs ont été trouvés dans le cache pour ce parcours d'index, et combien ont dû être demandés au système d'exploitation ?
- Quelles sont les requêtes en cours d'exécution ?
- Combien de buffers ont été écrits par le processus *background writer* ? Par les processus *backend* eux-mêmes ? durant un checkpoint ?

Il ne faut pas confondre les statistiques d'activité avec celles sur les données (taille des tables, des enregistrements, fréquences des valeurs…), qui sont à destination de l'optimiseur de requête !

Pour des raisons de performance, ces stastistiques restent en mémoire (ou dans des fichiers temporaires jusque PostgreSQL 14) et ne sont stockées durablement qu'en cas d'arrêt propre. Un arrêt brutal implique donc leur réinitialisation !

Voici les paramètres concernés par cette collecte d'informations.

`track_activities` (`on` par défaut) précise si les processus doivent mettre à jour leur activité dans `pg_stat_activity`.

`track_counts` (`on` par défaut) indique que les processus doivent collecter des informations sur leur activité. Il est vital pour le déclenchement de l'autovacuum.

`track_activity_query_size` est la taille maximale du texte de requête pouvant être stocké dans `pg_stat_activity`. 1024 caractères est un défaut souvent insuffisant, à monter vers 10 000 si les requêtes sont longues, voire plus ; cette modification nécessite un redémarrage vu qu'elle touche au dimensionnement de la mémoire partagée.

Disponible depuis la version 14, `compute_query_id` permet d'activer le calcul de l'identifiant de la requête. Ce dernier sera visible dans le champ `query_id` de la vue `pg_stat_activity`, ainsi que dans les traces.

`track_io_timing` (`off` par défaut) précise si les processus doivent collecter des informations de chronométrage sur les lectures et écritures, pour compléter les champs `blk_read_time` et `blk_write_time` des vues `pg_stat_database` et `pg_stat_statements`, ainsi que les plans d'exécutions appelés avec `EXPLAIN (ANALYZE,BUFFERS)` et les traces de l'autovacuum (pour un `VACUUM` comme un `ANALYZE`). Avant de l'activer sur une machine peu performante, vérifiez l'impact avec l'outil `pg_test_timing` (il doit montrer des durées de chronométrage essentiellement sous la microseconde).

`track_functions` indique si les processus doivent aussi collecter des informations sur l'exécution des routines stockées. Les valeurs sont `none` (par défaut), `pl` pour ne tracer que les routines en langages procéduraux, `all` pour tracer aussi les routines en C et en SQL.

`update_process_title` permet de modifier le titre du processus, visible par exemple avec `ps -ef` sous Unix. Il est à `on` par défaut sous Unix, mais il faut le laisser à `off` sous Windows pour des raisons de performance.

Avant la version 15, `stats_temp_directory` servait à indiquer le répertoire de stockage temporaire des statistiques, avant copie dans `pg_stat/` lors d'un arrêt propre. Ce répertoire peut devenir gros, est réécrit fréquemment, et peut devenir source de contention. Il est conseillé de le stocker ailleurs que dans le répertoire de l'instance PostgreSQL, par exemple sur un RAM disque ou `tmpfs` (c'est le défaut sous Debian).

Ce répertoire existe toujours en version 15, notamment si vous utilisez le module `pg_stat_statements`. Cependant, en dehors de ce module, rien d'autre ne l'utilise. Quant au paramètre `stats_temp_directory`, il a disparu.

9.8.1 Statistiques d'activité collectées

- Accès logiques (`INSERT`, `SELECT`…) par table et index
- Accès physiques (blocs) par table, index et séquence
- Activité du *Background Writer*
- Activité par base
- Liste des sessions et informations sur leur activité

9.8.2 Vues système

- Supervision / métrologie
- Diagnostiquer
- Vues système :
  - `pg_stat_user_*`
  - `pg_statio_user_*`
  - `pg_stat_activity` (requêtes)
  - `pg_stat_bgwriter`, `pg_stat_checkpointer` (v17+)
  - `pg_locks`
- Réinitialisation des compteurs : `pg_stat_reset_shared()`

PostgreSQL propose de nombreuses vues, accessibles en SQL, pour obtenir des informations sur son fonctionnement interne. Il est possible d'avoir des informations sur le fonctionnement des bases, des processus d'arrière-plan, des tables, les requêtes en cours…

Statistiques sur les objets :

Pour les statistiques sur les objets, le système fournit à chaque fois trois vues différentes :

- Une pour tous les objets du type. Elle contient *all* dans le nom, `pg_statio_all_tables` par exemple ;
- Une pour uniquement les objets systèmes. Elle contient *sys* dans le nom, `pg_statio_sys_tables` par exemple ;
- Une pour uniquement les objets non-systèmes. Elle contient *user* dans le nom, `pg_statio_user_tables` par exemple.

Les accès logiques aux objets (tables, index et routines) figurent dans les vues `pg_stat_xxx_tables`, `pg_stat_xxx_indexes` et `pg_stat_user_functions`.

```
SELECT * FROM pg_stat_user_tables WHERE relname = 'pgbench_accounts' \gx
```

```
-[ RECORD 1 ]-------+-----------------------------
relid               | 200784
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 6
last_seq_scan       | 2024-11-14 11:37:16.851753+01
seq_tup_read        | 104077729
idx_scan            | 1
last_idx_scan       | 2024-11-13 15:48:32.962717+01
idx_tup_fetch       | 1319553
n_tup_ins           | 0
n_tup_upd           | 0
```

```
n_tup_del           | 0
n_tup_hot_upd       | 0
n_tup_newpage_upd   | 0
n_live_tup          | 0
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ø
last_autovacuum     | ø
last_analyze        | ø
last_autoanalyze    | ø
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
```

Les accès physiques aux objets sont visibles dans les vues `pg_statio_xxx_indexes`, `pg_statio_xxx_tables` et `pg_statio_xxx_sequences`. Une vision plus globale est disponible dans `pg_stat_io` (apparue avec PostgreSQL 16).

```
SELECT * FROM pg_statio_user_tables WHERE relname = 'pgbench_accounts' \gx
```

```
-[ RECORD 1 ]---+-----------------
relid           | 200784
schemaname      | public
relname         | pgbench_accounts
heap_blks_read  | 2993285
heap_blks_hit   | 8720337
idx_blks_read   | 277804
idx_blks_hit    | 6114553
toast_blks_read | ø
toast_blks_hit  | ø
tidx_blks_read  | ø
tidx_blks_hit   | ø
```

Des statistiques globales par base sont aussi disponibles, dans `pg_stat_database` : le nombre de transactions validées et annulées, quelques statistiques sur les sessions, et quelques statistiques sur les accès physiques et en cache, ainsi que sur les opérations logiques.

```
SELECT * FROM pg_stat_database WHERE datname ='pgbench' \gx
```

```
-[ RECORD 1 ]-------+------------------------------
relid               | 200784
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 7
last_seq_scan       | 2024-11-14 15:25:08.632708+01
seq_tup_read        | 199954505
idx_scan            | 1001638
last_idx_scan       | 2024-11-14 15:33:15.346497+01
idx_tup_fetch       | 38260013
n_tup_ins           | 0
n_tup_upd           | 91491186
n_tup_del           | 0
n_tup_hot_upd       | 304994
```

```
n_tup_newpage_upd   | 91186192
n_live_tup          | 98976910
n_dead_tup          | 16426
n_mod_since_analyze | 1481762
n_ins_since_vacuum  | 0
last_vacuum         | 2024-11-14 14:41:20.893236+01
last_autovacuum     | 2024-11-14 16:06:01.192891+01
last_analyze        | 2024-11-14 14:41:45.544659+01
last_autoanalyze    | ø
vacuum_count        | 1
autovacuum_count    | 1
analyze_count       | 1
autoanalyze_count   | 0
```

`pg_stat_bgwriter` stocke les statistiques d'écriture des buffers des *background writer*, et du *checkpointer* (jusqu'en version 16 incluse) et des sessions elles-mêmes. On peut ainsi voir si les *backends* écrivent beaucoup ou peu. À partir de PostgreSQL 17, apparaît `pg_stat_checkpointer` qui reprend les champs sur les *checkpoints* et en ajoute quelques-uns. Cette vue permet de vérifier que les *checkpoints* sont réguliers, donc peu gênants.

Exemple (version 17) :

```
TABLE pg_stat_bgwriter \gx
```

```
-[ RECORD 1 ]----+-------------------------------
buffers_clean    | 3004
maxwritten_clean | 26
buffers_alloc    | 24399160
stats_reset      | 2024-11-05 15:12:27.556173+01
```

```
TABLE pg_stat_checkpointer \gx
```

```
-[ RECORD 1 ]-------+-------------------------------
num_timed           | 282
num_requested       | 2
restartpoints_timed | 0
restartpoints_req   | 0
restartpoints_done  | 0
write_time          | 605908
sync_time           | 3846
buffers_written     | 20656
stats_reset         | 2024-11-05 15:12:27.556173+01
```

Écritures :

`pg_stat_bgwriter` stocke les statistiques d'écriture des buffers des Background Writer, Checkpointer et des sessions elles-mêmes.

Requêtes

`pg_stat_activity` est une des vues les plus utilisées et est souvent le point de départ d'une recherche. Elle donne la liste des processus en cours sur l'instance, en incluant entre autres :

- le numéro de processus sur le serveur (`pid`) ;
- la base de données, le nom dutilisateur, l'adresse et le port du client ;

- les dates de début d'ordre, de transaction ou de session ;
- son statut (active ou non) ;
- la requête en cours, ou la dernière requête si la session ne fait rien ;
- le nom de l'application s'il a été renseigné avec le paramètre `application_name` ;
- le type de processus : session d'un utilisateur (*client backend*), processus interne…

```
SELECT datname, pid, usename, application_name,
       backend_start, state, backend_type, query
FROM   pg_stat_activity \gx

-[ RECORD 1 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26378
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236776+02
state            | ¤
backend_type     | autovacuum launcher
query            |
-[ RECORD 2 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26380
usename          | postgres
application_name |
backend_start    | 2019-10-24 18:25:28.238157+02
state            | ¤
backend_type     | logical replication launcher
query            |
-[ RECORD 3 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22324
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.167611+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + -3810 WHERE…
-[ RECORD 4 ]----+-------------------------------------------------------------
datname          | postgres
pid              | 22429
usename          | postgres
application_name | psql
backend_start    | 2019-10-28 10:27:09.599426+01
state            | active
backend_type     | client backend
query            | select datname, pid, usename, application_name, backend_start…
-[ RECORD 5 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22325
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.172585+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 4360 WHERE…
```

```
-[ RECORD 6 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22326
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.178514+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 2865 WHERE…
-[ RECORD 7 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26376
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235574+02
state            | ¤
backend_type     | background writer
query            |
-[ RECORD 8 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26375
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235064+02
state            | ¤
backend_type     | checkpointer
query            |
-[ RECORD 9 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26377
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236239+02
state            | ¤
backend_type     | walwriter
query            |
```

Les textes des requêtes sont tronqués à 1024 caractères : c'est un problème courant. Il est conseillé de monter le paramètre `track_activity_query_size` à plusieurs kilooctets.

Cette vue fournit aussi les *wait events*, qui indiquent ce qu'une session est en train d'attendre. Cela peut être très divers et inclut la levée d'un verrou sur un objet, celle d'un verrou interne, la fin d'une entrée-sortie… L'absence de *wait event* indique que la requête s'exécute. À noter qu'une session avec un *wait event* peut rester en statut `active`.

Les détails sur les champs `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente) sont disponibles dans le tableau des événements d'attente[9]. de la documentation.

À partir de PostgreSQL 17, la vue `pg_wait_events` peut être directement jointe à `pg_stat_activity`, et son champ `description` évite d'aller voir la documentation :

[9] https://docs.postgresql.fr/current/monitoring-stats.html#WAIT-EVENT-TABLE

```
SELECT datname, application_name, pid,
       wait_event_type, wait_event, query, w.description
FROM pg_stat_activity a
  LEFT OUTER JOIN pg_wait_events w
  ON (a.wait_event_type = w.type AND a.wait_event = w.name)
WHERE backend_type='client backend'
AND   wait_event IS NOT NULL
ORDER BY wait_event DESC LIMIT 4 \gx
```

```
-[ RECORD 1 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786146
wait_event_type  | LWLock
wait_event       | WALWriteLock
query            | UPDATE pgbench_accounts SET abalance = abalance + 4055 WHERE…
description      | ø
-[ RECORD 2 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786190
wait_event_type  | IO
wait_event       | WalSync
query            | UPDATE pgbench_accounts SET abalance = abalance + -1859  WHERE…
description      | Waiting for a WAL file to reach durable storage
-[ RECORD 3 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786145
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 3553 WHERE…
description      | Waiting for a read from a relation data file
-[ RECORD 4 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786143
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 1929 WHERE…
description      | Waiting for a read from a relation data file
```

Le processus de la ligne 2 attend une synchronisation sur disque du journal de transaction (WAL), et les deux suivants une lecture d'un fichier de données. (La description vide en ligne 1 est un souci de la version 17.2).

Pour entrer dans le détail des champs liés aux connexions :

- `backend_type` est le type de processus : on filtrera généralement sur `client backend`, mais on y trouvera aussi des processus de tâche de fond comme `checkpointer`, `walwriter`, `autovacuum launcher` et autres processus de PostgreSQL, ou encore des *workers* lancés par des extensions ;
- `datname` est le nom de la base à laquelle la session est connectée, et `datid` est son identifiant (OID) ;

- `pid` est le processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client, qui durera le temps de la session (sauf parallélisation) ;
- `usename` est le nom de l'utilisateur connecté, et `usesysid` est son OID dans `pg_roles` ;
- `application_name` est un nom facultatif, et il est recommandé que l'application cliente le renseigne autant que possible avec `SET application_name TO 'nom_outil_client'` ;
- `client_addr` est l'adresse IP du client connecté (`NULL` si connexion sur socket Unix), et `client_hostname` est le nom associé à cette IP, renseigné uniquement si `log_hostname` a été passé à `on` (cela peut ralentir les connexions à cause de la résolution DNS) ;
- `client_port` est le numéro de port sur lequel le client est connecté, toujours s'il s'agit d'une connexion IP.

Une requête parallélisée occupe plusieurs processus, et apparaîtra sur plusieurs lignes de `pid` différents. Le champ `leader_pid` indique le processus principal. Les autres processus disparaîtront dès la requête terminée.

Pour les champs liés aux durées de session, transactions et requêtes :

- `backend_start` est le timestamp de l'établissement de la session ;
- `xact_start` est le timestamp de début de la transaction ;
- `query_start` est le timestamp de début de la requête en cours, ou de la dernière requête exécutée ;
- `status` vaut soit `active`, soit `idle` (la session ne fait rien) soit `idle in transaction` (en attente pendant une transaction) ;
- `backend_xid` est l'identifiant de la transaction en cours, s'il y en a une ;
- `backend_xmin` est l'horizon des transactions visibles, et dépend aussi des autres transactions en cours.

Rappelons qu'une session durablement en statut `idle in transaction` bloque le fonctionnement de l'autovacuum car `backend_xmin` est bloqué. Cela peut mener à des tables fragmentées et du gaspillage de place disque.

Depuis PostgreSQL 14, `pg_stat_activity` peut afficher un champ `query_id`, c'est-à-dire un identifiant de requête normalisée (dépouillée des valeurs de paramètres). Il faut que le paramètre `compute_query_id` soit à `on` ou `auto` (le défaut, et alors une extension peut l'activer). Ce champ est utile pour retrouver une requête dans la vue de l'extension `pg_stat_statements`, par exemple.

Certains champs de cette vue ne sont renseignés que si le paramètre `track_activities` est à `on` (valeur par défaut, qu'il est conseillé de laisser ainsi).

À noter qu'il ne faut pas interroger `pg_stat_activity` au sein d'une transaction, son contenu pourrait sembler figé.

Verrous :

`pg_locks` permet de voir les verrous posés sur les objets (principalement les relations comme les tables et les index). Le processus (`pid`) est la clé commune pour la rapprocher de `pg_stat_activity`.

Archivage et réplication :

`pg_stat_archiver` donne des informations sur l'archivage des journaux de transaction et notamment sur les erreurs d'archivage. L'exemple suivant montre un archivage en erreur :

```
TABLE pg_stat_archiver \gx
```

```
-[ RECORD 1 ]------+-----------------------------
archived_count     | 1637
last_archived_wal  | 000000010000000700000E3
last_archived_time | 2024-11-14 16:00:00.418887+01
failed_count       | 13254
last_failed_wal    | 000000010000000700000E4
last_failed_time   | 2024-11-14 16:01:37.347793+01
stats_reset        | 2024-11-05 14:58:00.515774+01
```

`pg_stat_replication` donne des informations sur les serveurs secondaires connectés. Les statistiques sur les conflits entre application de la réplication et requêtes en lecture seule sont disponibles dans `pg_stat_database_conflicts`.

Si les réplications se font par des slots de réplication (optionnels en réplication physique), `pg_stat_replication_slots` donne des informations sur leur état et leur retard.

Autres vues :

Des vues plus spécialisées existent :

`pg_stat_ssl` donne des informations sur les connexions SSL : version SSL, suite de chiffrement, nombre de bits pour l'algorithme de chiffrement, compression, Distinguished Name (DN) du certificat client.

`pg_stat_progress_vacuum`, `pg_stat_progress_analyze`, `pg_stat_progress_create_index`, `pg_stat_progress_cluster`, `pg_stat_progress_basebackup` et `pg_stat_progress_copy` donnent respectivement des informations sur la progression des `VACUUM`, des `ANALYZE`, des créations d'index, des commandes de `VACUUM FULL` et `CLUSTER`, de la commande de réplication `BASE BACKUP` et des `COPY`.

`pg_stat_slru` permet de suivre les accès à différents petits caches internes de PostgreSQL (à partir de PostgreSQL 17).

Réinitialisation :

Ces vues contiennent des compteurs cumulatifs. L'évolution en fonction du temps est souvent gérée par les nombreux outils qui font appel à ces vues. Il existe une fonction pour réinitialiser les compteurs de certaines vues (pas toutes) :

```
SELECT pg_stat_reset_shared('archiver') ;
```

9.9 STATISTIQUES SUR LES DONNÉES

- Statistiques sur les données : `pg_stats`
  - collectées par échantillonnage (`default_statistics_target`)
  - `ANALYZE table`
  - table par table (et pour certains index)
  - colonne par colonne
  - pour de meilleurs plans d'exécution
- Affiner :
  - Échantillonnage

```
ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300 ;
```

  - Statistiques multicolonnes sur demande

```
CREATE STATISTICS nom ON champ1, champ2… FROM nom_table ;
```

Afin de calculer les plans d'exécution des requêtes au mieux, le moteur a besoin de statistiques sur les données qu'il va interroger. Il est très important pour lui de pouvoir estimer la sélectivité d'une clause `WHERE`, l'augmentation ou la diminution du nombre d'enregistrements entraînée par une jointure, tout cela afin de déterminer le coût approximatif d'une requête, et donc de choisir un bon plan d'exécution.

Il ne faut pas les confondre avec les statistiques d'activité, vues précédemment !

Les statistiques sont collectées dans la table `pg_statistic`. La vue `pg_stats` affiche le contenu de cette table système de façon plus accessible.

Les statistiques sont collectées sur :

- chaque colonne de chaque table ;
- les index fonctionnels.

Le recueil des statistiques s'effectue quand on lance un ordre `ANALYZE` sur une table, ou que l'autovacuum le lance de son propre chef.

Les statistiques sont calculées sur un échantillon égal à 300 fois le paramètre `STATISTICS` de la colonne (ou, s'il n'est pas précisé, du paramètre `default_statistics_target`, 100 par défaut).

La vue `pg_stats` affiche les statistiques collectées :

```
\d pg_stats
```

```
                    View "pg_catalog.pg_stats"
         Column          |   Type   | Collation | Nullable | Default
-------------------------+----------+-----------+----------+---------
 schemaname              | name     |           |          |
 tablename               | name     |           |          |
 attname                 | name     |           |          |
 inherited               | boolean  |           |          |
 null_frac               | real     |           |          |
 avg_width               | integer  |           |          |
 n_distinct              | real     |           |          |
 most_common_vals        | anyarray |           |          |
 most_common_freqs       | real[]   |           |          |
 histogram_bounds        | anyarray |           |          |
 correlation             | real     |           |          |
 most_common_elems       | anyarray |           |          |
 most_common_elem_freqs  | real[]   |           |          |
 elem_count_histogram    | real[]   |           |          |
```

- `inherited` : la statistique concerne-t-elle un objet utilisant l'héritage (table parente, dont héritent plusieurs tables) ;
- `null_frac` : fraction d'enregistrements dont la colonne vaut NULL ;
- `avg_width` : taille moyenne de cet attribut dans l'échantillon collecté ;
- `n_distinct` : si positif, c'est le nombre de valeurs distinctes ; si négatif, c'est la fraction de valeurs distinctes pour cette colonne dans la table. Il est possible de forcer la valeur de ce champ s'il est constaté que la collecte des statistiques le calcule mal. Par exemple, pour indiquer à l'optimiseur que chaque valeur apparaît statistiquement deux fois :

```
ALTER TABLE matable ALTER COLUMN yyy SET (n_distinct = -0.5) ;
ANALYZE matable ;
```

- `most_common_vals` et `most_common_freqs` : les valeurs les plus fréquentes de la table, et leur fréquence. Le nombre de valeurs collectées est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target`. Le défaut de 100 échantillons sur 30 000 lignes peut être modifié comme ci-après (sachant que le temps de planification augmente exponentiellement avec ce paramètre, et qu'il vaut mieux ne pas dépasser la valeur 1000) :

```
ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300 ;
```

- `histogram_bounds` : les limites d'histogramme sur la colonne. Les histogrammes permettent d'évaluer la sélectivité d'un filtre par rapport à sa valeur précise. Ils permettent par exemple à l'optimiseur de déterminer que 4,3 % des enregistrements d'une colonne `noms` commencent par un A, ou 0,2 % par AL. Le principe est de regrouper les enregistrements triés dans des groupes de tailles approximativement identiques, et de stocker les limites de ces groupes (on ignore les `most_common_vals`, pour lesquelles il y a déjà une mesure plus précise). Le nombre d' `histogram_bounds` est calculé de la même façon que les `most_common_vals` ;
- `correlation` : le facteur de corrélation statistique entre l'ordre physique et l'ordre logique des enregistrements de la colonne. Il vaudra par exemple `1` si les enregistrements sont physiquement stockés dans l'ordre croissant, `-1` si ils sont dans l'ordre décroissant, ou `0` si ils sont totalement aléatoirement répartis. Ceci sert à affiner le coût d'accès aux enregistrements ;

- `most_common_elems` et `most_common_elems_freqs` : les valeurs les plus fréquentes si la colonne est un tableau (NULL dans les autres cas), et leur fréquence. Le nombre de valeurs collectées est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target` ;
- `elem_count_histogram` : les limites d'histogramme sur la colonne si elle est de type tableau.

Parfois, il est intéressant de calculer des statistiques sur un ensemble de colonnes ou d'expressions. Dans ce cas, il faut créer un objet statistique en indiquant les colonnes et/ou expressions à traiter et le type de statistiques à calculer (voir la documentation de `CREATE STATISTICS`).

9.10 OPTIMISEUR

- SQL est un langage déclaratif :
  - décrit le résultat attendu (projection, sélection, jointure, etc.)…
  - …mais pas comment l'obtenir
  - c'est le rôle de l'optimiseur

Le langage SQL décrit le résultat souhaité. Par exemple :

```
SELECT path, filename
FROM file
JOIN path ON (file.pathid=path.pathid)
WHERE path LIKE '/usr/%'
```

Cet ordre décrit le résultat souhaité. Nous ne précisons pas au moteur comment accéder aux tables `path` et `file` (par index ou parcours complet par exemple), ni comment effectuer la jointure (PostgreSQL dispose de plusieurs méthodes). C'est à l'optimiseur de prendre la décision, en fonction des informations qu'il possède.

Les informations les plus importantes pour lui, dans le contexte de cette requête, seront :

- quelle fraction de la table `path` est ramenée par le critère `path LIKE '/usr/%'` ?
- y a-t-il un index utilisable sur cette colonne ?
- y a-t-il des index utilisables sur `file.pathid`, sur `path.pathid` ?
- quelles sont les tailles des deux tables ?

La stratégie la plus efficace ne sera donc pas la même suivant les informations retournées par toutes ces questions.

Par exemple, il pourrait être intéressant de charger les deux tables séquentiellement, supprimer les enregistrements de `path` ne correspondant pas à la clause `LIKE`, trier les deux jeux d'enregistrements et fusionner les deux jeux de données triés (cette technique est un *merge join*). Cependant, si les tables sont assez volumineuses, et que le `LIKE` est très discriminant (il ramène peu d'enregistrements de la table `path`), la stratégie d'accès sera totalement différente : nous pourrions préférer récupérer les quelques enregistrements de `path` correspondant au `LIKE` par un index, puis pour chacun de ces enregistrements, aller chercher les informations correspondantes dans la table `file` (c'est un *nested loop*).

9.10.1 Optimisation par les coûts

- L'optimiseur évalue les coûts respectifs des différents plans
- Il calcule tous les plans possibles tant que c'est possible
- Le coût de planification exhaustif est exponentiel par rapport au nombre de jointures de la requête
- Il peut falloir d'autres stratégies
- Paramètres principaux :
  - `seq_page_cost`, `random_page_cost`, `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost`
  - `parallel_setup_cost`, `parallel_tuple_cost`
  - `effective_cache_size`

Afin de choisir un bon plan, le moteur essaie des plans d'exécution. Il estime, pour chacun de ces plans, le coût associé. Afin d'évaluer correctement ces coûts, il utilise plusieurs informations :

- Les statistiques sur les données, qui lui permettent d'estimer le nombre d'enregistrements ramenés par chaque étape du plan et le nombre d'opérations de lecture à effectuer pour chaque étape de ce plan ;
- Des informations de paramétrage lui permettant d'associer un coût arbitraire à chacune des opérations à effectuer. Ces informations sont les suivantes :
  - `seq_page_cost` (1 par défaut) : coût de la lecture d'une page disque de façon séquentielle (au sein d'un parcours séquentiel de table par exemple) ;
  - `random_page_cost` (4 par défaut) : coût de la lecture d'une page disque de façon aléatoire (lors d'un accès à une page d'index par exemple) ;
  - `cpu_tuple_cost` (0,01 par défaut) : coût de traitement par le processeur d'un enregistrement de table ;
  - `cpu_index_tuple_cost` (0,005 par défaut) : coût de traitement par le processeur d'un enregistrement d'index ;
  - `cpu_operator_cost` (0,0025 par défaut) : coût de traitement par le processeur de l'exécution d'un opérateur.

Ce sont les coûts relatifs de ces différentes opérations qui sont importants : l'accès à une page de façon aléatoire est par défaut 4 fois plus coûteux que de façon séquentielle, du fait du déplacement des têtes de lecture sur un disque dur classique à plateaux. Ceci prend déjà en considération un potentiel effet du cache. Sur une base fortement en cache, il est donc possible d'être tenté d'abaisser le `random_page_cost` à 3, voire 2,5, ou des valeurs encore bien moindres dans le cas de bases totalement en mémoire.

Le cas des disques SSD est particulièrement intéressant. Ces derniers n'ont pas à proprement parler

de tête de lecture. De ce fait, comme les paramètres `seq_page_cost` et `random_page_cost` sont principalement là pour différencier un accès direct et un accès après déplacement de la tête de lecture, la différence de configuration entre ces deux paramètres n'a pas lieu d'être si les index sont placés sur des disques SSD. Dans ce cas, une configuration très basse et pratiquement identique (voire identique) de ces deux paramètres est préconisée :

```
# pour un SSD
seq_page_cost = 1
random_page_cost = 1.1
```

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[10], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. Ces valeurs doivent être réduites sur un système très chargé. Une valeur excessive mène au gaspillage de CPU. Le défaut d `effective_io_concurrency` est seulement de 1, et la valeur maximale est 1000.

(Avant la version 13, le principe était le même, mais la valeur exacte de ce paramètre devait être 2 à 5 fois plus basse comme le précise la formule des notes de version de la version 13[11].)

Le paramètre `maintenance_io_concurrency` a le même sens que `effective_io_concurrency`, mais pour les opérations de maintenance. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Le défaut est de 10, et il faut penser à le monter aussi comme `effective_io_concurrency` sur des disques plus rapides qu'un simple disque magnétique.

`seq_page_cost`, `random_page_cost`, `effective_io_concurrency` et `maintenance_io_concurrency` peuvent être paramétrés par tablespace, afin de refléter les caractéristiques de disques différents.

La mise en place du parallélisme dans une requête représente un coût : il faut mettre en place une mémoire partagée, lancer des processus… Ce coût est pris en compte par le planificateur à l'aide du paramètre `parallel_setup_cost`. Par ailleurs, le transfert d'enregistrement entre un worker et le processus principal a également un coût représenté par le paramètre `parallel_tuple_cost`.

Ainsi une lecture complète d'une grosse table peut être moins coûteuse sans parallélisation du fait que le nombre de lignes retournées par les workers est très important. En revanche, en filtrant les résultats, le nombre de lignes retournées peut être moins important, la répartition du filtrage entre différents processeurs devient « rentable » et le planificateur peut être amené à choisir un plan comprenant la parallélisation.

Certaines autres informations permettent de nuancer les valeurs précédentes. `effective_cache_size` est la taille totale du cache. Il permet à PostgreSQL de modéliser plus finement le coût réel d'une opération disque, en prenant en compte la probabilité que cette information se trouve dans le cache du système d'exploitation ou dans celui de l'instance, et soit donc moins coûteuse à accéder.

Le parcours de l'espace des solutions est un parcours exhaustif. Sa complexité est principalement liée au nombre de jointures de la requête et est de type exponentiel. Par exemple, planifier de façon ex-

[10] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[11] https://docs.postgresql.fr/13/release.html

haustive une requête à une jointure dure 200 microsecondes environ, contre 7 secondes pour 12 jointures. Une autre stratégie, l'optimiseur génétique, est donc utilisée pour éviter le parcours exhaustif quand le nombre de jointure devient trop élevé.

Pour plus de détails, voir l'article sur les coûts de planification[12] issu de notre base de connaissance.

9.10.2 Nombre de tables considérées par le planificateur

- Réordonne les tables :
  - `join_collapse_limit`
  - `from_collapse_limit`
  - défaut 8, parfois besoin de plus
  - attention au temps de planification, selon le besoin
- GEQO :
  - optimiseur génétique
  - rapide, mais non optimal
  - `geqo` & `geqo_threshold` (≥ 12 tables)

Les paramètres suivants peuvent être modifiés par session.

Nombre de tables considérées :

Les paramètres `join_collapse_limit` et `from_collapse_limit` sont trop peu connus, mais peuvent améliorer radicalement les performances si vous joignez souvent plus de huit tables.

En principe, le planificateur ne tient pas compte de l'ordre des tables dans la clause `FROM` et peut décider de commencer la requête par n'importe laquelle. Cependant, la complexité des plans d'exécution à étudier explose avec le nombre de tables et de combinaisons de jointures, avec toutes leurs possibilités. Les paramètres `from_collapse_limit` et `join_collapse_limit` définissent le nombre de tables prises en compte à la fois.

- `join_collapse_limit` (à 8 par défaut) définit le nombre de tables prises en compte dans le `FROM` (vision simplifiée). S'il y a plus de tables, elles seront jointes au premier résultat dans un deuxième temps. Cela peut donner des résultats catastrophiques si le critère de filtrage le plus utile est sur la neuvième table du `FROM` !

[12] https://support.dalibo.com/kb/cout_planification

- `from_collapse_limit` remonte les tables dans une sous-requête dans le `FROM`, pourvu de ne pas dépasser cette valeur. On le définit habituellement à la même valeur que `join_collapse_limit`.

Comme exemple de plan désastreux à cause d'un critère de filtrage sur une table non prise en compte, voir ce plan[13]. Monter `join_collapse_limit` de 8 à 9 bascule vers un plan bien meilleur[14]. Une autre solution aurait été de remonter la table `INNER JOIN contacts`, qui porte le critère, plus haut dans le `FROM`. Mais il n'est pas toujours possible de modifier le code, qui d'ailleurs peut être dynamique.

Pour éviter de se soucier de ce problème, il est fréquent de monter les valeurs de `join_collapse_limit` et `from_collapse_limit` à 10 ou 12 quand on utilise parfois autant de tables. Des valeurs plus élevées (jusque 20 ou plus) sont plus dangereuses, car le temps de planification peut monter très haut (centaines de millisecondes, voire pire). Mais elles se justifient dans certains domaines. Il est alors préférable de positionner ces valeurs élevées uniquement au niveau de la session, de l'écran, ou de l'utilisateur avec `SET` ou `ALTER ROLE … SET …`.

À l'inverse, descendre `join_collapse_limit` à 1 permet de dicter l'ordre d'exécution au planificateur[15], une mesure de dernier recours.

Parallèment, il ne sera pas inutile de juger de la pertinence d'une requête avec autant de tables.

Optimiseur génétique :

Ce qui suit suppose que `join_collapse_limit` dépasse 12.

PostgreSQL, pour les requêtes trop complexes, bascule vers un optimiseur appelé GEQO (*GEnetic Query Optimizer*). Comme tout algorithme génétique, il fonctionne par introduction de mutations aléatoires sur un état initial donné. Il permet de planifier rapidement une requête complexe, et de fournir un plan d'exécution acceptable.

Le code source de PostgreSQL décrit le principe[16], résumé aussi dans ce schéma :

[13] https://explain.dalibo.com/plan/D0
[14] https://explain.dalibo.com/plan/EQN
[15] https://docs.postgresql.fr/current/explicit-joins.html
[16] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f

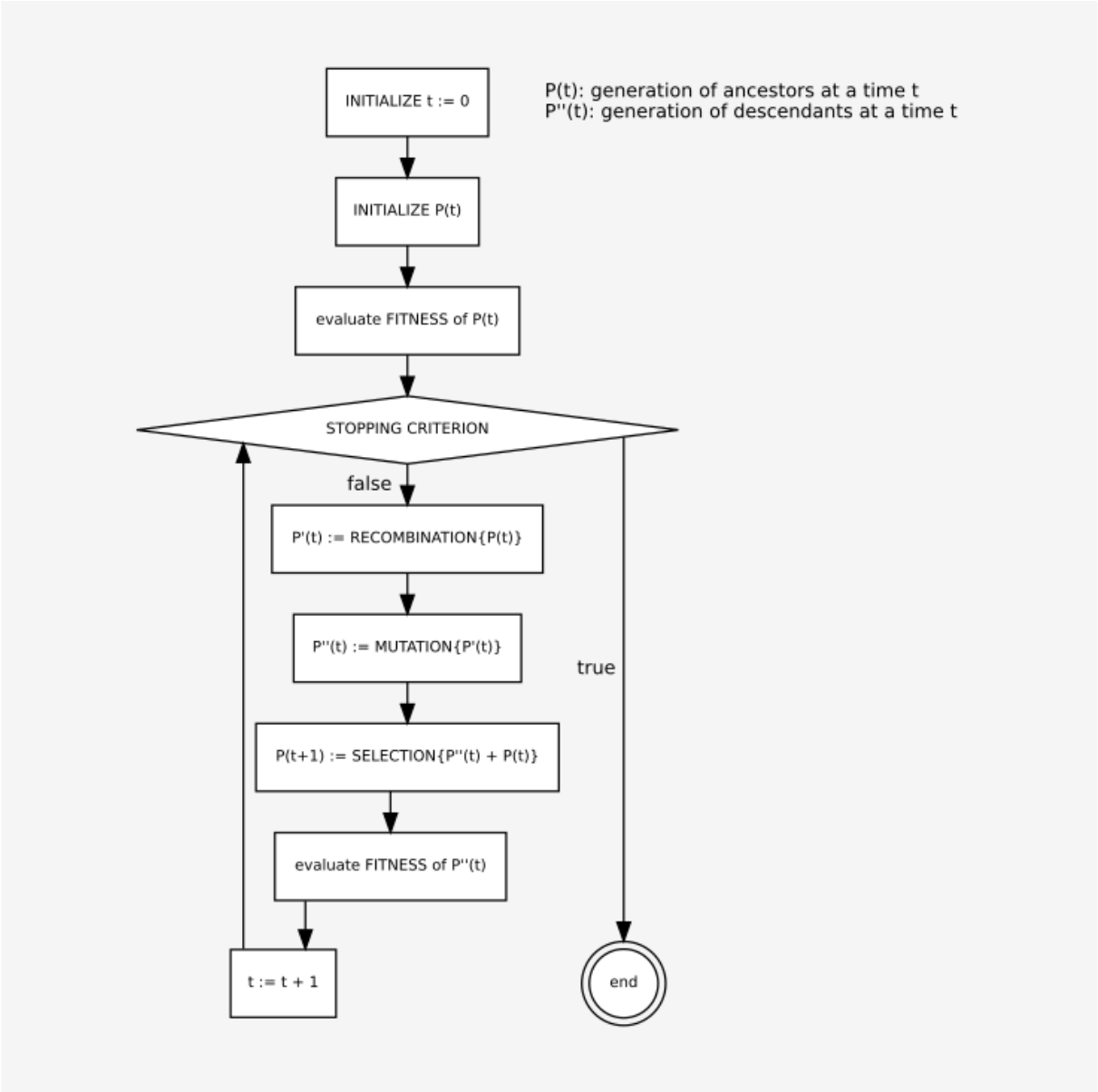


Figure 9/ .1: Principe d'un algorithme génétique (schéma de la documentation officielle, licence PostgreSQL)

Ce mécanisme est configuré par des paramètres dont le nom commence par `geqo`. Exceptés ceux ci-dessous, il est déconseillé d'y toucher sans une bonne connaissance des algorithmes génétiques.

- `geqo`, par défaut à `on`, permet d'activer/désactiver GEQO ;
- `geqo_threshold`, par défaut à 12, est le nombre d'éléments minimum à joindre dans un `FROM` avant d'optimiser celui-ci par GEQO au lieu du planificateur exhaustif ;
- `geqo_seed` (la « graine » pour la génération aléatoire, 0 par défaut) permet de forcer la re-

cherche d'autres plans.

À `geqo_seed` identique, les plans générés seront toujours les mêmes[17] (toutes autres choses identiques par ailleurs).

En production, on ne montera pas `geqo_threshold` ou à peine, et en limitant dans une session. Le nombre de plans à étudier sans l'optimisation du GEQO peut devenir colossal et consommer beaucoup de RAM et de CPU.

9.10.3 Paramètres supplémentaires de l'optimiseur

- Requêtes préparées
  - `plan_cache_mode`
- Partitionnement
  - `constraint_exclusion`
  - `enable_partition_pruning`
- Curseurs
  - `cursor_tuple_fraction`
- Mutualiser les entrées-sorties
  - `synchronize_seqscans`

Tous les paramètres suivants peuvent être modifiés par session.

Requêtes préparées :

Pour les requêtes préparées, par défaut l'optimiseur génère des plans personnalisés pour les cinq premières exécutions d'une requête préparée, puis il bascule sur un plan générique dès que celui-ci devient plus intéressant que la moyenne des plans personnalisés précédents. Le paramètre de configuration `plan_cache_mode` permet de modifier cela :

- `auto` active le comportement par défaut ;
- `force_custom_plan` force le recalcul systématique d'un plan personnalisé pour la requête (on n'économise plus le temps de planification, mais le plan est calculé pour être optimal pour les paramètres, tout en conservant la protection contre les injections SQL permise par les requêtes préparées) ;
- `force_generic_plan` force l'utilisation d'un seul et même plan dès le départ.

[17] https://www.postgresql.org/docs/current/geqo-pg-intro.html#GEQO-PG-INTRO-GEN-POSSIBLE-PLANS

Partitionnement :

Avant la version 10, PostgreSQL ne connaissait qu'un partitionnement par héritage, où l'on crée une table parente et des tables filles héritent de celle-ci, possédant des contraintes `CHECK` comme critères de partitionnement, par exemple `CHECK (date >='2011-01-01' and date < '2011-02-01')` pour une table fille d'un partitionnement par mois. Afin que PostgreSQL ne parcourt que les partitions correspondant à la clause `WHERE` d'une requête, le paramètre `constraint_exclusion` doit valoir `partition` (la valeur par défaut) ou `on`. `partition` est moins coûteux dans un contexte d'utilisation classique car les contraintes d'exclusion ne seront examinées que dans le cas de requêtes `UNION ALL`, qui sont les requêtes générées par le partitionnement.

Pour le partitionnement déclaratif (à favoriser à partir de PostgreSQL 10), `enable_partition_pruning`, activé par défaut, est le paramètre équivalent.

Curseurs :

Lors de l'utilisation de curseurs, le moteur n'a aucun moyen de connaître le nombre d'enregistrements que souhaite récupérer réellement l'utilisateur : peut-être seulement les premiers enregistrements. Si c'est le cas, le plan d'exécution optimal ne sera plus le même. Le paramètre `cursor_tuple_fraction`, par défaut à 0,1, permet d'indiquer à l'optimiseur la fraction du nombre d'enregistrements qu'un curseur souhaitera vraisemblablement récupérer, et lui permettra donc de choisir un plan en conséquence. Si vous utilisez des curseurs, il vaut mieux indiquer explicitement le nombre d'enregistrements dans les requêtes avec `LIMIT`, et passer `cursor_tuple_fraction` à 1,0.

Synchronisation des parcours de table :

Quand plusieurs requêtes souhaitent accéder séquentiellement à la même table, les processus se rattachent à ceux déjà en cours de parcours, afin de profiter des entrées-sorties que ces processus effectuent, le but étant que le système se comporte comme si un seul parcours de la table était en cours, et réduise donc fortement la charge disque. Le seul problème de ce mécanisme est que les processus se rattachant ne parcourent pas la table dans son ordre physique : elles commencent leur parcours de la table à l'endroit où se trouve le processus auquel elles se rattachent, puis rebouclent sur le début de la table. Les résultats n'arrivent donc pas forcément toujours dans le même ordre, ce qui n'est normalement pas un problème (on est censé utiliser `ORDER BY` dans ce cas). Mais il est toujours possible de désactiver ce mécanisme en passant `synchronize_seqscans` à `off`.

9.10.4 Débogage de l'optimiseur

- Permet de valider qu'on est en face d'un problème d'optimiseur.
- Les paramètres sont assez grossiers :
  - défavoriser très fortement un type d'opération
  - pour du diagnostic, pas pour de la production

Ces paramètres dissuadent le moteur d'utiliser un type de nœud d'exécution (en augmentant énormément son coût). Ils permettent de vérifier ou d'invalider une erreur de l'optimiseur. Par exemple :

```
-- création de la table de test
CREATE TABLE test2(a integer, b integer);

-- insertion des données de tests
INSERT INTO test2 SELECT 1, i FROM generate_series(1, 500000) i;

-- analyse des données
ANALYZE test2;

-- désactivation de la parallélisation (pour faciliter la lecture du plan)
SET max_parallel_workers_per_gather TO 0;

-- récupération du plan d'exécution
EXPLAIN ANALYZE SELECT * FROM test2 WHERE a<3;
```

```
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test2  (cost=0.00..8463.00 rows=500000 width=8)
             (actual time=0.031..63.194 rows=500000 loops=1)
   Filter: (a < 3)
 Planning Time: 0.411 ms
 Execution Time: 86.824 ms
```

Le moteur a choisi un parcours séquentiel de table. Si l'on veut vérifier qu'un parcours par l'index sur la colonne `a` n'est pas plus rentable :

```
-- désactivation des parcours SeqScan, IndexOnlyScan et BitmapScan
SET enable_seqscan TO off;
SET enable_indexonlyscan TO off;
SET enable_bitmapscan TO off;

-- création de l'index
CREATE INDEX ON test2(a);

-- récupération du plan d'exécution
EXPLAIN ANALYZE SELECT * FROM test2 WHERE a<3;
```

```
                              QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test2_a_idx on test2  (cost=0.42..16462.42 rows=500000 width=8)
                                  (actual time=0.183..90.926 rows=500000 loops=1)
   Index Cond: (a < 3)
 Planning Time: 0.517 ms
 Execution Time: 111.609 ms
```

Non seulement le plan est plus coûteux, mais il est aussi (et surtout) plus lent.

Attention aux effets du cache : le parcours par index est ici relativement performant à la deuxième exécution parce que les données ont été trouvées dans le cache disque. La requête, sinon, aurait été bien plus lente. La requête initiale est donc non seulement plus rapide, mais aussi plus **sûre** : son temps d'exécution restera prévisible même en cas d'erreur d'estimation sur le nombre d'enregistrements.

Si nous supprimons l'index, nous constatons que le *sequential scan* n'a pas été désactivé. Il a juste été rendu très coûteux par ces options de débogage :

```
-- suppression de l'index
DROP INDEX test2_a_idx;

-- récupération du plan d'exécution
EXPLAIN ANALYZE SELECT * FROM test2 WHERE a<3;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on test2  (cost=10000000000.00..10000008463.00 rows=500000 width=8)
                              (actual time=0.044..60.126 rows=500000 loops=1)
   Filter: (a < 3)
 Planning Time: 0.313 ms
 Execution Time: 82.598 ms
```

Le « très coûteux » est un coût majoré de 10 milliards pour l'exécution d'un nœud interdit.

Voici la liste des options de désactivation :

- `enable_bitmapscan` ;
- `enable_gathermerge` ;
- `enable_hashagg` ;
- `enable_hashjoin` ;
- `enable_incremental_sort` ;
- `enable_indexonlyscan` ;
- `enable_indexscan` ;
- `enable_material` ;
- `enable_mergejoin` ;
- `enable_nestloop` ;
- `enable_parallel_append` ;
- `enable_parallel_hash` ;
- `enable_partition_pruning` ;
- `enable_partitionwise_aggregate` ;
- `enable_partitionwise_join` ;
- `enable_seqscan` ;
- `enable_sort` ;
- `enable_tidscan`.

9.11 CONCLUSION

- Nombreuses fonctionnalités
  - donc nombreux paramètres

9.11.1 Questions

N'hésitez pas, c'est le moment !

9.12 QUIZ

https://dali.bo/m2_quiz

9.13 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m2_solutions.

9.13.1 Tablespace

But : Ajouter un tablespace

Créer un tablespace nommé `ts1` pointant vers `/opt/ts1`.

Se connecter à la base de données `b1`. Créer une table `t_dans_ts1` avec une colonne `id` de type integer dans le tablespace `ts1`.

Récupérer le chemin du fichier correspondant à la table `t_dans_ts1` avec la fonction `pg_relation_filepath`.

Supprimer le tablespace `ts1`. Qu'observe-t-on ?

9.13.2 Statistiques d'activités, tables et vues système

But : Consulter les statistiques d'activité

Créer une table `t3` avec une colonne `id` de type integer.

Insérer 1000 lignes dans la table `t3` avec `generate_series`.

Lire les statistiques d'activité de la table `t3` à l'aide de la vue système `pg_stat_user_tables`.

Créer un utilisateur **pgbench** et créer une base `pgbench` lui appartenant.

Écrire une requête SQL qui affiche le nom et l'identifiant de toutes les bases de données, avec le nom du propriétaire et l'encodage. (Utiliser les table `pg_database` et `pg_roles`).

Comparer la requête avec celle qui est exécutée lorsque l'on tape la commande `\l` dans la console (penser à `\set ECHO_HIDDEN`).

Pour voir les sessions connectées :

- dans un autre terminal, ouvrir une session `psql` sur la base `b0`, qu'on n'utilisera plus ;
- se connecter à la base `b1` depuis une autre session ;
- la vue `pg_stat_activity` affiche les sessions connectées. Qu'y trouve-t-on ?

9.13.3 Statistiques sur les données

But : Consulter les statistiques sur les données

Se connecter à la base de données `b1` et créer une table `t4` avec une colonne `id` de type entier.

Empêcher l'autovacuum d'analyser automatiquement la table `t4`.

Insérer 1 million de lignes dans `t4` avec `generate_series`.

Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

Exécuter la commande `ANALYZE` sur la table `t4`.

Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

Ajouter un index sur la colonne `id` de la table `t4`.

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

> Modifier le contenu de la table `t4` avec `UPDATE t4 SET id = 100000;`

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

> Exécuter la commande `ANALYZE` sur la table `t4`.

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

9.14 TRAVAUX PRATIQUES (SOLUTIONS)

9.14.1 Tablespace

> Créer un tablespace nommé `ts1` pointant vers `/opt/ts1`.

En tant qu'utilisateur **root** :

```
# mkdir /opt/ts1
# chown postgres:postgres /opt/ts1
```

En tant qu'utilisateur **postgres** :

```
$ psql
```

```
postgres=# CREATE TABLESPACE ts1 LOCATION '/opt/ts1';
CREATE TABLESPACE

postgres=# \db
```

```
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
 pg_global  | postgres     |
 ts1        | postgres     | /opt/ts1
```

> Se connecter à la base de données `b1`. Créer une table `t_dans_ts1` avec une colonne `id` de type integer dans le tablespace `ts1`.

```
b1=# CREATE TABLE t_dans_ts1 (id integer) TABLESPACE ts1;
CREATE TABLE
```

> Récupérer le chemin du fichier correspondant à la table `t_dans_ts1` avec la fonction `pg_relation_filepath`.

```
b1=# SELECT current_setting('data_directory') || '/'
     || pg_relation_filepath('t_dans_ts1')
     AS chemin;
```

```
                              chemin
---------------------------------------------------------------------
  /var/lib/pgsql/15/data/pg_tblspc/16394/PG_15_202107181/16393/16395
```

Le fichier n'a pas été créé dans un sous-répertoire du répertoire `base`, mais dans le tablespace indiqué par la commande `CREATE TABLE`. `/opt/ts1` n'apparaît pas ici : il y a un lien symbolique dans le chemin.

```
$ ls -l $PGDATA/pg_tblspc/
total 0
lrwxrwxrwx 1 postgres postgres 8 Apr 16 16:26 16394 -> /opt/ts1
```

```
$ cd /opt/ts1/PG_15_202107181/
$ ls -lR
.:
total 0
drwx------ 2 postgres postgres 18 Apr 16 16:26 16393
./16393:
total 0
-rw------- 1 postgres postgres 0 Apr 16 16:26 16395
```

Il est à noter que ce fichier se trouve réellement dans un sous-répertoire de `/opt/ts1` mais que PostgreSQL le retrouve à partir de `pg_tblspc` grâce à un lien symbolique.

> Supprimer le tablespace `ts1`. Qu'observe-t-on ?

La suppression échoue tant que le tablespace est utilisé. Il faut déplacer la table dans le tablespace par défaut :

```
b1=# DROP TABLESPACE ts1 ;
ERROR:  tablespace "ts1" is not empty

b1=# ALTER TABLE t_dans_ts1 SET TABLESPACE pg_default ;
ALTER TABLE

b1=# DROP TABLESPACE ts1 ;
DROP TABLESPACE
```

9.14.2 Statistiques d'activités, tables et vues système

> Créer une table `t3` avec une colonne `id` de type integer.

```
b1=# CREATE TABLE t3 (id integer);
CREATE TABLE
```

> Insérer 1000 lignes dans la table `t3` avec `generate_series`.

```
b1=# INSERT INTO t3 SELECT generate_series(1, 1000);
INSERT 0 1000
```

> Lire les statistiques d'activité de la table `t3` à l'aide de la vue système `pg_stat_user_tables`.

```
b1=# \x
Expanded display is on.
b1=# SELECT * FROM pg_stat_user_tables WHERE relname = 't3';

-[ RECORD 1 ]-----+-------
relid             | 24594
schemaname        | public
relname           | t3
seq_scan          | 0
```

```
seq_tup_read       | 0
idx_scan           |
idx_tup_fetch      |
n_tup_ins          | 1000
n_tup_upd          | 0
n_tup_del          | 0
n_tup_hot_upd      | 0
n_live_tup         | 1000
n_dead_tup         | 0
last_vacuum        |
last_autovacuum    |
last_analyze       |
last_autoanalyze   |
vacuum_count       | 0
autovacuum_count   | 0
analyze_count      | 0
autoanalyze_count  | 0
```

Les statistiques indiquent bien que 1000 lignes ont été insérées.

> Créer un utilisateur **pgbench** et créer une base `pgbench` lui appartenant.

```
b1=# CREATE ROLE pgbench LOGIN ;
CREATE ROLE

b1=# CREATE DATABASE pgbench OWNER pgbench ;
CREATE DATABASE
```

> Écrire une requête SQL qui affiche le nom et l'identifiant de toutes les bases de données, avec le nom du propriétaire et l'encodage. (Utiliser les table `pg_database` et `pg_roles`).

La liste des bases de données se trouve dans la table `pg_database` :

```
SELECT db.oid, db.datname, datdba
FROM pg_database db ;
```

Une jointure est possible avec la table `pg_roles` pour déterminer le propriétaire des bases :

```
SELECT db.datname, r.rolname, db.encoding
FROM pg_database db, pg_roles r
WHERE db.datdba = r.oid ;
```

d'où par exemple :

```
  datname   | rolname   | encoding
------------+----------+----------
 b1         | postgres |        6
 b0         | postgres |        6
 template0  | postgres |        6
 template1  | postgres |        6
 postgres   | postgres |        6
 pgbench    | pgbench  |        6
```

L'encodage est numérique, il reste à le rendre lisible.

> Comparer la requête avec celle qui est exécutée lorsque l'on tape la commande `\l` dans la console (penser à `\set ECHO_HIDDEN`).

Il est possible de positionner le paramètre `\set ECHO_HIDDEN on`, ou sortir de la console et la lancer de nouveau psql avec l'option `-E` :

```
$ psql -E
```

Taper la commande `\l`. La requête envoyée par `psql` au serveur est affichée juste avant le résultat :

```
\l
********* QUERY **********
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges"
FROM pg_catalog.pg_database d
ORDER BY 1;
**************************
```

L'encodage se retrouve donc en appelant la fonction `pg_encoding_to_char` :

```
b1=# SELECT db.datname, r.rolname, db.encoding,
 ↪  pg_catalog.pg_encoding_to_char(db.encoding)
FROM pg_database db, pg_roles r
WHERE db.datdba = r.oid ;

  datname   | rolname  | encoding | pg_encoding_to_char
------------+----------+----------+---------------------
 b1         | postgres |        6 | UTF8
 b0         | postgres |        6 | UTF8
 template0  | postgres |        6 | UTF8
 template1  | postgres |        6 | UTF8
 postgres   | postgres |        6 | UTF8
 pgbench    | pgbench  |        6 | UTF8
```

> Pour voir les sessions connectées :
>
> - dans un autre terminal, ouvrir une session `psql` sur la base `b0`, qu'on n'utilisera plus ;
> - se connecter à la base `b1` depuis une autre session ;
> - la vue `pg_stat_activity` affiche les sessions connectées. Qu'y trouve-t-on ?

```
# terminal 1
$ psql b0

# terminal 2
$ psql b1
```

La table a de nombreux champs, affichons les plus importants :

```
# SELECT datname, pid, state, usename, application_name AS appp, backend_type, query
  FROM pg_stat_activity ;
 datname | pid  | state  | usename  | appp |         backend_type         | query
---------+------+--------+----------+------+------------------------------+--------
         | 6179 |        |          |      | autovacuum launcher          |
         | 6181 |        | postgres |      | logical replication launcher |
 b0      | 6870 | idle   | postgres | psql | client backend               |
 b1      | 6872 | active | postgres | psql | client backend               | SELECT…
         | 6177 |        |          |      | background writer            |
         | 6176 |        |          |      | checkpointer                 |
         | 6178 |        |          |      | walwriter                    |
(7 rows)
```

La session dans `b1` est `idle`, c'est-à-dire en attente. La seule session active (au moment où elle tournait) est celle qui exécute la requête. Les autres lignes correspondent à des processus système.

Remarque : Ce n'est qu'à partir de la version 10 de PostgreSQL que la vue `pg_stat_activity` liste les processus d'arrière-plan (checkpointer, background writer….). Les connexions clientes peuvent s'obtenir en filtrant sur la colonne `backend_type` le contenu *client backend*.

```
SELECT datname, count(*)
FROM pg_stat_activity
WHERE backend_type = 'client backend'
GROUP BY datname
HAVING count(*)>0;
```

Ce qui donnerait par exemple :

```
 datname | count
---------+-------
 pgbench |    10
 b0      |     5
```

9.14.3 Statistiques sur les données

> Se connecter à la base de données `b1` et créer une table `t4` avec une colonne `id` de type entier.

```
b1=# CREATE TABLE t4 (id integer);
CREATE TABLE
```

> Empêcher l'autovacuum d'analyser automatiquement la table `t4`.

```
b1=# ALTER TABLE t4 SET (autovacuum_enabled=false);
ALTER TABLE
```

NB : ceci n'est à faire qu'à titre d'exercice ! En production, c'est une très mauvaise idée.

> Insérer 1 million de lignes dans `t4` avec `generate_series`.

```
b1=# INSERT INTO t4 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE id = 100000;

                              QUERY PLAN
----------------------------------------------------------------------
 Gather  (cost=1000.00..11866.15 rows=5642 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t4  (cost=0.00..10301.95 rows=2351 width=4)
         Filter: (id = 100000)
```

> Exécuter la commande `ANALYZE` sur la table `t4`.

```
b1=# ANALYZE t4;
ANALYZE
```

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE id = 100000;

                             QUERY PLAN
-------------------------------------------------------------------
 Gather  (cost=1000.00..10633.43 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t4  (cost=0.00..9633.33 rows=1 width=4)
         Filter: (id = 100000)
```

Les statistiques sont beaucoup plus précises. PostgreSQL sait maintenant qu'il ne va récupérer qu'une seule ligne, sur le million de lignes dans la table. C'est le cas typique où un index serait intéressant.

> Ajouter un index sur la colonne `id` de la table `t4`.

```
b1=# CREATE INDEX ON t4(id);
CREATE INDEX
```

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE id = 100000;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using t4_id_idx on t4  (cost=0.42..8.44 rows=1 width=4)
   Index Cond: (id = 100000)
```

Après création de l'index, nous constatons que PostgreSQL choisit un autre plan qui permet d'utiliser cet index.

> Modifier le contenu de la table `t4` avec `UPDATE t4 SET id = 100000;`

```
b1=# UPDATE t4 SET id = 100000;
UPDATE 1000000
```

Toutes les lignes ont donc à présent la même valeur.

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN ANALYZE SELECT * FROM t4 WHERE id = 100000;

                  QUERY PLAN
-------------------------------------------------------
 Index Only Scan using t4_id_idx on t4
       (cost=0.43..8.45 rows=1 width=4)
       (actual time=0.040..265.573 rows=1000000 loops=1)
   Index Cond: (id = 100000)
   Heap Fetches: 1000001
 Planning time: 0.066 ms
 Execution time: 303.026 ms
```

Là, un parcours séquentiel serait plus performant. Mais comme PostgreSQL n'a plus de statistiques à jour, il se trompe de plan et utilise toujours l'index.

> Exécuter la commande `ANALYZE` sur la table `t4`.

```
b1=# ANALYZE t4;
ANALYZE
```

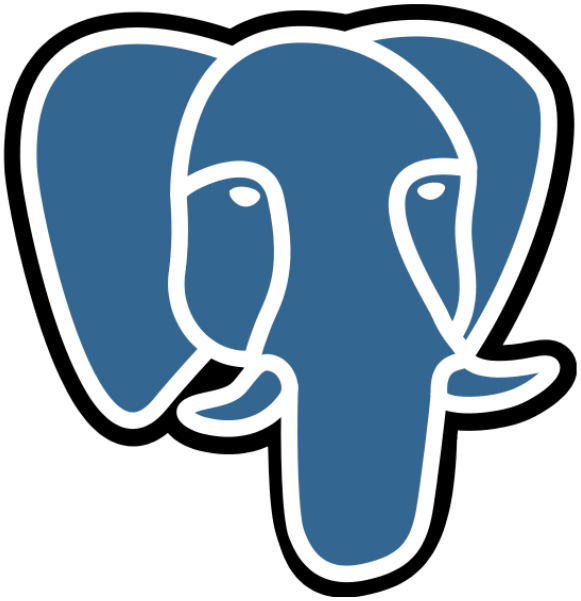
> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN ANALYZE SELECT * FROM t4 WHERE id = 100000;

                     QUERY PLAN
---------------------------------------------------------
 Seq Scan on t4
       (cost=0.00..21350.00 rows=1000000 width=4)
       (actual time=75.185..186.019 rows=1000000 loops=1)
   Filter: (id = 100000)
 Planning time: 0.122 ms
 Execution time: 223.357 ms
```

Avec des statistiques à jour et malgré la présence de l'index, PostgreSQL va utiliser un parcours séquentiel qui, au final, sera plus performant.

Si l'autovacuum avait été activé, les modifications massives dans la table auraient provoqué assez rapidement la mise à jour des statistiques.

10/ Mémoire et journalisation dans PostgreSQL

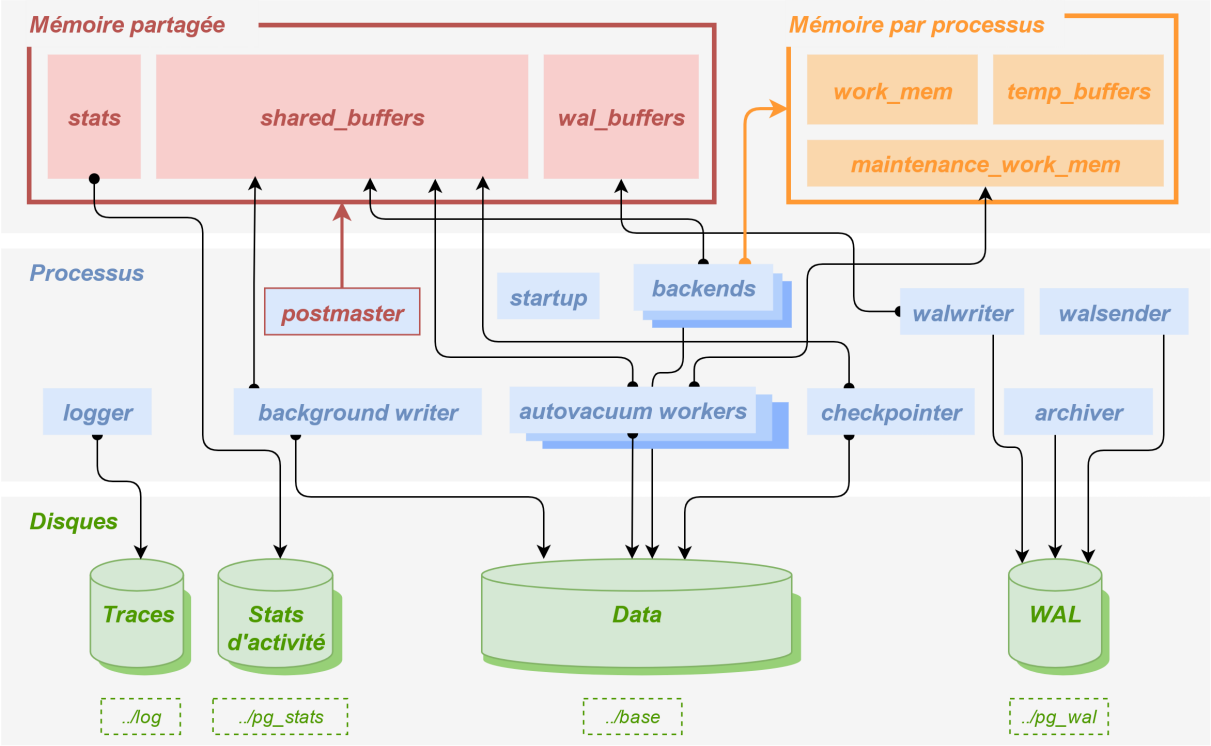


10.1 AU MENU

La mémoire & PostgreSQL :

- Mémoire partagée
- Mémoire des processus
- Les *shared buffers* & la gestion du cache
- La journalisation

10.2 RAPPEL DE L'ARCHITECTURE DE POSTGRESQL



10.3 MÉMOIRE PARTAGÉE

10.3.1 Zones de la mémoire partagée

- `shared_buffers`
  - cache disque des fichiers de données
- `wal_buffers`
  - cache disque des journaux de transactions
- `max_connections`
  - 100… ou plus ?
- `track_activity_query_size`
  - à monter
- verrous
  - `max_connections`, `max_locks_per_transaction`
- SLRU, etc…
- Modification → redémarrage

La zone de mémoire partagée statique est allouée au démarrage de l'instance. Le type de mémoire partagée est configuré avec le paramètre `shared_memory_type`. Sous Linux, il s'agit par défaut de `mmap`, sachant qu'une très petite partie utilise toujours `sysv` (System V). (Il est en principe possible de basculer uniquement en `sysv` mais ceci n'est pas recommandé et nécessite un paramétrage du noyau Linux.) Sous Windows, le type est `windows`.

Les principales zones de mémoire partagées décrites ici sont fixes, et les tailles calculées en fonction de paramètres. Nous verrons en détail l'utilité de certaines de ces zones dans les chapitres suivants.

Shared buffers :

Les *shared buffers* sont le cache des fichiers de données présents sur le disque. Ils représentent de loin la volumétrie la plus importante.

Paramètre associé : `shared_buffers` (à adapter systématiquement)

Wal buffers :

Les *wal buffers* sont le cache des journaux de transaction.

Paramètre associé : `wal_buffers` (rarement modifié)

Données liées aux sessions :

Cet espace mémoire sert à gérer les sessions ouvertes, celles des utilisateurs, mais aussi celles ouvertes par de nombreux processus internes.

Principaux paramètres associés :

- `max_connections`, soit le nombre de connexions simultanées possibles (défaut : 100, souvent suffisant mais à adapter) ;
- `track_activity_query_size` (défaut : 1024 ; souvent monté à 10 000 s'il y a des requêtes de grande taille) ;

Verrous :

La gestion des verrous est décrite dans un autre module[1]. Lors des mises à jour ou suppressions dans les tables, les verrous sur les lignes sont généralement stockés dans les lignes mêmes ; mais de nombreux autres verrous sont stockés en mémoire partagée. Les paramètres associés pour le dimensionnement sont surtout :

- `max_connections` à nouveau ;
- `max_locks_per_transaction`, soit le nombre de verrous possible pour une transaction (défaut : 64, généralement suffisant) ;
- `max_pred_locks_per_relation`, nombre de verrous possible pour une table si le niveau d'isolation « sérialisation » est choisi ;
- mais encore les paramètres liés aux nombres de divers processus internes, ou de transactions préparées.

Les SLRU :

Les SLRU (*Simple Least Recently Used Buffers*) sont de petits caches pour certaines métadonneés de diverses natures : transactions et sous-transactions en cours, horodatage des *commits*, notifications par `NOTIFY`…

Le mécanisme a été revu en PostgreSQL 17 et, depuis, il est possible de modifier la taille des SLRU avec les paramètres `commit_timestamp_buffers`, `multixact_member_buffers`, `multixact_offset_buffers`, `notify_buffers`, `serializable_buffers`, `subtransaction_buffers` et `transaction_buffers`.

Les valeurs par défaut ne font parfois que 16 blocs. Les augmenter peut être intéressant si l'on repère une contention sur l'un d'eux. Un indice est la présence récurrente de *wait events* avec « SLRU » dans le nom. (Pour la liste des *wait events*, voir la table `pg_wait_events`, à partir de la version 17 également. Ils se voient dans `pg_stat_activity`. Il existe aussi une vue `pg_stat_slru`.)

Modification :

Toute modification des paramètres régissant la mémoire partagée imposent un redémarrage de l'instance.

[1] https://dali.bo/m4_html#verrouillage-et-mvcc

10.3.2 Taille de la mémoire partagée

```
-- v15+
SHOW shared_memory_size ;
SHOW shared_memory_size_in_huge_pages ;
```

À partir de la version 15, le paramètre `shared_memory_size` permet de connaître la taille complète de mémoire partagée allouée (un peu supérieure à `shared_buffers` en pratique). Dans le cadre de l'utilisation des *Huge Pages*, il est possible de consulter le paramètre `shared_memory_size_in_huge_pages` pour connaître le nombre de pages mémoires nécessaires (mais on ne peut savoir ici si elles sont utilisées) :

```
postgres=# \dconfig shared*
               Liste des paramètres de configuration
            Paramètre             |             Valeur
----------------------------------+---------------------------------
 shared_buffers                   | 12GB
 shared_memory_size               | 12835MB
 shared_memory_size_in_huge_pages | 6418
 ...
```

Des zones de mémoire partagée non statiques peuvent exister : par exemple, à l'exécution d'une requête parallélisée, les processus impliqués utilisent de la mémoire partagée dynamique. Depuis PostgreSQL 14, une partie peut être pré-allouée avec le paramètre `min_dynamic_shared_memory` (0 par défaut).

10.4 MÉMOIRE PAR PROCESSUS

10.4.1 work_mem, maintenance_work_mem

- `work_mem`
  - × `hash_mem_multiplier`
- `maintenance_work_mem`
  - `autovacuum_work_mem`
- `temp_buffers`
- Pas de limite stricte à la consommation mémoire d'une session !
  - ni à la consommation totale
- Augmenter prudemment & superviser

Les processus de PostgreSQL ont accès à la mémoire partagée, définie principalement par `shared_buffers`, mais ils ont aussi leur mémoire propre. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

- Le paramètre le plus important est `work_mem`, qui définit la taille maximale de la mémoire de travail d'un `ORDER BY`, de certaines jointures, pour la déduplication… que peut utiliser un processus sur un nœud de requête, principalement lors d'opérations de tri ou regroupement.
- Autre paramètre capital, `maintenance_work_mem` qui est la mémoire utilisable pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, `REINDEX`, ajouts de clé étrangère…

Cette mémoire liée au processus est rendue immédiatement après la fin de l'ordre concerné.

- Il existe aussi `logical_decoding_work_mem` (défaut : 64 Mo), utilisable pour chacun des flux de réplication logique (s'il y en a, ils sont rarement nombreux).

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go, voire plus sur les machines récentes, car il concerne des opérations lourdes (indexation, nettoyage des index par `VACUUM`…). Leurs consommations de RAM s'additionnent, mais en pratique ces opérations sont rarement exécutées plusieurs fois simultanément.

Monter au-delà de 1 Go n'a d'intérêt que pour la création ou la réindexation de très gros index.

`autovacuum_work_mem` est la mémoire que s'autorise à prendre l'autovacuum pour les nettoyages d'index, et ce pour chaque *worker* de l'autovacuum (3 maximum par défaut). Par défaut, ce paramètre reprend `maintenance_work_mem`, et est généralement laissé tel quel.

Paramétrage de work_mem :

Pour `work_mem`, c'est beaucoup plus compliqué.

Si `work_mem` est trop bas, beaucoup d'opérations de tri, y compris nombre de jointures, ne s'effectueront pas en RAM. Par exemple, si une jointure par hachage impose d'utiliser 100 Mo en mémoire, mais que `work_mem` vaut 10 Mo, PostgreSQL écrira des dizaines de Mo sur disque à chaque appel de la jointure. Si, par contre, le paramètre `work_mem` vaut 120 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément la requête.

Trop de fichiers temporaires peuvent ralentir les opérations, voire saturer le disque. Un `work_mem` trop bas peut aussi contraindre le planificateur à choisir des plans d'exécution moins optimaux.

Par contre, si `work_mem` est trop haut, et que trop de requêtes le consomment simultanément, le danger est de saturer la RAM. Il n'existe en effet pas de limite à la consommation des sessions de PostgreSQL, ni globalement ni par session !
Or le paramétrage de l'overcommit sous Linux est par défaut très permissif, le noyau ne bloquera rien. La première conséquence de la saturation de mémoire est l'assèchement du cache système (complémentaire de celui de PostgreSQL), et la dégradation des performances. Puis le système va se mettre à swapper, avec à la clé un ralentissement général et durable. Enfin le noyau, à court de mémoire, peut être amené à tuer un processus de PostgreSQL. Cela mène à l'arrêt de l'instance, ou plus fréquemment à son redémarrage brutal avec coupure de toutes les connexions et requêtes en cours.

Toutefois, si l'administrateur paramètre correctement l'overcommit (voir https://dali.bo/j1_html#configuration-de-la-surréservation-mémoire), Linux refusera d'allouer la RAM et la requête tombera en erreur, mais le cache système sera préservé, et PostgreSQL ne tombera pas.

Suivant la complexité des requêtes, il est possible qu'un processus utilise plusieurs fois `work_mem` (par exemple si une requête fait une jointure et un tri, ou qu'un nœud est parallélisé). À l'inverse, beaucoup de requêtes ne nécessitent aucune mémoire de travail.

La valeur de `work_mem` dépend donc beaucoup de la mémoire disponible, des requêtes et du nombre de connexions actives.

Si le nombre de requêtes simultanées est important, `work_mem` devra être faible. Avec peu de requêtes simultanées, `work_mem` pourra être augmenté sans risque.

Il n'y a pas de formule de calcul miracle. Une première estimation courante, bien que très conservatrice, peut être :

> `work_mem` = mémoire / `max_connections`

On obtient alors, sur un serveur dédié avec 16 Go de RAM et 200 connexions autorisées :

```
work_mem = 80MB
```

Mais `max_connections` est fréquemment surdimensionné, et beaucoup de sessions sont inactives. `work_mem` est alors sous-dimensionné.

Plus finement, Christophe Pettus propose en première intention[2] :

> `work_mem` = 4 × mémoire libre / `max_connections`

Soit, pour une machine dédiée avec 16 Go de RAM, donc 4 Go de *shared buffers*, et 200 connections :

```
work_mem = 240MB
```

Dans l'idéal, si l'on a le temps pour une étude, on montera `work_mem` jusqu'à voir disparaître l'essentiel des fichiers temporaires dans les traces, tout en restant loin de saturer la RAM lors des pics de charge.

En pratique, le défaut de 4 Mo est très conservateur, souvent insuffisant. Généralement, la valeur varie entre 10 et 100 Mo. Au-delà de 100 Mo, il y a souvent un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisés pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Augmenter globalement la valeur du `work_mem` peut parfois mener à une consommation excessive de mémoire. Il est possible de ne la modifier que le temps d'une session pour les besoins d'une requête ou d'un traitement particulier :

```
SET work_mem TO '30MB' ;
```

hash_mem_multiplier :

`hash_mem_multiplier` est un paramètre multiplicateur, qui peut s'appliquer à certaines opérations (le hachage, lors de jointures ou agrégations). Par défaut, il vaut 1 en versions 13 et 14, et 2 à partir de la 15. Le seuil de consommation fixé par `work_mem` est alors multiplié d'autant. `hash_mem_multiplier` permet de donner plus de RAM à ces opérations sans augmenter globalement `work_mem`. Il peut lui aussi être modifié dans une session.

Tables temporaires

Les tables temporaires (et leurs index) sont locales à chaque session, et disparaîtront avec elle. Elles sont tout de même écrites sur disque dans le répertoire de la base.

Le cache dédié à ces tables pour minimiser les accès est séparé des *shared buffers*, parce qu'il est propre à la session. Sa taille dépend du paramètre `temp_buffers`. La valeur par défaut (8 Mo) peut

[2] https://thebuild.com/blog/2023/03/13/everything-you-know-about-setting-work_mem-is-wrong/

être insuffisante dans certains cas pour éviter les accès aux fichiers de la table. Elle doit être augmentée dans la session *avant* la création de la table temporaire.

Il ne faut pas laisser inutilement ouvertes des sessions ayant créé des tables temporaires, sinon la mémoire n'est jamais rendue.

10.5 SHARED BUFFERS

10.5.1 Utilité des shared buffers

Shared buffers ou blocs de mémoire partagée

- partage les blocs entre les processus
- cache en lecture ET écriture
- double emploi partiel avec le cache du système
  - pas de *direct I/O*
- caches importants pour les performances !
  - voir `effective_cache_size` (~⅔ RAM)

PostgreSQL dispose de son propre mécanisme de cache. Toute donnée lue l'est de ce cache. Si la donnée n'est pas dans le cache, le processus devant effectuer cette lecture l'y recopie avant d'y accéder dans le cache.

L'unité de travail du cache est le bloc (de 8 ko par défaut) de données. C'est-à-dire qu'un processus charge toujours un bloc dans son entier quand il veut lire un enregistrement. Chaque bloc du cache correspond donc exactement à un bloc d'un fichier d'un objet. Cette information est d'ailleurs, bien sûr, stockée en en-tête du bloc de cache.

Tous les processus accèdent à ce cache unique. C'est la zone la plus importante, par la taille, de la mémoire partagée. Toute modification de données est tracée dans le journal de transaction, **puis** modifiée dans ce cache. Elle n'est donc pas écrite sur le disque par le processus effectuant la modification, sauf en dernière extrémité (voir *Synchronisation en arrière plan*).

Tout accès à un bloc nécessite la prise de verrous. Un *pin lock*, qui est un simple compteur, indique qu'un processus se sert du buffer, et qu'il n'est donc pas réutilisable. C'est un verrou potentiellement de longue durée. Il existe de nombreux autres verrous, de plus courte durée, pour obtenir le droit de modifier le contenu d'un buffer, d'un enregistrement dans un buffer, le droit de recycler un buffer… mais tous ces verrous n'apparaissent pas dans la table `pg_locks`, car ils sont soit de très courte durée, soit partagés (comme le *spin lock*). Il est donc très rare qu'ils soient sources de contention, mais le diagnostic d'une contention à ce niveau est difficile.

Les lectures et écritures de PostgreSQL passent toutefois toujours par le cache du système. Il n'y a pas de *direct I/O* comme dans certains SGBD concurrents. Les deux caches risquent donc de stocker les mêmes informations. Les algorithmes d'éviction sont différents entre le système et PostgreSQL, PostgreSQL disposant de davantage d'informations sur l'utilisation des données, et le type d'accès. La redondance est habituellement limitée, mais il existe des cas problématiques. En conséquence, des travaux sont en cours pour contourner le cache du système quand cela est pertinent. Cela implique de réimplémenter certaines fonctionnalités fournies par le noyau (*prefetching*, *read-ahead*), en

les optimisant pour PostgreSQL. Une première étape apparaît en version 17. PostgreSQL sait depuis longtemps lire plusieurs blocs d'un coup (fonction `pread()`), mais ce n'est pas très bien adapté à la nature fragmentée de son cache. Avec l'utilisation des vectored I/O[3], PostgreSQL peut lire plusieurs blocs contigus sur le système de fichiers, et les écrire à des positions *disjointes* en mémoire virtuelle, avec un seul appel système `preadv()` (voir la page de manuel[4]), (Avec un OS qui ne connait pas la fonction, comme Windows, PostgreSQL se contente d'appeler `pread()` plusieurs fois.) Avec la valeur par défaut d'un nouveau paramètre, `io_combine_limit`, et des blocs de taille standard, on a jusqu'à 16 fois moins d'appels système pour la lecture des blocs en dehors du cache PostgreSQL. En version 17, seule l'implémentation du parcours séquentiel, celle de `ANALYZE`, et celle de `pg_prewarm` utilisent cette nouvelle API. Nous conseillons de ne pas changer la valeur de ce paramètre sans tests sérieux montrant qu'un changement est bénéfique.

10.5.2 Dimensionnement des shared buffers

- En première intention & avant tests :

```
shared_buffers = 25 % RAM généralement
```

- Si > 8 Go :
  - *Huge Pages*,
  - `max_wal_size`, `checkpoint_timeout`...

Dimensionner correctement ce cache est important pour de nombreuses raisons.

Un cache trop petit :

- ralentit l'accès aux données, car des données importantes risquent de ne plus s'y trouver ;
- force l'écriture de données sur le disque, ralentissant les sessions qui auraient pu effectuer uniquement des opérations en mémoire ;
- limite le regroupement d'écritures, dans le cas où un bloc viendrait à être modifié plusieurs fois.

Un cache trop grand :

- limite l'efficacité du cache système en augmentant la redondance de données entre les deux caches ;
- peut ralentir PostgreSQL, car la gestion des `shared_buffers` a un coût de traitement ;
- réduit la mémoire disponible pour d'autres opérations (tris en mémoire notamment), et peut provoquer plus d'erreurs de saturation de la mémoire.

[3] https://en.wikipedia.org/wiki/Vectored_I/O
[4] https://man.freebsd.org/cgi/man.cgi?query=pread&sektion=2

La documentation officielle[5] conseille ceci pour dimensionner `shared_buffers` :

> Un bon point de départ est 25 % de la mémoire vive totale. Ne pas dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine dédiée de 32 Go de RAM, cela donne donc :

```
shared_buffers = 8GB
```

Le défaut de 128 Mo n'est donc pas adapté à un serveur sur une machine récente.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Le cache système limite la plupart du temps l'impact d'un mauvais paramétrage de `shared_buffers`, et il est moins grave de sous-dimensionner un peu `shared_buffers` que de le sur-dimensionner.

> Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres liés aux journaux et *checkpoints* comme `max_wal_size`. Il faut aussi s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…) et donc adapter `work_mem`.

> Modifier `shared_buffers` impose de redémarrer l'instance.

Un cache supplémentaire est disponible pour PostgreSQL : celui du système d'exploitation. Il est donc intéressant de préciser à PostgreSQL la taille approximative du cache, ou du moins de la part du cache qu'occupera PostgreSQL. Le paramètre `effective_cache_size` n'a pas besoin d'être très précis, mais il permet une meilleure estimation des coûts par le moteur. Il est paramétré habituellement aux alentours des ⅔ de la taille de la mémoire vive du système d'exploitation, pour un serveur dédié.

Par exemple pour une machine avec 32 Go de RAM, on peut paramétrer en première intention dans `postgresql.conf` :

```
shared_buffers = '8GB'
effective_cache_size = '21GB'
```

Cela sera à ajuster en fonction du comportement observé de l'application.

[5] https://docs.postgresql.fr/current/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-MEMORY

10.5.3 Notions essentielles de gestion du cache

- Buffer pin
- Buffer dirty/clean
- Compteur d'utilisation
- Clocksweep

Les principales notions à connaître pour comprendre le mécanisme de gestion du cache de PostgreSQL sont :

Buffer pin

Un processus voulant accéder à un bloc du cache (*buffer*) doit d'abord épingler (*to pin*) ce bloc pour forcer son maintien en cache. Pour ce faire, il incrémente le compteur *buffer pin*, puis le décrémente quand il a fini. Un buffer dont le pin est différent de 0 est donc utilisé et ne peut être recyclé.

Buffer dirty/clean

Un buffer est *dirty* (sale) si son contenu a été modifié en mémoire, mais pas encore sur disque. Le fichier de données n'est donc plus à jour (mais a bien été pérennisée sur disque dans les journaux de transactions). Il faudra écrire ce bloc avant de pouvoir le supprimer du cache. Nous verrons plus loin comment.

Un buffer est *clean* (propre) s'il n'a pas été modifié. Le bloc peut être supprimé du cache sans nécessiter le coût d'une écriture sur disque.

Compteur d'utilisation

Cette technique vise à garder dans le cache les blocs les plus utilisés.

À chaque fois qu'un processus a fini de se servir d'un buffer (quand il enlève son pin), ce compteur est incrémenté (à hauteur de 5 dans l'implémentation actuelle). Il est décrémenté par le *clocksweep* évoqué plus bas.

Seul un buffer dont le compteur est à zéro peut voir son contenu remplacé par un nouveau bloc.

Clocksweep (ou algorithme de balayage)

Un processus ayant besoin de charger un bloc de données dans le cache doit trouver un buffer disponible. Soit il y a encore des buffers vides (cela arrive principalement au démarrage d'une instance), soit il faut libérer un buffer.

L'algorithme *clocksweep* parcourt la liste des buffers de façon cyclique à la recherche d'un buffer *unpinned* dont le compteur d'utilisation est à zéro. Tout buffer visité voit son compteur décrémenté de 1. Le système effectue autant de passes que nécessaire sur tous les blocs jusqu'à trouver un buffer à 0. Ce *clocksweep* est effectué par chaque processus, au moment où ce dernier a besoin d'un nouveau buffer.

10.5.4 Ring buffer

But : Le *ring buffer* permet de ne pas purger le cache à cause :

- des grandes tables
- de certaines opérations
  - *Seq Scan*
  - `VACUUM` (écritures)
  - `COPY`, `CREATE TABLE AS SELECT…`
  - etc.
- À partir de PG 16 : `vacuum_buffer_usage_limit`

```
VACUUM (ANALYZE, BUFFER_USAGE_LIMIT '16MB') ;
```

```
vacuumdb --analyze --buffer-usage-limit='16MB'
```

Une table peut être plus grosse que les *shared buffers*. Sa lecture intégrale (lors d'un parcours complet ou d'une opération de maintenance) ne doit pas mener à l'éviction de tous les blocs du cache. PostgreSQL utilise donc plutôt un *ring buffer* quand la taille de la relation dépasse ¼ de `shared_buffers`. Un *ring buffer* est une zone de mémoire gérée à l'écart des autres blocs du cache.

Pour un parcours complet d'une table, cette zone est de 256 ko (taille choisie pour tenir dans un cache L2). Si un bloc y est modifié (`UPDATE`…), il est traité hors du *ring buffer* comme un bloc sale normal.

Pour un `VACUUM` ou un `ANALYZE`, la même technique est utilisée, mais les écritures se font dans le *ring buffer*. Sa taille est de 2 Mo (256 ko seulement jusque PostgreSQL 16 inclus). À partir de PostgreSQL 16, elle peut être augmentée pour accélérer les opérations avec le paramètre `vacuum_buffer_usage_limit` ou ponctuellement ainsi :

```
vacuumdb --buffer-usage-limit='16MB'
```

ou ainsi :

```
VACUUM (ANALYZE, BUFFER_USAGE_LIMIT '16MB') ;
```

En montant à quelques mégaoctets, l'accélération de la vitesse du `VACUUM` peut être notable[6]. C'est aussi très intéressant pour accélérer `ANALYZE`.

Pour les écritures en masse (notamment `COPY` ou `CREATE TABLE AS SELECT`), une technique similaire utilise un *ring buffer* de 16 Mo.

Le site *The Internals of PostgreSQL*[7] et un README[8] dans le code de PostgreSQL entrent plus en détail sur tous ces sujets tout en restant lisibles.

[6] https://blog.dalibo.com/2024/03/26/strategies-acces.html
[7] https://www.interdb.jp/pg/pgsql08.html
[8] https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README

10.5.5 Contenu du cache

2 extensions en « contrib » :

- `pg_buffercache`
- `pg_prewarm`

Deux extensions sont livrées dans les *contribs* de PostgreSQL qui impactent le cache.

`pg_buffercache` permet de consulter le contenu du cache (à utiliser de manière très ponctuelle). La requête suivante indique les objets non système de la base en cours, présents dans le cache et s'ils sont *dirty* ou pas :

```
pgbench=# CREATE EXTENSION pg_buffercache ;

pgbench=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

        relname         | isdirty | blocs | taille
------------------------+---------+-------+---------
 pgbench_accounts       | f       |  8398 | 66 MB
 pgbench_accounts       | t       |  4622 | 36 MB
 pgbench_accounts_pkey  | f       |  2744 | 21 MB
 pgbench_branches       | f       |    14 | 112 kB
 pgbench_branches       | t       |     2 | 16 kB
 pgbench_branches_pkey  | f       |     2 | 16 kB
 pgbench_history        | f       |   267 | 2136 kB
 pgbench_history        | t       |   102 | 816 kB
 pgbench_tellers        | f       |    13 | 104 kB
 pgbench_tellers_pkey   | f       |     2 | 16 kB
```

L'extension `pg_prewarm` permet de précharger un objet dans le cache de PostgreSQL (si le cache est assez gros, bien sûr) :

```
=# CREATE EXTENSION pg_prewarm ;
=# SELECT pg_prewarm ('nom_table_ou_index', 'buffer') ;
```

Il permet même de recharger dès le démarrage le contenu du cache lors d'un arrêt (voir la documentation[9]).

[9] https://docs.postgresql.fr/current/pgprewarm.html

Ces deux outils sont décrits dans le module de formation X2[10].

10.5.6 Synchronisation en arrière-plan

Pour synchroniser les blocs « dirty » :

- **Checkpointer** essentiellement :
  - lors des checkpoints (surtout périodiques)
  - synchronise toutes les pages *dirty*
- **Background writer** :
  - de façon anticipée, selon l'activité
  - une portion des pages
- **Backends**
  - en dernière extrémité

Afin de limiter les attentes des sessions interactives, PostgreSQL dispose de deux processus complémentaires, le `background writer` et le `checkpointer`. Tous deux ont pour rôle d'écrire sur le disque les buffers *dirty* (« sales ») de façon asynchrone. Le but est de ne pas impacter les temps de traitement des requêtes des utilisateurs, donc que les écritures soient lissées sur de grandes plages de temps, pour ne pas saturer les disques, mais en nettoyant assez souvent le cache pour qu'il y ait toujours assez de blocs libérables en cas de besoin.

Le `checkpointer` écrit généralement l'essentiel des blocs *dirty*. Lors des *checkpoints*, il synchronise sur disque tous les blocs modifiés. Son rôle est de lisser cette charge sans saturer les disques.

Comme le `checkpointer` n'est pas censé passer très souvent, le `background writer` anticipe les besoins des sessions, et écrit lui-même une partie des blocs *dirty*.

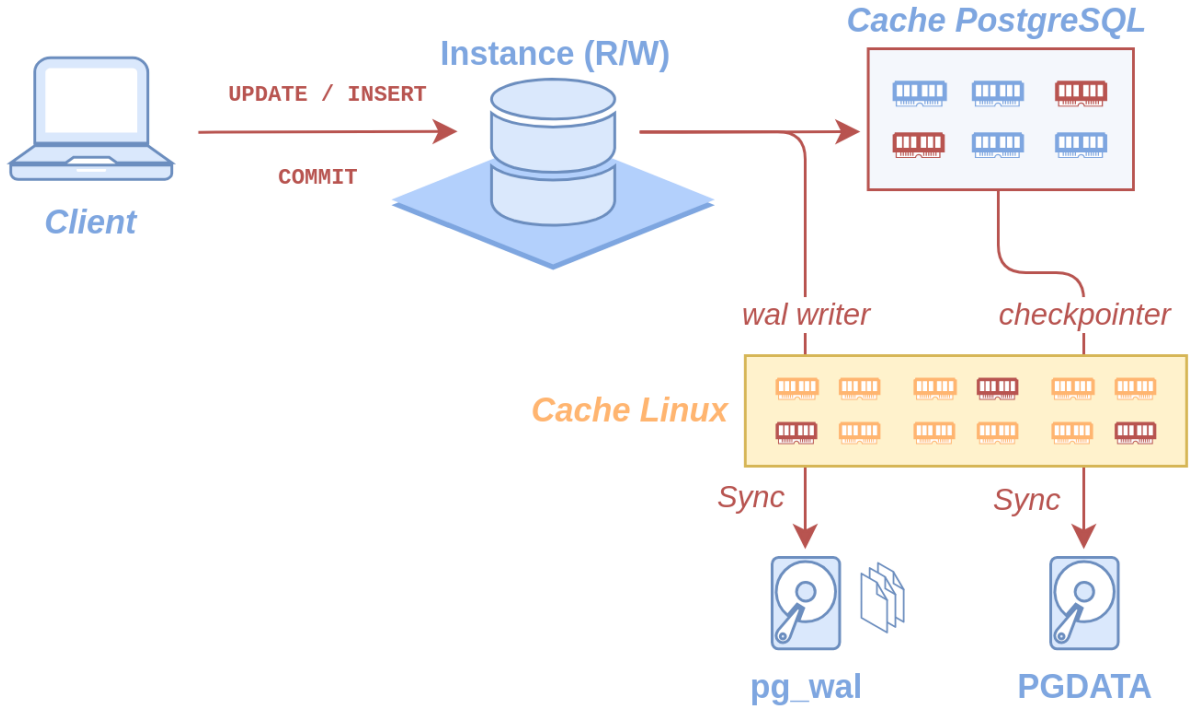
Lors d'écritures intenses, il est possible que ces deux mécanismes soient débordés. Les processus *backend* ne trouvent alors plus en cache de bloc libre ou libérable, et doivent alors écrire eux-mêmes dans les fichiers de données (après les journaux de transaction, bien sûr). Cette situation est évidemment à éviter, ce qui implique généralement de rendre le `background writer` plus agressif.

Nous verrons plus loin les paramètres concernés.

[10] https://dali.bo/x2_html

10.6 JOURNALISATION

10.6.1 Principe de la journalisation



10.6.2 Intégrité & durabilité

- Intégrité : la base reste cohérente malgré :
  - arrêt brutal des processus
  - crash machine
  - …
- Durabilité garantie si `COMMIT`
- Écriture des modifications dans un journal **avant** les fichiers de données
- WAL : *Write Ahead Logging*

La journalisation, sous PostgreSQL, permet de garantir l'intégrité des fichiers, et la durabilité des opérations :

- l'intégrité : la base reste cohérente quoi qu'il arrive ; un arrêt d'urgence ne corrompra pas la base.
- la durabilité : toute donnée validée (`COMMIT`) est écrite physiquement, et un arrêt brutal immédiatement ne va pas la faire disparaître (excepté la perte de tous les disques de stockage et réplicas, bien sûr).

Pour cela, le mécanisme est relativement simple : toute modification affectant un fichier sera d'abord écrite dans le journal. Les modifications affectant les vrais fichiers de données ne sont écrites qu'en mémoire, dans les *shared buffers*.

Les écritures dans le journal, bien que synchrones, sont relativement performantes, car elles sont séquentielles (moins de déplacement de têtes pour les disques magnétiques). Il n'y a que le fichier en cours à synchroniser à chaque `COMMIT`.

Ce n'est généralement que bien plus tard que les modificationss seront écrites de façon asynchrone, soit par un processus recherchant un buffer libre, soit par le `background writer`, soit par le `checkpointer`. Ce dernier processus sait étaler la charge en écriture dans les fichiers de données sur plusieurs minutes, et dans l'idéal il est seul à s'en charger.

Il existe plusieurs configurations et astuces pour arbitrer entre le niveau de durabilité des données exigé et les contraintes de performances : secondaire en réplication synchrone pour une sécurité maximale, désactivation partielle du mécanisme d'enregistrement synchrone des journaux dans une session, tables de travail non journalisées (*unlogged*), regroupement des insertions pour réduire l'impact des `COMMIT`… Aucune ne remet en cause l'intégrité définie dans les modèle de données.

10.6.3 Journaux de transaction (rappels)

Essentiellement :

- `pg_wal/` : journaux de transactions
  - sous-répertoire `archive_status`
  - nom : *timeline*, journal, segment
  - ex : `00000002 00000142 000000FF`
- `pg_xact/` : état des transactions
- **Ces fichiers sont vitaux !**

Rappelons que les journaux de transaction sont des fichiers de 16 Mo par défaut, stockés dans `PGDATA/pg_wal`, dont les noms comportent le numéro de *timeline*, un numéro de journal de 4 Go et un numéro de segment, en hexadécimal.

```
$ ls -l
total 2359320
...
```

```
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001420000007C
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 000000020000001420000007D
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000023
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000024
drwx------ 2 postgres postgres    16384 Mar 26 16:28 archive_status
```

Le sous-répertoire `archive_status` est lié à l'archivage.

D'autres plus petits répertoires comme `pg_xact`, qui contient les statuts des transactions passées, ou `pg_commit_ts`, `pg_multixact`, `pg_serial`, `pg_snapshots`, `pg_subtrans` ou encore `pg_twophase` sont également impliqués.

Tous ces répertoires sont critiques, gérés par PostgreSQL, et ne doivent pas être modifiés !

10.6.4 Que contiennent les journaux de transactions ?

- Outils : `pg_waldump` ou `pg_walinspect`

```
 start_lsn   | record_type |      description       |     block_ref
-------------+-------------+------------------------+----------------
89/3B6B6868 | INSERT_LEAF | off: 1                 | …1663/5/1673829
89/3B6B68A8 | COMMIT      | 2025-01-20 16:40:30.   | ø
89/3B6B68D0 | INSERT      | off: 2, flags: 0x00    | …1663/5/1673826
89/3B6B6910 | INSERT_LEAF | off: 2                 | …1663/5/1673829
89/3B6B6950 | ABORT       | 2025-01-20 16:40:30.   | ø
89/3B6B6978 | UPDATE      | old_xmax: 34696089,    | …1663/5/1673826
89/3B6B69C0 | INSERT_LEAF | off: 3                 | …1663/5/1673829
89/3B6B6A00 | COMMIT      | 2025-01-20 16:40:30.   | ø
```

Il est peu courant de consulter le contenu des journaux, mais ce peut être utile pour chercher le moment exact d'une mauvaise manipulation au niveau applicatif, et savoir à quel moment restaurer l'instance (par exemple à cause d'une table supprimée[11]).

Les journaux de transactions sont des fichiers binaires. Deux outils existent pour les lire :

- `pg_walinspect` est une extension présente depuis PostgreSQL 15, utilisable en SQL, qui nécessite qu'un journal soit encore dans le `pg_wal/` de PostgreSQL pour être étudié (il doit donc être récent) ;
- `pg_waldump` est un outil en ligne de commande, un peu moins souple à manipuler, qui peut être utilisé sur tout journal qu'on a pu récupérer (dans l'archive PITR par exemple).

Exemple de résultat de `pg_walinspect` sur quelques lignes :

[11] https://blog.hagander.net/locating-the-recovery-point-just-before-a-dropped-table-230/

```
/* LSN de départ */
SELECT pg_current_wal_lsn () AS lsn1 \gset

/* Une nouvelle table, une insertion,
   une insertion annulée, une mise à jour*/
CREATE TABLE demo (i int PRIMARY KEY);
INSERT INTO demo SELECT 1 ;
BEGIN ;
  INSERT INTO demo SELECT 2 ;
ROLLBACK ;
UPDATE demo SET i = 3 WHERE i=1;

CHECKPOINT ;

/* LSN final */
SELECT pg_current_wal_lsn () AS lsn2 \gset

SELECT :'lsn1' AS lsn1 ,  :'lsn2' AS lsn2 ;
```

```
    lsn1     |    lsn2
-------------+-------------
 89/3B69CC20 | 89/3B6B6AF8
```

```
CREATE EXTENSION IF NOT EXISTS pg_walinspect ;
/* Affichag des enregistrements de journaux des opérations ci-dessus */
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2') \gx
```

Ce petit exemple génère 80 enregistrements dans les journaux, donc seul un petit extrait figure plus haut. L'essentiel de ces lignes correspondent à des enregistrements dans diverses tables systèmes. Il n'y a que quelques enregistrements concernant directement les données insérées ou modifiées.

Noter les numéros de LSN et de transaction (`xid`) qui peuvent servir de point de restauration.

On trouvera par exemple 3 enregistrements de type `COMMIT`, avec leur heure exacte : le premier correspond à la création de la table, les deux suivants aux insertions d'une ligne validées.

```
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2')
WHERE record_type = 'COMMIT' \gx
```

```
-[ RECORD 1 ]-----------------------------------------------------------------
start_lsn        | 89/3B6B6508
end_lsn          | 89/3B6B6790
prev_lsn         | 89/3B6B6448
xid              | 34696086
resource_manager | Transaction
record_type      | COMMIT
record_length    | 645
main_data_length | 616
fpi_length       | 0
description      | 2025-01-20 16:40:30.534955+01; inval msgs: catcache 80 catcache 79
 ↪  catcache 80 catcache 79 catcache 55 catcache 54 catcache 7 catcache 6 catcache 7
 ↪  catcache 6 catcache 7 catcache 6 catcache 7 catcache 6 catcache 7 catcache 6
 ↪  catcache 7 catcache 6 catcache 7 catcache 6 catcache 55 catcache 54 catcache 7
 ↪  catcache 6 catcache 32 catcache 19 catcache 55 catcache 54 catcache 55 catcache
 ↪  54 snapshot 2608 relcache 1673826 relcache 1673829 relcache 1673826 snapshot
 ↪  2608 relcache 1673826 relcache 1673829
```

```
block_ref        | ø
-[ RECORD 2 ]-----------------------------------------------------------------
start_lsn        | 89/3B6B68A8
end_lsn          | 89/3B6B68D0
prev_lsn         | 89/3B6B6868
xid              | 34696087
resource_manager | Transaction
record_type      | COMMIT
record_length    | 34
main_data_length | 8
fpi_length       | 0
description      | 2025-01-20 16:40:30.547764+01
block_ref        | ø
-[ RECORD 3 ]-----------------------------------------------------------------
start_lsn        | 89/3B6B6A00
end_lsn          | 89/3B6B6A28
prev_lsn         | 89/3B6B69C0
xid              | 34696089
resource_manager | Transaction
record_type      | COMMIT
record_length    | 34
main_data_length | 8
fpi_length       | 0
description      | 2025-01-20 16:40:30.549943+01
block_ref        | ø
```

De même on trouvera un enregistrement de type `ABORT` pour la transaction terminée par `ROLLBACK`.

Il y a non pas un, mais deux, enregistrements de type `CREATE` :

```
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2')
WHERE record_type = 'CREATE' \gx
```

```
-[ RECORD 1 ]----+---------------
start_lsn        | 89/3B69CC50
end_lsn          | 89/3B69CC80
prev_lsn         | 89/3B69CC20
xid              | 34696086
resource_manager | Storage
record_type      | CREATE
record_length    | 42
main_data_length | 16
fpi_length       | 0
description      | base/5/1673826
block_ref        | ø
-[ RECORD 2 ]----+---------------
start_lsn        | 89/3B6AD2C0
end_lsn          | 89/3B6AD2F0
prev_lsn         | 89/3B6AB7C0
xid              | 34696086
resource_manager | Storage
record_type      | CREATE
record_length    | 42
main_data_length | 16
fpi_length       | 0
```

```
description      | base/5/1673829
block_ref        | ø
```

En effet, la table est accompagnée d'un index pour sa clé primaire, et le champ `description` stocke les chemins physiques des objets :

```
SELECT pg_relation_filepath ('demo') as demo_node,
       pg_relation_filepath ('demo_pkey') AS pk_node ;
```

```
   demo_node     |     pk_node
----------------+----------------
 base/5/1673826 | base/5/1673829
```

Dans les insertions, ce sont aussi des informations physiques qui vont apparaître, comme la relation (`rel`), la partie de la table (`main`, c'est-à-dire la partie principale), le bloc, l'offset dans le bloc…

```
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2')
WHERE record_type LIKE 'INSERT%' AND block_ref LIKE '%1673826%' \gx
```

```
…
-[ RECORD 7 ]----+-----------------------------------------------
start_lsn        | 89/3B6B68D0
end_lsn          | 89/3B6B6910
prev_lsn         | 89/3B6B68A8
xid              | 34696088
resource_manager | Heap
record_type      | INSERT
record_length    | 59
main_data_length | 3
fpi_length       | 0
description      | off: 2, flags: 0x00
block_ref        | blkref #0: rel 1663/5/1673826 fork main blk 0
```

On trouvera également un enregistrement de type `UPDATE`, et de nombreux enregistrements de type `INSERT_LEAF` correspondant à des ajouts dans les feuilles d'index. Noter que tous ces enregistrements figurent même pour la transaction annulée, celle-ci alourdit donc inutilement les journaux.

10.6.5 Checkpoint

- « Point de reprise »
- À partir d'où rejouer les journaux ?
- Données écrites au moins au niveau du checkpoint
  - il peut durer
- Processus `checkpointer`

PostgreSQL trace les modifications de données dans les journaux WAL. Ceux-ci sont générés au fur et à mesure des écritures.

Si le système ou l'instance sont arrêtés brutalement, il faut que PostgreSQL puisse appliquer le contenu des journaux non traités sur les fichiers de données. Il a donc besoin de savoir à partir d'où rejouer ces données. Ce point est ce qu'on appelle un *checkpoint*, ou « point de reprise ».

Les principes sont les suivants :

Toute entrée dans les journaux est idempotente, c'est-à-dire qu'elle peut être appliquée plusieurs fois, sans que le résultat final ne soit changé. C'est nécessaire, au cas où la récupération serait interrompue, ou si un fichier sur lequel la reprise est effectuée était plus récent que l'entrée qu'on souhaite appliquer.

Tout fichier de journal antérieur au dernier point de reprise valide **peut être supprimé** ou recyclé, car il n'est plus nécessaire à la récupération.

PostgreSQL a besoin des fichiers de données qui contiennent toutes les données jusqu'au point de reprise. Ils peuvent être plus récents et contenir des informations supplémentaires, ce n'est pas un problème.

Un checkpoint n'est pas un « instantané » cohérent de l'ensemble des fichiers. C'est simplement l'endroit à partir duquel les journaux doivent être rejoués. Il faut donc pouvoir garantir que tous les blocs modifiés dans le cache *au démarrage du checkpoint* auront été synchronisés sur le disque quand le checkpoint sera terminé, et marqué comme dernier checkpoint valide. Un checkpoint peut donc durer plusieurs minutes, sans que cela ne bloque l'activité.

C'est le processus `checkpointer` qui est responsable de l'écriture des buffers devant être synchronisés durant un checkpoint.

10.6.6 Déclenchement & comportement des checkpoints - 1

- Déclenchement périodique (idéal)
  - `checkpoint_timeout`
- ou : Quantité de journaux
  - `max_wal_size` (pas un plafond !)
- ou : `CHECKPOINT`
- À la fin :
  - sync
  - recyclage des journaux
- Espacer les checkpoints peut réduire leur volumétrie

Plusieurs paramètres influencent le comportement des checkpoints.

Dans l'idéal les checkpoints sont périodiques. Le temps maximum entre deux checkpoints est fixé par `checkpoint_timeout` (par défaut 300 secondes). C'est parfois un peu court pour les instances actives.

Le checkpoint intervient aussi quand il y a beaucoup d'écritures et que le volume des journaux dépasse le seuil défini par le paramètre `max_wal_size` (1 Go par défaut). Un checkpoint est alors déclenché.

L'ordre `CHECKPOINT` déclenche aussi un *checkpoint* sans attendre. En fait, il sert surtout à des utilitaires.

Une fois le checkpoint terminé, les journaux sont à priori inutiles. Ils peuvent être effacés pour redescendre en-dessous de la quantité définie par `max_wal_size`. Ils sont généralement « recyclés », c'est-à-dire renommés, et prêt à être réécris.

Cependant, les journaux peuvent encore être retenus dans `pg_wal/` si l'archivage a été activé et que certains n'ont pas été sauvegardés, ou si l'on garde des journaux pour des serveurs secondaires.

À cause de cela, le volume de l'ensemble des fichiers WAL peut largement dépasser la taille fixée par `max_wal_size`. Ce n'est **pas** une valeur plafond !

Il existe un paramètre `min_wal_size` (défaut : 80 Mo) qui fixe la quantité minimale de journaux à tout moment, même sans activité en écriture. Ils seront donc vides et prêts à être remplis en cas d'écriture

imprévue. Bien sûr, s'il y a des grosses écritures, PostgreSQL créera au besoin des journaux supplémentaires, jusque `max_wal_size`, voire au-delà. Mais il lui faudra les créer et les remplir intégralement de zéros avant utilisation.

Après un gros pic d'activité suivi d'un checkpoint et d'une période calme, la quantité de journaux va très progressivement redescendre de `max_wal_size` à `min_wal_size`.

Le dimensionnement de ces paramètres est très dépendant du contexte, de l'activité habituelle, et de la régularité des écritures. Le but est d'éviter des gros pics d'écriture, et donc d'avoir des checkpoints essentiellement périodiques, même si des opérations ponctuelles peuvent y échapper (gros chargements, grosse maintenance…).

Des checkpoints espacés ont aussi pour effet de réduire la quantité totale de journaux écrits. En effet, par défaut, un bloc modifié est intégralement écrit dans les journaux à sa première modification après un checkpoint, mais par la suite seules les modifications de ce bloc sont journalisées. Espacer les checkpoints peut économiser beaucoup de place disque quand les journaux sont archivés, et du réseau s'ils sont répliqués. Par contre, un écart plus grand entre checkpoints peut allonger la restauration après un arrêt brutal, car il y aura plus de journaux à rejouer.

En pratique, une petite instance se contentera du paramétrage de base ; une plus grosse montera `max_wal_size` à plusieurs gigaoctets.

> Si l'on monte `max_wal_size`, par cohérence, il faudra penser à augmenter aussi `checkpoint_timeout`, et vice-versa.

Pour `min_wal_size`, rien n'interdit de prendre une valeur élevée pour mieux absorber les montées d'activité brusques.

Enfin, le checkpoint comprend un *sync* sur disque final. Toujours pour éviter des à-coups d'écriture, PostgreSQL demande au système d'exploitation de forcer un vidage du cache quand `checkpoint_flush_after` a déjà été écrit (par défaut 256 ko). Avant PostgreSQL 9.6, ceci se paramétrait au niveau de Linux en abaissant les valeurs des *sysctl* `vm.dirty_*`. Il y a un intérêt à continuer de le faire, car PostgreSQL n'est pas seul à écrire de gros fichiers (exports `pg_dump`, copie de fichiers…).

10.6.7 Déclenchement & comportement des checkpoints - 2

- Dilution des écritures
  - `checkpoint_completion_target` × durée moy. entre 2 checkpoints
- Surveillance :
  - `checkpoint_warning`
  - `log_checkpoints`
  - Gardez de la place ! sinon crash…

Quand le checkpoint démarre, il vise à lisser au maximum le débit en écriture. La durée d'écriture des données se calcule à partir d'une fraction de la durée d'exécution des précédents checkpoints, fraction fixée par le paramètre `checkpoint_completion_target`. Sa valeur par défaut est celle préconisée par la documentation pour un lissage maximum, soit 0,9 (depuis la version 14, et auparavant le défaut de 0,5 était fréquemment corrigé). Par défaut, PostgreSQL prévoit donc une durée maximale de 300 × 0,9 = 270 secondes pour opérer son checkpoint, mais cette valeur pourra évoluer ensuite suivant la durée réelle des checkpoints précédents.

Il est possible de suivre le déroulé des checkpoints dans les traces si `log_checkpoints` est à `on`. De plus, si deux checkpoints sont rapprochés d'un intervalle de temps inférieur à `checkpoint_warning` (défaut : 30 secondes), un message d'avertissement sera tracé. Une répétition fréquente indique que `max_wal_size` est bien trop petit.

Enfin, répétons que `max_wal_size` n'est pas une limite en dur de la taille de `pg_wal/`.

La partition de `pg_wal/` doit être taillée généreusement. Sa saturation entraîne l'arrêt immédiat de l'instance !

10.6.8 Paramètres du background writer

Nettoyage selon l'activité, en plus du `checkpointer` :

- `bgwriter_delay`
- `bgwriter_lru_maxpages`
- `bgwriter_lru_multiplier`
- `bgwriter_flush_after`

Comme le `checkpointer` ne s'exécute pas très souvent, et ne s'occupe pas des blocs salis depuis son exécution courante, il est épaulé par le `background writer`. Celui-ci pour but de nettoyer une partie des blocs *dirty* pour faire de la place à d'autres. Il s'exécute beaucoup plus fréquemment que le `checkpointer` mais traite moins de blocs à chaque fois.

À intervalle régulier, le `background writer` synchronise un nombre de buffers proportionnel à l'activité sur l'intervalle précédent. Quatre paramètres régissent son comportement :

- `bgwriter_delay` (défaut : 200 ms) : la fréquence à laquelle se réveille le `background writer` ;
- `bgwriter_lru_maxpages` (défaut : 100) : le nombre maximum de pages pouvant être écrites sur chaque tour d'activité. Ce paramètre permet d'éviter que le `background writer` ne veuille synchroniser trop de pages si l'activité des sessions est trop intense : dans ce cas, autant les laisser effectuer elles-mêmes les synchronisations, étant donné que la charge est forte ;
- `bgwriter_lru_multiplier` (defaut : 2) : le coefficient multiplicateur utilisé pour calculer le nombre de buffers à libérer par rapport aux demandes d'allocation sur la période précédente ;
- `bgwriter_flush_after` (défaut : 512 ko sous Linux, 0 ou désactivé ailleurs) : à partir de quelle quantité de données écrites une synchronisation sur disque est demandée.

Pour les paramètres `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier`, *lru* signifie *Least Recently Used* (« moins récemment utilisé »). Ainsi le `background writer` synchronisera les pages du cache qui ont été utilisées le moins récemment.

Ces paramètres permettent de rendre le `background writer` plus agressif si la supervision montre que les processus *backends* écrivent trop souvent les blocs de données eux-mêmes, faute de blocs libres dans le cache, ce qui évidemment ralentit l'exécution du point de vue du client.

Évidemment, certaines écritures massives ne peuvent être absorbées par le `background writer`. Elles provoquent des écritures par les processus *backend* et des checkpoints déclenchés.

10.6.9 WAL buffers : journalisation en mémoire

- Mutualiser les écritures entre transactions
- Un processus d'arrière plan : `walwriter`
- Paramètres notables :
  - `wal_buffers`
  - `wal_writer_flush_after`
- Fiabilité :
  - `fsync` = `on`
  - `full_page_writes` = `on`
  - sinon **corruption** !

La journalisation s'effectue par écriture dans les journaux de transactions. Toutefois, afin de ne pas effectuer des écritures synchrones pour chaque opération dans les fichiers de journaux, les écritures sont préparées dans des tampons (*buffers*) en mémoire. Les processus écrivent donc leur travail de journalisation dans des *buffers*, ou *WAL buffers*. Ceux-ci sont vidés quand une session demande validation de son travail (`COMMIT`), qu'il n'y a plus de *buffer* disponible, ou que le *walwriter* se réveille (`wal_writer_delay`).

Écrire un ou plusieurs blocs séquentiels de façon synchrone sur un disque a le même coût à peu de chose près. Ce mécanisme permet donc de réduire fortement les demandes d'écriture synchrone sur le journal, et augmente donc les performances.

Afin d'éviter qu'un processus n'ait tous les buffers à écrire à l'appel de `COMMIT`, et que cette opération ne dure trop longtemps, un processus d'arrière-plan appelé *walwriter* écrit à intervalle régulier tous les buffers à synchroniser.

Ce mécanisme est géré par ces paramètres, rarement modifiés :

- `wal_buffers` : taille des *WAL buffers*, soit par défaut 1/32e de `shared_buffers` avec un maximum de 16 Mo (la taille d'un segment), des valeurs supérieures (par exemple 128 Mo[12]) pouvant être intéressantes pour les très grosses charges ;
- `wal_writer_delay` (défaut : 200 ms) : intervalle auquel le *walwriter* se réveille pour écrire les buffers non synchronisés ;
- `wal_writer_flush_after` (défaut : 1 Mo) : au-delà de cette valeur, les journaux écrits sont synchronisés sur disque pour éviter l'accumulation dans le cache de l'OS.

Pour la fiabilité, on ne touchera pas à ceux-ci :

[12] https://thebuild.com/blog/2023/02/08/xtreme-postgresql/

- `wal_sync_method` : appel système à utiliser pour demander l'écriture synchrone (sauf très rare exception, PostgreSQL détecte tout seul le bon appel système à utiliser) ;
- `full_page_writes` : doit-on réécrire une image complète d'une page suite à sa première modification après un checkpoint ? Sauf cas très particulier, comme un système de fichiers *Copy On Write* comme ZFS ou btrfs, ce paramètre doit rester à `on` pour éviter des corruptions de données (et il est alors conseillé d'espacer les checkpoints pour réduire la volumétrie des journaux) ;
- `fsync` : doit-on réellement effectuer les écritures synchrones ? Le défaut est `on` et **il est très fortement conseillé de le laisser ainsi en production**. Avec `off`, les performances en écritures sont certes très accélérées, mais en cas d'arrêt d'urgence de l'instance, les données seront totalement corrompues ! Ce peut être intéressant pendant le chargement initial d'une nouvelle instance par exemple, sans oublier de revenir à `on` après ce chargement initial. (D'autres paramètres et techniques existent pour accélérer les écritures et sans corrompre votre instance, si vous êtes prêt à perdre certaines données non critiques : `synchronous_commit` à `off`, les tables *unlogged*…)

10.6.10 Compression des journaux

- `wal_compression`
  - compression des enregistrements
  - moins de journaux
  - un peu de CPU
  - `off` (défaut)
  - `pglz` (`on`), `lz4`, `zstd` (v15)

`wal_compression` compresse les blocs complets enregistrés dans les journaux de transactions, réduisant le volume des WAL, la charge en écriture sur les disques, la volumétrie des journaux archivés des sauvegardes PITR.

Comme il y a moins de journaux, leur rejeu est aussi plus rapide, ce qui accélère la réplication et la reprise après un crash. Le prix est une augmentation de la consommation en CPU.

Les détails et un exemple figurent dans ce billet du blog Dalibo[13].

Depuis PostgreSQL 15, on peut même choisir l'algorithme : `pglz`, `lz4` ou `zstd`. `on` est le synonyme de `pglz`… qui est sans doute le moins bon des trois (voir ce petit test[14]), surtout en terme de consommation CPU.

[13] https://blog.dalibo.com/2024/01/05/cambouis.html
[14] https://www.postgresql.org/message-id/YMmlvyVyAFlxZ%2B/H%40paquier.xyz

10.6.11 Limiter le coût de la journalisation

- `synchronous_commit`
  - perte potentielle de données validées
- `commit_delay` / `commit_siblings`
- Par session

Le coût d'un `fsync` est parfois rédhibitoire. Avec certains sacrifices, il est parfois possible d'améliorer les performances sur ce point.

Le paramètre `synchronous_commit` (défaut : `on`) indique si la validation de la transaction en cours doit déclencher une écriture synchrone dans le journal. Le défaut permet de garantir la pérennité des données dès la fin du `COMMIT`.

Mais ce paramètre peut être modifié dans chaque session par une commande `SET`, et passé à `off` **s'il est possible d'accepter une petite perte de données** pourtant committées. La perte peut monter à `3 × wal_writer_delay` (600 ms) ou `wal_writer_flush_after` (1 Mo) octets écrits. On accélère ainsi notablement les flux des petites transactions. Les transactions où le paramètre reste à `on` continuent de profiter de la sécurité maximale. La base restera, quoi qu'il arrive, cohérente. (Ce paramètre permet aussi de régler le niveau des transactions synchrones avec des secondaires.)

Il existe aussi `commit_delay` (défaut : `0`) et `commit_siblings` (défaut : `5`) comme mécanisme de regroupement de transactions[15]. S'il y au moins `commit_siblings` transactions en cours, PostgreSQL attendra jusqu'à `commit_delay` (en microsecondes) avant de valider une transaction pour permettre à d'autres transactions de s'y rattacher. Ce mécanisme, désactivé par défaut, accroît la latence de certaines transactions afin que plusieurs soient écrites ensembles, et n'apporte un gain de performance global qu'avec de nombreuses petites transactions en parallèle, et des disques classiques un peu lents. (En cas d'arrêt brutal, il n'y a pas à proprement parler de perte de données puisque les transactions délibérément retardées n'ont pas été signalées comme validées.)

[15] https://docs.postgresql.fr/current/wal-configuration.html

10.7 AU-DELÀ DE LA JOURNALISATION

- Sauvegarde PITR
- Réplication physique
  - par *log shipping*
  - par *streaming*

Le système de journalisation de PostgreSQL étant très fiable, des fonctionnalités très intéressantes ont été bâties dessus.

10.7.1 L'archivage des journaux

- Repartir à partir :
  - d'une vieille sauvegarde
  - les journaux archivés
- Sauvegarde à chaud
- Sauvegarde en continu
- Paramètres
  - `wal_level`, `archive_mode`
  - `archive_command` ou `archive_library`

Les journaux permettent de rejouer, suite à un arrêt brutal de la base, toutes les modifications depuis le dernier checkpoint. Les journaux devenus obsolète depuis le dernier *checkpoint* (l'avant-dernier avant la version 11) sont à terme recyclés ou supprimés, car ils ne sont plus nécessaires à la réparation de la base.

Le but de l'archivage est de stocker ces journaux, afin de pouvoir rejouer leur contenu, non plus depuis le dernier checkpoint, mais **depuis une sauvegarde**. Le mécanisme d'archivage permet de repartir d'une sauvegarde binaire de la base (c'est-à-dire des fichiers, pas un `pg_dump`), et de réappliquer le contenu des journaux archivés.

Il suffit de rejouer tous les journaux depuis le checkpoint précédent la sauvegarde jusqu'à la fin de la sauvegarde, ou même à un point précis dans le temps. L'application de ces journaux permet de rendre

à nouveau cohérents les fichiers de données, même si ils ont été sauvegardés en cours de modification.

Ce mécanisme permet aussi de fournir une sauvegarde continue de la base, alors même que celle-ci travaille.

Tout ceci est vu dans le module *Point In Time Recovery*[16].

Même si l'archivage n'est pas en place, il faut connaître les principaux paramètres impliqués :

wal_level :

Il vaut `replica` par défaut depuis la version 10. Les journaux contiennent les informations nécessaires pour une sauvegarde PITR ou une réplication vers une instance secondaire.

Si l'on descend à `minimal` (défaut jusqu'en version 9.6 incluse), les journaux ne contiennent plus que ce qui est nécessaire à une reprise après arrêt brutal sur le serveur en cours. Ce peut être intéressant pour réduire, parfois énormément, le volume des journaux générés, si l'on a bien une sauvegarde non PITR par ailleurs.

Le niveau `logical` est destiné à la réplication logique[17].

(Avant la version 9.6 existaient les niveaux intermédiaires `archive` et `hot_standby`, respectivement pour l'archivage et pour un serveur secondaire en lecture seule. Ils sont toujours acceptés, et assimilés à `replica`.)

archive_mode & **archive_command**/**archive_library** :

Il faut qu'`archive_mode` soit à `on` pour activer l'archivage. Les journaux sont alors copiés grâce à une commande shell à fournir dans `archive_command` ou grâce à une bibliothèque partagée indiquée dans `archive_library` (version 15 ou postérieure). En général on y indiquera ce qu'exige un outil de sauvegarde dédié (par exemple pgBackRest ou barman) dans sa documentation.

10.7.2 Réplication

- *Log shipping* : fichier par fichier
- *Streaming* : entrée par entrée (en flux continu)
- Serveurs secondaires très proches de la production, en lecture

La restauration d'une sauvegarde peut se faire en continu sur un autre serveur, qui peut même être actif (bien que forcément en lecture seule). Les journaux peuvent être :

- envoyés régulièrement vers le secondaire, qui les rejouera : c'est le principe de la réplication par *log shipping* ;

[16] https://dali.bo/i2_html
[17] https://dali.bo/w5_html

- envoyés par fragments vers cet autre serveur : c'est la réplication par *streaming*.

Ces thèmes ne seront pas développés ici. Signalons juste que la réplication par *log shipping* implique un archivage actif sur le primaire, et l'utilisation de `restore_command` (et d'autres pour affiner) sur le secondaire. Le *streaming* permet de se passer d'archivage, même si coupler *streaming* et sauvegarde PITR est une bonne idée. Sur un PostgreSQL récent, le primaire a par défaut le nécessaire activé pour se voir doté d'un secondaire : `wal_level` est à `replica` ; `max_wal_senders` permet d'ouvrir des processus dédiés à la réplication ; et l'on peut garder des journaux en paramétrant `wal_keep_size` (ou `wal_keep_segments` avant la version 13) pour limiter les risques de décrochage du secondaire.

Une configuration supplémentaire doit se faire sur le serveur secondaire, indiquant comment récupérer les fichiers de l'archive, et comment se connecter au primaire pour récupérer des journaux. Elle a lieu dans les fichiers `recovery.conf` (jusqu'à la version 11 comprise), ou (à partir de la version 12) `postgresql.conf` dans les sections évoquées plus haut, ou `postgresql.auto.conf`.

10.8 CONCLUSION

Mémoire et journalisation :

- complexe
- critique
- mais fiable
- et le socle de nombreuses fonctionnalités évoluées

10.8.1 Questions

N'hésitez pas, c'est le moment !

10.9 QUIZ

https://dali.bo/m3_quiz

10.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m3_solutions.

10.10.1 Mémoire partagée

But : constater l'effet du cache sur les accès.

Se connecter à la base de données **b0** et créer une table `t2` avec une colonne `id` de type `integer`.

Insérer 500 lignes dans la table `t2` avec `generate_series`.

Pour réinitialiser les statistiques de `t2` :

- utiliser la fonction `pg_stat_reset_single_table_counters`
- l'OID en paramètre est dans la table des relations `pg_class`, ou peut être trouvé avec `'t2'::regclass`

Afin de vider le cache, redémarrer l'instance PostgreSQL.

Se connecter à la base de données **b0** et lire les données de la table `t2`.

Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

10.10.2 Mémoire de tri

But : constater l'influence de la mémoire de tri

Ouvrir un premier terminal et laisser défiler le fichier de traces.

Dans un second terminal, activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Activer le chronométrage dans la session (`\timing on`). Lire les données de la table `t2` en triant par la colonne `id` Qu'observe-t-on ?

Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

10.10.3 Cache disque de PostgreSQL

But : constater l'effet du cache de PostgreSQL

Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

Créer une table `t2` avec une colonne `id` de type `integer`.

Insérer un million de lignes dans la table `t2` avec `generate_series`.

> Pour vider le cache de PostgreSQL, redémarrer l'instance.

> Pour vider le cache du système d'exploitation, sous **root** :
>
> ```
> # sync && echo 3 > /proc/sys/vm/drop_caches
> ```

> Se connecter à la base de données **b1**. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ? (Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

> Activer l'affichage de la durée des requêtes. Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

> Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

> Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

> Redémarrer l'instance PostgreSQL.

> Se connecter à la base de données **b1** et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

> Modifier le contenu de la table `t2`, par exemple avec :
>
> ```
> UPDATE t2 SET id = 0 WHERE id < 1000 ;
> ```
>
> Que contient le cache de PostgreSQL ?

> Exécuter un checkpoint. Que contient le cache de PostgreSQL ?

10.10.4 Journaux

But : Observer la génération de journaux

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

> Exécuter un checkpoint. Que se passe-t-il au niveau du répertoire `pg_wal` ?

10.11 TRAVAUX PRATIQUES (SOLUTIONS)

10.11.1 Mémoire partagée

> Se connecter à la base de données **b0** et créer une table `t2` avec une colonne `id` de type `integer`.

```
$ psql b0
```

```
b0=# CREATE TABLE t2 (id integer);
CREATE TABLE
```

> Insérer 500 lignes dans la table `t2` avec `generate_series`.

```
b0=# INSERT INTO t2 SELECT generate_series(1, 500);
INSERT 0 500
```

> Pour réinitialiser les statistiques de `t2` :
>
> - utiliser la fonction `pg_stat_reset_single_table_counters`
> - l'OID en paramètre est dans la table des relations `pg_class`, ou peut être trouvé avec `'t2'::regclass` Cette fonction attend un OID comme paramètre :

```
b0=# \df pg_stat_reset_single_table_counters
```

```
List of functions
-[ RECORD 1 ]-------+---------------------
Schema              | pg_catalog
Name                | pg_relation_filepath
Result data type    | text
Argument data types | regclass
Type                | func
```

L'OID est une colonne présente dans la table `pg_class` :

```
b0=# SELECT relname, pg_stat_reset_single_table_counters(oid)
     FROM pg_class WHERE relname = 't2';
```

```
 relname | pg_stat_reset_single_table_counters
---------+-------------------------------------
 t2      |
```

Il y a cependant un raccourci à connaître :

```
SELECT pg_stat_reset_single_table_counters('t2'::regclass) ;
```

> Afin de vider le cache, redémarrer l'instance PostgreSQL.

```
# systemctl restart postgresql-15
```

> Se connecter à la base de données **b0** et lire les données de la table `t2`.

```
b0=# SELECT * FROM t2;
[...]
```

> Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

```
b0=# \x
Expanded display is on.

b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2' ;

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 0
idx_blks_read   |
idx_blks_hit    |
toast_blks_read |
toast_blks_hit  |
tidx_blks_read  |
tidx_blks_hit   |
```

3 blocs ont été lus en dehors du cache de PostgreSQL (colonne `heap_blks_read`).

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2;
[...]
b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2';

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 3
…
```

Les 3 blocs sont maintenant lus à partir du cache de PostgreSQL (colonne `heap_blks_hit`).

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2;
[...]
b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2';

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
```

```
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 6
…
```

Quelle que soit la session, le cache étant partagé, tout le monde profite des données en cache.

10.11.2 Mémoire de tri

> Ouvrir un premier terminal et laisser défiler le fichier de traces.

Le nom du fichier dépend de l'installation et du moment. Pour suivre tout ce qui se passe dans le fichier de traces, utiliser `tail -f` :

```
$ tail -f /var/lib/pgsql/15/data/log/postgresql-Tue.log
```

> Dans un second terminal, activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Dans la session :

```
postgres=# SET client_min_messages TO log;
SET
postgres=# SET log_temp_files TO 0;
SET
```

Les paramètres `log_temp_files` et `client_min_messages` peuvent aussi être mis en place une fois pour toutes dans `postgresql.conf` (recharger la configuration). En fait, c'est généralement conseillé.

> Insérer un million de lignes dans la table `t2` avec `generate_series`.

```
b0=# INSERT INTO t2 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

> Activer le chronométrage dans la session (`\timing on`). Lire les données de la table `t2` en triant par la colonne `id` Qu'observe-t-on ?

```
b0=# \timing on
b0=# SELECT * FROM t2 ORDER BY id;
```

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp1197.0", size 14032896
   id
---------
       1
       1
       2
       2
       3
[...]
Time: 436.308 ms
```

Le message `LOG` apparaît aussi dans la trace, et en général il se trouvera là.

PostgreSQL a dû créer un fichier temporaire pour stocker le résultat temporaire du tri. Ce fichier s'appelle `base/pgsql_tmp/pgsql_tmp1197.0`. Il est spécifique à la session et sera détruit dès qu'il ne sera plus utile. Il fait 14 Mo.

Écrire un fichier de tri sur disque prend évidemment un certain temps, c'est généralement à éviter si le tri peut se faire en mémoire.

> Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

```
b0=# SET work_mem TO '100MB';
SET
```

> Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2 ORDER BY id;

   id
---------
       1
       1
       2
       2
[...]
Time: 240.565 ms
```

Il n'y a plus de fichier temporaire généré. La durée d'exécution est bien moindre.

10.11.3 Cache disque de PostgreSQL

> Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

```
b1=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

> Créer une table `t2` avec une colonne `id` de type `integer`.

```
b1=# CREATE TABLE t2 (id integer);
CREATE TABLE
```

> Insérer un million de lignes dans la table `t2` avec `generate_series`.

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

> Pour vider le cache de PostgreSQL, redémarrer l'instance.

```
# systemctl restart postgresql-15
```

> Pour vider le cache du système d'exploitation, sous **root** :
>
> `# sync && echo 3 > /proc/sys/vm/drop_caches`

> Se connecter à la base de données **b1**. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ? (Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

```
b1=# SELECT relfilenode, count(*)
  FROM pg_buffercache
  GROUP BY 1
  ORDER BY 2 DESC
  LIMIT 10;

 relfilenode | count
-------------+-------
             | 16181
        1249 |    57
        1259 |    26
        2659 |    15
[...]
```

Les valeurs exactes peuvent varier. La colonne `relfilenode` correspond à l'identifiant système de la table. La deuxième colonne indique le nombre de blocs. Il y a ici 16 181 blocs non utilisés pour l'instant dans le cache (126 Mo), ce qui est logique vu que PostgreSQL vient de redémarrer. Il y a quelques blocs utilisés par des tables systèmes, mais aucune table utilisateur (on les repère par leur OID supérieur à 16384).

> Activer l'affichage de la durée des requêtes. Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

```
b1=# \timing on
Timing is on.

b1=# SELECT * FROM t2;

   id
---------
       1
       2
       3
       4
       5
[...]
Time: 277.800 ms

b1=# SELECT relfilenode, count(*) FROM pg_buffercache
     GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;
```

```
 relfilenode | count
-------------+-------
             | 16220
       16410 |    32
        1249 |    29
        1259 |     9
        2659 |     8
[...]
Time: 30.694 ms
```

32 blocs ont été alloués pour la lecture de la table `t2` (*filenode* 16410). Cela représente 256 ko alors que la table fait 35 Mo :

```
b1=# SELECT pg_size_pretty(pg_table_size('t2'));

 pg_size_pretty
----------------
 35 MB
(1 row)

Time: 1.913 ms
```

Un simple `SELECT *` ne suffit donc pas à maintenir la table dans le cache. Par contre, ce deuxième accès était déjà beaucoup rapide, ce qui suggère que le système d'exploitation, lui, a probablement gardé les fichiers de la table dans son propre cache.

> Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

```
b1=# SELECT * FROM t2;

   id
---------
[...]
Time: 184.529 ms

b1=#  SELECT relfilenode, count(*) FROM pg_buffercache
      GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+-------
             | 16039
        1249 |    85
       16410 |    64
        1259 |    39
        2659 |    22
[...]
```

Il y en en a un peu plus dans le cache (en fait, 2 fois 32 ko). Plus vous exécuterez la requête, et plus le nombre de blocs présents en cache augmentera. Sur le long terme, les 4425 blocs de la table `t2` peuvent se retrouver dans le cache.

> Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

Pour cela, il faut ouvrir le fichier de configuration `postgresql.conf` et modifier la valeur du paramètre `shared_buffers` à un quart de la mémoire. Par exemple :

```
shared_buffers = 2GB
```

> Redémarrer l'instance PostgreSQL.

```
# systemctl restart postgresql-15
```

> Se connecter à la base de données **b1** et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

```
b1=# \timing on
b1=# SELECT * FROM t2;

   id
---------
       1
[...]
Time: 340.444 ms

b1=# SELECT relfilenode, count(*) FROM pg_buffercache
     GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+--------
             | 257581
       16410 |   4425
        1249 |     29
[...]
```

PostgreSQL se retrouve avec toute la table directement dans son cache, et ce dès la première exécution.

PostgreSQL est optimisé principalement pour une utilisation multiutilisateur. Dans ce cadre, il faut pouvoir exécuter plusieurs requêtes en même temps. Une requête ne doit donc pas monopoliser tout le cache, juste une partie. Mais plus le cache est gros, plus la partie octroyée est grosse.

> Modifier le contenu de la table `t2`, par exemple avec :
>
> ```
> UPDATE t2 SET id = 0 WHERE id < 1000 ;
> ```
>
> Que contient le cache de PostgreSQL ?

```
b1=# UPDATE t2 SET id=0 WHERE id < 1000;
UPDATE 999

b1=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;
```

```
 relname | isdirty | blocs | taille
---------+---------+-------+--------
 t2      | f       |  4419 | 35 MB
 t2      | t       |    15 | 120 kB
```

15 blocs ont été modifiés (`isdirty` est à `true`), le reste n'a pas bougé.

> Exécuter un checkpoint. Que contient le cache de PostgreSQL ?

```
b1=# CHECKPOINT;
CHECKPOINT
```

```
b1=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;
```

```
 relname | isdirty | blocs | taille
---------+---------+-------+--------
 t2      | f       |  4434 | 35 MB
```

Les blocs *dirty* ont tous été écrits sur le disque et sont devenus « propres ».

10.11.4 Journaux

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

```
b1=# INSERT INTO t2 SELECT generate_series(1, 10000000);
INSERT 0 10000000
```

```
$ ls -al $PGDATA/pg_wal
total 131076
$ ls -al $PGDATA/pg_wal
total 638984
drwx------  3 postgres postgres     4096 Apr 16 17:55 .
drwx------ 20 postgres postgres     4096 Apr 16 17:48 ..
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000033
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000034
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000035
…
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000054
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000055
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000056
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000057
```

```
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000058
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000059
drwx------  2 postgres postgres        6 Apr 16 15:01 archive_status
```

Des journaux de transactions sont écrits lors des écritures dans la base. Leur nombre varie avec l'activité récente.

> Exécuter un checkpoint. Que se passe-t-il au niveau du répertoire `pg_wal` ?

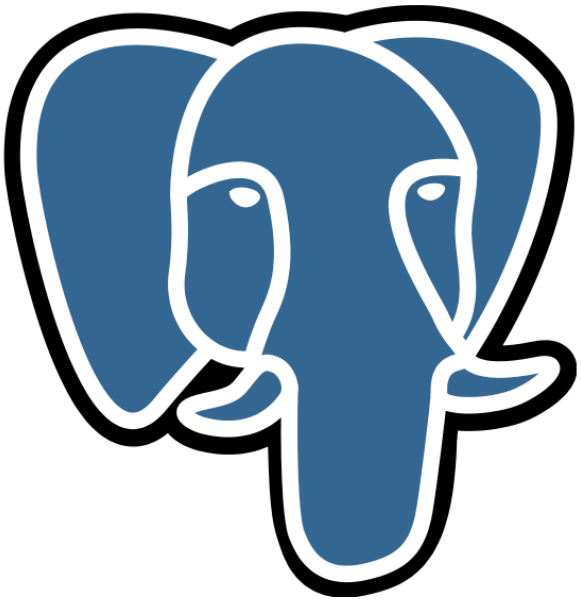
```
b1=# CHECKPOINT;
CHECKPOINT
```

```
$ ls -al $PGDATA/pg_wal
total 131076
total 638984
drwx------  3 postgres postgres     4096 Apr 16 17:56 .
drwx------ 20 postgres postgres     4096 Apr 16 17:48 ..
-rw-------  1 postgres postgres 16777216 Apr 16 17:56 000000010000000000000059
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000005A
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000005B
…
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000079
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007A
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007B
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007C
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007D
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007E
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007F
drwx------  2 postgres postgres        6 Apr 16 15:01 archive_status
```

Le nombre de journaux n'a pas forcément décru, mais le dernier journal d'avant le checkpoint est à présent le plus ancien (selon l'ordre des noms des journaux).

Ici, il n'y a ni PITR ni archivage. Les anciens journaux sont donc totalement inutiles et sont donc recyclés : renommés, il sont prêts à être remplis à nouveau. Noter que leur date de création n'a pas été mise à jour !

11/ Mécanique du moteur transactionnel & MVCC



11.1 INTRODUCTION

PostgreSQL utilise un modèle appelé **MVCC** (*Multi-Version Concurrency Control*).

- Gestion concurrente des transactions
- Excellente concurrence
- Impacts sur l'architecture

PostgreSQL s'appuie sur un modèle de gestion de transactions appelé MVCC. Nous allons expliquer cet acronyme, puis étudier en profondeur son implémentation dans le moteur.

Cette technologie a en effet un impact sur le fonctionnement et l'administration de PostgreSQL.

11.2 AU MENU

- Présentation de MVCC
- Niveaux d'isolation
- Implémentation de MVCC de PostgreSQL
- Les verrous
- Le mécanisme TOAST

11.3 PRÉSENTATION DE MVCC

11.3.1 Définitions

- *MultiVersion Concurrency Control*
- Contrôle de Concurrence Multi-Version
- Plusieurs versions du même enregistrement
- Granularité : l'enregistrement (pas le champ !)

MVCC est un sigle signifiant *MultiVersion Concurrency Control*, ou « contrôle de concurrence multiversion ».

Le principe est de faciliter l'accès concurrent de plusieurs utilisateurs (sessions) à la base en disposant en permanence de plusieurs versions différentes d'un même enregistrement. Chaque session peut travailler simultanément sur la version qui s'applique à son contexte (on parle d'« instantané » ou de *snapshot*).

Par exemple, une transaction modifiant un enregistrement va créer une nouvelle version de cet enregistrement. Mais celui-ci ne devra pas être visible des autres transactions tant que le travail de modification n'est pas validé en base. Les autres transactions *verront* donc une ancienne version de cet enregistrement. La dénomination technique est « lecture cohérente » (*consistent read* en anglais).

Précisons que la granularité des modifications est bien l'enregistrement (ou ligne) d'une table. Modifier un champ (colonne) revient à modifier la ligne. Deux transactions ne peuvent pas modifier deux champs différents d'un même enregistrement sans entrer en conflit, et les verrous portent toujours sur des lignes entières.

11.3.2 Alternative à MVCC : un seul enregistrement en base

- Verrouillage en lecture et exclusif en écriture
- Nombre de verrous ?
- Contention ?
- Cohérence ?
- Annulation ?

Avant d'expliquer en détail MVCC, voyons l'autre solution de gestion de la concurrence qui s'offre à nous, afin de comprendre le problème que MVCC essaye de résoudre.

Une table contient une liste d'enregistrements.

- Une transaction voulant consulter un enregistrement doit le verrouiller (pour s'assurer qu'il n'est pas modifié) de façon partagée, le consulter, puis le déverrouiller.
- Une transaction voulant modifier un enregistrement doit le verrouiller de façon exclusive (personne d'autre ne doit pouvoir le modifier ou le consulter), le modifier, puis le déverrouiller.

Cette solution a l'avantage de la simplicité : il suffit d'un gestionnaire de verrous pour gérer l'accès concurrent aux données. Elle a aussi l'avantage de la performance, dans le cas où les attentes de verrous sont peu nombreuses, la pénalité de verrouillage à payer étant peu coûteuse.

Elle a par contre des inconvénients :

- Les verrous sont en mémoire. Leur nombre est donc probablement limité. Que se passe-t-il si une transaction doit verrouiller 10 millions d'enregistrements ? Des mécanismes de promotion de verrou sont implémentés. Les verrous lignes deviennent des verrous bloc, puis des verrous table. **Le nombre de verrous est limité, et une promotion de verrou peut avoir des conséquences dramatiques** ;
- Un processus devant lire un enregistrement devra attendre la fin de la modification de celui-ci. Ceci entraîne rapidement de gros problèmes de contention. **Les écrivains bloquent les lecteurs, et les lecteurs bloquent les écrivains**. Évidemment, les écrivains se bloquent entre eux, mais cela est normal (il n'est pas possible que deux transactions modifient le même enregistrement simultanément, chacune sans conscience de ce qu'a effectué l'autre) ;
- Un ordre SQL (surtout s'il dure longtemps) n'a aucune garantie de voir des données cohérentes du début à la fin de son exécution : si, par exemple, durant un `SELECT` long, un écrivain modifie à la fois des données déjà lues par le `SELECT`, et des données qu'il va lire, le `SELECT` n'aura pas une vue cohérente de la table. Il pourrait y avoir un total faux sur une table comptable par exemple, le `SELECT` ayant vu seulement une partie des données validées par une nouvelle transaction ;
- Comment annuler une transaction ? Il faut un moyen de défaire ce qu'une transaction a effectué, au cas où elle ne se terminerait pas par une validation mais par une annulation.

11.3.3 Implémentation de MVCC par *undo*

- MVCC par *undo* :
  - une version de l'enregistrement dans la table
  - sauvegarde des anciennes versions
  - l'adresse physique d'un enregistrement ne change pas
  - la lecture cohérente est complexe
  - l'*undo* est complexe à dimensionner… et parfois insuffisant
  - l'annulation est lente
- Exemple : Oracle

C'est l'implémentation d'Oracle, par exemple. Un enregistrement, quand il doit être modifié, est recopié précédemment dans le tablespace d'UNDO. La nouvelle version de l'enregistrement est ensuite écrite par-dessus. Ceci implémente le MVCC (les anciennes versions de l'enregistrement sont toujours disponibles), et présente plusieurs avantages :

- Les enregistrements ne sont pas dupliqués dans la table. Celle-ci ne grandit donc pas suite à une mise à jour (si la nouvelle version n'est pas plus grande que la version précédente) ;
- Les enregistrements gardent la même adresse physique dans la table. Les index correspondant à des données non modifiées de l'enregistrement n'ont donc pas à être modifiés eux-mêmes, les index permettant justement de trouver l'adresse physique d'un enregistrement par rapport à une valeur.

Elle a aussi des défauts :

- La gestion de l'*undo* est très complexe : comment décider ce qui peut être purgé ? Il arrive que la purge soit trop agressive, et que des transactions n'aient plus accès aux vieux enregistrements (erreur `SNAPSHOT TOO OLD` sous Oracle, par exemple) ;
- La lecture cohérente est complexe à mettre en œuvre : il faut, pour tout enregistrement modifié, disposer des informations permettant de retrouver l'image avant modification de l'enregistrement (et la bonne image, il pourrait y en avoir plusieurs). Il faut ensuite pouvoir le reconstituer en mémoire ;
- Il est difficile de dimensionner correctement le fichier d'*undo*. Il arrive d'ailleurs qu'il soit trop petit, déclenchant l'annulation d'une grosse transaction. Il est aussi potentiellement une source de contention entre les sessions ;
- L'annulation (`ROLLBACK`) est très lente : il faut, pour toutes les modifications d'une transaction, défaire le travail, donc restaurer les images contenues dans l'*undo*, les réappliquer aux tables (ce qui génère de nouvelles écritures). Le temps d'annulation peut être supérieur au temps de traitement initial devant être annulé.

11.3.4 L'implémentation MVCC de PostgreSQL

- *Copy On Write* (duplication à l'écriture)
- Une version d'enregistrement n'est jamais modifiée
- Toute modification entraîne une nouvelle version
- Pas d'*undo* : pas de contention, `ROLLBACK` instantané

Dans une table PostgreSQL, un enregistrement peut être stocké dans plusieurs versions. Une modification d'un enregistrement entraîne l'écriture d'une nouvelle version de celui-ci. Une ancienne version ne peut être recyclée que lorsqu'aucune transaction ne peut plus en avoir besoin, c'est-à-dire qu'aucune transaction n'a un instantané de la base plus ancien que l'opération de modification de cet enregistrement, et que cette version est donc invisible pour tout le monde. Chaque version

d'enregistrement contient bien sûr des informations permettant de déterminer s'il est visible ou non dans un contexte donné.

Les avantages de cette implémentation stockant plusieurs versions dans la table principale sont multiples :

- La lecture cohérente est très simple à mettre en œuvre : à chaque session de lire la version qui l'intéresse. La visibilité d'une version d'enregistrement est simple à déterminer ;
- Il n'y a pas d'*undo*. C'est un aspect de moins à gérer dans l'administration de la base ;
- Il n'y a pas de contention possible sur l'*undo* ;
- Il n'y a pas de recopie dans l'*undo* avant la mise à jour d'un enregistrement. La mise à jour est donc moins coûteuse ;
- L'annulation d'une transaction est instantanée : les anciens enregistrements sont toujours disponibles.

Cette implémentation a quelques défauts :

- Il faut supprimer régulièrement les versions obsolètes des enregistrements ;
- Il y a davantage de maintenance d'index (mais moins de contentions sur leur mise à jour) ;
- Les enregistrements embarquent des informations de visibilité, qui les rendent plus volumineux.

11.4 NIVEAUX D'ISOLATION

11.4.1 Principe des niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point :
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Le niveau d'isolation au démarrage d'une transaction peut être spécifié :
  - `BEGIN ISOLATION LEVEL xxx`;

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé est un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction). Quatre niveaux sont définis, ils ne sont pas tous implémentés par PostgreSQL.

11.4.2 Niveau READ UNCOMMITTED

- Non disponible sous PostgreSQL
  - si demandé, s'exécute en `READ COMMITTED`
- Lecture de données modifiées par d'autres transactions **non** validées
- Aussi appelé *dirty reads*
- Dangereux
- Pas de blocage entre les sessions

Ce niveau d'isolation n'est disponible que pour les SGBD non-MVCC. Il est très dangereux : il est possible de lire des données invalides, ou temporaires, puisque tous les enregistrements de la table sont lus, quels que soient leurs états. Il est utilisé dans certains cas où les performances sont cruciales, au détriment de la justesse des données.

Sous PostgreSQL, ce mode n'est pas disponible. Une transaction qui demande le niveau d'isolation `READ UNCOMMITTED` s'exécute en fait en `READ COMMITTED`.

11.4.3 Niveau READ COMMITTED

- Niveau d'isolation par défaut
- La transaction ne lit que les données validées en base
- Un ordre SQL s'exécute dans un instantané (les tables semblent figées sur la durée de l'ordre)
- L'ordre suivant s'exécute dans un instantané différent

Ce mode est le mode par défaut, et est suffisant dans de nombreux contextes. PostgreSQL étant MVCC, les écrivains et les lecteurs ne se bloquent pas mutuellement, et chaque ordre s'exécute sur un instantané de la base (ce n'est pas un prérequis de `READ COMMITTED` dans la norme SQL). Il n'y a plus de lectures d'enregistrements non valides (*dirty reads*). Il est toutefois possible d'avoir deux problèmes majeurs d'isolation dans ce mode :

- Les lectures non-répétables (*non-repeatable reads*) : une transaction peut ne pas voir les mêmes enregistrements d'une requête sur l'autre, si d'autres transactions ont validé des modifications entre temps ;
- Les lectures fantômes (*phantom reads*) : des enregistrements peuvent ne plus satisfaire une clause `WHERE` entre deux requêtes d'une même transaction.

11.4.4 Niveau REPEATABLE READ

- Instantané au début de la transaction
- Ne voit donc plus les modifications des autres transactions
- Voit toujours ses propres modifications
- Peut entrer en conflit avec d'autres transactions si modification des mêmes enregistrements

Ce mode, comme son nom l'indique, permet de ne plus avoir de lectures non-répétables. Deux ordres SQL consécutifs dans la même transaction retourneront les mêmes enregistrements, dans la même version. Ceci est possible car la transaction voit une image de la base figée. L'image est figée non au démarrage de la transaction, mais à la première commande non TCL (*Transaction Control Language*) de la transaction, donc généralement au premier `SELECT` ou à la première modification.

Cette image sera utilisée pendant toute la durée de la transaction. En lecture seule, ces transactions ne peuvent pas échouer. Elles sont entre autres utilisées pour réaliser des exports des données : c'est ce que fait `pg_dump`.

Dans le standard, ce niveau d'isolation souffre toujours des lectures fantômes, c'est-à-dire de lecture d'enregistrements différents pour une même clause `WHERE` entre deux exécutions de requêtes. Cependant, PostgreSQL est plus strict et ne permet pas ces lectures fantômes en `REPEATABLE READ`. Autrement dit, un même `SELECT` renverra toujours le même résultat.

En écriture, par contre (ou `SELECT FOR UPDATE`, `FOR SHARE`), si une autre transaction a modifié les enregistrements ciblés entre temps, une transaction en `REPEATABLE READ` va échouer avec l'erreur suivante :

```
ERROR: could not serialize access due to concurrent update
```

Il faut donc que l'application soit capable de la rejouer au besoin.

11.4.5 Niveau SERIALIZABLE

- Niveau d'isolation le plus élevé
- Chaque transaction se croit seule sur la base
  - sinon annulation d'une transaction en cours
- Avantages :
  - pas de « lectures fantômes »
  - évite des verrous, simplifie le développement
- Inconvénients :
  - pouvoir rejouer les transactions annulées
  - toutes les transactions impliquées doivent être sérialisables

PostgreSQL fournit un mode d'isolation appelé `SERIALIZABLE` :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
…
COMMIT / ROLLBACK ;
```

Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base, et comme si elles se déroulaient les unes à la suite des autres. Dès que cette garantie ne peut plus être apportée, PostgreSQL annule celle qui entraînera le moins de perte de données.

Le niveau `SERIALIZABLE` est utile quand le résultat d'une transaction peut être influencé par une transaction tournant en parallèle, par exemple quand des valeurs de lignes dépendent de valeurs d'autres lignes : mouvements de stocks, mouvements financiers… avec calculs de stocks. Autrement dit, si une transaction lit des lignes, elle a la garantie que leurs valeurs ne seront pas modifiées jusqu'à son `COMMIT`, y compris par les transactions qu'elle ne voit pas — ou bien elle tombera en erreur.

Au niveau `SERIALIZABLE` (comme en `REPEATABLE READ`), il est donc essentiel de pouvoir rejouer une transaction en cas d'échec. Par contre, nous simplifions énormément tous les autres points du développement. Il n'y a plus besoin de `SELECT FOR UPDATE`, solution courante mais très gênante pour les transactions concurrentes. Les triggers peuvent être utilisés sans soucis pour valider des opérations.

Ce mode doit être mis en place globalement, car toute transaction non sérialisable peut en théorie s'exécuter n'importe quand, ce qui rend inopérant le mode sérialisable sur les autres.

La sérialisation utilise le « verrouillage de prédicats ». Ces verrous sont visibles dans la vue `pg_locks` sous le nom `SIReadLock`, et ne gênent pas les opérations habituelles, du moins tant que la sérialisation est respectée. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation.

Le wiki PostgreSQL[1], et la documentation officielle[2] donnent des exemples, et ajoutent quelques conseils pour l'utilisation de transactions sérialisables. Afin de tenter de réduire les verrous et le nombre d'échecs :

- faire des transactions les plus courtes possibles (si possible uniquement ce qui a trait à l'intégrité) ;
- limiter le nombre de connexions actives ;
- utiliser les transactions en mode `READ ONLY` dès que possible, voire en `SERIALIZABLE READ ONLY DEFERRABLE` (au risque d'un délai au démarrage) ;
- augmenter certains paramètres liés aux verrous, c'est-à-dire augmenter la mémoire dédiée ; car si elle manque, des verrous de niveau ligne pourraient être regroupés en verrous plus larges et plus gênants ;
- éviter les parcours de tables (*Seq Scan*), et donc privilégier les accès par index.

[1] https://wiki.postgresql.org/wiki/SSI/fr
[2] https://docs.postgresql.fr/current/transaction-iso.html#XACT-SERIALIZABLE

11.5 BLOCS & LIGNES

11.5.1 Structure d'un bloc

- 1 bloc = 8 ko
- `ctid` = (bloc, item dans le bloc)

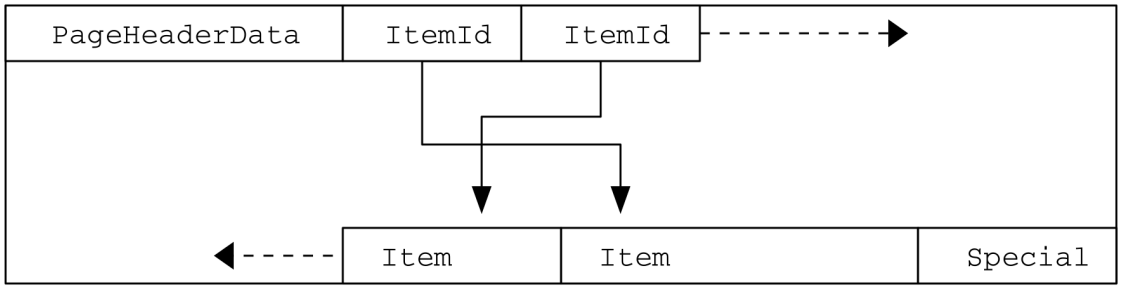


Figure 11/ .1: Répartition des lignes au sein d'un bloc (schéma de la documentation officielle, licence PostgreSQL)

Le bloc (ou page) est l'unité de base de transfert pour les I/O, le cache mémoire… Il fait généralement 8 ko (ce qui ne peut être modifié qu'en recompilant). Les lignes y sont stockées avec des informations d'administration telles que décrites dans le schéma ci-dessus. Une ligne ne fait jamais partie que d'un seul bloc (si cela ne suffit pas, un mécanisme que nous verrons plus tard, nommé TOAST, se déclenche).

Nous distinguons dans ce bloc :

- un entête de page avec diverses informations, notamment la somme de contrôle (si activée) ;
- des identificateurs de 4 octets, pointant vers les emplacements des lignes au sein du bloc ;
- les lignes, stockées à rebours depuis la fin du bloc ;
- un espace spécial, vide pour les tables ordinaires, mais utilisé par les blocs d'index.

Le `ctid` identifie une ligne, en combinant le numéro du bloc (à partir de 0) et l'identificateur dans le bloc (à partir de 1). Comme la plupart des champs administratifs liés à une ligne, il suffit de l'inclure dans un `SELECT` pour l'afficher. L'exemple suivant affiche les premiers et derniers éléments des deux blocs d'une table et vérifie qu'il n'y a pas de troisième bloc :

```
# CREATE TABLE deuxblocs AS SELECT i, i  AS j FROM generate_series(1, 452) i;
SELECT 452

# SELECT ctid, i,j FROM deuxblocs
WHERE ctid in ( '(1, 1)', '(0, 226)', '(1, 1)', '(1, 226)', '(1, 227)', '(2, 0)' );
```

```
  ctid    |  i  |  j
---------+-----+-----
 (0,1)    |   1 |   1
 (0,226)  | 226 | 226
 (1,1)    | 227 | 227
 (1,226)  | 452 | 452
```

Un `ctid` ne doit jamais servir à désigner une ligne de manière pérenne et ne doit pas être utilisé dans des requêtes ! Il peut changer n'importe quand, notamment en cas d'`UPDATE` ou de `VACUUM FULL` !

La documentation officielle[3] contient évidemment tous les détails.

Pour observer le contenu d'un bloc, à titre pédagogique, vous pouvez utiliser l'extension pageinspect[4].

11.5.2 xmin & xmax

Table initiale :

xmin	xmax	Nom	Solde
100		M. Durand	1500
100		Mme Martin	2200

PostgreSQL stocke des informations de visibilité dans chaque version d'enregistrement.

- `xmin` : l'identifiant de la transaction créant cette version.
- `xmax` : l'identifiant de la transaction invalidant cette version.

Ici, les deux enregistrements ont été créés par la transaction 100. Il s'agit peut-être, par exemple, de la transaction ayant importé tous les soldes à l'initialisation de la base.

[3] https://docs.postgresql.fr/current/storage-page-layout.html
[4] https://docs.postgresql.fr/current/pageinspect.html#PAGEINSPECT-HEAP-FUNCS

11.5.3 xmin & xmax (suite)

```
BEGIN;
UPDATE soldes SET solde = solde - 200 WHERE nom = 'M. Durand';
```

xmin	xmax	Nom	Solde
100	**150**	M. Durand	1500
100		Mme Martin	2200
150		M. Durand	1300

Nous décidons d'enregistrer un virement de 200 € du compte de M. Durand vers celui de Mme Martin. Ceci doit être effectué dans une seule transaction : l'opération doit être atomique, sans quoi de l'argent pourrait apparaître ou disparaître de la table.

Nous allons donc tout d'abord démarrer une transaction (ordre `SQL BEGIN`). PostgreSQL fournit donc à notre session un nouveau numéro de transaction (150 dans notre exemple). Puis nous effectuerons :

```
UPDATE soldes SET solde = solde - 200 WHERE nom = 'M. Durand';
```

11.5.4 xmin & xmax (suite)

```
UPDATE soldes SET solde = solde + 200 WHERE nom = 'Mme Martin';
```

xmin	xmax	Nom	Solde
100	150	M. Durand	1500
100	**150**	Mme Martin	2200
150		M. Durand	1300
150		Mme Martin	2400

Puis nous effectuerons :

```
UPDATE soldes SET solde = solde + 200 WHERE nom = 'Mme Martin';
```

Nous avons maintenant deux versions de chaque enregistrement. Notre session ne voit bien sûr plus que les nouvelles versions de ces enregistrements, sauf si elle décidait d'annuler la transaction, auquel cas elle reverrait les anciennes données.

Pour une autre session, la version visible de ces enregistrements dépend de plusieurs critères :

- La transaction 150 a-t-elle été validée ? Sinon elle est invisible ;
- La transaction 150 a-t-elle été validée après le démarrage de la transaction en cours, et sommes-nous dans un niveau d'isolation (*repeatable read* ou *serializable*) qui nous interdit de voir les modifications faites depuis le début de notre transaction ? ;
- La transaction 150 a-t-elle été validée après le démarrage de la requête en cours ? Une requête, sous PostgreSQL, voit un instantané cohérent de la base, ce qui implique que toute transaction validée après le démarrage de la requête doit être ignorée.

Dans le cas le plus simple, 150 ayant été validée, une transaction 160 ne verra pas les premières versions : `xmax` valant 150, ces enregistrements ne sont pas visibles. Elle verra les secondes versions, puisque `xmin` = 150, et pas de `xmax`.

11.5.5 xmin & xmax (suite)

xmin	xmax	Nom	Solde
100	150	M. Durand	1500
100	**150**	Mme Martin	2200
150		M. Durand	1300
150		Mme Martin	2400

- Comment est effectuée la suppression d'un enregistrement ?
- Comment est effectuée l'annulation de la transaction 150 ?

- La suppression d'un enregistrement s'effectue simplement par l'écriture d'un `xmax` dans la version courante de la ligne ;
- Le `COMMIT` ou le `ROLLBACK` consiste à écrire le statut de la transaction dans le CLOG (*Commit Log*), une zone mémoire dédiée au statut des transactions.

11.5.6 CLOG

- Le CLOG (*Commit Log*) enregistre l'état des transactions.
- Chaque transaction occupe 2 bits de CLOG (4 statuts)
- `COMMIT` ou `ROLLBACK` très rapide
- Référence pour savoir si une ligne est visible ou pas
- Puis reporté dans les lignes (*hint bits*)

Le CLOG permet à PostgreSQL de savoir si les numéros de transaction portés par une ligne sont validés ou non. Le CLOG est stocké dans une série de fichiers de 256 ko, stockés dans le répertoire `pg_xact/` de PGDATA (répertoire racine de l'instance PostgreSQL). Chaque transaction est créée dans ce fichier dès son démarrage et est encodée sur deux bits puisqu'une transaction peut avoir quatre états :

- `TRANSACTION_STATUS_IN_PROGRESS` signifie que la transaction est en cours, c'est l'état initial ;
- `TRANSACTION_STATUS_COMMITTED` signifie que la la transaction a été validée ;
- `TRANSACTION_STATUS_ABORTED` signifie que la transaction a été annulée ;
- `TRANSACTION_STATUS_SUB_COMMITTED` signifie que la transaction comporte des sous-transactions, afin de valider l'ensemble des sous-transactions de façon atomique.

Nous avons donc un million d'états de transactions par fichier de 256 ko.

Ainsi, annuler une transaction (`ROLLBACK`) est quasiment instantané sous PostgreSQL : il suffit d'écrire `TRANSACTION_STATUS_ABORTED` dans l'entrée de CLOG correspondant à la transaction. Les lignes insérées par cette transaction seront simplement ignorées, de même que des versions de lignes créées par un `UPDATE`. Les versions de lignes marquées comme périmées par ce même `UPDATE` dans la transaction resteront visibles.

Toute modification dans le CLOG, comme toute modification d'un fichier de données (table, index, séquence, vue matérialisée), est bien sûr enregistrée tout d'abord dans les journaux de transactions (dans le répertoire `pg_wal/`).

Pour éviter de se référer au CLOG trop souvent, chaque ligne porte des *hint bits* contenant, entre autres, le statut des numéros de transaction portés par la ligne. Mais ce n'est pas la requête qui a modifié les lignes qui les renseigne. La première requête qui lit une ligne après une modification, même un `SELECT`, lit le CLOG et met à jour ses *hint bits*, ce qui occasionne de nouvelles écritures. (Et génèrent même de nouveaux journaux si `wal_log_hints` est `on` ou si les sommes de contrôle sont activées, ce qui est recommandé.)

11.6 AVANTAGES & INCONVÉNIENTS DU MVCC DE POSTGRESQL

11.6.1 Avantages du MVCC de PostgreSQL

- Avantages classiques de MVCC (concurrence d'accès)
- Implémentation simple et performante
- Peu de sources de contention
- Verrouillage simple d'enregistrement
- `ROLLBACK` instantané
- Données conservées aussi longtemps que nécessaire

Reprenons les avantages du MVCC tel qu'implémenté par PostgreSQL :

- Les lecteurs ne bloquent pas les écrivains, ni les écrivains les lecteurs ;
- Le code gérant les instantanés est simple, ce qui est excellent pour la fiabilité, la maintenabilité et les performances ;
- Les différentes sessions ne se gênent pas pour l'accès à une ressource commune (l'*undo*) ;
- Un enregistrement est facilement identifiable comme étant verrouillé en écriture : il suffit qu'il ait une version ayant un `xmax` correspondant à une transaction en cours ;
- L'annulation est instantanée : il suffit d'écrire le nouvel état de la transaction annulée dans la CLOG. Pas besoin de restaurer les valeurs précédentes, elles sont toujours là ;
- Les anciennes versions restent en ligne aussi longtemps que nécessaire. Elles ne pourront être effacées de la base qu'une fois qu'aucune transaction ne les considérera comme visibles.

(Précisons toutefois que ceci est une vision un peu simplifiée pour les cas courants. La signification du `xmax` est parfois altérée par des bits positionnés dans des champs systèmes inaccessibles par l'utilisateur. Cela arrive, par exemple, quand des transactions insèrent des lignes portant une clé étrangère, pour verrouiller la ligne pointée par cette clé, laquelle ne doit pas disparaître pendant la durée de cette transaction.)

11.6.2 Inconvénients du MVCC de PostgreSQL

- Nettoyage des enregistrements
  - `VACUUM`
  - automatisation : autovacuum
- Tables plus volumineuses
- Écritures amplifiées
- Pas de visibilité des lignes dans les index
- Les colonnes supprimées impliquent reconstruction

Comme toute solution complexe, l'implémentation MVCC de PostgreSQL est un compromis. Les avantages cités précédemment sont obtenus au prix de concessions. Bien sûr, divers mécanismes tentent de mitiger ces soucis.

11.6.2.1 VACUUM

Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande `VACUUM`. Il a un avantage sur la technique de l'*undo* : le nettoyage n'est pas effectué par un client faisant des mises à jour (et créant donc des enregistrements morts), et le ressenti est donc meilleur.

`VACUUM` peut se lancer à la main, mais dans le cas général on s'en remet à l'autovacuum, un démon qui lance les `VACUUM` (et bien plus) en arrière-plan quand il le juge nécessaire. Tout cela sera traité en détail par la suite.

11.6.2.2 Bloat

Les tables sont forcément plus volumineuses que dans l'implémentation par *undo*, pour deux raisons :

- les informations de visibilité y sont stockées, il y a donc un surcoût d'une douzaine d'octets par enregistrement ;
- il y a toujours des enregistrements morts dans une table, une sorte de *fond de roulement*, qui se stabilise quand l'application est en régime stationnaire.

Ces enregistrements sont recyclés à chaque passage de `VACUUM`.

11.6.2.3 Amplification des écritures

Toute écriture a tendance à en déclencher d'autres. D'abord, toute modification est écrite dans les journaux de transaction, comme le fait tout moteur de base de données relationnelle fiable, et éventuellement archivée et/ou répliquée.

Des suppressions, insertions ou mises à jour, même suivies d'un `ROLLBACK`, écrivent toujours dans les fichiers de données eux-mêmes. (Ce sont les numéros de transaction et le CLOG qui définiront la visibilité des lignes.) Le `VACUUM` nettoie les lignes, et cela genère aussi des écritures et des journaux. Nous verrons également le coût du problème de *wraparound*. La mise à jour des *hints bits*, même lors d'un `SELECT`, génère de nouvelles écritures voire des journaux.

Comme les index pointent vers un bloc et une ligne, les index sont également souvent à modifier suite à une écriture (et cela est aussi journalisé).

11.6.2.4 Visibilité

Les index n'ont pas d'information de visibilité. Il est donc nécessaire d'aller vérifier dans la table associée que l'enregistrement trouvé dans l'index est bien visible. Cela a un impact sur le temps d'exécution de requêtes comme `SELECT count(*)` sur une table : dans le cas le plus défavorable, il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles. La *visibility map* permet de limiter cette vérification aux données les plus récentes. Il existe également des mécanismes de nettoyage des index quand une requête rencontre des *tuples* périmés.

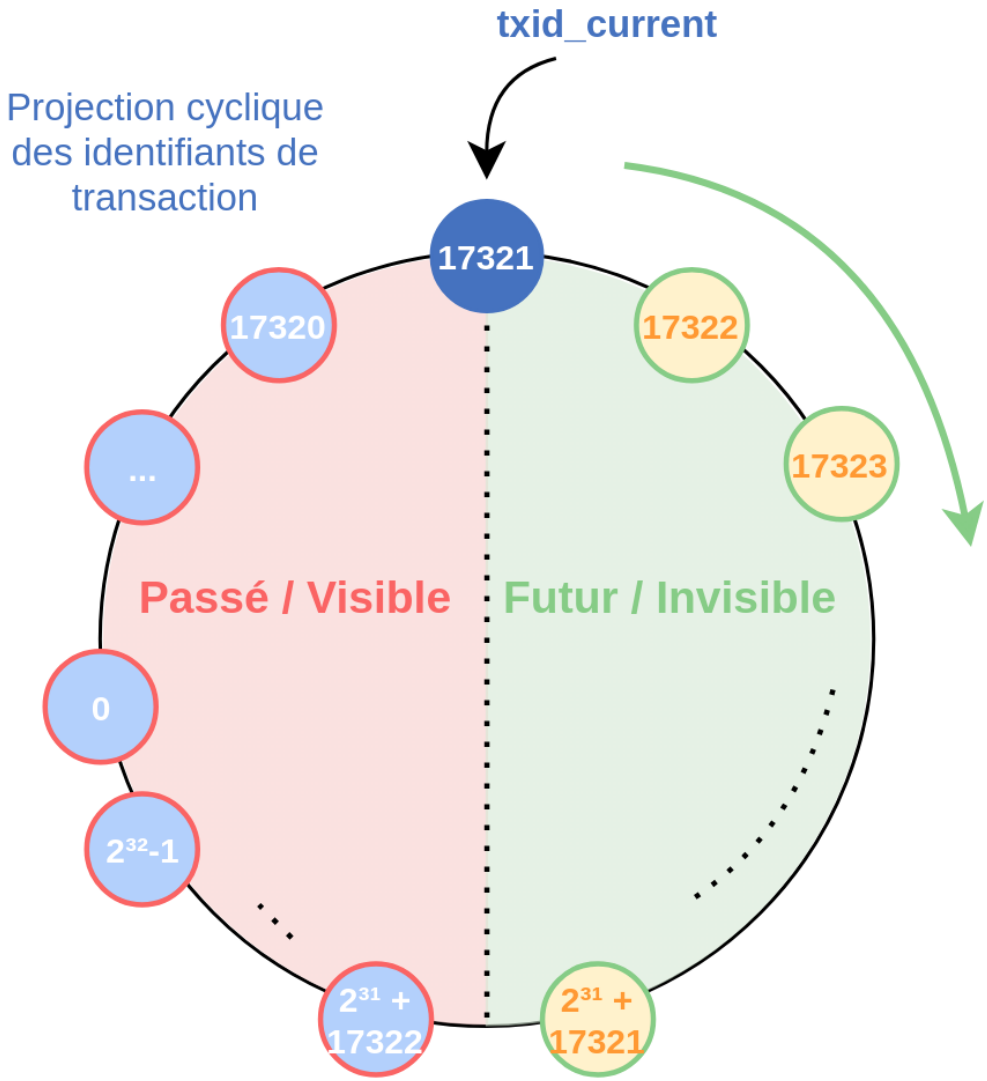
11.6.2.5 Colonnes supprimées

Un `VACUUM` ne s'occupe pas de l'espace libéré par des colonnes supprimées (fragmentation verticale). Un `VACUUM FULL` est nécessaire pour reconstruire la table.

11.7 LE WRAPAROUND

11.7.1 Le wraparound (1)

- *Wraparound* : bouclage du compteur de `xmin` / `xmax`
- 32 bits ~ 4 milliards



Le numéro de transaction stocké sur chaque ligne est sur 32 bits, même si PostgreSQL utilise en interne 64 bits. Il y aura donc dépassement de ce compteur au bout de 4 milliards de transactions. Sur les

machines actuelles, avec une activité soutenue, cela peut être atteint relativement rapidement.

En fait, ce compteur est cyclique. Cela peut se voir ainsi :

```
SELECT pg_current_xact_id(), *
FROM pg_control_checkpoint()\gx
```

```
-[ RECORD 1 ]--------+-------------------------
pg_current_xact_id   | 10549379902
checkpoint_lsn       | 62/B902F128
redo_lsn             | 62/B902F0F0
redo_wal_file        | 000000010000006200000B9
timeline_id          | 1
prev_timeline_id     | 1
full_page_writes     | t
next_xid             | 2:1959445310
next_oid             | 24614
next_multixact_id    | 1
next_multi_offset    | 0
oldest_xid           | 1809610092
oldest_xid_dbid      | 16384
oldest_active_xid    | 1959445310
oldest_multi_xid     | 1
oldest_multi_dbid    | 13431
oldest_commit_ts_xid | 0
newest_commit_ts_xid | 0
checkpoint_time      | 2023-12-29 16:39:25+01
```

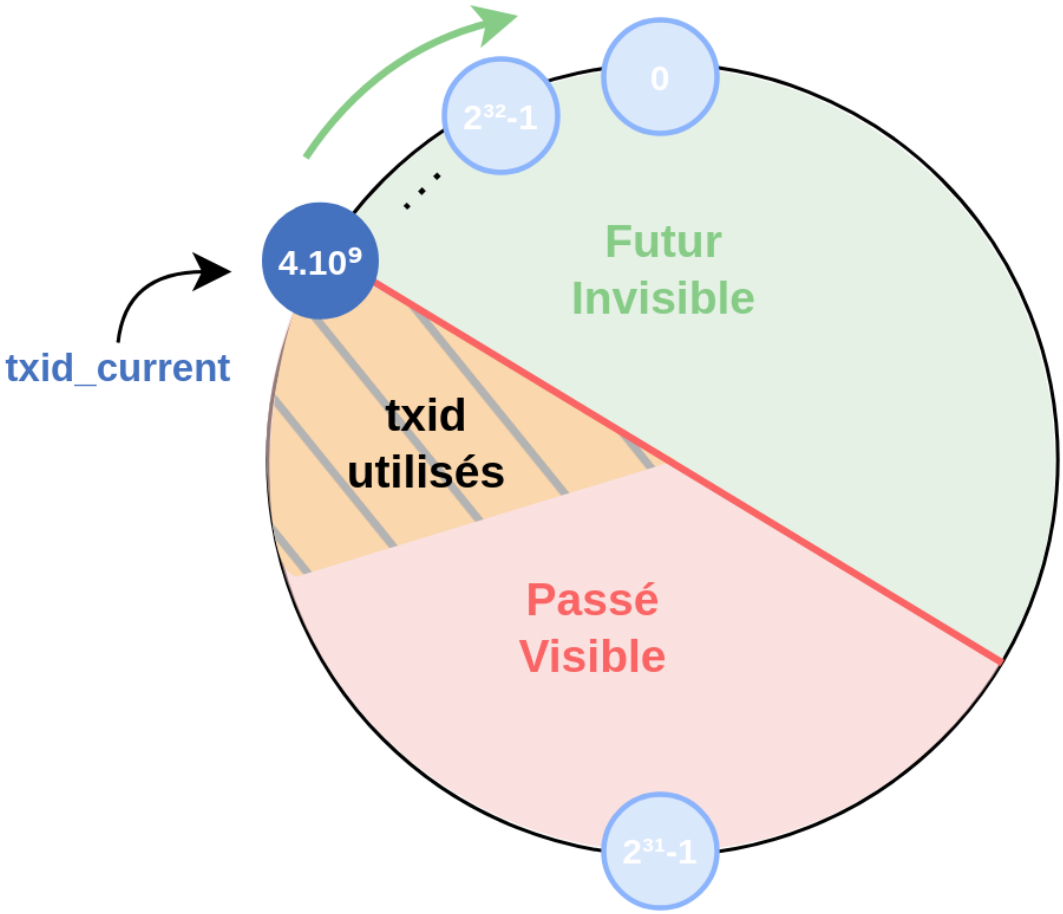
`pg_current_xact_id()` renvoie le numéro de transaction en cours, soit 10 549 379 902 (sur 64 bits). Le premier chiffre de la ligne `next_xid` indique qu'il y a déjà eu deux « époques », et le numéro de transaction sur 32 bits dans ce troisième cycle est 1 959 445 310. On vérifie que 10 549 379 902 = 1 959 445 310 + 2×2^{32}.

Le nombre d'époques n'est enregistré ni dans les tables ni dans les lignes. Pour savoir quelles transactions sont visibles depuis la transaction en cours (ou exactement depuis le « *snapshot* » de l'état de la base associé), PostgreSQL considère que les 2 milliards de numéros de transactions supérieurs à celle en cours correspondent à des transactions futures, et les 2 milliards de numéros inférieurs à des transactions dans le passé.

Si on se contentait de boucler, une fois dépassé le numéro de transaction 2^{31}-1 (environ 2 milliards), de nombreux enregistrements deviendraient invisibles, car validés par des transactions à présent situées dans le futur. Il y a donc besoin d'un mécanisme supplémentaire.

11.7.2 Le wraparound (2)

Après 4 milliards de transactions :



PostgreSQL se débrouille afin de ne jamais laisser dans les tables des numéros de transactions visibles situées dans ce qui serait le futur de la transaction actuelle.

En temps normal, les numéros de transaction visibles se situent tous dans des valeurs un peu inférieures à la transaction en cours. Les lignes plus vieilles sont « gelées », c'est-à-dire marquées comme assez anciennes pour qu'on ignore le numéro de transaction. (Cela implique bien sûr de réécrire la ligne sur le disque.) Il n'y a alors plus de risque de présence de numéros de transaction qui serait abusivement dans le futur de la transaction actuelle. Les numéros de transaction peuvent être réutilisés

sans risque de mélange avec ceux issus du cycle précédent.

Ce cycle des numéros de transactions peut se poursuivre indéfiniment.

Ces recyclages sont visibles dans `pg_control_checkpoint()`, comme vu plus haut. La même fonction renvoie `oldest_xid`, qui est le numéro de la transaction (32 bits) la plus ancienne dans l'instance, qui est par définition celui de la base la plus vieille, ou plutôt de la ligne la plus vieille dans la base :

```
SELECT datname, datfrozenxid, age (datfrozenxid)
FROM pg_database ORDER BY 3 DESC ;
```

```
  datname   | datfrozenxid |    age
-----------+--------------+-----------
 pgbench    |   1809610092 | 149835222
 template0  |   1957896953 |   1548361
 template1  |   1959012415 |    432899
 postgres   |   1959445305 |         9
```

La ligne la plus ancienne dans la base **pgbench** a moins de 150 millions de transactions d'âge, il n'y a pas de confusion possible.

Un peu plus de 50 millions de transactions plus tard, la même requête renvoie :

```
  datname   | datfrozenxid |   age
-----------+--------------+----------
 template0  |   1957896953 | 52338522
 template1  |   1959012415 | 51223060
 postgres   |   1959445305 | 50790170
 pgbench    |   1959625471 | 50610004
```

Les lignes les plus anciennes de `pgbench` semblent avoir disparu.

11.7.3 Le wraparound (3)

Concrètement ?

- `VACUUM FREEZE`
  - géré par l'autovacuum
  - au pire, d'office
  - potentiellement beaucoup d'écritures

Concrètement, l'opération de « gel » des lignes qui possèdent des identifiants de transaction suffisamment anciens est appelée `VACUUM FREEZE`.

Ce dernier peut être déclenché manuellement, mais il fait aussi partie des tâches de maintenance habituellement gérées par le démon `autovacuum`, en bonne partie en même temps que les `VACUUM`

habituels. Un `VACUUM FREEZE` n'est pas bloquant, mais les verrous sont parfois plus gênants que lors d'un `VACUUM` simple. Les écritures générées sont très variables, et parfois gênantes.

Si cela ne suffit pas, le moteur déclenche automatiquement un `VACUUM FREEZE` sur une table avec des lignes trop âgées, et ce, même si autovacuum est désactivé.

Ces opérations suffisent dans l'immense majorité des cas. Dans le cas contraire, l'`autovacuum` devient de plus en plus agressif. Si le stock de transactions disponibles descend en-dessous de 40 millions (10 millions avant la version 14), des messages d'avertissements apparaissent dans les traces.

Dans le pire des cas, après bien des messages d'avertissements, le moteur refuse toute nouvelle transaction dès que le stock de transactions disponibles se réduit à 3 millions (1 million avant la version 14 ; valeurs codées en dur). Il faudra alors lancer un `VACUUM FREEZE` manuellement. Ceci ne peut arriver qu'exceptionnellement (par exemple si une transaction préparée a été oubliée depuis 2 milliards de transactions et qu'aucune supervision ne l'a détectée).

`VACUUM FREEZE` sera développé dans le module VACUUM et autovacuum[5]. La documentation officielle[6] contient aussi un paragraphe sur ce sujet.

[5] https://dali.bo/m5_html
[6] https://docs.postgresql.fr/current/maintenance.html

11.8 OPTIMISATIONS DE MVCC

- HOT
- Free Space Map
- Visibility Map

Les fonctionnalités que nous allons décrire visent à mitiger les inconvénients du MVCC de PostgreSQL.

11.8.1 Mise à jour jour HOT

HOT = *Heap-Only Tuples*

- Si place dans le bloc
- Si aucune colonne indexée modifiée
- Alors la mise à jour se fait dans le même bloc
  - gain en mise à jour des index
  - gain en écritures et en utilisation du cache
- Favorisée par un *fillfator* < 100

Le mécanisme des *Heap-Only Tuples* (HOT) permet de stocker, sous condition, plusieurs versions du même enregistrement dans le même bloc, avec une forme de chaînage entre plusieurs versions d'une ligne dans le bloc. Ceci permet au fur et à mesure des mises à jour de supprimer automatiquement les anciennes versions, sans attendre `VACUUM`. Cela permet aussi de ne pas toucher aux index, qui pointent donc grâce à cela sur plusieurs versions du même enregistrement. Il y a donc un gain en écriture comme en utilisation du cache. Les conditions sont les suivantes :

- Aucune colonne indexée n'a été modifiée par l'opération : l'index n'a pas besoin d'être modifié et peut continuer à pointer vers le même bloc ;
- Le bloc contient assez de place pour la nouvelle version (les enregistrements ne sont pas chaînés entre plusieurs blocs).

Afin que cette dernière condition ait plus de chance d'être vérifiée, il peut être utile de baisser la valeur du paramètre `fillfactor` pour une table donnée (cf documentation officielle[7]). Par défaut, le `fillfactor` vaut 100% pour économiser la place. Sacrifier un peu d'espace en le baissant peut être rentable car le mécanisme HOT peut réduire la fragmentation.

[7] https://docs.postgresql.fr/current/sql-createtable.html#SQL-CREATETABLE-STORAGE-PARAMETERS

Le nombre de mises à jour HOT sur une table peut être suivi en comparant les champs `n_tup_hot_upd` (décompte des mises à jour HOT) et `n_tup_upd` (toutes les mises à jour) dans la vue pg_stat_user_tables[8].

Documentation officielle : Heap-Only Tuples (HOT)[9]

11.8.2 Free Space Map

- Fichier pointant les espaces libres des blocs
- Optimisation des insertions

Chaque table possède une *Free Space Map* avec une liste des espaces libres de chaque bloc, organisée en arbre. PostgreSQL l'utilise pour savoir où insérer de nouvelles lignes ou versions de lignes. Elle est stockée dans les fichiers `*_fsm` associés à chaque table.

Documentation officielle : Carte des espaces libres[10]

11.8.3 Visibility Map

- Quels blocs sont intégralement visibles ?
- Mise à jour par `VACUUM`
- Utilisation :
  - Accélérer `VACUUM` et `VACUUM FREEZE`
  - *Index Only Scan*

La *Visibility Map* permet de savoir si tous les enregistrements d'un bloc sont visibles de toutes les sessions en cours. C'est le cas si toutes les transactions de modifications sont terminées, et si les lignes mortes ont été nettoyées. Il n'y a donc plus besoin de consulter les numéros de transaction et la CLOG pour toutes les lignes du bloc. En cas de doute, ou d'enregistrement non visible, le bloc n'est pas marqué comme totalement visible.

Cela accélère grandement le `VACUUM` : sa phase 1 ne parcourt pas toute la table, mais uniquement les enregistrements pour lesquels la *Visibility Map* est à *faux*, car des données sont potentiellement

[8] https://docs.postgresql.fr/current/monitoring-stats.html#MONITORING-PG-STAT-ALL-TABLES-VIEW
[9] https://docs.postgresql.fr/current/storage-hot.html
[10] https://docs.postgresql.fr/current/storage-fsm.html

obsolètes dans le bloc. Les blocs intégralement visibles peuvent être ignorés. Après nettoyage d'un bloc, `VACUUM` positionne sa réference dans la *Visibility Map* à *vrai*, si toutes les lignes sont bien visibles pour toutes les sessions.

À l'inverse, un *Index Only Scan* (parcours d'index seuls) utilise cette *Visibility Map* pour savoir si les données de l'index suffisent ou s'il faut aller vérifier la visibilité de la ligne dans la table. Dans ce dernier cas, le plan indique des *Heap Fetches*, qui dégradent les performances. Un `VACUUM` fréquent améliore donc les performances si des *Index Only Scans* sont utilisés. (Les *Index Scans* ou *Bitmap Scans* ne sont pas concernés.)

La même *Visibility Map* sert aussi à noter les bloc entièrement « gelés » pour accélérer les `VACUUM FREEZE`, là encore pour ne pas avoir à les relire.

Documentation officielle : Carte de visibilité[11]

[11] https://docs.postgresql.fr/current/storage-vm.html#STORAGE-VM

11.9 VERROUS

11.9.1 Verrouillage et MVCC

La gestion des verrous est liée à l'implémentation de MVCC

- Verrouillage d'objets en mémoire
- Verrouillage d'objets sur disque
- Paramètres

11.9.2 Le gestionnaire de verrous

PostgreSQL possède un gestionnaire de verrous

- Verrous d'objet
- Niveaux de verrouillage
- Empilement des verrous
- *Deadlock*
- Vue `pg_locks`

Le gestionnaire de verrous de PostgreSQL est capable de gérer des verrous sur des tables, sur des enregistrements, sur des ressources virtuelles. De nombreux types de verrous sont disponibles, chacun entrant en conflit avec d'autres.

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler. Si le verrou ne peut être obtenu immédiatement, par défaut PostgreSQL attendra indéfiniment qu'il soit libéré.

Ce verrou en attente peut lui-même imposer une attente à d'autres sessions qui s'intéresseront au même objet. Si ce verrou en attente est bloquant (cas extrême : un `VACUUM FULL` sans `SKIP_LOCKED` lui-même bloqué par une session qui tarde à faire un `COMMIT`), il est possible d'assister à un phénomène d'empilement de verrous en attente.

Les noms des verrous peuvent prêter à confusion : `ROW SHARE` par exemple est un verrou de table, pas un verrou d'enregistrement. Il signifie qu'on a pris un verrou sur une table pour y faire des `SELECT FOR UPDATE` par exemple. Ce verrou est en conflit avec les verrous pris pour un `DROP TABLE`, ou pour un `LOCK TABLE`.

Le gestionnaire de verrous détecte tout verrou mortel (*deadlock*) entre deux sessions. Un *deadlock* est

la suite de prise de verrous entraînant le blocage mutuel d'au moins deux sessions, chacune étant en attente d'un des verrous acquis par l'autre.

Il est possible d'accéder aux verrous actuellement utilisés sur une instance par la vue `pg_locks`.

11.9.3 Verrous sur enregistrement

- Le gestionnaire de verrous possèdes des verrous sur enregistrements
  - transitoires
  - le temps de poser le `xmax`
- Utilisation de verrous sur disque
  - pas de risque de pénurie
- Les verrous entre transaction se font sur leurs ID

Le gestionnaire de verrous fournit des verrous sur enregistrement. Ceux-ci sont utilisés pour verrouiller un enregistrement le temps d'y écrire un `xmax`, puis libérés immédiatement.

Le verrouillage réel est implémenté comme suit :

- D'abord, chaque transaction verrouille son objet « identifiant de transaction » de façon exclusive.
- Une transaction voulant mettre à jour un enregistrement consulte le `xmax`. Si ce `xmax` est celui d'une transaction en cours, elle demande un verrou exclusif sur l'objet « identifiant de transaction » de cette transaction, qui ne lui est naturellement pas accordé. La transaction est donc placée en attente.
- Enfin, quand l'autre transaction possédant le verrou se termine (`COMMIT` ou `ROLLBACK`), son verrou sur l'objet « identifiant de transaction » est libéré, débloquant ainsi l'autre transaction, qui peut reprendre son travail.

Ce mécanisme ne nécessite pas un nombre de verrous mémoire proportionnel au nombre d'enregistrements à verrouiller, et simplifie le travail du gestionnaire de verrous, celui-ci ayant un nombre bien plus faible de verrous à gérer.

Le mécanisme exposé ici est évidemment simplifié.

11.9.4 La vue pg_locks

- `pg_locks` :
  - visualisation des verrous en place
  - tous types de verrous sur objets
- Complexe à interpréter :
  - verrous sur enregistrements pas directement visibles

C'est une vue globale à l'instance.

```
# \d pg_locks
```

```
                      Vue « pg_catalog.pg_locks »
      Colonne        |   Type                    | Collationnement | NULL-able | …
--------------------+---------------------------+-----------------+-----------+-
 locktype            | text                      |                 |           |
 database            | oid                       |                 |           |
 relation            | oid                       |                 |           |
 page                | integer                   |                 |           |
 tuple               | smallint                  |                 |           |
 virtualxid          | text                      |                 |           |
 transactionid       | xid                       |                 |           |
 classid             | oid                       |                 |           |
 objid               | oid                       |                 |           |
 objsubid            | smallint                  |                 |           |
 virtualtransaction  | text                      |                 |           |
 pid                 | integer                   |                 |           |
 mode                | text                      |                 |           |
 granted             | boolean                   |                 |           |
 fastpath            | boolean                   |                 |           |
 waitstart           | timestamp with time zone  |                 |           |
```

- `locktype` est le type de verrou, les plus fréquents étant `relation` (table ou index), `transactionid` (transaction), `virtualxid` (transaction virtuelle, utilisée tant qu'une transaction n'a pas eu à modifier de données, donc à stocker des identifiants de transaction dans des enregistrements) ;
- `database` est la base dans laquelle ce verrou est pris ;
- `relation` est l'OID de la relation cible si locktype vaut `relation` (ou `page` ou `tuple`) ;
- `page` est le numéro de la page dans une relation (pour un verrou de type `page` ou `tuple`) cible ;
- `tuple` est le numéro de l'enregistrement cible (quand verrou de type `tuple`) ;
- `virtualxid` est le numéro de la transaction virtuelle cible (quand verrou de type `virtualxid`) ;

- `transactionid` est le numéro de la transaction cible ;
- `classid` est le numéro d'OID de la classe de l'objet verrouillé (autre que relation) dans `pg_class`. Indique le catalogue système, donc le type d'objet, concerné. Aussi utilisé pour les advisory locks ;
- `objid` est l'OID de l'objet dans le catalogue système pointé par `classid` ;
- `objsubid` correspond à l'ID de la colonne de l'objet `objid` concerné par le verrou ;
- `virtualtransaction` est le numéro de transaction virtuelle possédant le verrou (ou tentant de l'acquérir si `granted` vaut `f`) ;
- `pid` est le PID (l'identifiant de processus système) de la session possédant le verrou ;
- `mode` est le niveau de verrouillage demandé ;
- `granted` signifie si le verrou est acquis ou non (donc en attente) ;
- `fastpath` correspond à une information utilisée surtout pour le débogage (*fastpath* est le mécanisme d'acquisition des verrous les plus faibles) ;
- `waitstart` indique depuis quand le verrou est en attente.

La plupart des verrous sont de type relation, `transactionid` ou `virtualxid`. Une transaction qui démarre prend un verrou virtualxid sur son propre `virtualxid`. Elle acquiert des verrous faibles (`ACCESS SHARE`) sur tous les objets sur lesquels elle fait des `SELECT`, afin de garantir que leur structure n'est pas modifiée sur la durée de la transaction. Dès qu'une modification doit être faite, la transaction acquiert un verrou exclusif sur le numéro de transaction qui vient de lui être affecté. Tout objet modifié (table) sera verrouillé avec `ROW EXCLUSIVE`, afin d'éviter les `CREATE INDEX` non concurrents, et empêcher aussi les verrouillages manuels de la table en entier (`SHARE ROW EXCLUSIVE`).

11.9.5 Verrous - Paramètres

- Nombre :
  - `max_locks_per_transaction` (+ paramètres pour la sérialisation)
- Durée :
  - `lock_timeout` (éviter l'empilement des verrous)
  - `deadlock_timeout` (défaut 1 s)
- Trace :
  - `log_lock_waits`

Nombre de verrous :

`max_locks_per_transaction` sert à dimensionner un espace en mémoire partagée destinée aux verrous sur des objets (notamment les tables). Le nombre de verrous est :

```
max_locks_per_transaction × max_connections
```

Si les transactions préparées (liées aux transactions en deux phases[12]) sont activées, en montant `max_prepared_transactions` au-delà de 0, le nombre de verrous devient :

```
max_locks_per_transaction × (max_connections + max_prepared_transactions)
```

La valeur par défaut de 64 est largement suffisante la plupart du temps. Il peut arriver qu'il faille le monter, par exemple si l'on utilise énormément de partitions, mais le message d'erreur est explicite.

Le nombre maximum de verrous d'une session n'est pas limité à `max_locks_per_transaction`. C'est une valeur moyenne. Une session peut acquérir autant de verrous qu'elle le souhaite pourvu qu'au total la table de hachage interne soit assez grande. Les verrous de lignes sont stockés sur les lignes et donc potentiellement en nombre infini.

Pour la sérialisation, les verrous de prédicat possèdent des paramètres spécifiques. Pour économiser la mémoire, les verrous peuvent être regroupés par bloc ou relation (voir `pg_locks` pour le niveau de verrouillage). Les paramètres respectifs sont :

- `max_pred_locks_per_transaction` (64 par défaut) ;
- `max_pred_locks_per_page` (par défaut 2, donc 2 lignes verrouillées entraînent le verrouillage de tout le bloc, du moins pour la sérialisation) ;
- `max_pred_locks_per_relation` (voir la documentation[13] pour les détails).

Durées maximales de verrou :

Si une session attend un verrou depuis plus longtemps que `lock_timeout`, la requête est annulée. Il est courant de poser cela avant un ordre assez intrusif, même bref, sur une base utilisée. Par exemple, il faut éviter qu'un `VACUUM FULL`, s'il est bloqué par une transaction un peu longue, ne bloque lui-même toutes les transactions suivantes (phénomène d'empilement des verrous) :

```
postgres=# SET lock_timeout TO '3s' ;
SET
postgres=# VACUUM FULL t_grosse_table ;
ERROR:  canceling statement due to lock timeout
```

Il faudra bien sûr retenter le `VACUUM FULL` plus tard, mais la production n'est pas bloquée plus de 3 secondes.

PostgreSQL recherche périodiquement les *deadlocks* entre transactions en cours. La périodicité par défaut est de 1 s (paramètre `deadlock_timeout`), ce qui est largement suffisant la plupart du temps : les *deadlocks* sont assez rares, alors que la vérification est quelque chose de coûteux. L'une des transactions est alors arrêtée et annulée, pour que les autres puissent continuer :

```
postgres=*# DELETE FROM t_centmille_int WHERE i < 50000;
```

[12] https://docs.postgresql.fr/current/two-phase.html
[13] https://docs.postgresql.fr/current/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-RELATION

```
ERROR:  deadlock detected
DÉTAIL : Process 453259 waits for ShareLock on transaction 3010357;
blocked by process 453125.
Process 453125 waits for ShareLock on transaction 3010360;
blocked by process 453259.
ASTUCE : See server log for query details.
CONTEXTE : while deleting tuple (0,1) in relation "t_centmille_int"
```

Trace des verrous :

Pour tracer les attentes de verrous un peu longue, il est fortement conseillé de passer `log_lock_waits` à `on` (le défaut est `off`).

Le seuil est également défini par `deadlock_timeout` (1 s par défaut) Ainsi, une session toujours en attente de verrou au-delà de cette durée apparaîtra dans les traces :

```
LOG:  process 457051 still waiting for ShareLock on transaction 35373775
      after 1000.121 ms
DETAIL:  Process holding the lock: 457061. Wait queue: 457051.
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
```

S'il ne s'agit pas d'un *deadlock*, la transaction continuera, et le moment où elle obtiendra son verrou sera également tracé :

```
LOG:  process 457051 acquired ShareLock on transaction 35373775 after
      18131.402 ms
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
LOG:  duration: 18203.059 ms  statement: DELETE FROM t_centmille_int ;
```

11.10 DURÉES DE SESSIONS, TRANSACTIONS & ORDRES

11.10.1 Quelle durée pour les sessions & transactions ?

Divers seuils possibles, jamais globalement.

```
SET …_timeout TO '5s' ;
ALTER ROLE … IN DATABASE … SET ..._timeout TO '…s'
```

Paramètre	Cible du seuil
`lock_timeout`	Attente de verrou
`statement_timeout`	Ordre en cours
`idle_session_timeout`	Session inactive
`idle_in_transaction_session_timeout`	Transaction en cours, inactive
`transaction_timeout` (v17)	Transaction en cours

PostgreSQL possède divers seuils pour éviter des ordres, transactions ou sessions trop longues. Le dépassement d'un seuil provoque la fin et l'annulation de l'ordre, transaction ou session. Par défaut, aucun n'est activé.

Ne définissez jamais des paramètres globalement, les ordres de maintenance ou les batchs de nuit seraient aussi concernés, tout comme les superutilisateurs. Utilisez un ordre `SET` dans la session, ou `ALTER ROLE … IN DATABASE … SET …`.

Il est important que l'application sache gérer l'arrêt de connexion ou l'annulation d'un ordre ou d'une transaction et réagir en conséquence (nouvelle tentative, abandon avec erreur…). Dans le cas contraire, l'application pourrait rester déconnectée, et suivant les cas des données pourraient être perdues.

`lock_timeout` permet d'interrompre une requête qui mettrait trop de temps à acquérir son verrou. Il faut l'activer par précaution avant une opération lourde (un `VACUUM FULL` notamment) pour éviter un « empilement des verrous ». En effet, si l'ordre ne peut pas s'exécuter immédiatement, il bloque toutes les autres requêtes sur la table.

`statement_timeout` est la durée maximale d'un ordre. Défini au niveau d'un utilisateur ou d'une session, cet ordre permet d'éviter que des requêtes aberrantes chargent la base pendant des heures ou des jours ; ou encore que des requêtes « s'empilent » sur une base totalement saturée ou avec trop de contention. Autre exemple : les utilisateurs de supervision et autres utilisateurs non critiques.

À noter : l'extrait suivant d'une sauvegarde par `pg_dump` montre que l'outil inhibe ces paramètres par précaution, afin que la restauration n'échoue pas.

```
-- Dumped from database version 17.2 (Debian 17.2-1.pgdg120+1)
-- Dumped by pg_dump version 17.2 (Debian 17.2-1.pgdg120+1)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
…
```

L'idée peut être reprise pour des scripts batchs, par exemple.

11.10.2 Quelle durée pour une session ?

- Courte
  - coût & temps des connexions
  - pooler ?
- Longue
  - risque de saturation du nombre de connexions
  - (rare) gaspillage mémoire par les backends

Il est généralement conseillé d'utiliser des sessions assez longues, car la création d'une connexion à PostgreSQL est une opération lourde. Si l'applicatif ne cesse de se connecter/déconnecter, il faudra penser à un pooler (pgBouncer, ou côté applicatif).

Des sessions très longues et inactives ne sont en général pas un souci. Un pooler garde justement ses sessions longtemps. Mais si les sessions ne se ferment jamais, le nombre maximal de sessions (`max_connections`) peut être atteint, et de nouvelles connexions refusées.

De plus, pour les performances, il n'est pas très bon qu'il y ait des milliers de sessions ouvertes, malgré les progrès des versions récentes de PostgreSQL. Enfin, si la consommation mémoire du *backend* associé à une session est raisonnable, il arrive, exceptionnellement, qu'il se mette à occuper durablement de la mémoire (par exemple en cas d'accès à des milliers de tables et index durant son existence).

La durée maximale des sessions peut être réglée par `idle_session_timeout`, ou au niveau du pooler s'il y en a un.

11.10.3 Quelle durée pour une transaction ?

- Courte
  - synchronisation fréquente coûteuse
- Longue
  - verrous bloquants

Les transactions ouvertes et figées durablement sans faire de `COMMIT` ou `ROLLBACK` sont un problème sérieux, car l'autovacuum ne pourra pas nettoyer de ligne que cette session pourrait encore voir. Si l'applicatif ne peut être corrigé, ou si des utilisateurs ouvrent un outil quelconque sans le refermer, une solution est de définir `idle_in_transaction_session_timeout`, par exemple à une ou plusieurs heures.

Pour les performances, il faut éviter les sessions trop courtes car le coût de la synchronisation sur disque des journaux lors d'un `COMMIT` est coûteux (sauf table *unlogged* ou `synchronous_commit`). Typiquement, un traitement de batch ou de chargement regroupera de nombreuses opérations en une seule transaction.

D'un autre côté, chaque transaction maintient ses verrous jusqu'à sa fin, et peut donc bloquer d'autres transactions et ralentir toute l'application. En utilisation transactionnelle, il vaut donc mieux que les transactions soient courtes et n'en fassent pas plus que ce qui est dicté par le besoin fonctionnel, et si possible le plus tard possible dans la transaction.

`transaction_timeout` (à partir de PostgreSQL 17) peut alors servir comme sécurité en cas de problème. À noter que cette limite ne concerne pas les transactions préparées, liées au transactions en deux phases[14], et dont la longueur est parfois un souci.

[14] https://docs.postgresql.fr/current/two-phase.html

11.11 TOAST

11.11.1 Mécanisme TOAST

TOAST : *The Oversized-Attribute Storage Technique*
Que faire si une ligne dépasse d'un bloc ?

- Compresser
- Déporter dans une table
- Ou les deux
- Inutile de le faire dans l'applicatif
- Politique par champ
  - `PLAIN` / `MAIN` / `EXTERNAL` / `EXTENDED`

Principe :

Une ligne ne peut déborder d'un bloc, et un bloc fait 8 ko par défaut. Cela ne suffit pas pour certains champs qui peuvent être beaucoup plus longs, comme certains textes, mais aussi des types composés (`json`, `jsonb`, `hstore`), ou binaires (`bytea`), et même `numeric`.

Le mécanisme TOAST consiste en deux mécanismes. Le premier compresse le contenu des champs, mais ça ne suffit pas forcément. Le second consiste à découper les champs trop longs et à déporter les morceaux dans une autre table associée à la table principale, table gérée de manière transparente pour l'utilisateur. La combinaison des deux mécanismes est possible. Ces deux mécanismes permettent au final d'éviter qu'un enregistrement ne dépasse la taille d'un bloc.

Politiques de stockage :

Chaque champ possède une propriété de stockage :

```
CREATE TABLE demotoast (i int, t text, png bytea, j jsonb);

# \d+ unetable
                    Table « public.demotoast »
 Colonne |  Type   |  Col…  | NULL-able | Par défaut | Stockage | …
---------+---------+--------+-----------+------------+----------+--
 i       | integer |        |           |            | plain    |
 t       | text    |        |           |            | extended |
 png     | bytea   |        |           |            | extended |
 j       | jsonb   |        |           |            | extended |
Méthode d'accès : heap
```

Les différentes politiques de stockage sont :

- `PLAIN` : stockage uniquement dans la table, sans compression (champs numériques ou dates notamment) ;

- `MAIN` : stockage dans la table autant que possible, éventuellement compressé (politique rarement utilisée) ;
- `EXTERNAL` : stockage au besoin dans la table TOAST, sans chercher à compresser ;
- `EXTENDED` : stockage au besoin dans la table TOAST, éventuellement compressé (cas général des champs texte ou binaire).

Il est rare d'avoir à modifier ce paramétrage, mais cela arrive. Par exemple, certains longs champs (souvent binaires, par exemple du JPEG, des PNG, des PDF…) sont déjà compressés et il ne vaut pas la peine de gaspiller du CPU dans cette tâche. Dans le cas extrême où le champ compressé est plus grand que l'original, PostgreSQL revient à la valeur originale, mais du CPU a été consommé inutilement. Il peut alors être intéressant de passer de `EXTENDED` à `EXTERNAL`, pour un gain de temps parfois non négligeable :

```
ALTER TABLE demotoast ALTER COLUMN png SET STORAGE EXTERNAL ;
```

Lors d'un changement de mode de stockage, les données existantes ne sont pas modifiées. Il faut modifier les champs d'une manière ou d'une autre pour forcer le changement.

Performances :

Le découpage et la compression restent des opérations coûteuses. Il reste déconseillé de stocker des données binaires de grande taille dans une base de données ! De plus, cela surcharge les journaux de transaction et les sauvegardes.

Il faut rappeler qu'accéder aux champs « toastés » sans en avoir besoin est une mauvaise pratique :

```
INSERT INTO demotoast
SELECT i, rpad('0',100000,'0') AS t,
pg_read_binary_file ('/tmp/slonik.png'),
(SELECT jsonb_agg(to_json(rq)) FROM (SELECT * FROM pg_stat_all_tables) AS rq)
FROM generate_series (1,1000) i;
```

```
EXPLAIN (ANALYZE,VERBOSE,BUFFERS,SERIALIZE) SELECT i FROM demotoast ;
```

```
                            QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on public.demotoast  (cost=0.00..177.00 rows=1000 width=4) (actual
 ↪  time=0.008..0.178 rows=1000 loops=1)
   Output: i
   Buffers: shared hit=167
 Query Identifier: -2257406617252911738
 Planning Time: 0.029 ms
 Serialization: time=0.073 ms  output=9kB  format=text
 Execution Time: 0.321 ms
```

```
EXPLAIN (ANALYZE,VERBOSE,BUFFERS,SERIALIZE) SELECT * FROM demotoast ;
```

```
                            QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on public.demotoast  (cost=0.00..177.00 rows=1000 width=1196) (actual
 ↪  time=0.012..0.182 rows=1000 loops=1)
```

```
  Output: i, t, png, j
  Buffers: shared hit=167
Query Identifier: 698565989149231362
Planning Time: 0.055 ms
Serialization: time=397.480 ms  output=307084kB  format=text
  Buffers: shared hit=14500
Execution Time: 397.776 ms
```

L'effet du `SELECT *` est ici désastreux, avec 14 500 blocs lus supplémentaires. (Noter que l'effet n'est visible dans `EXPLAIN (ANALYZE)` qu'avec l'option `SERIALIZE`, qui n'est disponible qu'à partir de PostgreSQL 17. Sans cette option, `EXPLAIN(ANALYZE)` affiche un résultat trompeur par rapport aux requêtes réelles !)

11.11.2 TOAST & table de débordement

- Table de débordement `pg_toast_XXX`
  - masquée, transparente
- Jusqu'à 1 Go par champ (déconseillé)
  - texte, JSON, binaire…
  - compression optionnelle
- Une raison de plus d'éviter les `SELECT *`

Principe des tables de débordement :

Le débordement dans des tables séparées de la table principale a d'autres intérêts :

- la partie principale d'une table ayant des champs très longs est moins grosse, alors que les « gros champs » ont moins de chance d'être accédés systématiquement par le code applicatif ;
- ces champs peuvent aussi être compressés de façon transparente, avec souvent de gros gains en place ;
- si un `UPDATE` ne modifie pas un de ces champs « toastés », la table TOAST n'est pas mise à jour : le pointeur vers l'enregistrement de cette table est juste « cloné » dans la nouvelle version de l'enregistrement.

Les tables pg_toast_XXX :

À chaque table utilisateur se voit associée une table TOAST, et ce dès sa création si la table possède un champ « toastable ». Les enregistrements y sont stockés en morceaux (*chunks*) d'un peu moins de 2 ko. Tous les champs « toastés » d'une table se retrouvent dans la même table `pg_toast_XXX`, dans un espace de nommage séparé nommé `pg_toast`.

Pour l'utilisateur, les tables TOAST sont totalement transparentes. Un développeur doit juste savoir qu'il n'a pas besoin de déporter des champs texte (ou JSON, ou binaires…) imposants dans une table séparée pour des raisons de volumétrie de la table principale : PostgreSQL le fait déjà, et de manière efficace ! Il est donc souvent inutile de se donner la peine de compresser les données au niveau applicatif juste pour réduire le stockage.

La présence de ces tables n'apparaît guère que dans `pg_class`, par exemple ainsi :

```
SELECT * FROM pg_class c
WHERE c.relname = 'demotoast'
OR c.oid = (SELECT reltoastrelid FROM pg_class
            WHERE relname = 'demotoast');
```

```
-[ RECORD 1 ]-------+-----------------
oid                 | 1315757
relname             | pg_toast_1315754
relnamespace        | 99
reltype             | 0
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 1315757
reltablespace       | 0
relpages            | 9500
reltuples           | 38000
relallvisible       | 9500
reltoastrelid       | 0
relhasindex         | t
…
-[ RECORD 2 ]-------+-----------------
oid                 | 1315754
relname             | demotoast
relnamespace        | 2200
reltype             | 1315756
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 1315754
reltablespace       | 0
relpages            | 167
reltuples           | 1000
relallvisible       | 0
reltoastrelid       | 1315757
relhasindex         | f
…
```

La partie TOAST est une table à part entière, avec une clé primaire. On ne peut ni ne doit y toucher !

```
\d+ pg_toast.pg_toast_1315754

Table TOAST « pg_toast.pg_toast_1315754 »
  Colonne    |  Type   | Stockage
-------------+---------+----------
 chunk_id    | oid     | plain
 chunk_seq   | integer | plain
 chunk_data  | bytea   | plain
```

```
Table propriétaire : « public.demotoast »
Index :
    "pg_toast_1315754_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Méthode d'accès : heap
```

L'index de la table TOAST est toujours utilisé pour accéder aux *chunks*.

La volumétrie des différents éléments (partie principale, TOAST, index éventuels) peut se calculer grâce à cette requête dérivée du wiki[15] :

```
SELECT
    oid AS table_oid,
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltoastrelid,
    reltoastrelid::regclass::text AS table_toast,
    reltuples AS nb_lignes_estimees,
    pg_size_pretty(pg_table_size(c.oid)) AS " Table (dont TOAST)",
    pg_size_pretty(pg_relation_size(c.oid)) AS "  Heap",
    pg_size_pretty(pg_relation_size(reltoastrelid)) AS "  Toast",
    pg_size_pretty(pg_indexes_size(reltoastrelid)) AS  "  Toast (PK)",
    pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
    pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname = 'demotoast'
\gx
```

```
-[ RECORD 1 ]-------+-------------------------
table_oid           | 1315754
table               | public.demotoast
reltoastrelid       | 1315757
table_toast         | pg_toast.pg_toast_1315754
nb_lignes_estimees  | 1000
 Table (dont TOAST) | 76 MB
  Heap              | 1336 kB
  Toast             | 74 MB
  Toast (PK)        | 816 kB
 Index              | 0 bytes
Total               | 76 MB
```

On constate que la plus grande partie de la volumétrie de la table est en fait dans la partie TOAST.

La taille des éventuels index sur les champs susceptibles d'être toastés est comptabilisée avec tous les index de la table (la clé primaire de la table TOAST est à part).

Détails du mécanisme TOAST :

Les détails techniques du mécanisme TOAST sont dans la documentation officielle[16]. En résumé, le mécanisme TOAST est déclenché sur un enregistrement quand la taille d'un enregistrement dépasse 2 ko. Les champs « toastables » peuvent alors être compressés pour que la taille de l'enregistrement redescende en-dessous de 2 ko. Si cela ne suffit pas, des champs sont alors découpés et déportés vers la table TOAST. Dans ces champs de la table principale, l'enregistrement ne contient plus qu'un pointeur vers la table TOAST associée.

[15] https://wiki.postgresql.org/wiki/Disk_Usage
[16] https://doc.postgresql.fr/current/storage-toast.html

Un champ MAIN peut tout de même être stocké dans la table TOAST, si l'enregistrement, même compressé, dépasse 2 ko : mieux vaut « toaster » que d'empêcher l'insertion.

Cette valeur de 2 ko convient généralement. Au besoin, on peut l'augmenter en utilisant le paramètre de stockage `toast_tuple_target` ainsi :

```
ALTER TABLE demotoast SET (toast_tuple_target = 3000);
```

mais cela est rarement utile.

Le mécanisme peut différer entre deux lignes pour un même champ, en fonction de la taille réelle de celui-ci sur la ligne.

Il est possible d'avoir le détail champ par champ :

```
SELECT i,
        pg_column_compression (png) AS png_c,
        pg_column_toast_chunk_id (png) AS ck_png,
        pg_column_size(png) AS png_size,
        pg_column_compression (j) AS j_c,
        pg_column_toast_chunk_id (j) AS ck_j,
        pg_column_size (j) AS j_size
FROM demotoast
LIMIT 3 ;
```

```
 i | png_c | ck_png  | png_size | j_c  |  ck_j   | j_size
---+-------+---------+----------+------+---------+--------
 1 |       | 1315760 |    58635 | pglz | 1315759 |  14418
 2 |       | 1315762 |    58635 | pglz | 1315761 |  14418
 3 |       | 1315764 |    58635 | pglz | 1315763 |  14418
```

(La fonction `pg_column_compression()` existe depuis PostgreSQL 14 et indique l'algorithme de compression éventuel (voir plus bas), et `pg_column_toast_chunk_id()` depuis PostgreSQL 17.)

On constate que le champ `png` n'est pas compressé (puisqu'on l'a défini comme `EXTERNAL`), et que le champ de `j` de type `jsonb` est compressé avec l'algorithme indiqué. Chaque champ de chaque ligne a son `chunk_id` propre qui permet de compter le nombre de *chunks*

```
SELECT chunk_id, count(*)
  FROM pg_toast.pg_toast_1315754
 WHERE chunk_id IN ( 1315760, 1315759 )
GROUP BY 1;
```

```
 chunk_id | count
----------+-------
  1315759 |     8
  1315760 |    30
```

En multipliant par 2 ko, on retrouve à peu près les tailles sur disque (compressées ou pas) renvoyées par `pg_column_size()`.

<!– TODO : faire une requête pour savoir le % de compressé et de déporté dans chaque champ de chaque table ? ça réclame du dynamique… – > KB ? –>

Administration des tables TOAST :

Les tables TOAST restent forcément dans le même tablespace que la table principale. Leur maintenance (notamment le nettoyage par `autovacuum`) s'effectue en même temps que la table principale, comme le montre un `VACUUM (VERBOSE)`. Une variante de l'ordre permet d'ailleurs de passer outre le nettoyage d'une table TOAST pour accélérer un nettoyage urgent (depuis PostgreSQL 14) :

```
VACUUM (VERBOSE, PROCESS_TOAST off) longs_textes ;
```

11.11.3 TOAST & compression

- `pglz` (`zlib`) : défaut
- `lz4` à préférer
  - généralement plus rapide
  - compression équivalente (à vérifier)
- Mise en place :

```
default_toast_compression = lz4
```

ou :

```
ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4 ;
```

Depuis la version 14, il est possible de modifier l'algorithme de compression. Ceci est défini par le nouveau paramètre `default_toast_compression` dont la valeur par défaut est :

```
SHOW default_toast_compression ;

 default_toast_compression
---------------------------
 pglz
```

c'est-à-dire que PostgreSQL utilise la zlib, seule compression disponible jusqu'en version 13 incluse.

À partir de la version 14, un nouvel algorithme `lz4` est disponible (et intégré dans les paquets distribués par le PGDG).

De manière générale, l'algorithme `lz4` ne compresse pas mieux les données courantes que `pglz`, mais cela dépend des usages. Surtout, `lz4` est **beaucoup** plus rapide à compresser, et parfois à décompresser.
Nous conseillons d'utiliser `lz4` pour la compression de vos TOAST, même si, en toute rigueur, l'arbitrage entre consommations CPU en écriture ou lecture et place disque ne peut se faire qu'en testant soigneusement avec les données réelles.

Pour changer l'algorithme, vous pouvez :

- soit changer la valeur par défaut dans `postgresql.conf` :

```
default_toast_compression = lz4
```

- soit déclarer la méthode de compression à la création de la table :

```
CREATE TABLE t1 (
  c1 bigint GENERATED ALWAYS AS identity,
  c2 text   COMPRESSION lz4
  ) ;
```

- soit après création :

```
ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4 ;
```

`lz4` accélère aussi les restaurations logiques comportant beaucoup de données toastées et compressées. Si `lz4` n'a pas été activé par défaut, il peut être utilisé dès le chargement :

```
$ PGOPTIONS='-c default_toast_compression=lz4'  pg_restore …
```

Des lignes d'une même table peuvent être compressées de manières différentes. En effet, l'utilisation de `SET COMPRESSION` sur une colonne préexistante ne recompresse pas les données de la table TOAST associée. De plus, des données toastées lues par une requête, puis réinsérées sans être modifiées, sont recopiées vers les champs cibles telles quelles, sans étapes de décompression/recompression, et ce même si la compression de la cible est différente, même s'il s'agit d'une autre table. Même un `VACUUM FULL` sur la table principale réécrit la table TOAST, en recopiant simplement les champs compressés tels quels.

Pour forcer la recompression de toutes les données d'une colonne, il faut modifier leur contenu. Et en fait, c'est peu intéressant car l'écart de volumétrie est généralement faible. Noter qu'il existe une fonction `pg_column_compression (nom_champ)`[17] pour consulter la compression d'un champ sur chaque ligne concernée.

Pour aller plus loin :

- Blog Dalibo : Les mains dans le cambouis #2 - Le mécanisme de TOAST[18] (2024)
- Blog Fujitsu PostgreSQL : What is the new LZ4 TOAST compression in PostgreSQL 14, and how fast is it?[19] (2021), avec notamment un benchmark montrant l'intérêt de la compression `lz4`.

[17] https://docs.postgresql.fr/current/functions-admin.html#FUNCTIONS-ADMIN-DBOBJECT
[18] https://blog.dalibo.com/2024/02/27/toast.html
[19] https://www.postgresql.fastware.com/blog/what-is-the-new-lz4-toast-compression-in-postgresql-14

11.12 CONCLUSION

- PostgreSQL dispose d'une implémentation MVCC complète, permettant :
  - que les lecteurs ne bloquent pas les écrivains
  - que les écrivains ne bloquent pas les lecteurs
  - que les verrous en mémoire soient d'un nombre limité
- Cela impose par contre une mécanique un peu complexe, dont les parties visibles sont la commande `VACUUM` et le processus d'arrière-plan autovacuum.

11.12.1 Questions

N'hésitez pas, c'est le moment !

11.13 QUIZ

https://dali.bo/m4_quiz

11.14 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m4_solutions.

11.14.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ

But : Découvrir les niveaux d'isolation

Créer une nouvelle base de données nommée **b2**.

Dans la base de données **b2**, créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type `text`.

Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

Ouvrir une transaction.

Lire les données de la table `t1`.

Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

Revenir à la première session et lire de nouveau toute la table `t1`.

Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

Lire les données de la table `t1`.

Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

11.14.2 Niveau d'isolation SERIALIZABLE (Optionnel)

But : Découvrir le niveau d'isolation *serializable*

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette procédure de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement), 0)
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

Au sein de trois transactions successives :

- insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**
- chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`

- la sécurité fonctionne-t-elle ?

Pour le client **2**, ouvrir deux transactions en parallèle :

- dans chacune, procéder à retrait de 500 € ;
- dans chacune, appeler `CALL verifie_solde_positif (2)` ;
- dans chacune, valider la transaction ;
- la règle du solde positif est-elle respectée ?

- Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables : (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
- Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :

```
SELECT locktype, mode, pid, granted FROM pg_locks
WHERE relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes')
 ↪  ;
```

11.14.3 Effets de MVCC

But : Voir l'effet du MVCC dans les lignes

Créer une nouvelle table `t2` avec deux colonnes : un entier et un texte.

Insérer 5 lignes dans la table `t2`, de `(1, 'un')` à `(5, 'cinq')`.

Lire les données de la table `t2`.

Commencer une transaction. Mettre en majuscules le texte de la troisième ligne de la table `t2`.

Lire les données de la table `t2`. Que faut-il remarquer ?

Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

Afficher `xmin` et `xmax` lors de la lecture des données de la table `t2`, dans chaque session.

Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`, dans chaque session.

Valider la transaction.

Installer l'extension `pageinspect`.

La documentation[a] indique comment décoder le contenu du bloc 0 de la table `t2` :

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
```

Que faut-il remarquer ?

[a] https://docs.postgresql.fr/current/pageinspect.html

- Lancer `VACUUM` sur `t2`.
- Relancer la requête avec `pageinspect`.
- Comment est réorganisé le bloc ?

Pourquoi l'autovacuum n'a-t-il pas nettoyé encore la table ?

11.14.4 Verrous

But : Trouver des verrous

Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

> Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

> Récupérer la liste des verrous en attente pour la requête bloquée.

> Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

> Récupérer la liste des verrous sur cet objet. Quel processus a verrouillé la table `t1` ?

> Retrouver les informations sur la session bloquante.

> Retrouver cette information avec la fonction `pg_blocking_pids`.

> Détruire la session bloquant le `DROP TABLE`.

> Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

> Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

11.15 TRAVAUX PRATIQUES (SOLUTIONS)

11.15.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ

> Créer une nouvelle base de données nommée **b2**.

```
$ createdb b2
```

> Dans la base de données **b2**, créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type `text`.

```
CREATE TABLE t1 (c1 integer, c2 text);
```

```
CREATE TABLE
```

> Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

```
INSERT INTO t1 (c1, c2) VALUES
  (1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
```

```
INSERT 0 5
```

> Ouvrir une transaction.

```
BEGIN;
```

```
BEGIN
```

> Lire les données de la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

> Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

```
UPDATE t1 SET c2 = upper(c2) WHERE c1 = 3;
```

```
UPDATE 1
```

> Revenir à la première session et lire de nouveau toute la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

Les modifications réalisées par la deuxième transaction sont immédiatement visibles par la première transaction. C'est le cas des transactions en niveau d'isolation READ COMMITED.

> Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

```
ROLLBACK;
```

```
ROLLBACK;
```

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

> Lire les données de la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

> Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

```
UPDATE t1 SET c2 = upper(c2) WHERE c1 = 4;
```

```
UPDATE 1
```

> Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

En niveau d'isolation `REPEATABLE READ`, la transaction est certaine de ne pas voir les modifications réalisées par d'autres transactions (à partir de la première lecture de la table).

11.15.2 Niveau d'isolation SERIALIZABLE (Optionnel)

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette procédure de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement), 0)
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

> Au sein de trois transactions successives :
>
> - insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**
> - chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`
> - la sécurité fonctionne-t-elle ?

Ce client a bien 600 € :

```
SELECT * FROM mouvements_comptes WHERE client = 1 ;
```

```
 client | mouvement
--------+-----------
      1 |       100
      1 |       200
      1 |       300
```

Première transaction :

```
BEGIN ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (1, -300) ;
CALL verifie_solde_positif (1) ;
```

```
NOTICE:  Client 1 - Solde positif : 300
CALL
```

```
COMMIT ;
```

Lors d'une seconde transaction : les mêmes ordres renvoient :

```
NOTICE:  Client 1 - Solde positif : 0
```

Avec le troisième débit :

```
BEGIN ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (1, -300) ;
CALL verifie_solde_positif (1) ;
```

```
ERROR:  Client 1 - Solde négatif : -300 !
CONTEXTE : PL/pgSQL function verifie_solde_positif(integer) line 11 at RAISE
```

La transaction est annulée : il est interdit de retirer plus d'argent qu'il n'y en a.

> Pour le client **2**, ouvrir deux transactions en parallèle :
>
> - dans chacune, procéder à retrait de 500 € ;
> - dans chacune, appeler `CALL verifie_solde_positif (2)` ;
> - dans chacune, valider la transaction ;
> - la règle du solde positif est-elle respectée ?

Chaque transaction va donc se dérouler dans une session différente.

Première transaction :

```
BEGIN ; --session 1
INSERT INTO mouvements_comptes(client, mouvement) VALUES (2, -500) ;
CALL verifie_solde_positif (2) ;
```

```
NOTICE:  Client 2 - Solde positif : 100
```

On ne commite pas encore.

Dans la deuxième session, il se passe exactement la même chose :

```
BEGIN ; --session 2
INSERT INTO mouvements_comptes(client, mouvement) VALUES (2, -500) ;
CALL verifie_solde_positif (2) ;
```

```
NOTICE:  Client 2 - Solde positif : 100
```

En effet, cette deuxième session ne voit pas encore le débit de la première session.

Les deux tests étant concluants, les deux sessions committent :

```
COMMIT ; --session 1
```

```
COMMIT
```

```
COMMIT ; --session 2
```

```
COMMIT
```

Au final, le solde est négatif, ce qui est pourtant strictement interdit !

```
CALL verifie_solde_positif (2) ;
```

```
ERROR:  Client 2 - Solde négatif : -400 !
CONTEXTE : PL/pgSQL function verifie_solde_positif(integer) line 11 at RAISE
```

Les deux sessions en parallèle sont donc un moyen de contourner la sécurité, qui porte sur le résultat d'un ensemble de lignes, et non juste sur la ligne concernée.

> - Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables : (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
> - Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :
>
> ```
> SELECT locktype, mode, pid, granted FROM pg_locks
> WHERE relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes')
> ↪  ;
> ```

Première session :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (3, -500) ;
CALL verifie_solde_positif (3) ;
```

```
NOTICE:  Client 3 - Solde positif : 100
```

On ne committe pas encore.

Deuxième session :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (3, -500) ;
CALL verifie_solde_positif (3) ;
```

```
NOTICE:  Client 3 - Solde positif : 100
```

Les verrous en place sont les suivants :

```
SELECT locktype, mode, pid, granted
FROM   pg_locks
WHERE  relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;
```

```
 locktype |       mode       |   pid   | granted
----------+------------------+---------+---------
 relation | AccessShareLock  | 28304 | t
 relation | RowExclusiveLock | 28304 | t
 relation | AccessShareLock  | 28358 | t
 relation | RowExclusiveLock | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

`SIReadLock` est un verrou lié à la sérialisation : noter qu'il porte sur des lignes, portées par les deux sessions. `AccessShareLock` empêche surtout de supprimer la table. `RowExclusiveLock` est un verrou de ligne.

Validation de la première session :

```
COMMIT ;
```

```
COMMIT
```

Dans les verrous, il subsiste toujours les verrous `SIReadLock` de la session de PID 28304, qui a pourtant committé :

```
SELECT locktype, mode, pid, granted
FROM   pg_locks
WHERE  relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;
```

```
 locktype |       mode       |   pid   | granted
----------+------------------+---------+---------
 relation | AccessShareLock  | 28358 | t
 relation | RowExclusiveLock | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

Tentative de validation de la seconde session :

```
COMMIT ;
```

```
ERROR:  could not serialize access due to read/write dependencies among transactions
DÉTAIL : Reason code: Canceled on identification as a pivot, during commit attempt.
ASTUCE : The transaction might succeed if retried.
```

La transaction est annulée pour erreur de sérialisation. En effet, le calcul effectué pendant la seconde transaction n'est plus valable puisque la première a modifié les lignes qu'elle a lues.

La transaction annulée doit être rejouée de zéro, et elle tombera alors bien en erreur.

11.15.3 Effets de MVCC

> Créer une nouvelle table `t2` avec deux colonnes : un entier et un texte.

```
CREATE TABLE t2 (i int, t text) ;
```

```
CREATE TABLE
```

> Insérer 5 lignes dans la table `t2`, de `(1, 'un')` à `(5, 'cinq')`.

```
INSERT INTO t2(i, t)
VALUES
(1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
```

```
INSERT 0 5
```

> Lire les données de la table `t2`.

```
SELECT * FROM t2;
```

```
  i |   t
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

> Commencer une transaction. Mettre en majuscules le texte de la troisième ligne de la table `t2`.

```
BEGIN ;
UPDATE t2 SET t = upper(t) WHERE i = 3;
```

```
UPDATE 1
```

> Lire les données de la table `t2`. Que faut-il remarquer ?

```
SELECT * FROM t2;
```

```
  i |   t
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

La ligne mise à jour n'apparaît plus, ce qui est normal. Elle apparaît en fin de table. En effet, quand un `UPDATE` est exécuté, la ligne courante est considérée comme morte et une nouvelle ligne est ajoutée, avec les valeurs modifiées. Comme nous n'avons pas demandé de récupérer les résultats dans un certain ordre (pas d'`ORDER BY`), les lignes sont simplement affichées dans leur ordre de stockage dans les blocs de la table.

> Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

```
SELECT * FROM t2;
```

```
  i |   t
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

L'ordre des lignes en retour n'est pas le même. Les autres sessions voient toujours l'ancienne version de la ligne 3, puisque la transaction n'a pas encore été validée.

> Afficher `xmin` et `xmax` lors de la lecture des données de la table `t2`, dans chaque session.

Voici ce que renvoie la session qui a fait la modification :

```
SELECT xmin, xmax, * FROM t2;
```

```
 xmin | xmax |  i |   t
------+------+----+--------
  753 |    0 | 1 | un
  753 |    0 | 2 | deux
  753 |    0 | 4 | quatre
  753 |    0 | 5 | cinq
  754 |    0 | 3 | TROIS
```

Et voici ce que renvoie l'autre session :

```
SELECT xmin, xmax, * FROM t2;
```

```
 xmin | xmax |  i |   t
------+------+----+--------
  753 |    0 | 1 | un
  753 |    0 | 2 | deux
  753 |  754 | 3 | trois
  753 |    0 | 4 | quatre
  753 |    0 | 5 | cinq
```

La transaction 754 est celle qui a réalisé la modification. La colonne `xmin` de la nouvelle version de ligne contient ce numéro. De même pour la colonne `xmax` de l'ancienne version de ligne. PostgreSQL se base sur cette information pour savoir si telle transaction peut lire telle ou telle ligne.

> Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`, dans chaque session.

Voici ce que renvoie la session qui a fait la modification :

```
SELECT ctid, xmin, xmax, * FROM t2;
```

```
 ctid  | xmin | xmax | i |   t
-------+------+------+---+--------
 (0,1) |  753 |    0 | 1 | un
 (0,2) |  753 |    0 | 2 | deux
 (0,4) |  753 |    0 | 4 | quatre
 (0,5) |  753 |    0 | 5 | cinq
 (0,6) |  754 |    0 | 3 | TROIS
```

Et voici ce que renvoie l'autre session :

```
SELECT ctid, xmin, xmax, * FROM t2 ;
```

```
 ctid  | xmin | xmax |  i |   t
-------+------+------+----+--------
 (0,1) |  753 |    0 |  1 | un
 (0,2) |  753 |    0 |  2 | deux
 (0,3) |  753 |  754 |  3 | trois
 (0,4) |  753 |    0 |  4 | quatre
 (0,5) |  753 |    0 |  5 | cinq
```

La colonne `ctid` contient une paire d'entiers. Le premier indique le numéro de bloc, le second le numéro de l'enregistrement dans le bloc. Autrement dit, elle précise la position de l'enregistrement sur le fichier de la table.

En récupérant cette colonne, nous voyons que la première session voit la nouvelle position (enregistrement 6 du bloc 0) car elle est dans la transaction 754. Mais pour la deuxième session, cette nouvelle transaction n'est pas validée, donc l'information d'effacement de la ligne 3 n'est pas prise en compte, et on la voit toujours.

> Valider la transaction.

```
COMMIT;
```

```
COMMIT
```

> Installer l'extension `pageinspect`.

```
CREATE EXTENSION pageinspect ;
```

```
CREATE EXTENSION
```

> La documentation[a] indique comment décoder le contenu du bloc 0 de la table `t2` :
>
> ```
> SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
> ```
>
> Que faut-il remarquer ?
>
> [a] https://docs.postgresql.fr/current/pageinspect.html

Cette table est assez petite pour tenir dans le bloc 0 toute entière. `pageinspect` nous fournit le détail de ce qu'il y a dans les lignes (la sortie est coupée en deux pour l'affichage) :

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
```

```
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid |
----+--------+----------+--------+--------+--------+----------+--------+-
  1 |   8160 |        1 |     31 |    753 |      0 |        0 | (0,1)  |
  2 |   8120 |        1 |     33 |    753 |      0 |        0 | (0,2)  |
  3 |   8080 |        1 |     34 |    753 |    754 |        0 | (0,6)  |
  4 |   8040 |        1 |     35 |    753 |      0 |        0 | (0,4)  |
  5 |   8000 |        1 |     33 |    753 |      0 |        0 | (0,5)  |
  6 |   7960 |        1 |     34 |    754 |      0 |        0 | (0,6)  |
 lp | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid |          t_data
----+-------------+------------+--------+--------+-------+-------------------------
  1 |           2 |       2306 |     24 |        |       | \x0100000007756e
  2 |           2 |       2306 |     24 |        |       | \x020000000b64657578
  3 |       16386 |        258 |     24 |        |       | \x030000000d74726f6973
  4 |           2 |       2306 |     24 |        |       | \x040000000f717561747265
  5 |           2 |       2306 |     24 |        |       | \x050000000b63696e71
  6 |       32770 |      10242 |     24 |        |       | \x030000000d54524f4953
```

Tous les champs ne sont pas immédiatement compréhensibles, mais on peut lire facilement ceci :

- Les six lignes sont bien présentes, dont les deux versions de la ligne 3 ;
- Le `t_ctid` de l'ancienne ligne ne contient plus `(0,3)` mais l'adresse de la nouvelle ligne (soit `(0,6)`).

Mais encore :

- Les lignes sont stockées à rebours depuis la fin du bloc : la première a un *offset* de 8160 octets depuis le début, la dernière est à seulement 7960 octets du début ;
- la longueur de la ligne est indiquée par le champ `lp_len` : la ligne 4 est la plus longue ;
- `t_infomask2` est un champ de bits : la valeur 16386 pour l'ancienne version nous indique que le changement a eu lieu en utilisant la technologie HOT (la nouvelle version de la ligne est maintenue dans le même bloc et un chaînage depuis l'ancienne est effectué) ;
- le champ `t_data` contient les valeurs de la ligne : nous devinons `i` au début (01 à 05), et la fin correspond aux chaînes de caractères, précédée d'un octet lié à la taille.

La signification de tous les champs n'est pas dans la documentation mais se trouve dans le code de PostgreSQL[20].

> - Lancer `VACUUM` sur `t2`.
> - Relancer la requête avec `pageinspect`.
> - Comment est réorganisé le bloc ?

```
VACUUM (VERBOSE) t2 ;
```

```
INFO:  vacuuming "postgres.public.t2"
…
tuples: 1 removed, 5 remain, 0 are dead but not yet removable
…
VACUUM
```

Une seule ligne a bien été nettoyée. La requête suivante nous montre qu'elle a bien disparu :

[20] https://doxygen.postgresql.org/itemid_8h_source.html

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
 lp | lp_off | lp_flags | … | t_ctid |… |          t_data
----+--------+----------+---+--------+--+-------------------------
  1 |   8160 |        1 |   | (0,1)  |  | \x0100000007756e
  2 |   8120 |        1 |   | (0,2)  |  | \x020000000b64657578
  3 |      6 |        2 |   |        |  |
  4 |   8080 |        1 |   | (0,4)  |  | \x040000000f717561747265
  5 |   8040 |        1 |   | (0,5)  |  | \x050000000b63696e71
  6 |   8000 |        1 |   | (0,6)  |  | \x030000000d54524f4953
```

La 3è ligne ici a été remplacée par un simple pointeur sur la nouvelle version de la ligne dans le bloc (mise à jour HOT).

On peut aussi remarquer que les lignes non modifiées ont été réorganisées dans le bloc : là où se trouvait l'ancienne version de la 3è ligne (à 8080 octets du début de bloc) se trouve à présent la 4è. Le `VACUUM` opère ainsi une défragmentation des blocs qu'il nettoie.

> Pourquoi l'autovacuum n'a-t-il pas nettoyé encore la table ?

Le vacuum ne se déclenche qu'à partir d'un certain nombre de lignes modifiées ou effacées (50 + 20% de la table par défaut). On est encore très loin de ce seuil avec cette très petite table.

11.15.4 Verrous

> Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

```
BEGIN;
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | TROIS
  4 | QUATRE
  5 | CINQ
```

> Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

```
DROP TABLE t1;
```

La suppression semble bloquée.

> Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

En tant qu'utilisateur système `postgres` :

```
$ ps -o pid,cmd fx
  PID CMD
 2657 -bash
 2693  \_ psql
 2431 -bash
 2622  \_ psql
 2728      \_ ps -o pid,cmd fx
 2415 /usr/pgsql-17/bin/postmaster -D /var/lib/pgsql/15/data/
 2417  \_ postgres: logger
 2419  \_ postgres: checkpointer
 2420  \_ postgres: background writer
 2421  \_ postgres: walwriter
 2422  \_ postgres: autovacuum launcher
 2424  \_ postgres: logical replication launcher
 2718  \_ postgres: postgres b2 [local] DROP TABLE waiting
 2719  \_ postgres: postgres b2 [local] idle in transaction
```

La ligne intéressante est la ligne du `DROP TABLE`. Elle contient le mot clé `waiting`. Ce dernier indique que l'exécution de la requête est en attente d'un verrou sur un objet.

> Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

```
\x
```

```
Expanded display is on.
```

```
SELECT * FROM pg_stat_activity
  WHERE application_name='psql' AND wait_event IS NOT NULL;
```

```
-[ RECORD 1 ]----+-------------------------------
datid            | 16387
datname          | b2
pid              | 2718
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:45.38342+00
xact_start       | 2018-11-02 15:57:32.82511+00
query_start      | 2018-11-02 15:57:32.82511+00
state_change     | 2018-11-02 15:57:32.825112+00
wait_event_type  | Lock
wait_event       | relation
state            | active
backend_xid      | 575
backend_xmin     | 575
query_id         |
query            | drop table t1 ;
backend_type     | client backend
-[ RECORD 2 ]----+-------------------------------
datid            | 16387
datname          | b2
pid              | 2719
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:17.173784+00
xact_start       | 2018-11-02 15:57:25.311573+00
query_start      | 2018-11-02 15:57:25.311573+00
state_change     | 2018-11-02 15:57:25.311573+00
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT * FROM t1;
backend_type     | client backend
```

> Récupérer la liste des verrous en attente pour la requête bloquée.

```
SELECT * FROM pg_locks WHERE pid = 2718 AND NOT granted;
```

```
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
```

```
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 5/7
pid                | 2718
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
waitstart          |
```

> Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

```
SELECT relname FROM pg_class WHERE oid=16394;
```

```
-[ RECORD 1 ]
relname | t1
```

Noter que l'objet n'est visible dans `pg_class` que si l'on est dans la même base de données que lui. D'autre part, la colonne `oid` des tables systèmes n'est pas visible par défaut dans les versions antérieures à la 12, il faut demander explicitement son affichage pour la voir.

> Récupérer la liste des verrous sur cet objet. Quel processus a verrouillé la table `t1` ?

```
SELECT * FROM pg_locks WHERE relation = 16394;
```

```
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 4/10
pid                | 2719
mode               | AccessShareLock
granted            | t
fastpath           | f
waitstart          |
-[ RECORD 2 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
```

```
classid            |
objid              |
objsubid           |
virtualtransaction | 5/7
pid                | 2718
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
waitstart          |
```

Le processus de PID 2718 (le `DROP TABLE`) demande un verrou exclusif sur `t1`, mais ce verrou n'est pas encore accordé (`granted` est à `false`). La session `idle in transaction` a acquis un verrou `Access Share`, normalement peu gênant, qui n'entre en conflit qu'avec les verrous exclusifs.

> Retrouver les informations sur la session bloquante.

On retrouve les informations déjà affichées :

```
SELECT * FROM pg_stat_activity WHERE pid = 2719;
```

```
-[ RECORD 1 ]----+-------------------------------
datid            | 16387
datname          | b2
pid              | 2719
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:17.173784+00
xact_start       | 2018-11-02 15:57:25.311573+00
query_start      | 2018-11-02 15:57:25.311573+00
state_change     | 2018-11-02 15:57:25.311573+00
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT * FROM t1;
backend_type     | client backend
```

> Retrouver cette information avec la fonction `pg_blocking_pids`.

Il existe une fonction pratique indiquant quelles sessions bloquent une autre. En l'occurence, notre `DROP TABLE t1` est bloqué par :

```
SELECT pg_blocking_pids(2718);
```

```
-[ RECORD 1 ]----+-------
pg_blocking_pids | {2719}
```

Potentiellement, la session pourrait attendre la levée de plusieurs verrous de différentes sessions.

> Détruire la session bloquant le `DROP TABLE`.

À partir de là, il est possible d'annuler l'exécution de l'ordre bloqué, le `DROP TABLE`, avec la fonction `pg_cancel_backend()`. Si l'on veut détruire le processus bloquant, il faudra plutôt utiliser la fonction `pg_terminate_backend()` :

```
SELECT pg_terminate_backend (2719) ;
```

Dans ce dernier cas, vérifiez que la table a été supprimée, et que la session en statut `idle in transaction` affiche un message indiquant la perte de la connexion.

> Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

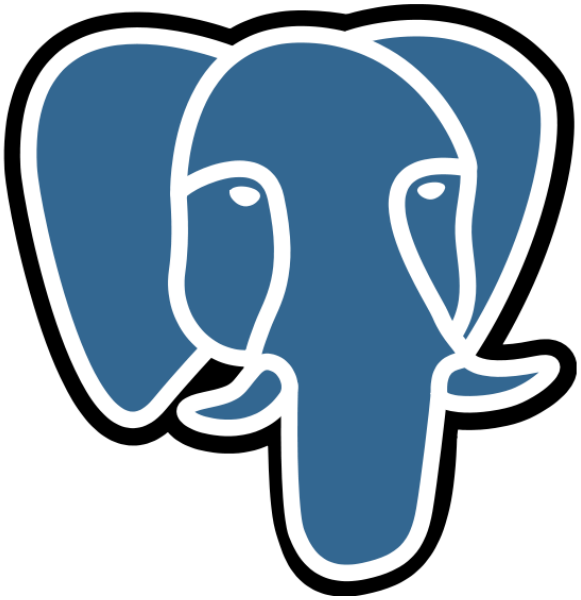
```
LOCK TABLE t1;
```

> Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

Le code source de la vue `pg_show_locks` est le suivant :

```
CREATE VIEW pg_show_locks as
SELECT
        a.pid,
        usename,
        (now() - query_start) as age,
        c.relname,
        l.mode,
        l.granted
FROM
        pg_stat_activity a
        LEFT OUTER JOIN pg_locks l
                ON (a.pid = l.pid)
        LEFT OUTER JOIN pg_class c
                ON (l.relation = c.oid)
WHERE
        c.relname !~ '^pg_'
ORDER BY
        pid;
```

12/ VACUUM et autovacuum



12.1 AU MENU

- Principe & fonctionnement du `VACUUM`
- Options : `VACUUM` seul, `ANALYZE`, `FULL`, `FREEZE`
  - ne pas les confondre !
- Suivi
- Autovacuum
- Paramétrages

VACUUM est la contrepartie de la flexibilité du modèle MVCC. Derrière les différentes options de `VACUUM` se cachent plusieurs tâches très différentes. Malheureusement, la confusion est facile. Il est capital de les connaître et de comprendre leur fonctionnement.

Autovacuum permet d'automatiser le VACUUM et allège considérablement le travail de l'administrateur.

Il fonctionne généralement bien, mais il faut savoir le surveiller et l'optimiser.

12.2 VACUUM ET AUTOVACUUM

- `VACUUM` : nettoie d'abord les lignes mortes
- Mais aussi d'autres opérations de maintenance
- Lancement :
  - manuel par `vacuumdb` (shell, pour appels en masse)
  - manuel par `VACUUM` (SQL)
  - par le démon `autovacuum` (seuils)

`VACUUM` est né du besoin de nettoyer les lignes mortes. Au fil du temps il a été couplé à d'autres ordres (`ANALYZE`, `VACUUM FREEZE`) et s'est occupé d'autres opérations de maintenance (création de la *visibility map* par exemple). Des options permettent de réguler son activité. Son paramétrage n'est donc pas toujours très clair.

L'outil `vacuumdb` est un outil qui génère des ordres `VACUUM` après s'être connecté à PostgreSQL. Il reprend simplement les options de `VACUUM` (voir sa page de manuel[1]). Il est plutôt destiné aux appels depuis le système ou comme tâche planifiée. Par rapport à un appel en SQL, `vacuumdb` facilite les appels en masse, avec notamment ces options :

- `--all` pour nettoyer toutes les bases de données les unes après les autres ;
- `--jobs` pour paralléliser sur plusieurs sessions.

`autovacuum` est un processus de l'instance PostgreSQL. Il est activé par défaut, et il fortement conseillé de le conserver ainsi. Dans le cas général, son fonctionnement convient et il ne gênera pas les utilisateurs. Au contraire, il faudra parfois le rendre plus agressif.

L'autovacuum ne gère pas toutes les variantes de `VACUUM` (notamment pas le `FULL`).

[1] https://docs.postgresql.fr/current/app-vacuumdb.html

12.3 FONCTIONNEMENT DE VACUUM

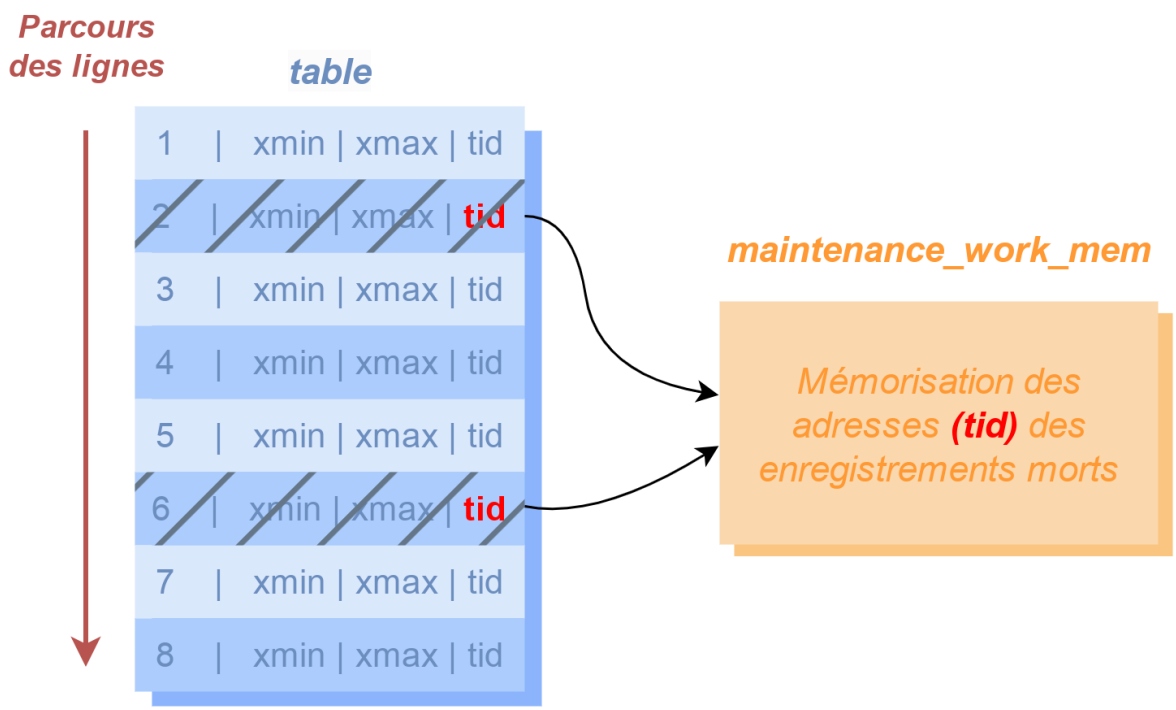


Figure 12/ .1: Phase 1/3 : recherche des enregistrements morts

Un ordre `VACUUM` vise d'abord à nettoyer les lignes mortes.

Le traitement `VACUUM` se déroule en trois passes. Cette première passe parcourt la table à nettoyer, à la recherche d'enregistrements morts. Un enregistrement est mort s'il possède un `xmax` qui correspond à une transaction validée, et que cet enregistrement n'est plus visible dans l'instantané d'aucune transaction en cours sur la base. D'autres lignes mortes portent un `xmin` d'une transaction annulée.

L'enregistrement mort ne peut pas être supprimé immédiatement : des enregistrements d'index pointent vers lui et doivent aussi être nettoyés. La session effectuant le `VACUUM` garde en mémoire la liste des adresses des enregistrements morts, à hauteur d'une quantité indiquée par le paramètre `maintenance_work_mem`. Si cet espace est trop petit pour contenir tous les enregistrements morts, `VACUUM` effectue plusieurs séries de ces trois passes.

12.3.1 Fonctionnement de VACUUM (suite)

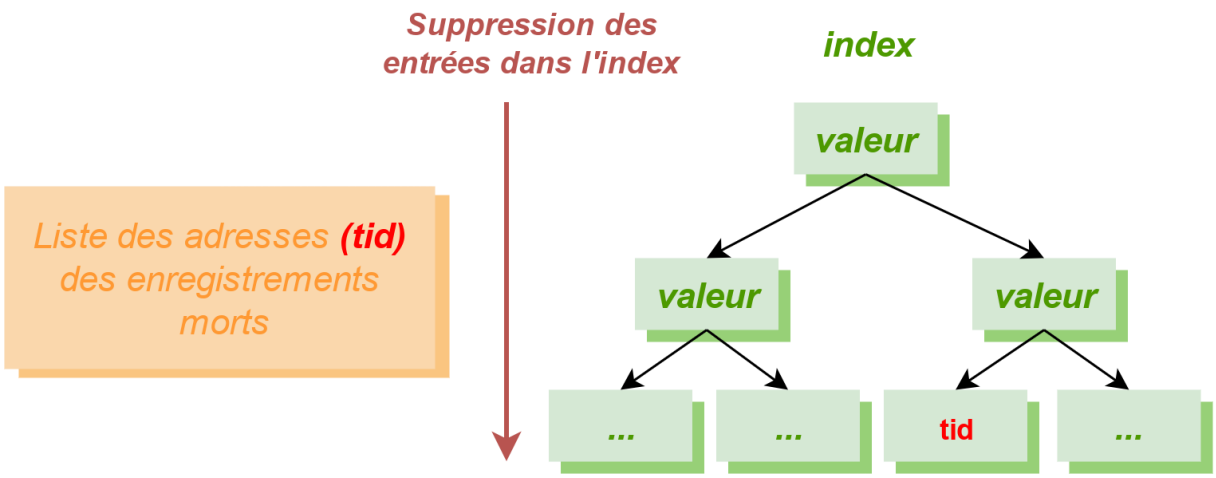


Figure 12/ .2: Phase 2/3 : nettoyage des index

La seconde passe se charge de nettoyer les entrées d'index. `VACUUM` possède une liste de `tid` (`tuple id`) à invalider. Il parcourt donc tous les index de la table à la recherche de ces `tid` et les supprime. En effet, les index sont triés afin de mettre en correspondance une valeur de clé (la colonne indexée par exemple) avec un `tid`. Il n'est par contre pas possible de trouver un `tid` directement. Les pages entièrement vides sont supprimées de l'arbre et stockées dans la liste des pages réutilisables, la *Free Space Map* (FSM).

Pour gagner du temps si le temps presse, cette phase peut être ignorée de deux manières. La première est de désactiver l'option `INDEX_CLEANUP` :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) nom_table ;
```

À partir de la version 14, un autre mécanisme, automatique cette fois, a été ajouté. Le but est toujours d'exécuter rapidement le `VACUUM`, mais uniquement pour éviter le *wraparound*. Quand la table atteint l'âge, très élevé, de 1,6 milliard de transactions (défaut des paramètres `vacuum_failsafe_age` et `vacuum_multixact_failsafe_age`), un `VACUUM` simple va automatiquement désactiver le nettoyage des index pour nettoyer plus rapidement la table et permettre d'avancer l'identifiant le plus ancien de la table.

Cette phase de nettoyage des index peut être parallélisée (clause `PARALLEL`), chaque index pouvant être traité par un CPU.

12.3.2 Fonctionnement de VACUUM (suite)

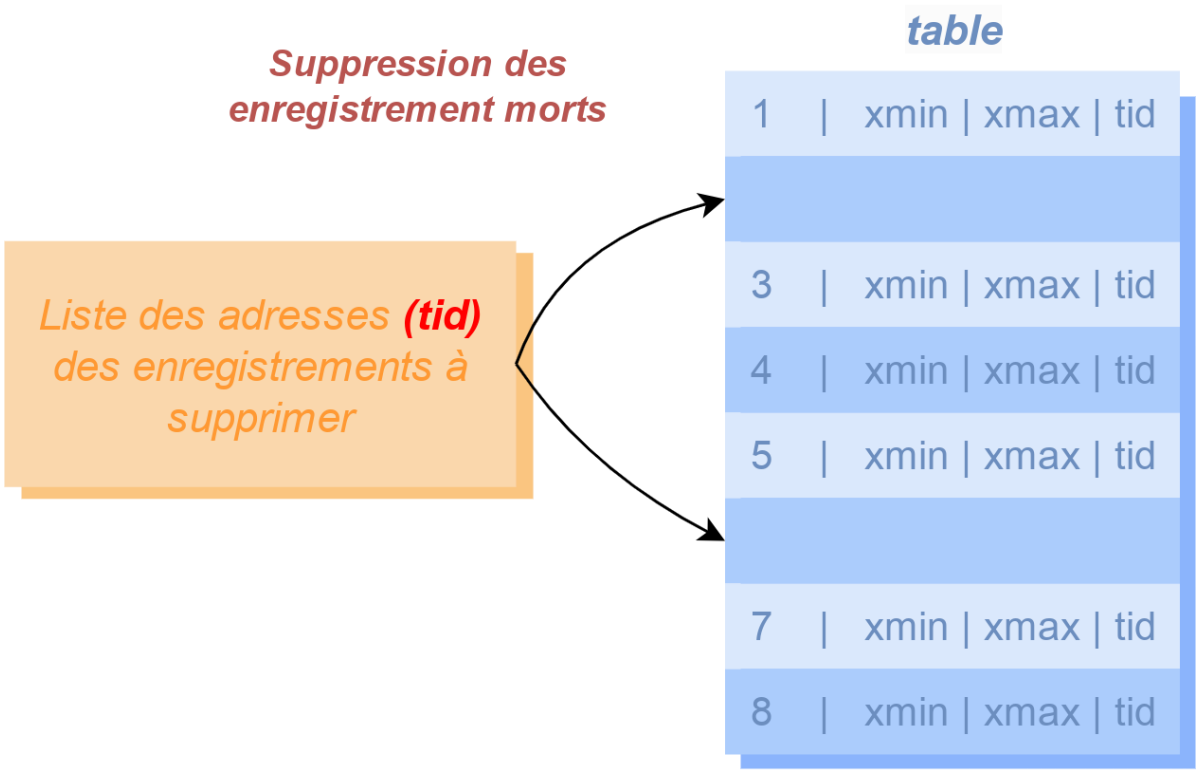


Figure 12/ .3: Phase 3/3 : suppression des enregistrements morts

NB : L'espace est rarement rendu à l'OS !

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements morts identifiés, ceux-ci peuvent disparaître. C'est le rôle de cette passe. Quand un enregistrement est supprimé d'un bloc, ce bloc est complètement réorganisé afin de consolider l'espace libre. Cet espace est renseigné dans la *Free Space Map* (FSM).

Une fois cette passe terminée, si le parcours de la table n'a pas été terminé lors de la passe précédente, le travail reprend où il en était du parcours de la table.

Si les derniers blocs de la table sont vides, ils sont rendus au système (si le verrou nécessaire peut être obtenu, et si l'option `TRUNCATE` n'est pas `off`). C'est le seul cas où `VACUUM` réduit la taille de la table. Les espaces vides (et réutilisables) au milieu de la table constituent le *bloat* (littéralement « boursouflure » ou « gonflement », que l'on peut aussi traduire par fragmentation).

Les statistiques d'activité sont aussi mises à jour.

12.4 LES OPTIONS DE VACUUM

- Quelle tâche ?
- Comment améliorer les performances ?
- Quelles options en cas d'urgence ?
- Autres options

12.4.1 Tâches d'un VACUUM

Ne pas confondre :

- `VACUUM` seul
  - nettoyage des lignes mortes, *visibility map*, *hint bits*
- `ANALYZE`
  - statistiques sur les données
- `VACUUM (ANALYZE)`
  - nettoyage & statistiques
- `VACUUM (FREEZE)`
  - gel des lignes
  - parfois gênant ou long
- `VACUUM FULL`
  - bloquant !
  - jamais lancé par l'autovacuum

VACUUM

Par défaut, `VACUUM` procède principalement au nettoyage des lignes mortes. Pour que cela soit efficace, il met à jour la *visibility map*, et la crée au besoin. Au passage, il peut geler certaines lignes rencontrées.

L'autovacuum le déclenchera sur les tables en fonction de l'activité.

Le verrou `SHARE UPDATE EXCLUSIVE` posé protège la table contre les modifications simultanées du schéma, et ne gêne généralement pas les opérations, sauf les plus intrusives (il empêche par exemple

un `LOCK TABLE`). L'autovacuum arrêtera spontanément un `VACUUM` qu'il aurait lancé et qui gênerait ; mais un `VACUUM` lancé manuellement continuera jusqu'à la fin.

VACUUM ANALYZE

`ANALYZE` existe en tant qu'ordre séparé, pour rafraîchir les statistiques sur un échantillon des données, à destination de l'optimiseur. L'autovacuum se charge également de lancer des `ANALYZE` en fonction de l'activité.

L'ordre `VACUUM ANALYZE` (ou `VACUUM (ANALYZE)`) force le calcul des statistiques sur les données en même temps que le `VACUUM`.

VACUUM FREEZE

`VACUUM FREEZE` procède au « gel » des lignes visibles par toutes les transactions en cours sur l'instance, afin de parer au problème du *wraparound* des identifiants de transaction.

Un ordre `FREEZE` n'existe pas en tant que tel.

Préventivement, lors d'un `VACUUM` simple, l'autovacuum procède au gel de certaines des lignes rencontrées. De plus, il lancera un `VACUUM FREEZE` sur une table dont les plus vieilles transactions dépassent un certain âge. Ce peut être très long, et très lourd en écritures si une grosse table doit être entièrement gelée d'un coup. Autrement, l'activité n'est qu'exceptionnellement gênée (voir plus bas).

L'opération de gel sera détaillée plus loin.

VACUUM FULL

L'ordre `VACUUM FULL` permet de reconstruire la table sans les espaces vides. C'est une opération très lourde, risquant de bloquer d'autres requêtes à cause du verrou exclusif qu'elle pose (on ne peut même plus lire la table !), mais il s'agit de la seule option qui permet de réduire la taille de la table au niveau du système de fichiers de façon certaine.

Il faut prévoir l'espace disque (la table est reconstruite à côté de l'ancienne, puis l'ancienne est supprimée). Les index sont reconstruits au passage. Un `VACUUM FULL` gèle agressivement les lignes, et effectue donc au passage l'équivalent d'un `FREEZE`.

L'autovacuum ne lancera jamais un `VACUUM FULL` !

Il existe aussi un ordre `CLUSTER`, qui permet en plus de trier la table suivant un des index.

12.4.2 Options de performance de VACUUM

- Index :
  - `PARALLEL`
- Taille du *buffer ring* (v16+)
  - `VACUUM (BUFFER_USAGE_LIMIT 2MB)`
  - paramètre `vacuum_buffer_usage_limit`
  - 256 ko ou 2 Mo par défaut, à monter
- `SKIP_DATABASE_STATS`, `ONLY_DATABASE_STATS` (v16+)
- Éviter les verrous
  - `SKIP_LOCKED`
  - `SET lock_timeout = '1s'`

PARALLEL :

L'option `PARALLEL` permet le traitement parallélisé des index. Le nombre indiqué après `PARALLEL` précise le niveau de parallélisation souhaité. Par exemple :

```
VACUUM (VERBOSE, PARALLEL 4) matable ;
```

```
INFO:  vacuuming "public.matable"
INFO:  launched 3 parallel vacuum workers for index cleanup (planned: 3)
```

SKIP_DATABASE_STATS, ONLY_DATABASE_STATS :

En fin d'exécution d'un `VACUUM`, même sur une seule table, le champ `pg_database.datfrozenxid` est mis à jour. Il contient le numéro de la transaction la plus ancienne non encore gélée dans toute la base de données. Cette opération impose de parcourir `pg_class` pour récupérer le plus ancien des numéros de transaction de chaque table (`relfrozenxid`), Or cette mise à jour n'est utile que pour l'autovacuum et le `VACUUM FREEZE`, et a rarement un caractère d'urgence.

Depuis la version 16, l'option `SKIP_DATABASE_STATS` demande au `VACUUM` d'ignorer la mise à jour de l'identifiant de transaction. Le principe est d'activer cette option pour les nettoyages en masse. À l'inverse, l'option `ONLY_DATABASE_STATS` demande de ne faire que la mise à jour du `datfrozenxid`, ce qui peut être fait une seule fois en fin de traitement.

L'outil `vacuumdb` procède ainsi automatiquement si le serveur est en version 16 minimum. Par exemple :

```
$ vacuumdb --echo --all
```

```
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « pgbench »
RESET search_path;
SELECT c.relname, ns.nspname FROM pg_catalog.pg_class c
 JOIN pg_catalog.pg_namespace ns ON c.relnamespace OPERATOR(pg_catalog.=) ns.oid
 LEFT JOIN pg_catalog.pg_class t ON c.reltoastrelid OPERATOR(pg_catalog.=) t.oid
 WHERE c.relkind OPERATOR(pg_catalog.=) ANY (array['r', 'm'])
 ORDER BY c.relpages DESC;
SELECT pg_catalog.set_config('search_path', '', false);
VACUUM (SKIP_DATABASE_STATS) public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_statistic;
…
…
VACUUM (ONLY_DATABASE_STATS);
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « postgres »
…
…
VACUUM (ONLY_DATABASE_STATS);
```

BUFFER_USAGE_LIMIT :

Cette option apparue en version 16 permet d'augmenter la taille de la mémoire partagée que peuvent utiliser `VACUUM`, et `ANALYZE`. Par défaut, cet espace nommé *buffer ring* n'est que de 256 ko de mémoire partagée, une valeur assez basse, cela pour protéger le cache (*shared buffers*). Si cette mémoire ne suffit pas, PostgreSQL doit recycler certains de ces buffers, d'où une écriture possible de journaux sur disque, avec un ralentissement à la clé.

Monter la taille du *buffer ring* avec `BUFFER_USAGE_LIMIT` permet une exécution plus rapide de VACUUM, et génére moins de journaux. De manière globale, on peut aussi modifier le paramètre `vacuum_buffer_usage_limit`. Les valeurs vont de 128 ko à 16 Go ; 0 désactive le *buffer ring*, il n'y a alors aucune limite en terme d'utilisation du cache. La taille par défaut n'est que de 2 Mo (et même seulement 256 ko jusque PostgreSQL 16 inclus).

Les machines modernes permettent de monter sans problème ce paramètre à quelques mégaoctets pour un gain en vitesse d'écriture très appréciable[2]. Il peut y avoir un impact négatif sur les autres requêtes si le débit en lecture ou écriture du `VACUUM` augmente trop. La valeur 0 est envisageable dans le cas d'une plage de maintenance où une purge du cache de PostgreSQL n'aurait pas de gros impact.

Exemples d'utilisation :

```
ANALYZE (BUFFER_USAGE_LIMIT '8MB');

VACUUM (BUFFER_USAGE_LIMIT '8MB');

VACUUM (ANALYZE, BUFFER_USAGE_LIMIT 0) ;  # sans limite

vacuumdb --analyze --buffer-usage-limit=8MB --echo -d pgbench
```

[2] https://blog.dalibo.com/2024/03/26/strategies-acces.html

```
…
VACUUM (SKIP_DATABASE_STATS, ANALYZE, BUFFER_USAGE_LIMIT '8MB')
       public.pgbench_accounts;
…
```

Les verrous, SKIP_LOCKED et lock_timeout :

L'option `SKIP_LOCKED` permet d'ignorer toute table pour laquelle la commande `VACUUM` ne peut pas obtenir immédiatement son verrou. Cela évite de bloquer le `VACUUM` sur une table, et permet d'éviter un empilement des verrous derrière celui que le `VACUUM` veut poser, surtout en cas de `VACUUM FULL`. La commande passe alors à la table suivante à traiter. Exemple :

```
VACUUM (FULL, SKIP_LOCKED) t_un_million_int, t_cent_mille_int ;
```

```
WARNING:  skipping vacuum of "t_un_million_int" --- lock not available
VACUUM
```

Une technique un peu différente est de paramétrer dans la session un petit délai avant abandon en cas de problème de verrou. Là encore, cela vise à limiter les empilements de verrou sur une base active. Par contre, comme l'ordre tombe immédiatement en erreur après le délai, il est plus adapté aux ordres ponctuels sur une table.

```
SET lock_timeout TO '100ms' ;
-- Un LOCK TABLE a été fait dans une autre session
VACUUM (verbose) pgbench_history,pgbench_tellers;
```

```
ERROR:  canceling statement due to lock timeout
Durée : 1000,373 ms (00:01,000)
RESET lock_timeout ;
```

12.4.3 Options pour un VACUUM en urgence

```
VACUUM (SKIP_DATABASE_STATS,    /* PG 16+ */
        INDEX_CLEANUP off,
        PROCESS_TOAST off,      /* PG 14+ */
        TRUNCATE      off,
        BUFFER_USAGE_LIMIT '1GB'  /* voire 0 (PG 16+) */
        ) ;
VACUUM (ONLY_DATABASE_STATS);   /* PG 16+ */
```

Ces options sont surtout destinées à désactiver certaines étapes d'un `VACUUM` quand le temps presse vraiment.

INDEX_CLEANUP :

L'option `INDEX_CLEANUP` (par défaut à `on` jusque PostgreSQL 13 compris) déclenche systématiquement le nettoyage des index. La commande `VACUUM` va supprimer les enregistrements de l'index qui

pointent vers des lignes mortes de la table. Quand il faut nettoyer des lignes mortes urgemment dans une grosse table, la valeur `off` fait gagner beaucoup de temps :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) unetable ;
```

Les index peuvent être nettoyés plus tard lors d'un autre `VACUUM`, ou reconstruits (`REINDEX`).

Cette option existe aussi sous la forme d'un paramètre de stockage (`vacuum_index_cleanup`) propre à la table pour que l'autovacuum en tienne aussi compte.

En version 14, le nouveau défaut est `auto`, qui indique que PostgreSQL décide de faire ou non le nettoyage des index suivant la quantité d'entrées à nettoyer. Il faut au minimum 2 % d'éléments à nettoyer pour que le nettoyage ait lieu.

PROCESS_TOAST :

À partir de la version 14, cette option active ou non le traitement de la partie TOAST associée à la table (parfois la partie la plus volumineuse d'une table). Elle est activée par défaut. Son utilité est la même que pour `INDEX_CLEANUP`.

```
VACUUM (VERBOSE, PROCESS_TOAST off) unetable ;
```

TRUNCATE :

L'option `TRUNCATE` (à `on` par défaut) permet de tronquer les derniers blocs vides d'une table. `TRUNCATE off` évite d'avoir à poser un verrou exclusif certes court, mais parfois gênant.

Cette option existe aussi sous la forme d'un paramètre de stockage de table (`vacuum_truncate`).

BUFFER_USAGE_LIMIT :

Le *ring buffer* du `VACUUM` étant par défaut très réduit, une augmentation, même modeste, accélère les écritures. À la limite, s'il n'y a pas d'autre activité, on peut lui octroyer tout le cache de PostgreSQL (valeur `0`).

12.4.4 Autres options de VACUUM

- `VERBOSE`
- Ponctuellement :
  - `DISABLE_PAGE_SKIPPING`

VERBOSE :

Cette option affiche un grand nombre d'informations sur ce que fait la commande. En général, c'est une bonne idée de l'activer :

```
VACUUM (VERBOSE) pgbench_accounts_5 ;
```

```
INFO:  vacuuming "public.pgbench_accounts_5"
INFO:  scanned index "pgbench_accounts_5_pkey" to remove 9999999 row versions
DÉTAIL : CPU: user: 12.16 s, system: 0.87 s, elapsed: 18.15 s
INFO:  "pgbench_accounts_5": removed 9999999 row versions in 163935 pages
DÉTAIL : CPU: user: 0.16 s, system: 0.00 s, elapsed: 0.20 s
INFO:  index "pgbench_accounts_5_pkey" now contains 100000000 row versions in 301613
 ↪  pages
DÉTAIL : 9999999 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "pgbench_accounts_5": found 10000001 removable,
       10000051 nonremovable row versions in 327870 out of 1803279 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 1071186825
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 1475409 frozen pages.
0 pages are entirely empty.
CPU: user: 13.77 s, system: 0.89 s, elapsed: 19.81 s.
VACUUM
```

DISABLE_PAGE_SKIPPING :

Par défaut, PostgreSQL ne traite que les blocs modifiés depuis le dernier `VACUUM`, ce qui est un gros gain en performance (l'information est stockée dans la *Visibility Map*, qui est généralement un tout petit fichier).

Activer l'option `DISABLE_PAGE_SKIPPING` force l'analyse de tous les blocs de la table. La table est intégralement reparcourue. Ce peut être utile en cas de problème, notamment pour reconstruire cette *Visibility Map*.

Mélange des options :

Il est possible de mélanger toutes ces options presque à volonté, et de préciser plusieurs tables à nettoyer :

```
VACUUM (VERBOSE, ANALYZE, INDEX_CLEANUP off, TRUNCATE off,
        DISABLE_PAGE_SKIPPING) bigtable, smalltable ;
```

12.5 SUIVI DU VACUUM

- `pg_stat_activity` ou `top`
- La table est-elle suffisamment nettoyée ?
- Vue `pg_stat_user_tables`
  - `last_vacuum` / `last_autovacuum`
  - `last_analyze` / `last_autoanalyze`
- `log_autovacuum_min_duration`

Un `VACUUM`, y compris lancé par l'autovacuum, apparaît dans `pg_stat_activity` et le processus est visible comme processus système avec `top` ou `ps` :

```
$ ps faux
…
postgres 3470724 0.0 0.0 12985308 6544  ? Ss 13:58 0:02 \_ postgres: 13/main:
 ↪  autovacuum launcher
postgres  795432 7.8 0.0 14034140 13424 ? Rs 16:22 0:01 \_ postgres: 13/main:
 ↪  autovacuum worker
                                                                    pgbench1000p10
…
```

Il est fréquent de se demander si l'autovacuum s'occupe suffisamment d'une table qui grossit ou dont les statistiques semblent périmées. La vue `pg_stat_user_tables` contient quelques informations. Dans l'exemple ci-dessous, nous distinguons les dates des `VACUUM` et `ANALYZE` déclenchés automatiquement ou manuellement (en fait par l'application `pgbench`). Si 44 305 lignes ont été modifiées depuis le rafraîchissement des statistiques, il reste 2,3 millions de lignes mortes à nettoyer (contre 10 millions vivantes).

```
# SELECT * FROM pg_stat_user_tables WHERE relname ='pgbench_accounts' \gx

-[ RECORD 1 ]-------+-----------------------------
relid               | 489050
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 1
seq_tup_read        | 10
idx_scan            | 686140
idx_tup_fetch       | 2686136
n_tup_ins           | 0
n_tup_upd           | 2343090
n_tup_del           | 452
n_tup_hot_upd       | 118551
n_live_tup          | 10044489
n_dead_tup          | 2289437
```

```
n_mod_since_analyze | 44305
n_ins_since_vacuum  | 452
last_vacuum         | 2020-01-06 18:42:50.237146+01
last_autovacuum     | 2020-01-07 14:30:30.200728+01
last_analyze        | 2020-01-06 18:42:50.504248+01
last_autoanalyze    | 2020-01-07 14:30:39.839482+01
vacuum_count        | 1
autovacuum_count    | 1
analyze_count       | 1
autoanalyze_count   | 1
```

Activer le paramètre `log_autovacuum_min_duration` avec une valeur relativement faible (dépendant des tables visées de la taille des logs générés), voire le mettre à 0, est également courant et conseillé.

12.5.1 Progression du VACUUM

- Pour `VACUUM` simple / `VACUUM FREEZE`
  - vue `pg_stat_progress_vacuum`
  - blocs parcourus / nettoyés
  - nombre de passes dans l'index
- Partie `ANALYZE`
  - `pg_stat_progress_analyze`
- Manuel ou via autovacuum
- Pour `VACUUM FULL`
  - vue `pg_stat_progress_cluster`

La vue `pg_stat_progress_vacuum` contient une ligne par `VACUUM` (simple ou `FREEZE`) en cours d'exécution :

```
TABLE pg_stat_progress_vacuum \gx
```

```
-[ RECORD 1 ]------+--------------
pid                | 2603780
datid              | 1308955
datname            | grossetable
relid              | 1308962
phase              | scanning heap
heap_blks_total    | 163935
heap_blks_scanned  | 3631
```

```
heap_blks_vacuumed   | 0
index_vacuum_count   | 0
max_dead_tuple_bytes | 67108864
dead_tuple_bytes     | 0
num_dead_item_ids    | 0
indexes_total        | 0
indexes_processed    | 0
```

Dans cet exemple, le `VACUUM` exécuté par le PID 4299 a parcouru 86 665 blocs (soit 68 % de la table), et en a traité 86 664.

À noter que le suivi du `VACUUM` dans les index (les deux derniers champs) nécessite au moins PostgreSQL 17. C'est souvent la partie la plus longue d'un `VACUUM`.

Dans le cas d'un `VACUUM ANALYZE`, la seconde partie de recueil des statistiques pourra être suivie dans `pg_stat_progress_analyze` :

```
SELECT * FROM pg_stat_progress_analyze ;
```

```
-[ RECORD 1 ]-------------+--------------------------------
pid                       | 1938258
datid                     | 748619
datname                   | grossetable
relid                     | 748698
phase                     | acquiring inherited sample rows
sample_blks_total         | 1875
sample_blks_scanned       | 1418
ext_stats_total           | 0
ext_stats_computed        | 0
child_tables_total        | 16
child_tables_done         | 6
current_child_table_relid | 748751
```

Les vues précédentes affichent aussi bien les opérations lancées manuellement que celles décidées par l'autovacuum.

Par contre, pour un `VACUUM FULL`, il faudra suivre la progression au travers de la vue `pg_stat_progress_cluster`. Cette vue est utilisable aussi avec l'ordre `CLUSTER`, d'où le nom. Exemple :

```
SELECT * FROM pg_stat_progress_cluster \gx
```

```
-[ RECORD 1 ]-------+------------------
pid                 | 21157
datid               | 13444
datname             | postgres
relid               | 16384
command             | VACUUM FULL
phase               | seq scanning heap
cluster_index_relid | 0
heap_tuples_scanned | 13749388
heap_tuples_written | 13749388
heap_blks_total     | 199105
heap_blks_scanned   | 60839
index_rebuild_count | 0
```

Ces vues n'affichent que les opérations en cours, elles n'historisent rien. Si aucun `VACUUM` n'est en cours, elles n'afficheront rien.

12.5.2 Droit de lancer un VACUUM

- Propriétaire
- Superutilisateur
- Inclus dans droit de maintenance (v17)

```
GRANT MAINTAIN ON matable TO dba ; -- granulaire
GRANT pg_maintain         TO dba ; -- global
```

`VACUUM` est une opération de maintenance ne pouvant être effectuée que par :

- le propriétaire de la table ;
- un superutilisateur ;
- à partir de PostgreSQL 17, un utilisateur possédant le droit de maintenir la table, soit via le rôle `pg_maintain` ou par un `GRANT MAINTAIN` sur la table. (Ce droit de maintenance inclut d'autres droits, comme les droits de `ANALYZE`, `CLUSTER`, `LOCK TABLE`, `REFRESH MATERIALIZED VIEW` et `REINDEX`, mais pas celui de lire les données.)

Sans droit de maintenance, le `VACUUM` ne fonctionne pas pour un utilisateur non propriétaire de la table :

```
$ vacuumdb --username maintenance --table t1
vacuumdb: vacuuming database "b3"
WARNING:  permission denied to vacuum "t1", skipping it
```

Si **maintenance** devient membre du rôle `pg_maintain`, tout fonctionne :

```
$ psql --username postgres --command 'GRANT pg_maintain TO maintenance'
GRANT ROLE

$ vacuumdb --username maintenance
vacuumdb: vacuuming database "b3"
```

Avant la version 17, il est toujours possible d'avoir un groupe propriétaire de l'objet et d'ajouter un rôle de maintenance comme membre de ce groupe. Il aura le droit de lire les données. Mais si jamais un objet est créé par quelqu'un sans transférer la propriété au groupe, le rôle de maintenance ne peut faire d'opération de maintenance sur cet objet.

12.6 AUTOVACUUM

- Processus autovacuum
- But : ne plus s'occuper de `VACUUM`
- Suit l'activité
- Seuil dépassé => worker dédié
- Gère : `VACUUM`, `ANALYZE`, `FREEZE`
  - mais pas `FULL`

Le principe est le suivant :

Le démon `autovacuum launcher` s'occupe de lancer des *workers* régulièrement sur les différentes bases. Ce nouveau processus inspecte les statistiques sur les tables (vue `pg_stat_all_tables`) : nombres de lignes insérées, modifiées et supprimées. Quand certains seuils sont dépassés sur un objet, le *worker* effectue un `VACUUM`, un `ANALYZE`, voire un `VACUUM FREEZE` (mais jamais, rappelons-le, un `VACUUM FULL`).

Le nombre de ces *workers* est limité, afin de ne pas engendrer de charge trop élevée.

12.6.1 Paramétrage du déclenchement de l'autovacuum

- `autovacuum` (on !)
- `autovacuum_naptime` (1 min)
- `autovacuum_max_workers` (3)
  - plusieurs *workers* simultanés sur une base
  - un seul par table

`autovacuum` (`on` par défaut) détermine si l'autovacuum doit être activé.

Il est fortement conseillé de laisser `autovacuum` à `on` !

S'il le faut vraiment, il est possible de désactiver l'autovacuum sur une table précise :

```
ALTER TABLE nom_table SET (autovacuum_enabled = off);
```

mais cela est très rare. La valeur `off` n'empêche pas le déclenchement d'un `VACUUM FREEZE` s'il devient nécessaire.

`autovacuum_naptime` est le temps d'attente entre deux périodes de vérification sur la même base (1 minute par défaut). Le déclenchement de l'autovacuum suite à des modifications de tables n'est donc pas instantané.

`autovacuum_max_workers` est le nombre maximum de *workers* que l'autovacuum pourra déclencher simultanément, chacun s'occupant d'une table (ou partition de table). Chaque table ne peut être traitée simultanément que par un unique *worker*. La valeur par défaut (3) est généralement suffisante. Néanmoins, s'il y a fréquemment trois *autovacuum workers* travaillant en même temps, et surtout si cela dure, il peut être nécessaire d'augmenter ce paramètre. Cela est fréquent quand il y a de nombreuses petites tables. Noter qu'il faudra sans doute augmenter certaines ressources allouées au nettoyage (paramètre `autovacuum_vacuum_cost_limit`, voir plus bas), car les *workers* se les partagent ; sauf `maintenance_work_mem` qui, lui, est une quantité de RAM utilisable par chaque *worker* indépendamment.

12.6.2 Déclenchement de l'autovacuum

Seuil de déclenchement =
threshold + *scale factor* × nb lignes de la table

L'autovacuum déclenche un `VACUUM` ou un `ANALYZE` à partir de seuils calculés sur le principe d'un nombre de lignes minimal (*threshold*) et d'une proportion de la table existante (*scale factor*) de lignes modifiées, insérées ou effacées. (Pour les détails précis sur ce qui suit, voir la documentation officielle[3].)

Ces seuils pourront être adaptés table par table.

[3] https://docs.postgresql.fr/current/routine-vacuuming.html#AUTOVACUUM

12.6.3 Déclenchement de l'autovacuum (suite)

- Pour `VACUUM`
  - `autovacuum_vacuum_scale_factor` (20 %)
  - `autovacuum_vacuum_threshold` (50)
  - `autovacuum_vacuum_insert_threshold` (1000)
  - `autovacuum_vacuum_insert_scale_factor` (20 %)
- Pour `ANALYZE`
  - `autovacuum_analyze_scale_factor` (10 %)
  - `autovacuum_analyze_threshold` (50)
- Adapter pour une grosse table :

```
ALTER TABLE table_name SET (autovacuum_vacuum_scale_factor = 0.1);
```

Pour le `VACUUM`, si on considère les enregistrements morts (supprimés ou anciennes versions de lignes), la condition de déclenchement est :

```
nb_enregistrements_morts (pg_stat_all_tables.n_dead_tup) >=
    autovacuum_vacuum_threshold
  + autovacuum_vacuum_scale_factor × nb_enregs (pg_class.reltuples)
```

où, par défaut :

- `autovacuum_vacuum_threshold` vaut 50 lignes ;
- `autovacuum_vacuum_scale_factor` vaut 0,2 soit 20 % de la table.

Donc, par exemple, dans une table d'un million de lignes, modifier 200 050 lignes provoquera le passage d'un `VACUUM`.

Pour les grosses tables avec de l'historique, modifier 20 % de la volumétrie peut être extrêmement long. Quand l'`autovacuum` lance enfin un `VACUUM`, celui-ci a donc beaucoup de travail et peut durer longtemps et générer beaucoup d'écritures. Il est donc fréquent de descendre la valeur de `autovacuum_vacuum_scale_factor` à quelques pour cent sur les grosses tables. (Une alternative est de monter `autovacuum_vacuum_threshold` à un nombre de lignes élevé et de descendre `autovacuum_vacuum_scale_factor` à 0, mais il faut alors calculer le nombre de lignes qui déclenchera le nettoyage, et cela dépend fortement de la table et de sa fréquence de mise à jour.)

S'il faut modifier un paramètre, il est préférable de ne pas le faire au niveau global mais de cibler les tables où cela est nécessaire. Par exemple, l'ordre suivant réduit à 5 % de la table le nombre de lignes

à modifier avant que l'`autovacuum` y lance un `VACUUM` :

```
ALTER TABLE nom_table SET (autovacuum_vacuum_scale_factor = 0.05);
```

À partir de PostgreSQL 13, le `VACUUM` est aussi lancé quand il n'y a que des insertions, avec deux nouveaux paramètres et un autre seuil de déclenchement :

```
nb_enregistrements_insérés (pg_stat_all_tables.n_ins_since_vacuum) >=
    autovacuum_vacuum_insert_threshold
  + autovacuum_vacuum_insert_scale_factor × nb_enregs (pg_class.reltuples)
```

Pour l'`ANALYZE`, le principe est le même. Il n'y a que deux paramètres, qui prennent en compte toutes les lignes modifiées *ou* insérées, pour calculer le seuil :

```
nb_insert + nb_updates + nb_delete (n_mod_since_analyze) >=
    autovacuum_analyze_threshold + nb_enregs × autovacuum_analyze_scale_factor
```

où, par défaut :

- `autovacuum_analyze_threshold` vaut 50 lignes ;
- `autovacuum_analyze_scale_factor` vaut 0,1, soit 10 %.

Dans notre exemple d'une table, modifier 100 050 lignes provoquera le passage d'un `ANALYZE`.

Là encore, il est fréquent de modifier les paramètres sur les grosses tables pour rafraîchir les statistiques plus fréquemment.

Les insertions ont toujours été prises en compte pour `ANALYZE`, puisqu'elles modifient le contenu de la table. (Par contre, jusque PostgreSQL 12 inclus, `VACUUM` ne tenait pas compte des lignes insérées pour déclencher son nettoyage. Or, cela avait des conséquences pour les tables à insertion seule : gel de lignes retardé, *Index Only Scan* impossibles… Pour cette raison, à partir de la version 13, les insertions sont aussi prises en compte pour déclencher un `VACUUM`.)

12.7 PARAMÉTRAGE DE VACUUM & AUTOVACUUM

- VACUUM vs autovacuum
- Mémoire
- Gestion des coûts
- Gel des lignes

En fonction de la tâche exacte, de l'agressivité acceptable ou de l'urgence, plusieurs paramètres peuvent être mis en place.

Ces paramètres peuvent différer (par le nom ou la valeur) selon qu'ils s'appliquent à un `VACUUM` lancé manuellement ou par script, ou à un processus lancé par l'autovacuum.

12.7.1 VACUUM vs autovacuum

VACUUM manuel	autovacuum
Urgent	Arrière-plan
Pas de limite	Peu agressif
Paramètres	Les mêmes + paramètres de surcharge

Quand on lance un ordre `VACUUM`, il y a souvent urgence, ou l'on est dans une période de maintenance, ou dans un batch. Les paramètres que nous allons voir ne cherchent donc pas, par défaut, à économiser des ressources.

À l'inverse, un `VACUUM` lancé par l'autovacuum ne doit pas gêner une production peut-être chargée. Il existe donc des paramètres `autovacuum_*` surchargeant les précédents, et beaucoup plus conservateurs.

12.7.2 Mémoire

- Quantité de mémoire allouable
  - `maintenance_work_mem`
  - `autovacuum_work_mem`
  - montés souvent à ½ à 1 Go
- Impact
  - `VACUUM`
  - construction d'index

`maintenance_work_mem` est la quantité de mémoire qu'un processus effectuant une opération de maintenance (c'est-à-dire n'exécutant pas des requêtes classiques comme `SELECT`, `INSERT`, `UPDATE`…) est autorisé à allouer pour sa tâche de maintenance.

Cette mémoire est utilisée lors de la construction d'index ou l'ajout de clés étrangères. et, dans le contexte de `VACUUM`, pour stocker les adresses des enregistrements pouvant être recyclés. Cette mémoire est remplie pendant la phase 1 du processus de `VACUUM`, tel qu'expliqué plus haut. Rappelons qu'une adresse d'enregistrement (`tid`, pour `tuple id`) a une taille de 6 octets et est composée du numéro dans la table, et du numéro d'enregistrement dans le bloc, par exemple `(0,1)`, `(3164,98)` ou `(5351510,42)`.

Le défaut de 64 Mo est assez faible. Si tous les enregistrements morts d'une table ne tiennent pas dans `maintenance_work_mem`, `VACUUM` est obligé de faire plusieurs passes de nettoyage, donc plusieurs parcours complets de chaque index. Une valeur assez élevée de `maintenance_work_mem` est donc conseillée : s'il est déjà possible de stocker plusieurs dizaines de millions d'enregistrements à effacer dans 256 Mo, 1 Go peut être utile lors de très grosses purges.

PostgreSQL 17 améliore beaucoup la consommation mémoire et la vitesse de nettoyage des index, et doit rendre rarissime les nettoyages d'index en plusieurs passes.

PostgreSQL 17 fait aussi disparaître une limite de 1 Go pour le nettoyage des index : `VACUUM` ne sait pas en utiliser plus jusque PostgreSQL 16. Par contre, l'indexation de grosses tables pourra toujours bénéficier d'une valeur supérieure à 1 Go.

Rappelons que plusieurs `VACUUM` ou `autovacuum` peuvent fonctionner simultanément et consommer chacun un `maintenance_work_mem` ! (Voir `autovacuum_max_workers` plus haut.)

`autovacuum_work_mem` permet de surcharger `maintenance_work_mem` spécifiquement pour l'autovacuum. Par défaut les deux sont identiques, et l'on conserve généralement cette configuration. Au besoin, `maintenance_work_mem` peut être surchargé le temps d'une session.

12.7.3 Bridage du VACUUM et de l'autovacuum

- Pauses régulières après une certaine activité
- Par bloc traité
  - `vacuum_cost_page_hit` / `_miss` / `_dirty` (1/ 10 ou 2 /20)
  - jusque total de : `vacuum_cost_limit` (200)
  - pause : `vacuum_cost_delay` (en manuel : 0 ms !)
- Surcharge pour l'autovacuum
  - `autovacuum_vacuum_cost_limit` (identique)
  - `autovacuum_vacuum_cost_delay` (2 ms)
  - => débit en écriture max : ~40 Mo/s
- Pour accélérer : augmenter la limite

Principe :

Les paramètres suivant permettent d'éviter qu'un `VACUUM` ne gêne les autres sessions en saturant le disque. Le principe est de provoquer une pause après qu'une certaine activité a été réalisée.

Paramètres de coûts :

Ces trois paramètres « quantifient » l'activité de `VACUUM`, affectant un coût arbitraire à chaque fois qu'une opération est réalisée :

- `vacuum_cost_page_hit` : coût d'accès à chaque page présente dans le cache de PostgreSQL (valeur : 1) ;
- `vacuum_cost_page_miss` : coût d'accès à chaque page hors de ce cache (valeur : 2 à partir de la PostgreSQL 14, 10 auparavant) ;
- `vacuum_cost_page_dirty` : coût de modification d'une page, et donc de son écriture (valeur : 20).

Il est déconseillé de modifier ces paramètres de coût.

Pause :

Quand le coût cumulé atteint un seuil, l'opération de nettoyage marque une pause. Elle est gouvernée par deux paramètres :

- `vacuum_cost_limit` : coût cumulé à atteindre avant de déclencher la pause (défaut : 200) ;
- `vacuum_cost_delay` : temps à attendre (défaut : 0 ms !)

En conséquence, les `VACUUM` lancés manuellement (en ligne de commande ou via `vacuumdb`) ne sont **pas** freinés par ce mécanisme et peuvent donc entraîner de fortes écritures ! Mais c'est généralement ce que l'on veut dans un batch ou en urgence, et il vaut mieux alors être le plus rapide possible.

(Pour les urgences, rappelons que l'option `INDEX_CLEANUP off` ou `PROCESS_TOAST off` permettent aussi d'ignorer le nettoyage des index ou des TOAST.)

Paramétrage pour le VACUUM manuel :

Il est conseillé de ne pas toucher au paramétrage par défaut de `vacuum_cost_limit` et `vacuum_cost_delay`.

Si on doit lancer un `VACUUM` manuellement en limitant son débit, procéder comme suit dans une session :

```
-- Reprise pour le VACUUM du paramétrage d'autovacuum
SET vacuum_cost_limit = 200 ;
SET vacuum_cost_delay = '2ms' ;
VACUUM (VERBOSE) matable ;
```

Avec `vacuumdb`, il faudra passer par la variable d'environnement `PGOPTIONS`.

Paramétrage pour l'autovacuum :

Les `VACUUM` d'autovacuum, eux, sont par défaut limités en débit pour ne pas gêner l'activité normale de l'instance. Deux paramètres surchargent les précédents :

- `autovacuum_cost_limit` vaut par défaut -1, donc reprend la valeur 200 de `vacuum_cost_limit` ;
- `autovacuum_vacuum_cost_delay` vaut par défaut 2 ms.

Un (`autovacuum_`) `vacuum_cost_limit` à 200 consiste à traiter au plus 200 blocs lus en cache (car `vacuum_cost_page_hit` = 1), soit 1,6 Mo, avant de faire la pause de 2 ms. Si ces blocs doivent être écrits, on descend en-dessous de 10 blocs traités avant chaque pause (`vacuum_cost_page_dirty` = 20) avant la pause de 2 ms, d'où un débit en écriture maximal de l'autovacuum de 40 Mo/s Cela s'observe aisément par exemple avec `iotop`.

Ce débit est partagé équitablement entre les différents *workers* lancés par l'autovacuum (sauf paramétrage spécifique au niveau de la table).

Pour rendre l'`autovacuum` plus rapide, il est préférable d'augmenter `autovacuum_vacuum_cost_limit` au-delà de 200, plutôt que de réduire `autovacuum_vacuum_cost_delay` qui n'est qu'à 2 ms, pour ne pas monopoliser le disque. (Exception : les versions antérieures à la 12, car `autovacuum_vacuum_cost_delay` valait alors 20 ms et le débit en écriture saturait à 4 Mo/s ! La valeur 2 ms tient mieux compte des disques actuels.).

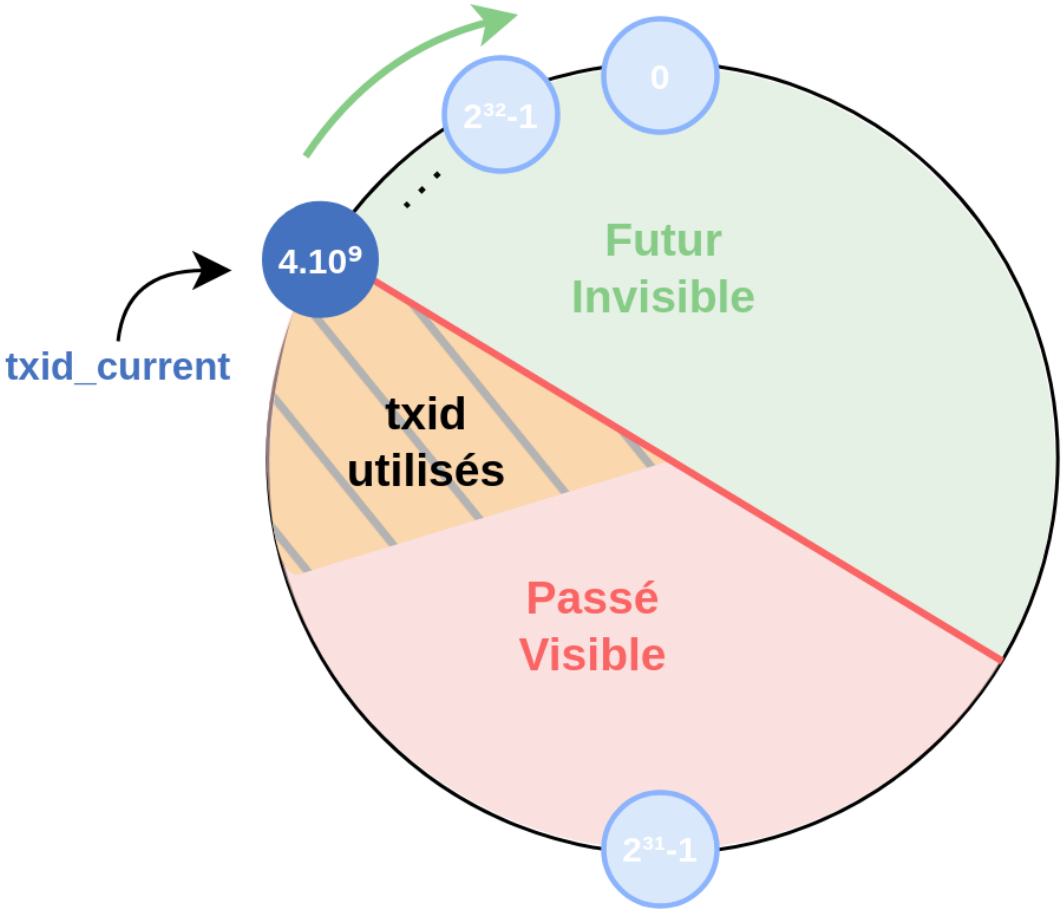
La prise en compte de la nouvelle valeur de la limite par les *workers* en cours sur les tables est automatique à partir de PostgreSQL 16. Dans les versions précédentes, il faut arrêter les *workers* en cours (avec `pg_cancel_backend()`) et attendre que l'`autovacuum` les relance. Quand `autovacuum_max_workers` est augmenté, prévoir aussi d'augmenter la limite. Il faut pouvoir assumer le débit supplémentaire pour les disques.

Sur le sujet, voir la conférence de Robert Haas à PGconf.EU 2023 à Prague[4].

[4] https://www.postgresql.eu/events/pgconfeu2023/sessions/session/4847/slides/432/Understanding%20and%20Fixing%20Autovacuum%20-%20PGCONF.EU%202023.pdf#page=14

12.7.4 Paramétrage du FREEZE (1)

Le but est de geler les numéros de transaction assez vite :



Rappelons que les numéros de transaction stockés sur les lignes ne sont stockés que sur 32 bits, et sont recyclés. Il y a donc un risque de mélanger l'avenir et le futur des transactions lors du rebouclage (*wraparound*). Afin d'éviter ce phénomène, l'opération `VACUUM FREEZE` « gèle » les vieux enregistrements, afin que ceux-ci ne se retrouvent pas brusquement dans le futur.

Concrètement, il s'agit de positionner un *hint bit* dans les entêtes des lignes concernées, indiquant qu'elle est plus vieille que tous les numéros de transactions actuellement actifs. (Avant la 9.4, la colonne système `xmin` était simplement remplacée par un `FrozenXid`, ce qui revient au même).

12.7.5 Paramétrage du FREEZE (2)

Quand le `VACUUM` gèle-t-il les lignes ?

- « Âge » d'une table : `age ( pgclass.relfrozenxid )`
  - Les blocs nettoyés/gelés sont notés dans la *visibility map*
- `vacuum_freeze_min_age` (50 Mtrx)
  - âge des lignes *rencontrées* à geler
- `vacuum_freeze_table_age` (150 Mtrx)
  - agressif (toute la table)
- Au plus tard, par l'autovacuum sur toute la table :
  - `autovacuum_freeze_max_age` (200 Mtrx)
- Attention après un import massif/migration logique !
  - `VACUUM FREEZE` préventif en période de maintenance

Geler une ligne ancienne implique de réécrire le bloc et donc des écritures dans les journaux de transactions et les fichiers de données. Il est inutile de geler trop tôt une ligne récente, qui sera peut-être bientôt réécrite.

Paramétrage :

Plusieurs paramètres règlent ce fonctionnement. Leurs valeurs par défaut sont satisfaisantes pour la plupart des installations et ne sont pour ainsi dire jamais modifiées. Par contre, il est important de bien connaître le fonctionnement pour ne pas être surpris.

Le numéro de transaction le plus ancien connu au sein d'une table est porté par `pgclass.relfrozenxid`, et est sur 32 bits. Il faut utiliser la fonction `age()` pour connaître l'écart par rapport au numéro de transaction courant (géré sur 64 bits en interne).

```
SELECT relname, relfrozenxid, round(age(relfrozenxid) /1e6,2) AS "age_Mtrx"
FROM pg_class c
WHERE relname LIKE 'pgbench%' AND relkind='r'
ORDER BY age(relfrozenxid) ;
```

```
       relname        | relfrozenxid | age_Mtrx
----------------------+--------------+----------
 pgbench_accounts_7   |    882324041 |     0.00
 pgbench_accounts_8   |    882324041 |     0.00
 pgbench_accounts_2   |    882324041 |     0.00
 pgbench_history      |    882324040 |     0.00
 pgbench_accounts_5   |    848990708 |    33.33
```

```
 pgbench_tellers      |    832324041 |    50.00
 pgbench_accounts_3   |    719860155 |   162.46
 pgbench_accounts_9   |    719860155 |   162.46
 pgbench_accounts_4   |    719860155 |   162.46
 pgbench_accounts_6   |    719860155 |   162.46
 pgbench_accounts_1   |    719860155 |   162.46
 pgbench_branches     |    719860155 |   162.46
 pgbench_accounts_10  |    719860155 |   162.46
```

Une partie du gel se fait lors d'un `VACUUM` normal. Si ce dernier rencontre un enregistrement plus vieux que `vacuum_freeze_min_age` (par défaut 50 millions de transactions écoulées), alors le *tuple* peut et doit être gelé. Cela ne concerne que les lignes dans des blocs qui ont des lignes mortes à nettoyer : les lignes dans des blocs un peu statiques y échappent. (Y échappent aussi les lignes qui ne sont pas forcément visibles par toutes les transactions ouvertes.)

`VACUUM` doit donc périodiquement déclencher un nettoyage plus agressif de toute la table (et non pas uniquement des blocs modifiés depuis le dernier `VACUUM`), afin de nettoyer tous les vieux enregistrements. C'est le rôle de `vacuum_freeze_table_age` (par défaut 150 millions de transactions). Si la table a atteint cet âge, un `VACUUM` (manuel ou automatique) lancé dessus deviendra « agressif » :

```
VACUUM (VERBOSE) pgbench_tellers ;
INFO:  aggressively vacuuming "public.pgbench_tellers"
```

C'est équivalent à l'option `DISABLE_PAGE_SKIPPING` : les blocs ne contenant que des lignes vivantes seront tout de même parcourus. Les lignes non gelées qui s'y trouvent et plus vieilles que `vacuum_freeze_min_age` seront alors gelées. Ce peut être long, ou pas, en fonction de l'efficacité de l'étape précédente.

À côté des numéros de transaction habituels, les identifiants `multixact`, utilisés pour supporter le verrouillage de lignes par des transactions multiples évitent aussi le *wraparound* avec des paramètres spécifiques (`vacuum_multixact_freeze_min_age`, `vacuum_multixact_freeze_table_age`) qui ont les mêmes valeurs que leurs homologues.

Enfin, il faut traiter le cas de tables sur lesquelles un `VACUUM` complet ne s'est pas déclenché depuis très longtemps. L'autovacuum y veille : `autovacuum_freeze_max_age` (par défaut 200 millions de transactions) est l'âge maximum que doit avoir une table. S'il est dépassé, un `VACUUM` agressif est automatiquement lancé sur cette table. Il est visible dans `pg_stat_activity` avec la mention caractéristique *to prevent wraparound* :

```
autovacuum: VACUUM public.pgbench_accounts (to prevent wraparound)
```

Ce traitement est lancé même si `autovacuum` est désactivé (c'est-à-dire à `off`).

En fait, un `VACUUM FREEZE` lancé manuellement équivaut à un `VACUUM` avec les paramètres `vacuum_freeze_table_age` (âge minimal de la table) et `vacuum_freeze_min_age` (âge minimal des lignes pour les geler) à 0. Il va geler toutes les lignes qui le peuvent, même « jeunes ».

Charge induite par le gel :

Le gel des lignes peut être très lourd s'il y a beaucoup de lignes à geler, ou très rapide si l'essentiel du travail a été fait par les nettoyages précédents. Si la table a déjà été entièrement gelée (parfois depuis des centaines de millions de transactions), il peut juste s'agir d'une mise à jour du `relfrozenxid`.

Les blocs déjà entièrement gelés sont recensés dans la *visibility map* (qui recense aussi les blocs sans ligne morte). ils ne seront pas reparcourus s'ils ne sont plus modifiés. Cela accélère énormément le `FREEZE` sur les grosses tables (avant PostgreSQL 9.6, il y avait forcément au moins un parcours complet de la table !) Si le `VACUUM` est interrompu, ce qu'il a déjà gelé n'est pas perdu, il ne faut donc pas hésiter à l'interrompre au besoin.

L'âge de la table peut dépasser `autovacuum_freeze_max_age` si le nettoyage est laborieux, ce qui explique la marge par rapport à la limite fatidique des 2 milliards de transactions.

Quelques problèmes possibles sont évoqués plus bas.

Âge d'une base :

Nous avons vu que l'âge d'une base est en fait l'âge de la table la plus ancienne, qui se calcule à partir de la colonne `pg_database.datfrozenxid` :

```
SELECT datname, datfrozenxid, age (datfrozenxid)
FROM pg_database ORDER BY 3 DESC ;
```

```
  datname   | datfrozenxid |    age
-----------+--------------+-----------
 pgbench    |   1809610092 | 149835222
 template0  |   1957896953 |   1548361
 template1  |   1959012415 |    432899
 postgres   |   1959445305 |         9
```

Concrètement, on verra l'âge d'une base de données approcher peu à peu des 200 millions de transactions, ce qui correspondra à l'âge des plus « vieilles » tables, souvent celles sur lesquelles l'autovacuum ne passe jamais. L'âge des tables évolue même si l'essentiel de leur contenu, voire la totalité, est déjà gelé (car il peut rester le `pg_class.relfrozenxid` à mettre à jour, ce qui sera bien sûr très rapide). Cet âge va retomber quand un gel sera forcé sur ces tables, puis remonter, etc.

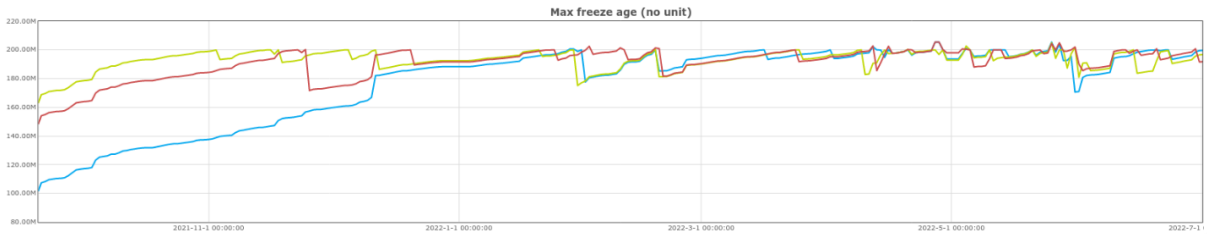


Figure 12/ .4: Évolution de l'âge des bases sur une instance

Rappelons que le `FREEZE` réécrit tous les blocs concernés. Il peut être quasi-instantané, mais le déclenchement inopiné d'un `VACUUM FREEZE` sur l'intégralité d'une très grosse table assez statique est une mauvaise surprise assez fréquente. La table est réécrite, les entrées-sorties sont chargées, la sauvegarde PITR enfle, évidemment à un moment où la base est chargée.
Une base chargée en bloc et peu modifiée peut même voir le `FREEZE` se déclencher sur toutes les tables en même temps. Le délai avant le déclenchement du gel par l'autovacuum dépend de la consommation des numéros de transaction sur l'instance migrée, et varie de quelques semaines à des années. Sont concernés tous les imports massifs et les migrations et restauration logiques (`pg_restore`, réplication logique, migration de bases depuis d'autres bases de données), mais pas les migrations et restaurations physiques.
Des ordres `VACUUM FREEZE` sur les plus grosses tables à des moments calmes permettent d'étaler ces écritures. Si ces ordres sont interrompus, l'essentiel de ce qu'ils auront pu geler n'est plus à re-geler plus tard.

Résumé :

Que retenir de ce paramétrage complexe ?

- le `VACUUM` gèlera une partie des lignes un peu anciennes lors de son fonctionnement habituel ;
- un bloc gelé non modifié ne sera plus à regeler ;
- de grosses tables statiques peuvent engendrer soudainement une grosse charge en écriture ; il vaut mieux être proactif.

12.8 AUTRES PROBLÈMES COURANTS

12.8.1 L'autovacuum dure trop longtemps

- Fréquence de passage ?
- Débit ?
- Nombre de workers ?
- Taille vraiment trop grosse ?

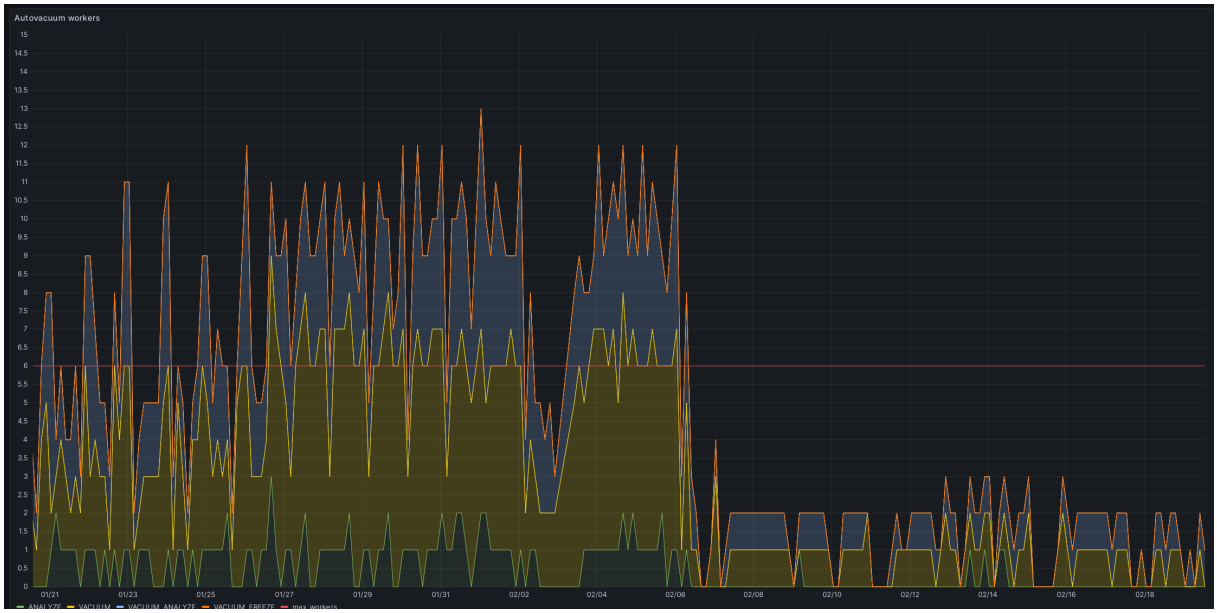


Figure 12/ .5: Nombre de workers avant et après la réduction d'autovacuum_vacuum_cost_delay

Nous avons vu que le paramétrage de l'autovacuum vise à limiter la charge sur la base. Le nettoyage d'une grosse table peut donc être parfois très long. Ce n'est pas forcément un problème majeur si l'opération arrive à terme dans un délai raisonnable, mais il vaut mieux savoir pourquoi. Il peut y avoir plusieurs causes, qui ne s'excluent pas mutuellement.

Il est fréquent que les grosses tables soient visitées trop peu souvent. Rappelons que la propriété `autovacuum_vacuum_scale_factor` de chaque table est par défaut à 20 % : lorsque l'autovacuum se déclenche, il doit donc traiter une volumétrie importante. Il est conseillé de réduire la valeur de ce paramètre (ou de jouer sur `autovacuum_vacuum_threshold`) pour un nettoyage plus fréquent.

Le débit en écriture peut être insuffisant (c'est fréquent sur les anciennes versions), auquel cas, avec des disques corrects, on peut baisser `autovacuum_vacuum_cost_delay` ou monter

`autovacuum_vacuum_cost_limit`. Sur le graphique ci-dessus, issu d'un cas réel, les trois workers semblaient en permanence occupés. Il risquait donc d'y avoir un retard pour nettoyer certaines tables, ou calculer des statistiques. La réduction de `autovacuum_vacuum_cost_delay` de 20 à 2 ms a mené à une réduction drastique de la durée de traitement de chaque worker.

Rappelons qu'un `VACUUM` manuel (ou planifié) n'est soumis à aucun bridage.

Le nombre de workers peut être trop bas, notamment s'il y a de nombreuses tables. Auquel cas ils semblent tous activés en permanence (comme ci-dessus). Monter `autovacuum_max_workers` au-delà de 3 nécessite d'augmenter le débit autorisé avec les paramètres ci-dessus.

Pour des grandes tables, le partitionnement permet de paralléliser l'activité de l'autovacuum. Les *workers* peuvent en effet travailler dans la même base, sur des tables différentes.

Un grand nombre de bases actives peut devenir un frein et augmenter l'intervalle entre deux nettoyages d'une base, bien que l'`autovacuum launcher` ignore les bases inutilisées.

Exceptionnellement, l'autovacuum peut tomber en erreur (bloc corrompu, index fonctionnel avec une fonction boguée…) et ne jamais finir (surveiller les traces).

12.8.2 Arrêter un VACUUM ?

- Lancement manuel ou script
  - risque avec certains verrous
- Autovacuum
  - interrompre s'il gêne
- Exception : *to prevent wraparound* lent **et** bloquant
  - `pg_cancel_backend` + `VACUUM FREEZE` manuel

Le cas des `VACUUM` manuels a été vu plus haut : ils peuvent gêner quelques verrous ou opérations DDL. Il faudra les arrêter manuellement au besoin.

C'est différent si l'autovacuum a lancé le processus : celui-ci sera arrêté si un utilisateur pose un verrou en conflit.

La seule exception concerne un `VACUUM FREEZE` lancé quand la table doit être gelée, donc avec la mention *to prevent wraparound* dans `pg_stat_activity` : celui-ci ne sera pas interrompu. Il ne pose qu'un verrou destinée à éviter les modifications de schéma simultanées (SHARE UPDATE EXCLUSIVE). Comme le débit en lecture et écriture est bridé par le paramétrage habituel de l'autovacuum, ce verrou peut durer assez longtemps (surtout avant PostgreSQL 9.6, où toute la table est relue à

chaque `FREEZE`). Cela peut s'avérer gênant avec certaines applications. Une solution est de réduire `autovacuum_vacuum_cost_delay`, surtout avant PostgreSQL 12 (voir plus haut).

Si les opérations sont impactées, on peut vouloir lancer soi-même un `VACUUM FREEZE` manuel, non bridé. Il faudra alors repérer le PID du `VACUUM FREEZE` en cours, l'arrêter avec `pg_cancel_backend`, puis lancer manuellement l'ordre `VACUUM FREEZE` sur la table concernée, (et rapidement avant que l'autovacuum ne relance un processus).

La supervision peut se faire avec `pg_stat_progress_vacuum` et `iotop`.

12.8.3 Ce qui peut bloquer le VACUUM FREEZE

- Causes :
  - sessions *idle in transaction* sur une longue durée
  - slot de réplication en retard/oublié
  - transactions préparées oubliées
  - erreur à l'exécution du `VACUUM`
- Conséquences :
  - processus autovacuum répétés
  - arrêt des transactions
  - mode single…
- Supervision :
  - `check_pg_activity` : `xmin`, `max_freeze_age`
  - surveillez les traces !

Il arrive que le fonctionnement du `FREEZE` soit gêné par un problème qui lui interdit de recycler les plus anciens numéros de transactions. (Ces causes gênent aussi un `VACUUM` simple, mais les symptômes sont alors surtout un gonflement des tables concernées).

Les causes possibles sont :

- des sessions *idle in transactions* durent depuis des jours ou des semaines (voir le statut `idle in transaction` dans `pg_stat_activity`, et au besoin fermer la session) : au pire, elles disparaissent après redémarrage ;
- des slots de réplication pointent vers un secondaire très en retard, voire disparu (consulter `pg_replication_slots`, et supprimer le slot) ;

- des transactions préparées (pas des requêtes préparées !) n'ont jamais été validées ni annulées, (voir `pg_prepared_xacts`, et annuler la transaction) : elles ne disparaissent pas après redémarrage ;
- l'opération de `VACUUM` tombe en erreur : corruption de table ou index, fonction d'index fonctionnel buggée, etc. (voir les traces et corriger le problème, supprimer l'objet ou la fonction, etc.).

Pour effectuer le `FREEZE` en urgence le plus rapidement possible, on peut utiliser :

```
VACUUM (FREEZE, VERBOSE, INDEX_CLEANUP off, TRUNCATE off) ;
```

Cette commande force le gel de toutes les lignes, ignore le nettoyage des index et ne supprime pas les blocs vides finaux (le verrou peut être gênant). Un `VACUUM` classique serait à prévoir ensuite à l'occasion.

En toute rigueur, une version sans l'option `FREEZE` est encore plus rapide : le mode agressif serait déclenché mais les lignes plus récentes que `vacuum_freeze_min_age` (50 millions de transaction) ne seraient pas encore gelées. On peut même monter ce paramètre dans la session pour alléger au maximum la charge sur une table dont les lignes ont des âges bien étalés.

Ne pas oublier de nettoyer toutes les bases de l'instance.

Dans le pire des cas, plus aucune transaction ne devient possible (y compris les opérations d'administration comme `DROP`, ou `VACUUM` sans `TRUNCATE off`) :

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database
↪  "db1"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
You might also need to commit or roll back old prepared transactions,
or drop stale replication slots.
```

En dernière extrémité, il reste un délai de grâce d'un million de transactions, qui ne sont accessibles que dans le très austère mode monoutilisateur[5] de PostgreSQL.

Avec la sonde Nagios check_pgactivity[6], et les services **max_freeze_age** et **oldest_xmin**, il est possible de vérifier que l'âge des bases ne dérive pas, ou de trouver quel processus porte le `xmin` le plus ancien. S'il y a un problème, il entraîne généralement l'apparition de nombreux messages dans les traces : lisez-les régulièrement !

[5] https://docs.postgresql.fr/current/app-postgres.html#APP-POSTGRES-SINGLE-USER
[6] https://github.com/OPMDG/check_pgactivity

12.9 RÉSUMÉ DES CONSEILS SUR L'AUTOVACUUM

> "Vacuuming is like exercising.
> If it hurts, you're not doing it enough!"
>
> (Robert Haas, PGConf.EU 2023, Prague, 13 décembre 2023)

Certains sont frileux à l'idée de passer un `VACUUM`. En général, ce n'est que reculer pour mieux sauter.

12.9.1 Résumé des conseils sur l'autovacuum (1/2)

- Laisser l'autovacuum faire son travail
- Augmenter le débit autorisé
- Surveiller `last_(auto)analyze` / `last_(auto)vacuum`
- Nombre de *workers*
- Grosses tables, par ex :

```
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.01) ;
ALTER TABLE table_name SET (autovacuum_vacuum_threshold = 1000000) ;
```

- Mais ne pas hésiter à planifier un `vacuumdb` quotidien

L'autovacuum fonctionne convenablement pour les charges habituelles. Il ne faut pas s'étonner qu'il fonctionne longtemps en arrière-plan : il est justement conçu pour ne pas se presser. Au besoin, ne pas hésiter à lancer manuellement l'opération, donc sans bridage en débit.

Si les disques sont bons, on peut augmenter le débit autorisé :

- en réduisant la durée de pause (`autovacuum_vacuum_cost_delay`), surtout avant la version 12 ;
- de préférence, en augmentant le coût à atteindre avant une pause (`autovacuum_vacuum_cost_limit`) ;
- en augmentant `vacuum_buffer_usage_limit` (en version 16).

Comme le déclenchement d'`autovacuum` est très lié à l'activité, il faut vérifier qu'il passe assez souvent sur les tables sensibles en surveillant `pg_stat_all_tables.last_autovacuum` et `last_autoanalyze`.

Si les statistiques peinent à se rafraîchir, ne pas hésiter à activer plus souvent l'autovacuum sur les grosses tables problématiques ainsi :

```
-- analyze après 5 % de modification au lieu du défaut de 10 %
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.05) ;
```

De même, si la fragmentation s'envole, descendre `autovacuum_vacuum_scale_factor`. (On peut préférer utiliser les variantes en `*_threshold` de ces paramètres, et mettre les `*_scale_factor` à 0).

Dans un modèle avec de très nombreuses tables actives, le nombre de *workers* doit parfois être augmenté.

12.9.2 Résumé des conseils sur l'autovacuum (2/2)

- Mode manuel
  - batchs / tables temporaires / tables à insertions seules (<v13)
  - si pressé !
- Danger du `FREEZE` brutal après migration logique ou gros import
  - prévenir
- `VACUUM FULL` : dernière extrémité

L'autovacuum n'est pas toujours assez rapide à se déclencher, par exemple entre les différentes étapes d'un batch : on intercalera des `VACUUM ANALYZE` manuels. Il faudra le faire systématiquement pour les tables temporaires (que l'autovacuum ne voit pas). De plus, avant PostgreSQL 13, pour les tables où il n'y a que des insertions, l'autovacuum ne lance spontanément que l'`ANALYZE` : il faudra effectuer un `VACUUM` explicite pour profiter de certaines optimisations.

Au final, un `vacuumdb` planifié quotidien à un moment calme est généralement une bonne idée : ce qu'il traite ne sera plus à nettoyer en journée, et il apporte la garantie qu'aucune table ne sera négligée. Cela ne dispense pas de contrôler l'activité de l'`autovacuum` sur les grosses tables très sollicitées.

Un point d'attention reste le gel brutal de grosses quantités de données chargées ou modifiées en même temps. Un `VACUUM FREEZE` préventif dans une période calme reste la meilleure solution.

Un `VACUUM FULL` sur une grande table est une opération très lourde, à réserver à la récupération d'une partie significative de son espace, qui ne serait pas réutilisé plus tard.

12.10 CONCLUSION

- `VACUUM` fait de plus en plus de choses au fil des versions
- Convient généralement
- Paramétrage apparemment complexe
  - en fait relativement simple avec un peu d'habitude

12.10.1 Questions

N'hésitez pas, c'est le moment !

12.11 QUIZ

https://dali.bo/m5_quiz

12.12 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m5_solutions.

12.12.1 Traiter la fragmentation

But : Traiter la fragmentation

Créer une table `t3` avec une colonne `id` de type `integer`.

Désactiver l'autovacuum pour la table `t3`.

Insérer un million de lignes dans la table `t3` avec la fonction `generate_series`.

Récupérer la taille de la table `t3`.

Supprimer les 500 000 premières lignes de la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Créer une table `t4` avec une colonne `id` de type `integer`.

Désactiver l'autovacuum pour la table `t4`.

Insérer un million de lignes dans la table `t4` avec `generate_series`.

Récupérer la taille de la table `t4`.

Supprimer les 500 000 **dernières** lignes de la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

Exécuter un `VACUUM` sur la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

12.12.2 Détecter la fragmentation

But : Détecter la fragmentation

Créer une table `t5` avec deux colonnes : `c1` de type `integer` et `c2` de type `text`.

Désactiver l'autovacuum pour la table `t5`.

Insérer un million de lignes dans la table `t5` avec `generate_series`.

- Installer l'extension `pg_freespacemap` (documentation : https://docs.postgresql.fr/current/pgfreespacemap.html)
- Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?

- Modifier exactement 200 000 lignes de la table `t5`.

- Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

Exécuter un `VACUUM` sur la table `t5`.

Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

Récupérer la taille de la table `t5`.

Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

12.12.3 Gestion de l'autovacuum

But : Voir fonctionner l'autovacuum

Créer une table `t6` avec une colonne `id` de type `integer`.

Insérer un million de lignes dans la table `t6` :

```
INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
```

Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Il faudra peut-être attendre une minute. (Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.)

Vérifier le nombre de lignes dans `pg_class.reltuples`.

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.

- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

- Modifier encore 200 000 autres lignes de la table `t6` :

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

12.13 TRAVAUX PRATIQUES (SOLUTIONS)

12.13.1 Traiter la fragmentation

> Créer une table `t3` avec une colonne `id` de type `integer`.

```
CREATE TABLE t3(id integer);
```

```
CREATE TABLE
```

> Désactiver l'autovacuum pour la table `t3`.

```
ALTER TABLE t3 SET (autovacuum_enabled = false);
```

```
ALTER TABLE
```

La désactivation de l'autovacuum ici a un but uniquement pédagogique. En production, c'est une très mauvaise idée !

> Insérer un million de lignes dans la table `t3` avec la fonction `generate_series`.

```
INSERT INTO t3 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

> Récupérer la taille de la table `t3`.

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 35 MB
```

> Supprimer les 500 000 premières lignes de la table `t3`.

```
DELETE FROM t3 WHERE id <= 500000;
```

```
DELETE 500000
```

> Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 35 MB
```

`DELETE` seul ne permet pas de regagner de la place sur le disque. Les lignes supprimées sont uniquement marquées comme étant mortes. Comme l'autovacuum est ici désactivé, PostgreSQL n'a pas encore nettoyé ces lignes.

> Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

```
VACUUM VERBOSE t3;
```

```
INFO:  vacuuming "public.t3"
INFO:  "t3": removed 500000 row versions in 2213 pages
INFO:  "t3": found 500000 removable, 500000 nonremovable row versions
             in 4425 out of 4425 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 3815272
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.09 s, system: 0.00 s, elapsed: 0.10 s.
VACUUM
```

L'indication :

```
removed 500000 row versions in 2213 pages
```

indique 500 000 lignes ont été nettoyées dans 2213 blocs (en gros, la moitié des blocs de la table).

Pour compléter, l'indication suivante :

```
found 500000 removable, 500000 nonremovable row versions in 4425 out of 4425 pages
```

reprend l'indication sur 500 000 lignes mortes, et précise que 500 000 autres ne le sont pas. Les 4425 pages parcourues correspondent bien à la totalité des 35 Mo de la table complète. C'est la première fois que `VACUUM` passe sur cette table, il est normal qu'elle soit intégralement parcourue.

> Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 35 MB
```

`VACUUM` ne permet pas non plus de gagner en espace disque. Principalement, il renseigne la structure FSM (*free space map*) sur les emplacements libres dans les fichiers des tables.

> Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

```
VACUUM FULL t3;
```

```
INFO:  vacuuming "public.t3"
INFO:  "t3": found 0 removable, 500000 nonremovable row versions in 4425 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.10 s, system: 0.01 s, elapsed: 0.21 s.
VACUUM
```

> Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 17 MB
```

Là, par contre, nous gagnons en espace disque. Le `VACUUM FULL` reconstruit la table et la fragmentation disparaît.

> Créer une table `t4` avec une colonne `id` de type `integer`.

```
CREATE TABLE t4(id integer);
```

```
CREATE TABLE
```

> Désactiver l'autovacuum pour la table `t4`.

```
ALTER TABLE t4 SET (autovacuum_enabled = false);
```

```
ALTER TABLE
```

> Insérer un million de lignes dans la table `t4` avec `generate_series`.

```
INSERT INTO t4(id) SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

> Récupérer la taille de la table `t4`.

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
 pg_size_pretty
----------------
 35 MB
```

> Supprimer les 500 000 **dernières** lignes de la table `t4`.

```
DELETE FROM t4 WHERE id > 500000;
```

```
DELETE 500000
```

> Récupérer la taille de la table `t4`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
 pg_size_pretty
----------------
 35 MB
```

Là aussi, nous n'avons rien perdu.

> Exécuter un `VACUUM` sur la table `t4`.

```
VACUUM t4;
```

```
VACUUM
```

> Récupérer la taille de la table `t4`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
 pg_size_pretty
----------------
 17 MB
```

En fait, il existe un cas où il est possible de gagner de l'espace disque suite à un `VACUUM` simple : quand l'espace récupéré se trouve en fin de table et qu'il est possible de prendre rapidement un verrou exclusif sur la table pour la tronquer. C'est assez peu fréquent mais c'est une optimisation intéressante.

12.13.2 Détecter la fragmentation

> Créer une table `t5` avec deux colonnes : `c1` de type `integer` et `c2` de type `text`.

```
CREATE TABLE t5 (c1 integer, c2 text);
```

```
CREATE TABLE
```

> Désactiver l'autovacuum pour la table `t5`.

```
ALTER TABLE t5 SET (autovacuum_enabled=false);
```

```
ALTER TABLE
```

> Insérer un million de lignes dans la table `t5` avec `generate_series`.

```
INSERT INTO t5(c1, c2) SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
```

```
INSERT 0 1000000
```

> - Installer l'extension `pg_freespacemap` (documentation : https://docs.postgresql.fr/current/pgfreespacemap.html)
> - Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?

```
CREATE EXTENSION pg_freespacemap;
```

```
CREATE EXTENSION
```

Cette extension installe une fonction nommée `pg_freespace`, dont la version la plus simple ne demande que la table en argument, et renvoie l'espace libre dans chaque bloc, en octets, *connu de la Free Space Map*.

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
 count | sum
-------+-----
  6274 |   0
```

et donc 6274 blocs (soit 51,4 Mo) sans aucun espace vide.

> - Modifier exactement 200 000 lignes de la table `t5`.
> - Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

```
UPDATE t5 SET c2 = upper(c2) WHERE c1 <= 200000;
```

```
UPDATE 200000
```

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
 count | sum
-------+-----
  7451 |  32
```

La table comporte donc 20 % de blocs en plus, où sont stockées les nouvelles versions des lignes modifiées. Le champ *avail* indique qu'il n'y a quasiment pas de place libre. (Ne pas prendre la valeur de 32 octets au pied de la lettre, la *Free Space Map* ne cherche pas à fournir une valeur précise.)

> Exécuter un `VACUUM` sur la table `t5`.

```
VACUUM VERBOSE t5;
```

```
INFO:  vacuuming "public.t5"
INFO:  "t5": removed 200000 row versions in 1178 pages
INFO:  "t5": found 200000 removable, 1000000 nonremovable row versions
       in 7451 out of 7451 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 8685974
         There were 0 unused item identifiers.
         Skipped 0 pages due to buffer pins, 0 frozen pages.
         0 pages are entirely empty.
         CPU: user: 0.11 s, system: 0.03 s, elapsed: 0.33 s.
INFO:  vacuuming "pg_toast.pg_toast_4160544"
INFO:  index "pg_toast_4160544_index" now contains 0 row versions in 1 pages
DÉTAIL : 0 index row versions were removed.
         0 index pages have been deleted, 0 are currently reusable.
         CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "pg_toast_4160544": found 0 removable, 0 nonremovable row versions in 0 out of
 ↪  0 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 8685974
        There were 0 unused item identifiers.
        Skipped 0 pages due to buffer pins, 0 frozen pages.
        0 pages are entirely empty.
        CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

> Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
 count |   sum
-------+---------
  7451 | 8806816
```

Il y a toujours autant de blocs, mais environ 8,8 Mo sont à présent repérés comme libres.

Il faut donc bien exécuter un `VACUUM` pour que PostgreSQL nettoie les blocs et mette à jour la structure FSM, ce qui nous permet de déduire le taux de fragmentation de la table.

> Récupérer la taille de la table `t5`.

```
SELECT pg_size_pretty(pg_table_size('t5'));
```

```
 pg_size_pretty
----------------
 58 MB
```

> Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

```
VACUUM (FULL, VERBOSE) t5;
```

```
INFO:  vacuuming "public.t5"
INFO:  "t5": found 200000 removable, 1000000 nonremovable row versions in 7451 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.49 s, system: 0.19 s, elapsed: 1.46 s.
VACUUM
```

> Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

```
SELECT count(blkno),sum(avail)FROM pg_freespace('t5'::regclass);
```

```
 count | sum
-------+-----
  6274 |   0
```

```
SELECT pg_size_pretty(pg_table_size('t5'));
```

```
 pg_size_pretty
----------------
 49 MB
```

`VACUUM FULL` a réécrit la table sans les espaces morts, ce qui nous a fait gagner entre 8 et 9 Mo. La taille de la table maintenant correspond bien à celle de l'ancienne table, moins la place prise par les lignes mortes.

12.13.3 Gestion de l'autovacuum

> Créer une table `t6` avec une colonne `id` de type `integer`.

```
CREATE TABLE t6 (id integer) ;
```

```
CREATE TABLE
```

> Insérer un million de lignes dans la table `t6` :
>
> ```
> INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
> ```

```
INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
```

```
INSERT 0 1000000
```

> Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Il faudra peut-être attendre une minute. (Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.)

```
\x
```

```
Expanded display is on.
```

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-------+-----------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 0
seq_tup_read        | 0
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 0
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:42:43.612269+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:42:43.719195+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 1
```

Les deux dates `last_autovacuum` et `last_autoanalyze` sont renseignées. Il faudra peut-être attendre une minute que l'autovacuum passe sur la table (voire plus sur une instance chargée par ailleurs).

Le seuil de déclenchement de l'autoanalyze est :
`autovacuum_analyze_scale_factor` × nombre de lignes

+ `autovacuum_analyze_threshold`
soit par défaut 10 % × 0 + 50 = 50. Quand il n'y a que des insertions, le seuil pour l'autovacuum est :
`autovacuum_vacuum_insert_scale_factor` × nombre de lignes
+ `autovacuum_vacuum_insert_threshold`
soit 20 % × 0 + 1000 = 1000.

Avec un million de nouvelles lignes, les deux seuils sont franchis.

Avec PostgreSQL 12 ou antérieur, seule la ligne `last_autoanalyze` sera remplie. S'il n'y a que des insertions, le démon autovacuum ne lance un `VACUUM` spontanément qu'à partir de PostgreSQL 13.

Jusqu'en PostgreSQL 12, il faut donc lancer manuellement :

```
ANALYZE t6 ;
```

> Vérifier le nombre de lignes dans `pg_class.reltuples`.

Vérifions que le nombre de lignes est à jour dans `pg_class` :

```
SELECT * FROM pg_class WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-------+--------
oid                 | 4160608
relname             | t6
relnamespace        | 2200
reltype             | 4160610
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 4160608
reltablespace       | 0
relpages            | 4425
reltuples           | 1e+06
...
```

L'autovacuum se base entre autres sur cette valeur pour décider s'il doit passer ou pas. Si elle n'est pas encore à jour, il faut lancer manuellement :

```
ANALYZE t6 ;
```

ce qui est d'ailleurs généralement conseillé après un gros chargement.

> - Modifier 60 000 lignes supplémentaires de la table `t6` avec :
>
> ```
> UPDATE t6 SET id=1 WHERE id > 940000 ;
> ```
>
> - Attendre une minute.

> - Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
> - Que faut-il en déduire ?

```
UPDATE t6 SET id = 0 WHERE id <= 150000 ;
```

```
UPDATE 150000
```

Le démon *autovacuum* ne se déclenche pas instantanément après les écritures, attendons un peu :

```
SELECT pg_sleep(60) ;
```

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 1
seq_tup_read        | 1000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 150000
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
n_dead_tup          | 150000
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:42:43.612269+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:43:43.561288+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 2
```

Seul `last_autoanalyze` a été modifié, et il reste entre 150 000 lignes morts (`n_dead_tup`). En effet, le démon autovacuum traite séparément l'`ANALYZE` (statistiques sur les valeurs des données) et le `VACUUM` (recherche des espaces morts). Si l'on recalcule les seuils de déclenchement, on trouve pour l'autoanalyze :
`autovacuum_analyze_scale_factor` × nombre de lignes
+ `autovacuum_analyze_threshold`
soit par défaut 10 % × 1 000 000 + 50 = 100 050, dépassé ici.

Pour l'autovacuum, le seuil est de :
`autovacuum_vacuum_insert_scale_factor` × nombre de lignes
+ `autovacuum_vacuum_insert_threshold`
soit 20 % × 1 000 000 + 50 = 200 050, qui n'est pas atteint.

> - Modifier 60 000 lignes supplémentaires de la table `t6` avec :
>
> ```
> UPDATE t6 SET id=1 WHERE id > 940000 ;
> ```
>
> - Attendre une minute.
> - Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
> - Que faut-il en déduire ?

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

```
UPDATE 60000
```

L'autovacuum ne passe pas tout de suite, les 210 000 lignes mortes au total sont bien visibles :

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6';
```

```
-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 3
seq_tup_read        | 3000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 210000
n_tup_del           | 0
n_tup_hot_upd       | 65
n_live_tup          | 1000000
n_dead_tup          | 210000
n_mod_since_analyze | 60000
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:42:43.612269+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:43:43.561288+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 2
```

Mais comme le seuil de 200 050 lignes modifiées à été franchi, le démon lance un `VACUUM` :

```
-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 3
seq_tup_read        | 3000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 210000
n_tup_del           | 0
n_tup_hot_upd       | 65
```

```
n_live_tup           | 896905
n_dead_tup           | 0
n_mod_since_analyze  | 60000
n_ins_since_vacuum   | 0
last_vacuum          | ¤
last_autovacuum      | 2021-02-22 17:47:43.740962+01
last_analyze         | ¤
last_autoanalyze     | 2021-02-22 17:43:43.561288+01
vacuum_count         | 0
autovacuum_count     | 2
analyze_count        | 0
autoanalyze_count    | 2
```

Noter que `n_dead_tup` est revenu à 0. `last_auto_analyze` indique qu'un nouvel `ANALYZE` n'a pas été exécuté : seules 60 000 lignes ont été modifiées (voir `n_mod_since_analyze`), en-dessous du seuil de 100 050.

> Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

```
ALTER TABLE t6 SET (autovacuum_vacuum_scale_factor=0.1);
```

```
ALTER TABLE
```

> - Modifier encore 200 000 autres lignes de la table `t6` :
>
> ```
> UPDATE t6 SET id=1 WHERE id > 740000 ;
> ```
>
> - Attendre une minute.
> - Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
> - Que faut-il en déduire ?

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

```
UPDATE 200000
```

```
SELECT pg_sleep(60);
```

```
SELECT * FROM pg_stat_user_tables WHERE relname='t6' ;
```

```
-[ RECORD 1 ]-------+-----------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 4
seq_tup_read        | 4000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 410000
n_tup_del           | 0
n_tup_hot_upd       | 65
n_live_tup          | 1000000
n_dead_tup          | 0
n_mod_since_analyze | 0
```

```
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:53:43.563671+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:53:43.681023+01
vacuum_count        | 0
autovacuum_count    | 3
analyze_count       | 0
autoanalyze_count   | 3
```

Le démon a relancé un `VACUUM` et un `ANALYZE`. Avec un facteur d'échelle à 10 %, il ne faut plus attendre que la modification de 100 050 lignes pour que le `VACUUM` soit déclenché par le démon. C'était déjà le seuil pour l' `ANALYZE`.

13/ Partitionnement déclaratif (introduction)

- Le partitionnement déclaratif apparaît avec PostgreSQL 10
- Préférer un PostgreSQL récent
- Ne plus utiliser l'ancien partitionnement par héritage.

Ce module introduit le partitionnement déclaratif introduit avec PostgreSQL 10, et amélioré dans les versions suivantes. PostgreSQL 13 au minimum est conseillé pour ne pas être gêné par les limitations des versions précédentes (qui ne sont d'ailleurs plus supportées). Les améliorations à chaque nouvelle version de PostgreSQL doivent mener à favoriser une version récente de PostgreSQL.

Le partitionnement par héritage, au fonctionnement totalement différent, reste utilisable, mais ne doit plus servir aux nouveaux développements, du moins pour les cas décrits ici.

13.1 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/

13.2 PARTITIONNEMENT DÉCLARATIF

- Table partitionnée
  - structure uniquement
  - index/contraintes répercutés sur les partitions
- Partitions :
  - 1 partition = 1 table classique, utilisable directement
  - clé de partitionnement (inclue dans PK/UK)
  - partition par défaut
  - sous-partitions possibles
  - FDW comme partitions possible
  - attacher/détacher une partition

En partitionnement déclaratif, une table partitionnée ne contient pas de données par elle-même. Elle définit la structure (champs, types) et les contraintes et index, qui sont répercutées sur ses partitions.

Une partition est une table à part entière, rattachée à une table partitionnée. Sa structure suit automatiquement celle de la table partitionnée et ses modifications. Cependant, des index ou contraintes supplémentaires propres à cette partition peuvent être ajoutées de la même manière que pour une table classique.

La partition se définit par une « clé de partitionnement », sur une ou plusieurs colonnes. Les lignes de même clé se retrouvent dans la même partition. La clé peut se définir comme :

- une liste de valeurs ;
- une plage de valeurs ;
- une valeur de hachage.

Les clés des différentes partitions ne doivent pas se recouvrir.

Une partition peut elle-même être partitionnée, sur une autre clé, ou la même.

Une table classique peut être attachée à une table partitionnée. Une partition peut être détachée et redevenir une table indépendante normale.

Même une table distante (utilisant *foreign data wrapper*) peut être définie comme partition, avec des restrictions. Pour les performances, préférer alors PostgreSQL 14 au moins.

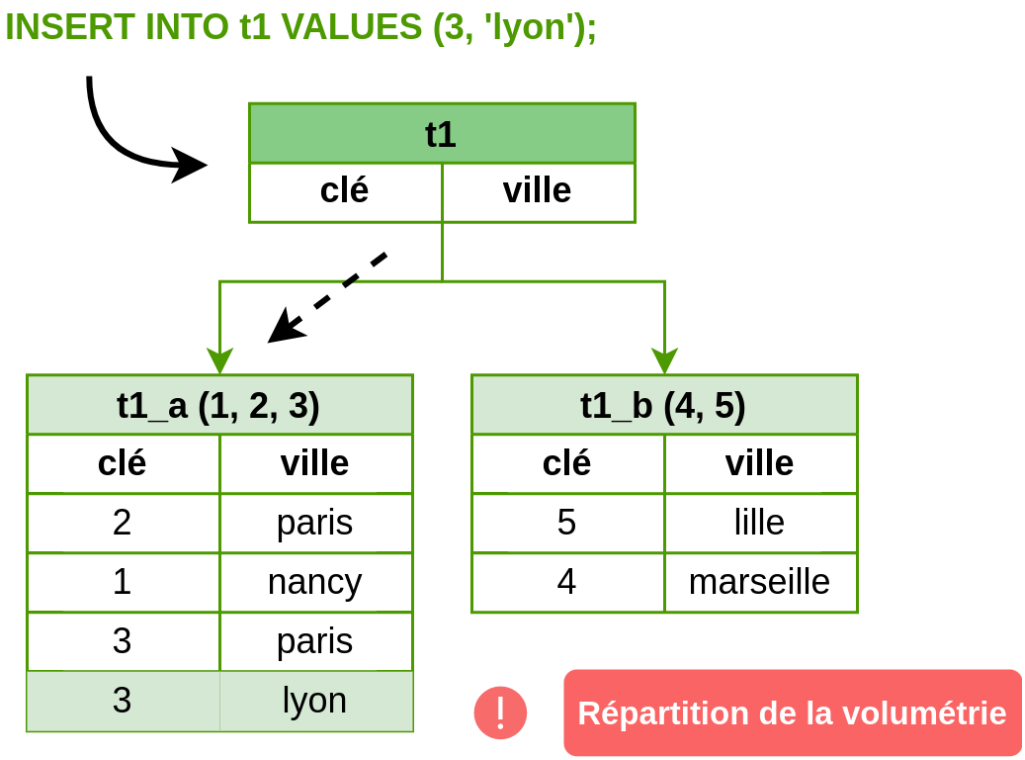
Les clés étrangères entre tables partitionnées ne sont pas un problème dans les versions récentes de PostgreSQL.

Le routage des données insérées ou modifiées vers la bonne partition est géré de façon automatique en fonction de la définition des partitions. La création d'une partition par défaut permet d'éviter des erreurs si aucune partition ne convient.

De même, à la lecture de la table partitionnée, les différentes partitions nécessaires sont accédées de manière transparente.

Pour le développeur, la table principale peut donc être utilisée comme une table classique. Il vaut mieux cependant qu'il connaisse le mode de partitionnement, pour utiliser la clé autant que possible. La complexité supplémentaire améliorera les performances. L'accès direct aux partitions par leur nom de table reste possible, et peut parfois améliorer les performances. Un développeur pourra aussi purger des données plus rapidement, en effectuant un simple `DROP` de la partition concernée.

13.2.1 Partitionnement par liste



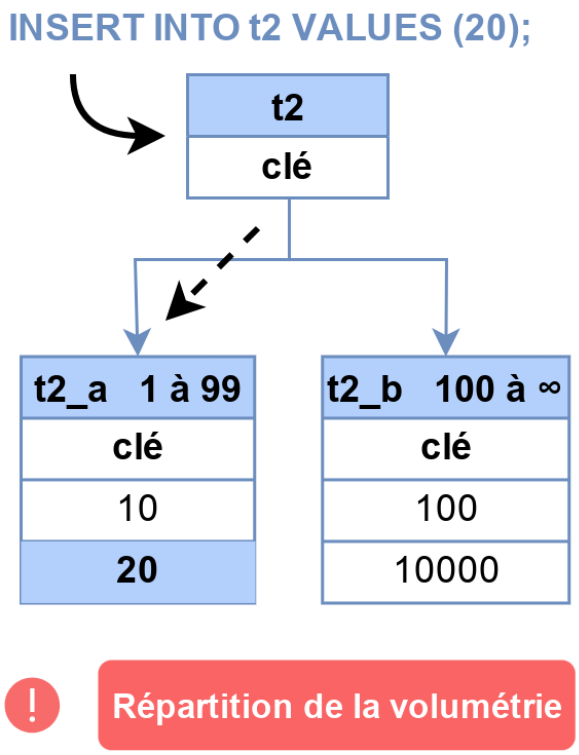
13.2.2 Partitionnement par liste : implémentation

```
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) ;
…
```

Le partitionnement par liste définit les valeurs d'une colonne acceptables dans chaque partition.

Utilisations courantes : partitionnement par année, par statut, par code géographique…

13.2.3 Partitionnement par intervalle



13.2.4 Partitionnement par intervalle : implémentation

```
CREATE TABLE logs ( d timestamptz, contenu text) PARTITION BY RANGE (d) ;

CREATE TABLE logs_201901 PARTITION OF logs
                         FOR VALUES FROM ('2019-01-01') TO ('2019-02-01');
CREATE TABLE logs_201902 PARTITION OF logs
                         FOR VALUES FROM ('2019-02-01') TO ('2019-03-01');
…
CREATE TABLE logs_201912 PARTITION OF logs
                         FOR VALUES FROM ('2019-12-01') TO ('2020-01-01');
…
CREATE TABLE logs_autres PARTITION OF logs
                         DEFAULT ;              -- pour ne rien perdre
```

Utilisations courantes : partitionnement par date, par plages de valeurs continues, alphabétiques…

L'exemple ci-dessus utilise le partitionnement par mois. Chaque partition est définie par des plages de date. Noter que la borne supérieure ne fait *pas* partie des données de la partition. Elle doit donc être aussi la borne inférieure de la partie suivante. La description de la table partitionnée devient :

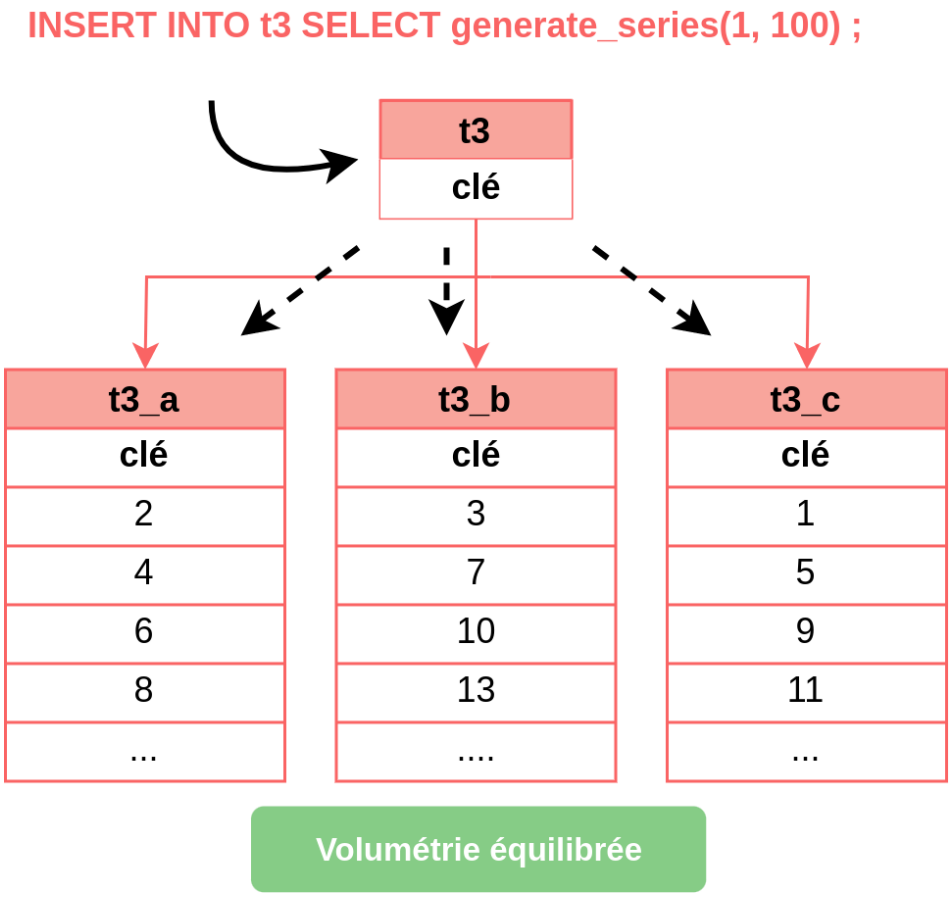
```
=# \d+ logs
                                   Table partitionnée « public.logs »
 Colonne |           Type           | ... | Stockage | …
---------+--------------------------+-----+----------+ …
 d       | timestamp with time zone | ... | plain    | …
 contenu | text                     | ... | extended | …
Clé de partition : RANGE (d)
Partitions: logs_201901 FOR VALUES FROM ('2019-01-01 00:00:00+01') TO ('2019-02-01
 ↪  00:00:00+01'),
            logs_201902 FOR VALUES FROM ('2019-02-01 00:00:00+01') TO ('2019-03-01
 ↪  00:00:00+01'),
            …
            logs_201912 FOR VALUES FROM ('2019-12-01 00:00:00+01') TO ('2020-01-01
 ↪  00:00:00+01'),
            logs_autres DEFAULT
```

La partition par défaut reçoit toutes les données qui ne vont dans aucune autre partition : cela évite des erreurs d'insertion. Il vaut mieux que la partition par défaut reste très petite.

Il est possible de définir des plages sur plusieurs champs :

```
CREATE TABLE tt_a PARTITION OF tt
FOR VALUES FROM (1,'2020-08-10') TO (100, '2020-08-11') ;
```

13.2.5 Partitionnement par hachage



13.2.6 Partitionnement par hachage : implémentation

- Hachage des valeurs
- Répartition homogène
- Indiquer un modulo et un reste

```
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);
```

Ce type de partitionnement vise à répartir la volumétrie dans plusieurs partitions de manière homogène, quand il n'y a pas de clé évidente. En général, il y aura plus que 3 partitions.

13.3 PERFORMANCES & PARTITIONNEMENT

- Insertions via la table principale
  - quasi aucun impact
- Lecture depuis la table principale
  - attention à la clé
- Purge
  - simple `DROP` ou `DETACH`
- Trop de partitions
  - attention au temps de planification

Des `INSERT` dans la table partitionnée seront redirigés directement dans les bonnes partitions avec un impact en performances quasi négligeable.

Lors des lectures ou jointures, il est important de préciser autant que possible la clé de jointure, si elle est pertinente. Dans le cas contraire, toutes les tables de la partition seront interrogées.

Dans cet exemple, la table comprend 10 partitions :

```
EXPLAIN (COSTS OFF)  SELECT COUNT(*) FROM pgbench_accounts ;
                                   QUERY PLAN
--------------------------------------------------------------------------------
Finalize Aggregate
-> Gather
   Workers Planned: 2
   ->  Parallel Append
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_1_pkey on
 ↪  pgbench_accounts_1 pgbench_accounts
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_2_pkey on
 ↪  pgbench_accounts_2 pgbench_accounts_1
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_3_pkey on
 ↪  pgbench_accounts_3 pgbench_accounts_2
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_4_pkey on
 ↪  pgbench_accounts_4 pgbench_accounts_3
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_5_pkey on
 ↪  pgbench_accounts_5 pgbench_accounts_4
     ->  Partial Aggregate
```

```
      ->  Parallel Index Only Scan using pgbench_accounts_6_pkey on
↪  pgbench_accounts_6 pgbench_accounts_5
    ->  Partial Aggregate
      ->  Parallel Index Only Scan using pgbench_accounts_7_pkey on
↪  pgbench_accounts_7 pgbench_accounts_6
    ->  Partial Aggregate
      ->  Parallel Index Only Scan using pgbench_accounts_8_pkey on
↪  pgbench_accounts_8 pgbench_accounts_7
    ->  Partial Aggregate
      ->  Parallel Index Only Scan using pgbench_accounts_9_pkey on
↪  pgbench_accounts_9 pgbench_accounts_8
    ->  Partial Aggregate
      ->  Parallel Index Only Scan using pgbench_accounts_10_pkey on
↪  pgbench_accounts_10 pgbench_accounts_9
```

Avec la clé, PostgreSQL se restreint à la (ou les) bonne(s) partition(s) :

```
EXPLAIN (COSTS OFF)  SELECT * FROM pgbench_accounts WHERE aid = 599999 ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 pgbench_accounts
   Index Cond: (aid = 599999)
```

Si l'on connaît la clé et que le développeur sait en déduire la table, il est aussi possible d'accéder directement à la partition :

```
EXPLAIN (COSTS OFF) SELECT * FROM pgbench_accounts_6 WHERE aid = 599999 ;
```

```
QUERY PLAN
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_6_pkey on pgbench_accounts_6
  Index Cond: (aid = 599999)
```

Cela allège la planification, surtout s'il y a beaucoup de partitions.

Exemples :

- dans une table partitionnée par statut de commande, beaucoup de requêtes ne s'occupent que d'un statut particulier, et peuvent donc n'appeler que la partition concernée (attention si le statut change…) ;
- dans une table partitionnée par mois de création d'une facture, la date de la facture permet de s'adresser directement à la bonne partition.

Il est courant que les ORM ne sachent pas exploiter cette fonctionnalité.

13.3.1 Attacher/détacher une partition

```
ALTER TABLE logs ATTACH PARTITION logs_archives
FOR VALUES FROM (MINVALUE) TO ('2019-01-01') ;
```

- Vérification du respect de la contrainte
  - parcours complet de la table: lent + verrou !

```
ALTER TABLE logs DETACH PARTITION logs_archives ;
```

- Rapide… mais verrou

Attacher une table existante à une table partitionnée implique de définir une clé de partitionnement. PostgreSQL vérifiera que les valeurs présentes correspondent bien à cette clé. Cela peut être long, surtout que le verrou nécessaire sur la table est gênant. Pour accélérer les choses, il est conseillé d'ajouter au préalable une contrainte `CHECK` correspondant à la clé, voire d'ajouter d'avance les index qui seraient ajoutés lors du rattachement.

Détacher une partition est beaucoup plus rapide qu'en attacher une. Cependant, là encore, le verrou peut être gênant.

13.3.2 Supprimer une partition

```
DROP TABLE logs_2018 ;
```

Là aussi, l'opération est simple et rapide, mais demande un verrou exclusif.

13.3.3 Limitations principales du partitionnement déclaratif

- Temps de planification ! Attention si > 100 partitions
- Création non automatique
- Pas d'héritage multiple, schéma fixe
- Limitations avant PostgreSQL 13/14

Certaines limitations du partitionnement sont liées à son principe. Les partitions ont forcément le même schéma de données que leur partition mère. Il n'y a pas de notion d'héritage multiple.

La création des partitions n'est pas automatique (par exemple dans un partitionnement par date). Il faudra prévoir de les créer par avance.

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 faut éviter de dépasser 100 partitions.

Pour contourner cette limite, il reste possible de manipuler directement les partitions s'il est facile de trouver leur nom.

Avant PostgreSQL 13, de nombreuses limitations rendent l'utilisation moins pratique ou moins performante. Si le partitionnement vous intéresse, il est conseillé d'utiliser une version la plus récente possible.

13.4 CONCLUSION

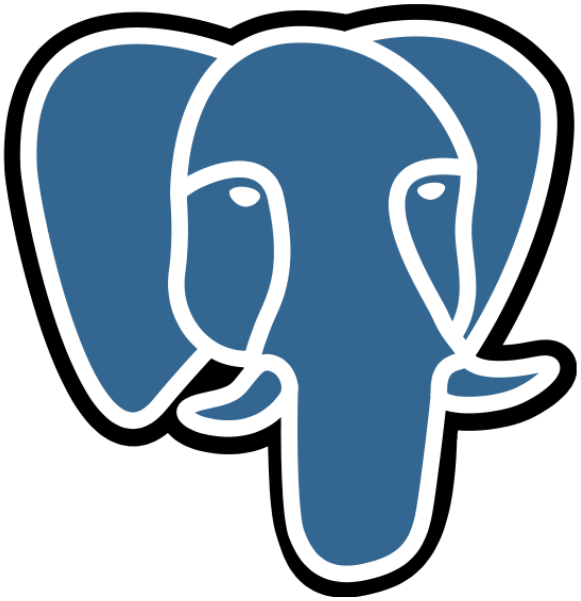
- Préférer une version récente de PostgreSQL
- Pour plus de détails sur le partitionnement
  - https://dali.bo/v1_html

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.

13.5 QUIZ

https://dali.bo/v0_quiz

14/ Sauvegarde physique à chaud et PITR



14.1 INTRODUCTION

- Sauvegarde traditionnelle
  - sauvegarde `pg_dump` à chaud
  - sauvegarde des fichiers à froid
- Insuffisant pour les grosses bases
  - long à sauvegarder
  - encore plus long à restaurer
- Perte de données potentiellement importante
  - car impossible de réaliser fréquemment une sauvegarde
- Une solution : la sauvegarde PITR

La sauvegarde traditionnelle, qu'elle soit logique ou physique à froid, répond à beaucoup de besoins. Cependant, ce type de sauvegarde montre de plus en plus ses faiblesses pour les gros volumes : la sauvegarde est longue à réaliser et encore plus longue à restaurer. Et plus une sauvegarde met du temps, moins fréquemment on l'exécute. La fenêtre de perte de données devient plus importante.

PostgreSQL propose une solution à ce problème avec la sauvegarde physique à chaud. On peut l'utiliser comme un simple mode de sauvegarde supplémentaire, mais elle permet bien d'autres possibilités, d'où le nom de PITR (*Point In Time Recovery*).

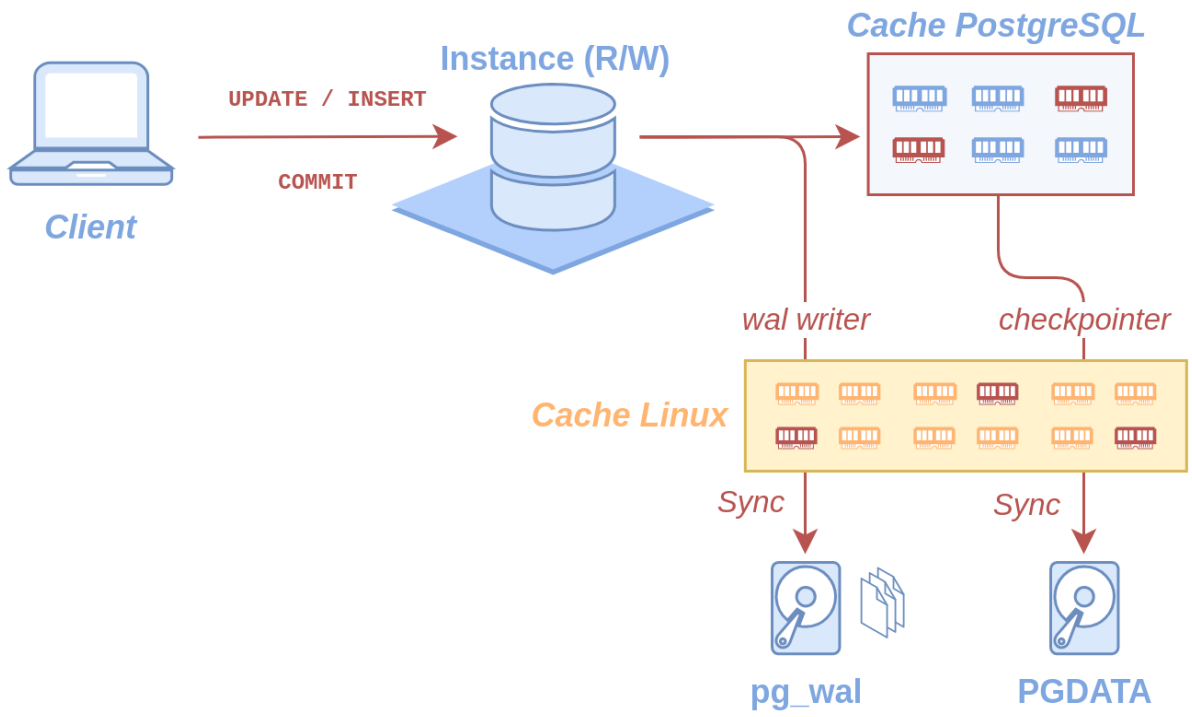
14.1.1 Au menu

- Rappel sur la journalisation
- Principe de la sauvegarde PITR
- Mise en place
  - sauvegarde : manuelle, ou avec `pg_basebackup`
  - archivage : manuel, ou avec `pg_receivewal`
- Restaurer une sauvegarde PITR
- Des outils

Ce module fait le tour de la sauvegarde PITR, de la mise en place de l'archivage (manuelle ou avec

l'outil `pg_receivewal`) à la sauvegarde des fichiers (en manuel, ou avec l'outil `pg_basebackup`). Il discute aussi de la restauration d'une telle sauvegarde. Nous évoquerons très rapidement quelques outils externes pour faciliter ces sauvegardes.

14.2 RAPPEL SUR LA JOURNALISATION



14.2.1 Journaux de transaction

- *Write Ahead Logs* (WAL)
- Chaque donnée est écrite **2 fois** sur le disque !
- Avantages :
  - sécurité infaillible (après `COMMIT`), intégrité, durabilité
  - écriture séquentielle rapide, et un seul *sync* sur le WAL
  - fichiers de données écrits en asynchrone
  - sauvegarde PITR et réplication fiables

Les journaux de transactions (appelés souvent WAL) sont une garantie contre les pertes de données. Il s'agit d'une technique standard de journalisation appliquée à toutes les transactions, pour garantir l'intégrité (la base reste cohérente quoiqu'il arrive) et la durabilité (ce qui est validé ne sera pas perdu).

Ainsi lors d'une modification de donnée, l'écriture au niveau du disque se fait généralement en deux temps :

- écriture des modifications dans le journal de transactions, avec écriture forcée sur disque (« synchronisation ») lors d'un `COMMIT` ;
- écriture dans le fichier de données bien plus tard, lors d'un « checkpoint ».

Ainsi en cas de crash :

- PostgreSQL redémarre ;
- il vérifie s'il reste des données non intégrées aux fichiers de données dans les journaux (mode *recovery*) ;
- si c'est le cas, ces données sont recopiées dans les fichiers de données afin de retrouver un état stable et cohérent ;
- PostgreSQL peut alors s'ouvrir sans perte des transactions validées lors du crash (une transaction interrompue, et donc jamais validée, est perdue).

Les écritures dans le journal se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture (sur un disque dur classique, c'est l'opération la plus coûteuse). De plus, comme nous n'écrivons que dans un seul fichier de transactions, la synchronisation sur disque, lors d'un `COMMIT`, peut se faire sur ce seul fichier, si le système de fichiers le supporte. Concrètement, ces journaux sont des fichiers de 16 Mo par défaut, avec des noms comme `000000010000002600000AF`, dans le répertoire `pg_wal/` de l'instance PostgreSQL (répertoire souvent sur une partition dédiée).

L'écriture définitive dans les fichiers de données est asynchrone, et généralement lissée, ce qui est meilleur pour les performances qu'une écriture immédiate. Cette opération est appelée « *checkpoint* » et périodique (5 minutes par défaut, ou plus).

Divers paramètres et fonctionnalités peuvent altérer ce comportement par défaut, par exemple pour des raisons de performances.

À côté de la sécurité et des performances, le mécanisme des journaux de transactions est aussi utilisé pour des fonctionnalités très intéressantes, comme le PITR et la réplication physique, basés sur le rejeu des informations stockées dans ces journaux.

Pour plus d'informations :

- *PostgreSQL et ses journaux de transactions*[1], Guillaume Lelarge, GNU/Linux Magazine n°108 (septembre 2008), encore d'actualité pour l'essentiel.

[1] https://public.dalibo.com/archives/publications/glmf108_postgresql_et_ses_journaux_de_transactions.pdf

14.3 PITR

- *Point In Time Recovery*
- À chaud
- En continu
- Cohérente

PITR est l'acronyme de *Point In Time Recovery*, autrement dit restauration à un point dans le temps.

C'est une sauvegarde à chaud et surtout en continu. Là où une sauvegarde logique du type `pg_dump` se fait au mieux une fois toutes les 24 h, la sauvegarde PITR se fait en continu grâce à l'archivage des journaux de transactions. De ce fait, ce type de sauvegarde diminue très fortement la fenêtre de perte de données.

Bien qu'elle se fasse à chaud, la sauvegarde est cohérente.

14.3.1 Principes du PITR

- Les journaux de transactions contiennent toutes les modifications
- Il faut les archiver
  - …et avoir une image des fichiers à un instant t (*base backup*)
- La restauration se fait en restaurant cette image
  - puis rejouant tous les journaux
  - dans l'ordre
  - entièrement
  - tous, jusqu'au moment voulu

Quand une transaction est validée, les données à écrire dans les fichiers de données sont d'abord écrites dans un journal de transactions. Ces journaux décrivent donc toutes les modifications survenant sur les fichiers de données, que ce soit les objets utilisateurs comme les objets systèmes. Pour reconstruire un système, il suffit donc d'avoir ces journaux et d'avoir un état des fichiers du répertoire des données à un instant t (*base backup*). Toutes les actions effectuées après cet instant t pourront être rejouées en demandant à PostgreSQL d'appliquer les actions contenues dans les journaux. Les opérations stockées dans les journaux correspondent à des modifications physiques de fichiers, il

faut donc partir d'une sauvegarde au niveau du système de fichier, un export avec `pg_dump` n'est pas utilisable.

Il est donc nécessaire de conserver ces journaux de transactions. Or PostgreSQL les recycle dès qu'il n'en a plus besoin. La solution est de demander au moteur de les archiver ailleurs avant ce recyclage. On doit aussi disposer de l'ensemble des fichiers qui composent le répertoire des données (incluant les tablespaces si ces derniers sont utilisés).

La restauration a besoin des journaux de transactions archivés. Il ne sera pas possible de restaurer et éventuellement revenir à un point donné avec la sauvegarde seule. En revanche, une fois la sauvegarde des fichiers restaurée et la configuration réalisée pour rejouer les journaux archivés, il sera possible de les rejouer, tous ou seulement une partie d'entre eux (en s'arrêtant à un certain moment).

Comme les journaux redéroulent toute l'activité depuis le début de la sauvegarde PITR, ils doivent impérativement être rejoués dans l'ordre de leur écriture (et donc de leur nom), et leur contenu entier est appliqué.

Le rejeu s'arrête quand les journaux à rejouer sont épuisés, ou si le DBA a demandé à s'arrêter à un moment précis.

Il est critique de ne perdre aucun journal. S'il en manque un, ou s'il est inutilisable, la restauration n'ira pas plus loin, les journaux suivants ne seront pas rejoués. La base sera cohérente et utilisable uniquement si l'on a pu réappliquer au moins les journaux générés *pendant* la sauvegarde (point de cohérence).

14.3.2 Avantages du PITR

- Sauvegarde à chaud
- Rejeu d'un grand nombre de journaux
- Moins de perte de données

Tout le travail est réalisé à chaud, que ce soit l'archivage des journaux ou la sauvegarde des fichiers de la base. En effet, il importe peu que les fichiers de données soient modifiés pendant la sauvegarde car les journaux de transactions archivés permettront de corriger toute incohérence par leur application.

Il est possible de rejouer un très grand nombre de journaux (une journée, une semaine, un mois, etc.). Évidemment, plus il y a de journaux à appliquer, plus cela prendra du temps. Mais il n'y a pas de limite au nombre de journaux à rejouer.

Dernier avantage, c'est le système de sauvegarde qui occasionnera le moins de perte de données (RPO). Avec l'archivage continu des journaux de transactions, la fenêtre de perte de données va être

fortement réduite. Plus l'activité est intense, plus la fenêtre de temps sera petite, car les fichiers des journaux sont de taille fixe, et ils ne sont archivés que complets. (À l'inverse, une sauvegarde logique avec `pg_dump` entraînera une perte de données bien plus importante. Si elle est lancée à 3 h et restaurée après un souci à 12 h, on perd 9 heures de données.)

Pour les systèmes n'ayant pas une grosse activité, il est aussi possible de forcer un changement de journal à intervalle régulier, ce qui a pour effet de forcer son archivage, et donc dans les faits de pouvoir s'assurer une perte correspondant au maximum à cet intervalle.

14.3.3 Inconvénients du PITR

- Sauvegarde/restauration de l'instance complète
- Impossible de changer d'architecture (même OS conseillé)
- Nécessite un grand espace de stockage (données + journaux)
- Interdiction de perdre un journal
- Risque d'accumulation des journaux
  - dans `pg_wal/` si échec d'archivage… et arrêt si plein !
  - dans le dépôt d'archivage si échec des sauvegardes
- Plus complexe

Le premier inconvénient vient directement du fait qu'on copie les fichiers : la sauvegarde et la restauration concernent l'instance complète. Il est impossible de ne restaurer qu'une seule base ou que quelques tables.

La restauration se fait impérativement sur la même architecture (x86/ARM, 32/64 bits, *little/big endian*). Il est même fortement conseillé de restaurer dans la même version du même système d'exploitation, sous peine de devoir réindexer l'instance ensuite (à cause d'une différence de définition des locales entre deux versions majeures d'une distribution).

Une sauvegarde PITR nécessite en plus un plus grand espace de stockage. Non seulement il faut sauvegarder les fichiers, y compris les index et la fragmentation, mais aussi tous les journaux de transactions sur une certaine période, ce qui peut être volumineux (en tout cas beaucoup plus que des `pg_dump`). La volumétrie des journaux dépend fortement de l'activité.

En cas de problème dans l'archivage et selon la méthode choisie, l'instance ne voudra pas effacer les journaux non archivés. Il y a donc un risque d'accumulation de ceux-ci. Il faudra donc surveiller la taille du `pg_wal`. En cas de saturation, PostgreSQL s'arrête !

Si un journal est perdu ou corrompu, la restauration ne pourra jamais aller au-delà de ce journal. Le risque augmente avec le nombre de journaux conservés.

Pour réduire le risque de perte d'un journal, et aussi pour accélérer le temps de rejeu, il est conseillé de procéder à des *base backups* (complet, différentiels… selon l'outil) assez fréquemment.

Enfin, la sauvegarde PITR est plus complexe à mettre en place qu'une sauvegarde `pg_dump`. Elle nécessite plus d'étapes, une réflexion sur l'architecture à mettre en œuvre et une meilleure compréhension des mécanismes internes à PostgreSQL pour en avoir la maîtrise.

14.4 COPIE PHYSIQUE À CHAUD PONCTUELLE AVEC PG_BASEBACKUP

(Non PITR)

14.4.1 pg_basebackup

- Réalise les différentes étapes d'une sauvegarde
  - via 1 ou 2 connexions de réplication + slots de réplication
  - base backup + journaux nécessaires
- Copie intégrale
  - image de la base à la **fin** du backup
  - peut servir de base pour du PITR plus tard
- Pas de copie incrémentale avant v17
- Configuration : *streaming* (rôle, droits, slots)

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

Description :

`pg_basebackup` est un produit qui a beaucoup évolué dans les dernières versions de PostgreSQL. De plus, le paramétrage par défaut le rend immédiatement utilisable.

Il permet de réaliser toute la sauvegarde de l'instance, à distance, via deux connexions de réplication : une pour les données, une pour les journaux de transactions qui sont générés pendant la copie. Sa compression permet d'éviter une durée de transfert ou une place disque occupée trop importante. Cela a évidemment un coût, notamment au niveau CPU, sur le serveur ou sur le client suivant le besoin. Il est simple à mettre en place et à utiliser, et permet d'éviter de nombreuses étapes que nous verrons par la suite.

Par contre, il ne permet pas de réaliser une sauvegarde incrémentale avant PostgreSQL 17. À partir de cette version, des sauvegardes incrémentales faite avec `pg_basebackup` peuvent se combiner avec un outil nommé `pg_combinebackup`, mais les fonctionnalités sont encore un peu sommaire.

Il ne permet pas non plus de continuer à archiver les journaux, contrairement aux outils de PITR classiques. Cependant, ceux-ci peuvent l'utiliser pour réaliser la première copie des fichiers d'une instance (c'est le cas de barman, notamment).

Mise en place :

`pg_basebackup` nécessite des connexions de réplication. Il peut utiliser un slot de réplication, une technique qui fiabilise la sauvegarde ou la réplication en indiquant à l'instance quels journaux elle doit conserver. Par défaut, tout est en place pour une connexion en local :

```
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
```

Ensuite, il faut configurer le fichier `pg_hba.conf` pour accepter la connexion du serveur où est exécutée `pg_basebackup`. Dans notre cas, il s'agit du même serveur avec un utilisateur dédié :

```
host  replication  sauve  127.0.0.1/32  scram-sha-256
```

Enfin, il faut créer un utilisateur dédié à la réplication (ici `sauve`) qui sera le rôle créant la connexion et lui attribuer un mot de passe :

```
CREATE ROLE sauve LOGIN REPLICATION;
\password sauve
```

Dans un but d'automatisation, le mot de passe finira souvent dans un fichier `.pgpass` ou équivalent.

Il ne reste plus qu'à :

- lancer `pg_basebackup`, ici en lui demandant une archive au format `tar` ;
- archiver les journaux en utilisant une connexion de réplication par *streaming* ;
- forcer le *checkpoint*.

Cela donne la commande suivante, ici pour une sauvegarde en local :

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/

644320/644320 kB (100%), 1/1 tablespace
```

Le résultat est ici un ensemble des deux archives : les journaux sont à part et devront être dépaquetés dans le `pg_wal` à la restauration.

```
$ ls -l /var/lib/postgresql/backups/
total 4163772
-rw------- 1 postgres postgres 659785216 Oct  9 11:37 base.tar
-rw------- 1 postgres postgres  16780288 Oct  9 11:37 pg_wal.tar
```

La cible doit être vide. En cas d'arrêt avant la fin, il faudra tout recommencer de zéro, c'est une limite de l'outil.

Restauration :

Pour restaurer, il suffit de remplacer le PGDATA corrompu par le contenu de l'archive, ou de créer une nouvelle instance et de remplacer son PGDATA par cette sauvegarde. Au démarrage, l'instance

repérera qu'elle est une sauvegarde restaurée et réappliquera les journaux. L'instance contiendra les données telles qu'elles étaient à la **fin** du `pg_basebackup`.

Noter que les fichiers de configuration ne sont PAS inclus s'ils ne sont pas dans le PGDATA, notamment sur Debian et ses versions dérivées.

Différences entre les versions :

Un slot de réplication temporaire sera créé par défaut pour garantir que le serveur gardera les journaux jusqu'à leur copie intégrale.

La commande `pg_basebackup` crée un fichier manifeste contenant la liste des fichiers sauvegardés, leur taille et une somme de contrôle. Cela permet de vérifier la sauvegarde avec l'outil `pg_verifybackup` (ce dernier ne fonctionne hélas que sur une sauvegarde au format `plain`, ou décompressée).

Lisez bien la documentation de pg_basebackup[2] pour votre version précise de PostgreSQL, des options ont changé de nom au fil des versions.

Même avec un serveur un peu ancien, il est possible d'utiliser un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

L'outil est développé plus en détail dans notre module I4[3].

[2] https://docs.postgresql.fr/current/app-pgbasebackup.html
[3] https://dali.bo/i4_html

14.5 SAUVEGARDE PITR

14.5.1 Étapes d'une sauvegarde PITR

2 étapes :

- Archivage des journaux de transactions
  - archivage interne
  - ou outil `pg_receivewal`
- Sauvegarde des fichiers
  - `pg_basebackup`
  - ou manuellement (outils de copie classiques)

Même si la mise en place est plus complexe qu'un `pg_dump`, la sauvegarde PITR demande peu d'étapes. La première chose à faire est de mettre en place l'archivage des journaux de transactions. Un choix est à faire entre un archivage classique et l'utilisation de l'outil `pg_receivewal`.

Lorsque cette étape est réalisée (et fonctionnelle), il est possible de passer à la seconde : la sauvegarde des fichiers. Là-aussi, il y a différentes possibilités : soit manuellement, soit `pg_basebackup`, soit son propre script ou un outil extérieur.

14.5.2 Méthodes d'archivage

- Deux méthodes :
  - processus interne `archiver`
  - outil `pg_receivewal` (flux de réplication)

La méthode historique, et la plus utilisée toujours, utilise le processus `archiver`. Ce processus fonctionne sur l'instance sauvegardée et fait partie des processus du serveur PostgreSQL. Seule sa (bonne) configuration incombe au DBA, notamment le paramètre `archive_command`.

Une autre méthode existe : `pg_receivewal`. Cet outil livré aussi avec PostgreSQL se comporte comme un serveur secondaire, tournant sur un autre serveur. Il reconstitue les journaux de transactions à partir du flux de réplication.

Chaque solution a ses avantages et inconvénients.

14.5.3 Choix du répertoire d'archivage

- À faire quelle que soit la méthode d'archivage
- Attention aux droits d'écriture dans le répertoire
  - la commande configurée pour la copie doit pouvoir écrire dedans
  - et potentiellement y lire

Dans le cas de l'archivage historique, le serveur PostgreSQL va exécuter une commande, dont le rôle sera de copier les journaux à l'extérieur de son répertoire de travail :

- sur un disque différent du même serveur (à éviter) ;
- sur un disque d'un autre serveur (montage réseau, transfert par SSH, bucket S3…) ;
- sur des bandes…

L'exemple pris ici utilise le répertoire `/mnt/nfs1/archivage` comme répertoire de copie. Ce répertoire est en fait un montage NFS. Il faut donc commencer par créer ce répertoire et s'assurer que l'utilisateur système **postgres** peut écrire dedans :

```
# mkdir /mnt/nfs1/archivage
# chown postgres:postgres /mnt/nfs1/archivage
```

Dans le cas de l'archivage avec `pg_receivewal`, c'est cet outil qui va écrire les journaux dans un répertoire de travail. Cette écriture ne peut se faire qu'en local. Cependant, le répertoire peut se trouver dans un montage réseau (NFS…).

14.5.4 Processus archiver : configuration (1/4)

Préalables :

- Dans `postgresql.conf` :
  - `wal_level = replica`
  - `archive_mode = on` (ou `always`)

Après avoir créé le répertoire d'archivage, il faut configurer PostgreSQL pour lui indiquer comment archiver.

Niveau d'archivage :

La valeur par défaut de `wal_level` est adéquate :

```
wal_level = replica
```

Ce paramètre indique le niveau des informations écrites dans les journaux de transactions. Avec un niveau `minimal`, les journaux ne servent qu'à garantir la cohérence des fichiers de données en cas de crash. Dans le cas d'un archivage, il faut écrire plus d'informations, d'où la nécessité du niveau `replica` (qui est celui par défaut). Le niveau `logical`, nécessaire à la réplication logique[4], convient également.

Mode d'archivage :

Il s'active ainsi sur une instance seule ou primaire :

```
archive_mode = on
```

(La valeur `always` permet d'archiver depuis un secondaire. Avec `on`, l'instance n'archive les journaux que si elle est primaire.) Le changement nécessite un redémarrage.

14.5.5 Processus archiver : configuration (2/4)

La commande d'archivage :

- Dans `postgresql.conf` :
  - `archive_command = '… une commande …'`
  - ou : `archive_library = '… une bibliothèque …'` (v15+)

Enfin, une commande d'archivage doit être définie par le paramètre `archive_command`. `archive_command` sert à archiver un seul fichier à chaque appel.

PostgreSQL l'appelle une fois pour chaque fichier WAL, impérativement dans l'ordre des fichiers. En cas d'échec, elle est répétée indéfiniment jusqu'à réussite, avant de passer à l'archivage du fichier suivant.

(À noter qu'à partir de la version 15, il existe une alternative, avec l'utilisation du paramètre `archive_library`. Il est possible d'indiquer une bibliothèque partagée qui fera ce travail d'archivage. Une telle bibliothèque, écrite en C, devrait être plus puissante et performante. Un module basique est fourni avec PostgreSQL : basic_archive[5]. Notre blog présente un exemple fonctionnel de module d'archivage[6] utilisant une extension en C pour compresser les journaux de transactions. En

[4] https://dali.bo/w5_html
[5] https://docs.postgresql.fr/current/basic-archive.html
[6] https://blog.dalibo.com/2023/07/28/hackingpg2.html

production, il vaudra mieux utiliser une bibliothèque fournie par un outil PITR reconnu. Cependant, à notre connaissance (en décembre 2024), aucun outil n'utilise encore cette fonctionnalité, qui est sans doute plutôt utilisée par des opérateurs *cloud*. L'utilisation simultanée de `archive_command` et `archive_library` est déconseillée, et interdite depuis PostgreSQL 16.)

14.5.6 Processus archiver : configuration (3/4)

- Exemples d'`archive_command` :

```
archive_command='cp %p /mnt/nfs1/archivage/%f && sync /mnt/nfs1/'
archive_command='test ! -f /arch/%f && cp %p /arch/%f'
archive_command='/usr/bin/rsync -az %p postgres@10.9.8.7:/archives/%f'
archive_command='/opt/mon_script.sh %p %f'
archive_command='/usr/bin/pgbackrest --stanza=prod archive-push %p'
archive_command='/usr/bin/barman-wal-archive backup prod %p'
archive_command='/bin/true'  # désactivation
```

- Ne pas oublier de forcer l'écriture de l'archive sur disque
- Code retour de l'archivage entre 0 (ok) et 125

PostgreSQL laisse le soin à l'administrateur de définir la méthode d'archivage des journaux de transactions suivant son contexte. Si vous utilisez un outil de sauvegarde, la commande vous sera probablement fournie. Une simple commande de copie suffit dans la plupart des cas. La directive `archive_command` peut alors être positionnée comme suit :

```
archive_command = 'cp %p /mnt/nfs1/archivage/%f'
```

Le joker `%p` est remplacé par le chemin complet vers le journal de transactions à archiver, par exemple `pg_wal/00000001000000A900000065`. Le joker `%f` correspond au nom du journal de transactions une fois archivé, par exemple `00000001000000A900000065`. La commande réellement exécutée ressemblera donc à ceci :

```
cp pg_wal/00000001000000A900000065 /mnt/nfs1/archivage/00000001000000A900000065
```

En toute rigueur, une copie du fichier ne suffit pas. Par exemple, dans le cas de la commande `cp`, le nouveau fichier n'est pas immédiatement écrit sur disque. La copie est effectuée dans le cache disque du système d'exploitation. En cas de crash juste après la copie, il est tout à fait possible de perdre l'archive. Il est donc essentiel d'ajouter une étape de synchronisation du cache sur disque (ordre `sync`).

La commande d'archivage suivante est donnée dans la documentation officielle à titre d'exemple :

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p
↪  /mnt/server/archivedir/%f'
```

Cette commande a deux inconvénients. Elle ne garantit pas que les données seront synchronisées sur disque. Et si le fichier existe ou a été copié partiellement à cause d'une erreur précédente, la copie ne s'effectuera pas.

Cette protection est une bonne chose. Cependant, il faut être vigilant lorsque l'on rétablit le fonctionnement de l'*archiver* suite à un incident ayant provoqué des écritures partielles dans le répertoire d'archive, comme une saturation de l'espace disque.

Il est aussi possible de placer dans `archive_command` le nom d'un script bash, perl ou autre. L'intérêt est de pouvoir faire plus qu'une simple copie. On peut y ajouter la demande de synchronisation du cache sur disque, ou de la gestion d'erreur plus complexe. Il peut aussi être intéressant de tracer l'action de l'archivage, ou de compresser le journal avant archivage.

Dans vos commandes et scripts, il faut s'assurer d'une chose : la commande d'archivage doit retourner 0 en cas de réussite et surtout une valeur différente de 0 en cas d'échec. Si le code retour de la commande est compris entre 1 et 125, PostgreSQL va tenter périodiquement d'archiver le fichier jusqu'à ce que la commande réussisse (autrement dit, renvoie 0).
Tant qu'un fichier journal n'est pas considéré comme archivé avec succès, PostgreSQL ne le supprimera ou recyclera pas ! Il ne cherchera pas non plus à archiver les fichiers suivants.

De plus si le code retour de la commande est supérieur à 125, le processus `archiver` redémarrera, et l'erreur ne sera pas comptabilisée dans la vue `pg_stat_archiver` !
Ce cas de figure inclut les erreurs de type `command not found` associées aux codes retours 126 et 127, ou le cas de `rsync`, qui renvoie un code retour 255 en cas d'erreur de syntaxe ou de configuration du ssh.

Il est donc important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur. Les causes d'échec sont nombreuses : problème réseau, montage inaccessible, erreur de paramétrage de l'outil, droits insuffisants ou expirés, génération de journaux trop rapide…

À titre d'exemple encore, les commandes fournies par pgBackRest ou barman ressemblent à ceci :

```
# pgBackRest
archive_command='/usr/bin/pgbackrest --stanza=prod archive-push %p'

# barman
archive_command='/usr/bin/barman-wal-archive backup prod %p'
```

Enfin, le paramétrage suivant archive « dans le vide ». Cette astuce est utilisée lors de certains dépannages, ou pour éviter le redémarrage que nécessiterait la désactivation de `archive_mode`.

```
archive_mode = on
archive_command = '/bin/true'
```

14.5.7 Processus archiver : configuration (4/4)

- Dans `postgresql.conf` (suite) :
  - période maximale entre deux archivages
  - `archive_timeout = '… min'`

Si l'activité en écriture est très réduite en volume, il peut se passer des heures entre deux archivages de journaux. Il est alors conseillé de forcer un archivage périodique, même si le journal n'a pas été rempli complètement, en indiquant un délai maximum entre deux archivages :

```
archive_timeout = '5min'
```

(La valeur par défaut, 0, désactive ce comportement.)

Ainsi, la perte de données maximale sera de cette durée.

Comme la taille d'un fichier journal, même incomplet, reste fixe (16 Mo par défaut), la consommation en terme d'espace disque sera plus importante (la compression par l'outil d'archivage peut compenser cela), et le temps de restauration plus long.

14.5.8 Processus archiver : lancement

- Redémarrage de PostgreSQL
  - si modification de `wal_level` et/ou `archive_mode`
- ou rechargement de la configuration

Il ne reste plus qu'à indiquer à PostgreSQL de recharger sa configuration pour que l'archivage soit en

place (avec `SELECT pg_reload_conf();` ou la commande `reload` adaptée au système). Dans le cas où il a fallu définir `wal_level = replica` ou `archive_mode = on`, il faudra relancer PostgreSQL.

14.5.9 Processus archiver : supervision

- Vue `pg_stat_archiver`
- `pg_wal/archive_status/`
  - fichiers `.ready` et `.done`
- Archivage dans l'ordre des fichiers
- Taille de `pg_wal`
  - si saturation : Arrêt !
- Traces

PostgreSQL archive les journaux impérativement dans l'ordre où ils ont été générés.

S'il y a un problème d'archivage d'un journal, les suivants ne seront pas archivés non plus, et vont s'accumuler dans `pg_wal` ! De plus, une saturation de la partition portant `pg_wal` mènera à l'arrêt de l'instance PostgreSQL !

La supervision se fait de quatre manières complémentaires.

Taille :

Si le répertoire `pg_wal` commence à grossir fortement, c'est que PostgreSQL n'arrive plus à recycler ses journaux de transactions : c'est un indicateur d'une commande d'archivage n'arrivant pas à faire son travail pour une raison ou une autre. Ce peut être temporaire si l'archivage est juste lent. Les causes classiques sont un réseau saturé, une compression des journaux trop lente, ou des écritures trop intenses. Si l'archivage est complètement bloqué (à cause d'un disque saturé par exemple), ce répertoire grossira indéfiniment.

Vue pg_stat_archiver :

La vue système `pg_stat_archiver` indique les derniers journaux archivés et les dernières erreurs. Dans l'exemple suivant, il y a eu un problème pendant quelques secondes, d'où 6 échecs, avant que l'archivage reprenne :

```
SELECT * FROM pg_stat_archiver \gx
```

```
-[ RECORD 1 ]------+-----------------------------
archived_count     | 156
last_archived_wal  | 000000020000000100000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count       | 6
last_failed_wal    | 000000020000000100000D7
last_failed_time   | 2020-01-17 18:24:24.463038+00
stats_reset        | 2020-01-17 16:08:37.980214+00
```

Comme dit plus haut, pour que cette supervision soit fiable, la commande exécutée doit renvoyer un code retour inférieur ou égal à 125. Dans le cas contraire, le processus `archiver` redémarre et l'erreur n'apparaît pas dans la vue !

> L'ordre `SELECT pg_switch_wal()` force un changement de journal, et donc l'archivage du journal en cours, à condition qu'il y ait eu une activité minimale. Cette commande est pratique pour tester.

Traces :

On trouvera la sortie et surtout les messages d'erreurs du script d'archivage dans les traces (qui dépendent bien sûr du script utilisé) :

```
2020-01-17 18:24:18.427 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:18.427 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/000000020000000100000D7 /opt/pgsql/archives/000000020000000100000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:19.456 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:19.456 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/000000020000000100000D7 /opt/pgsql/archives/000000020000000100000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:20.463 UTC [15431] LOG:  archive command failed with exit code 3
```

C'est donc le premier endroit à regarder en cas de souci ou lors de la mise en place de l'archivage.

pg_wal/archive_status :

Enfin, on peut monitorer les fichiers présents dans `pg_wal/archive_status`. Les fichiers `.ready`, de taille nulle, indiquent en permanence quels sont les journaux prêts à être archivés. Théoriquement, leur nombre doit donc rester faible et retomber rapidement à 0 ou 1. Le service `ready_archives` de la sonde Nagios check_pgactivity[7] se base sur ce répertoire.

```
SELECT * FROM pg_ls_dir ('pg_wal/archive_status') ORDER BY 1;

           pg_ls_dir
-------------------------------
```

[7] https://github.com/OPMDG/check_pgactivity

```
000000020000000100000DE.done
000000020000000100000DF.done
000000020000000100000E0.done
000000020000000100000E1.ready
000000020000000100000E2.ready
000000020000000100000E3.ready
000000020000000100000E4.ready
000000020000000100000E5.ready
000000020000000100000E6.ready
00000002.history.done
```

14.5.10 pg_receivewal

- Archivage via le protocole de réplication
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
  - toujours au plus près du primaire
- Slots de réplication obligatoires

`pg_receivewal` est un outil permettant de se faire passer pour un serveur secondaire utilisant la réplication en flux (*streaming replication*) dans le but d'archiver en continu les journaux de transactions. Il fonctionne habituellement sur un autre serveur, où seront archivés les journaux. C'est une alternative à l' `archiver`.

Comme il utilise le protocole de réplication, les journaux archivés ont un retard bien inférieur à celui induit par la configuration du paramètre `archive_command` ou du paramètre `archive_library`, les journaux de transactions étant écrits au fil de l'eau avant d'être complets. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le serveur primaire. On peut même utiliser une réplication synchrone (paramètres `synchronous_commit` et `synchronous_standby_names`) pour ne perdre aucune transaction, si l'on accepte un impact certain sur la latence des transactions.

Cet outil utilise les mêmes options de connexion que la plupart des outils PostgreSQL, avec en plus l'option `-D` pour spécifier le répertoire où sauvegarder les journaux de transactions. L'utilisateur spécifié doit bien évidemment avoir les attributs `LOGIN` et `REPLICATION`.

Comme il s'agit de conserver toutes les modifications effectuées par le serveur dans le cadre d'une sauvegarde permanente, il est nécessaire de s'assurer qu'on ne puisse pas perdre des journaux de transactions. Il n'y a qu'un seul moyen pour cela : utiliser la technologie des slots de réplication. En utilisant un slot de réplication, `pg_receivewal` s'assure que le serveur ne va pas recycler des journaux dont `pg_receivewal` n'aurait pas reçu les enregistrements. On retrouve donc le risque d'accumulation des journaux sur le serveur principal si `pg_receivewal` ne fonctionne pas.

Voici l'aide de cet outil en v15 :

```
$ pg_receivewal --help
pg_receivewal reçoit le flux des journaux de transactions PostgreSQL.

Usage :
  pg_receivewal [OPTION]...

Options :
  -D, --directory=RÉPERTOIRE     reçoit les journaux de transactions dans ce
                                 répertoire
  -E, --endpos=LSN               quitte après avoir reçu le LSN spécifié
      --if-not-exists            ne pas renvoyer une erreur si le slot existe
                                 déjà lors de sa création
  -n, --no-loop                  ne boucle pas en cas de perte de la connexion
      --no-sync                  n'attend pas que les modifications soient
                                 proprement écrites sur disque
  -s, --status-interval=SECS     durée entre l'envoi de paquets de statut au
                                 (par défaut 10)
  -S, --slot=NOMREP              slot de réplication à utiliser
      --synchronous              vide le journal de transactions immédiatement
                                 après son écriture
  -v, --verbose                  affiche des messages verbeux
  -V, --version                  affiche la version puis quitte
  -Z, --compress=METHOD[:DETAIL]
                                 compresse comme indiqué
  -?, --help                     affiche cette aide puis quitte

Options de connexion :
  -d, --dbname=CHAÎNE_CONNEX     chaîne de connexion
  -h, --host=HÔTE                hôte du serveur de bases de données ou
                                 répertoire des sockets
  -p, --port=PORT                numéro de port du serveur de bases de données
  -U, --username=UTILISATEUR     se connecte avec cet utilisateur
  -w, --no-password              ne demande jamais le mot de passe
  -W, --password                 force la demande du mot de passe (devrait
                                 survenir automatiquement)

Actions optionnelles :
      --create-slot              crée un nouveau slot de réplication
                                 (pour le nom du slot, voir --slot)
      --drop-slot                supprime un nouveau slot de réplication
                                 (pour le nom du slot, voir --slot)

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
Page d'accueil de PostgreSQL : <https://www.postgresql.org/>
```

`pg_receivewal` est utilisé par exemple par l'outil de sauvegarde PITR barman. Les auteurs de pgBackRest préfèrent utiliser `archive_command` car ils peuvent ainsi mieux paralléliser des débits élevés.

14.5.11 pg_receivewal - configuration serveur

- `postgresql.conf` :

```
# configuration  par défaut
max_wal_senders = 10
max_replication_slots = 10
```

- `pg_hba.conf` :

```
host  replication  repli_user  192.168.0.0/24  scram-sha-256
```

- Utilisateur de réplication :

```
CREATE ROLE repli_user LOGIN REPLICATION PASSWORD 'supersecret'
```

Le paramètre `max_wal_senders` indique le nombre maximum de connexions de réplication sur le serveur. Logiquement, une valeur de 1 serait suffisante, mais il faut compter sur quelques soucis réseau qui pourraient faire perdre la connexion à `pg_receivewal` sans que le serveur primaire n'en soit mis au courant, et du fait que certains autres outils peuvent utiliser la réplication. `max_replication_slots` indique le nombre maximum de slots de réplication. Pour ces deux paramètres, le défaut est 10 et suffit dans la plupart des cas.

Si l'on modifie un de ces paramètres, il est nécessaire de redémarrer le serveur PostgreSQL.

Les connexions de réplication nécessitent une configuration particulière au niveau des accès. D'où la modification du fichier `pg_hba.conf`. Le sous-réseau (192.168.0.0/24) est à modifier suivant l'adressage utilisé. Il est d'ailleurs préférable de n'indiquer que le serveur où est installé `pg_receivewal` (plutôt que l'intégralité d'un sous-réseau).

L'utilisation d'un utilisateur de réplication n'est pas obligatoire mais fortement conseillée pour des raisons de sécurité.

14.5.12 pg_receivewal - redémarrage du serveur

- Redémarrage de PostgreSQL
- Slot de réplication

```
SELECT pg_create_physical_replication_slot('archivage');
```

Enfin, nous devons créer le slot de réplication qui sera utilisé par `pg_receivewal`. La fonction `pg_create_physical_replication_slot()` est là pour ça. Il est à noter que la liste des slots est disponible dans le catalogue système `pg_replication_slots`.

14.5.13 pg_receivewal - lancement de l'outil

- Exemple de lancement

```
pg_receivewal -D /data/archives -S archivage
```

- Journaux créés en temps réel dans le répertoire de stockage
- Mise en place d'un script de démarrage
- S'il n'arrive pas à joindre le serveur primaire
  - Au démarrage de l'outil : `pg_receivewal` s'arrête
  - En cours d'exécution : `pg_receivewal` tente de se reconnecter
- Nombreuses options

On peut alors lancer `pg_receivewal` :

```
pg_receivewal -h 192.168.0.1 -U repli_user -D /data/archives -S archivage
```

Les journaux de transactions sont alors créés en temps réel dans le répertoire indiqué (ici, `/data/archives`) :

```
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000000E*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000000F*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000010.partial*
```

En cas d'incident sur le serveur primaire, il est alors possible de partir d'une sauvegarde physique et de rejouer les journaux de transactions disponibles (sans oublier de supprimer l'extension `.partial` du dernier journal).

Il faut mettre en place un script de démarrage pour que `pg_receivewal` soit redémarré en cas de redémarrage du serveur.

14.5.14 Avantages et inconvénients

- Méthode archiver
  - simple à mettre en place
  - perte au maximum d'un journal de transactions
- Méthode `pg_receivewal`
  - mise en place plus complexe
  - perte minimale voire nulle

La méthode archiver est la méthode la plus simple à mettre en place. Elle est gérée intégralement par PostgreSQL, donc il n'est pas nécessaire de créer et installer un script de démarrage. Cependant, un journal de transactions n'est archivé que quand PostgreSQL l'ordonne, soit parce qu'il a rempli le journal en question, soit parce qu'un utilisateur a forcé un changement de journal (avec la fonction `pg_switch_wal` ou suite à un `pg_backup_stop`), soit parce que le délai maximum entre deux archivages a été dépassé (paramètre `archive_timeout`). Il est donc possible de perdre un grand nombre de transactions (même si cela reste bien inférieur à la perte qu'une restauration d'une sauvegarde logique occasionnerait).

La méthode `pg_receivewal` est plus complexe à mettre en place. Il faut exécuter ce démon, généralement sur un autre serveur. Un script de démarrage doit donc être configuré. Par contre, elle a le gros avantage de ne perdre pratiquement aucune transaction, surtout en mode synchrone. Les enregistrements de transactions sont envoyés en temps réel à `pg_receivewal`. Ce dernier les place dans un fichier de suffixe `.partial`, qui est ensuite renommé pour devenir un journal de transactions complet.

14.6 SAUVEGARDE PITR MANUELLE

14.7 ÉTAPES D'UNE SAUVEGARDE PITR MANUELLE

- 3 étapes :
  - fonction de démarrage
  - copie des fichiers par outil externe
  - fonction d'arrêt
- Exclusive : simple… & obsolète ! (< v15)
- Concurrente : plus complexe à scripter
- Aucun impact pour les utilisateurs ; pas de verrou
- Préférer des outils dédiés qu'un script maison

Une fois l'archivage en place, une sauvegarde à chaud a lieu en trois temps :

- l'appel à la fonction de démarrage ;
- la copie elle-même par divers outils externes (PostgreSQL ne s'en occupe pas) ;
- l'appel à la fonction d'arrêt.

La fonction de démarrage s'appelle `pg_backup_start()` à partir de la version 15 mais avait pour nom `pg_start_backup()` auparavant. De la même façon, la fonction d'arrêt s'appelle `pg_backup_stop()` à partir de la version 15, mais `pg_stop_backup()` avant.

La sauvegarde exclusive était une méthode simple, et cela en faisait le choix par défaut. Il suffisait d'appeler les fonctions concernées avant et après la copie des fichiers. Il ne pouvait y en avoir qu'une à la fois. Elle ne fonctionnait que depuis un primaire.

À cause de ces limites et de différents problèmes, , la sauvegarde exclusive est déclarée obsolète depuis la 9.6, et n'est plus disponible depuis la version 15. Même sur les versions antérieures, il est conseillé d'utiliser dès maintenant des scripts utilisant les sauvegardes concurrentes.

Tout ce qui suit ne concerne plus que la sauvegarde concurrente.

La sauvegarde concurrente peut être lancée plusieurs fois en parallèle. C'est utile pour créer des secondaires alors qu'une sauvegarde physique tourne, par exemple. Elle est nettement plus complexe à gérer par script. Elle peut être exécutée depuis un serveur secondaire, ce qui allège la charge sur le primaire.

Pendant la sauvegarde, l'utilisateur ne verra aucune différence (à part peut-être la conséquence d'I/O saturées pendant la copie). Aucun verrou n'est posé. Lectures, écritures, suppression et création de tables, archivage de journaux et autres opérations continuent comme si de rien n'était.

La description du mécanisme qui suit est essentiellement destinée à la compréhension et l'expérimentation. En production, un script maison reste une possibilité, mais préférez des outils dédiés et fiables : pg_basebackup, pgBackRest…

Les sauvegardes manuelles servent cependant encore quand on veut utiliser une sauvegarde par snapshot de partition ou de baie, ou avec `rsync` (car pg_basebackup ne sait pas synchroniser vers une sauvegarde interrompue ou ancienne), et quand les outils conseillés ne sont pas utilisables ou disponibles sur le système.

14.7.1 Sauvegarde manuelle - 1/3 : pg_backup_start

`SELECT pg_backup_start (`

- `un_label` : texte
- `fast` : forcer un checkpoint ?

`)`

L'exécution de `pg_backup_start()` peut se faire depuis n'importe quelle base de données de l'instance.

(Rappelons que pour les versions avant la 15, la fonction s'appelle `pg_start_backup()`. Pour effectuer une sauvegarde non-exclusive avec ces versions, il faudra positionner un troisième paramètre[8] à `false`.)

Le label (le texte en premier argument) n'a aucune importance pour PostgreSQL (il ne sert qu'à l'administrateur, pour reconnaître le backup).

Le deuxième argument est un booléen qui permet de demander un *checkpoint* immédiat, si l'on est pressé et si un pic d'I/O n'est pas gênant. Sinon il faudra attendre souvent plusieurs minutes (selon la configuration du déclenchement du prochain checkpoint, dépendant des paramètres `checkpoint_timeout` et `max_wal_size` et de la rapidité d'écriture imposée par `checkpoint_completion_target`).

La session qui exécute la commande `pg_backup_start()` doit être la même que celle qui exécutera plus tard `pg_backup_stop()`. Nous verrons que cette dernière fonction fournira de quoi créer deux fichiers, qui devront être nommés `backup_label` et `tablespace_map`. Si la connexion est interrompue avant `pg_backup_stop()`, alors la sauvegarde doit être considérée comme invalide.

[8] https://docs.postgresql.fr/14/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-EXCLUSIVE

En plus de rester connectés à la base, les scripts qui gèrent la sauvegarde concurrente doivent donc récupérer et conserver les informations renvoyées par la commande de fin de sauvegarde.

La sauvegarde PITR est donc devenue plus complexe au fil des versions, et il est donc recommandé d'utiliser plutôt pg_basebackup ou des outils la supportant (barman, pgBackRest…).

14.7.2 Sauvegarde manuelle - 2/3 : copie des fichiers

- Cas courant : snapshot
  - cohérence ? redondance ?
- Sauvegarde des fichiers **à chaud**
  - répertoire principal des données
  - tablespaces
- Copie forcément incohérente (la restauration des journaux corrigera)
- `rsync` et autres outils
- Ignorer :
  - `postmaster.pid`, `log`, `pg_wal`, `pg_replslot` et quelques autres
- Ne pas oublier : configuration !

La deuxième étape correspond à la sauvegarde des fichiers. Le choix de l'outil dépend de l'administrateur. Cela n'a aucune incidence au niveau de PostgreSQL.

La sauvegarde doit comprendre aussi les tablespaces si l'instance en dispose.

Snapshot :

Il est possible d'effectuer cette étape de copie des fichiers par snapshot au niveau de la baie, de l'hyperviseur, ou encore de l'OS (LVM, ZFS…).

Un snaphost cohérent, y compris entre les tablespaces, permet théoriquement de réaliser une sauvegarde en se passant des étapes `pg_backup_start()` et `pg_backup_stop()`. La restauration de ce snapshot équivaudra pour PostgreSQL à un redémarrage brutal.
Pour une sauvegarde PITR, il faudra cependant toujours encadrer le snapshot des appels aux fonctions de démarrage et d'arrêt ci-dessus, et c'est généralement ce que font les outils comme Veeam ou Tina. L'utilisation d'un tel outil implique de vérifier qu'il sait gérer les sauvegardes non exclusives pour utiliser PostgreSQL 15 et supérieurs.

Le point noir de la sauvegarde par snapshot est d'être liée au même système matériel que l'instance PostgreSQL (disque, hyperviseur, datacenter…) Une défaillance grave du matériel, ou un bug de la baie, peut donc emporter, corrompre ou bloquer la sauvegarde en même temps que les données originales. La sécurité de l'instance est donc reportée sur celle de l'infrastructure sous-jacente : il vaut mieux que celle-ci soit répliquée sur plusieurs sites. Une copie parallèle, hors infastructure, des données de manière plus classique reste conseillée pour éviter un désastre total, et pour parer à la malveillance.

Copie manuelle :

La sauvegarde se fait à chaud : il est donc possible que pendant ce temps des fichiers changent, disparaissent avant d'être copiés ou apparaissent sans être copiés. Cela n'a pas d'importance en soi car les journaux de transactions corrigeront cela (leur archivage doit donc commencer **avant** le début de la sauvegarde et se poursuivre sans interruption jusqu'à la fin).
Il **faut** s'assurer que l'outil de sauvegarde supporte cela, c'est-à-dire qu'il soit capable de différencier les codes d'erreurs dus à « des fichiers ont bougé ou disparu lors de la sauvegarde » des autres erreurs techniques. `tar` par exemple convient : il retourne 1 pour le premier cas d'erreur, et 2 quand l'erreur est critique. `rsync` est très courant également.

Sur les plateformes Microsoft Windows, peu d'outils sont capables de copier des fichiers en cours de modification. Assurez-vous d'en utiliser un possédant cette fonctionnalité (il existe différents émulateurs des outils GNU sous Windows). Le plus sûr et simple est sans doute de renoncer à une copie manuelle des fichiers et d'utiliser `pg_basebackup`.

Exclusions :

Des fichiers et répertoires sont à ignorer, pour gagner du temps ou faciliter la restauration. Voici la liste exhaustive (disponible aussi dans la documentation officielle[9]) :

- `postmaster.pid`, `postmaster.opts`, `pg_internal.init` ;
- les fichiers de données des tables non journalisées (*unlogged*) ;
- `pg_wal`, ainsi que les sous-répertoires (mais à archiver séparément !) ;
- `pg_replslot` : les slots de réplication seront au mieux périmés, au pire gênants sur l'instance restaurée ;
- `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans` ne doivent pas être copiés (ils contiennent des informations propres à l'instance, ou qui ne survivent pas à un redémarrage) ;
- les fichiers et répertoires commençant par `pgsql_tmp` (fichiers temporaires) ;
- les fichiers autres que les fichiers et les répertoires standards (donc pas les liens symboliques).

[9] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

On n'oubliera pas les fichiers de configuration s'ils ne sont pas dans le PGDATA.

14.7.3 Sauvegarde manuelle - 3/3 : pg_backup_stop

Ne pas oublier !!
`SELECT * FROM pg_backup_stop (`

- `true` : attente de l'archivage

`)`

La dernière étape correspond à l'exécution de la procédure stockée `SELECT * FROM pg_backup_stop()`.

N'oubliez pas d'exécuter `pg_backup_stop()`, de vérifier qu'il finit avec succès et de récupérer les informations qu'il renvoie !
Cet oubli trop courant rend vos sauvegardes inutilisables !

PostgreSQL va alors :

- marquer cette fin de backup dans le journal des transactions (étape capitale pour la restauration) ;
- forcer la finalisation du journal de transactions courant et donc son archivage, afin que la sauvegarde (fichiers + archives) soit utilisable même en cas de crash juste l'appel à la fonction : `pg_backup_stop()` ne rendra pas la main (par défaut) tant que ce dernier journal n'aura pas été archivé avec succès.

La fonction renvoie :

- le *lsn* de fin de backup ;
- un champ destiné au fichier `backup_label` ;
- un champ destiné au fichier `tablespace_map`.

```
SELECT * FROM pg_stop_backup() \gx
```

```
NOTICE:  all required WAL segments have been archived
-[ RECORD 1 ]----------------------------------------------------------------
lsn         | 22/2FE5C788
labelfile   | START WAL LOCATION: 22/2B000028 (file 000000010000000220000002B)+
            | CHECKPOINT LOCATION: 22/2B000060                               +
            | BACKUP METHOD: streamed                                        +
            | BACKUP FROM: master                                            +
            | START TIME: 2019-12-16 13:53:41 CET                            +
```

```
           | LABEL: rr                                                  +
           | START TIMELINE: 1                                          +
           |
spcmapfile | 134141 /tbl/froid                                         +
           | 134152 /tbl/quota                                         +
           |
```

Ces informations se retrouvent aussi dans un fichier `.backup` mêlé aux journaux :

```
# cat /var/lib/postgresql/12/main/pg_wal/000000010000000220000002B.00000028.backup

START WAL LOCATION: 22/2B000028 (file 000000010000000220000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 000000010000000220000002F)
CHECKPOINT LOCATION: 22/2B000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2019-12-16 13:53:41 CET
LABEL: rr
START TIMELINE: 1
STOP TIME: 2019-12-16 13:54:04 CET
STOP TIMELINE: 1
```

Il faudra créer le fichier `tablespace_map` avec le contenu du champ `spcmapfile` :

```
134141 /tbl/froid
134152 /tbl/quota
```

… ce qui n'est pas trivial à scripter.

Ces deux fichiers devront être placés dans la sauvegarde, pour être présent d'entrée dans le PGDATA du serveur restauré.

À partir du moment où `pg_backup_stop()` rend la main, il est possible de restaurer la sauvegarde obtenue puis de rejouer les journaux de transactions suivants en cas de besoin, sur un autre serveur ou sur ce même serveur.

Tous les journaux archivés avant celui précisé par le champ `START WAL LOCATION` dans le fichier `backup_label` ne sont plus nécessaires pour la récupération de la sauvegarde du système de fichiers et peuvent donc être supprimés. Attention, il y a plusieurs compteurs hexadécimaux différents dans le nom du fichier journal, qui ne sont pas incrémentés de gauche à droite.

14.7.4 Sauvegarde de base à chaud : pg_basebackup

Outil de sauvegarde pouvant aussi servir au sauvegarde basique

- Backup de base ici **sans** les journaux :

```
$ pg_basebackup --format=tar --wal-method=none \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

`pg_basebackup` a été décrit plus haut. Il a l'avantage d'être simple à utiliser, de savoir quels fichiers ne pas copier, de fiabiliser la sauvegarde par un slot de réplication. Il ne réclame en général pas de configuration supplémentaire.

Si l'archivage est déjà en place, copier les journaux est inutile (`--wal-method=none`). Nous verrons plus tard comment lui indiquer où les chercher.

L'inconvénient principal de `pg_basebackup` reste son incapacité à reprendre une sauvegarde interrompue ou à opérer une sauvegarde différentielle ou incrémentale, du moins avant PostgreSQL 17.

14.7.5 Fréquence de la sauvegarde de base

- Dépend des besoins
- De tous les jours à tous les mois
- Plus elles sont espacées, plus la restauration est longue
  - et plus le risque d'un journal corrompu ou absent est important

La fréquence dépend des besoins. Une sauvegarde par jour est le plus commun, mais il est possible d'espacer encore plus la fréquence.

Cependant, il faut conserver à l'esprit que plus la sauvegarde est ancienne, plus la restauration sera longue, car un plus grand nombre de journaux seront à rejouer.

14.7.6 Suivi de la sauvegarde de base

- Vue `pg_stat_progress_basebackup`

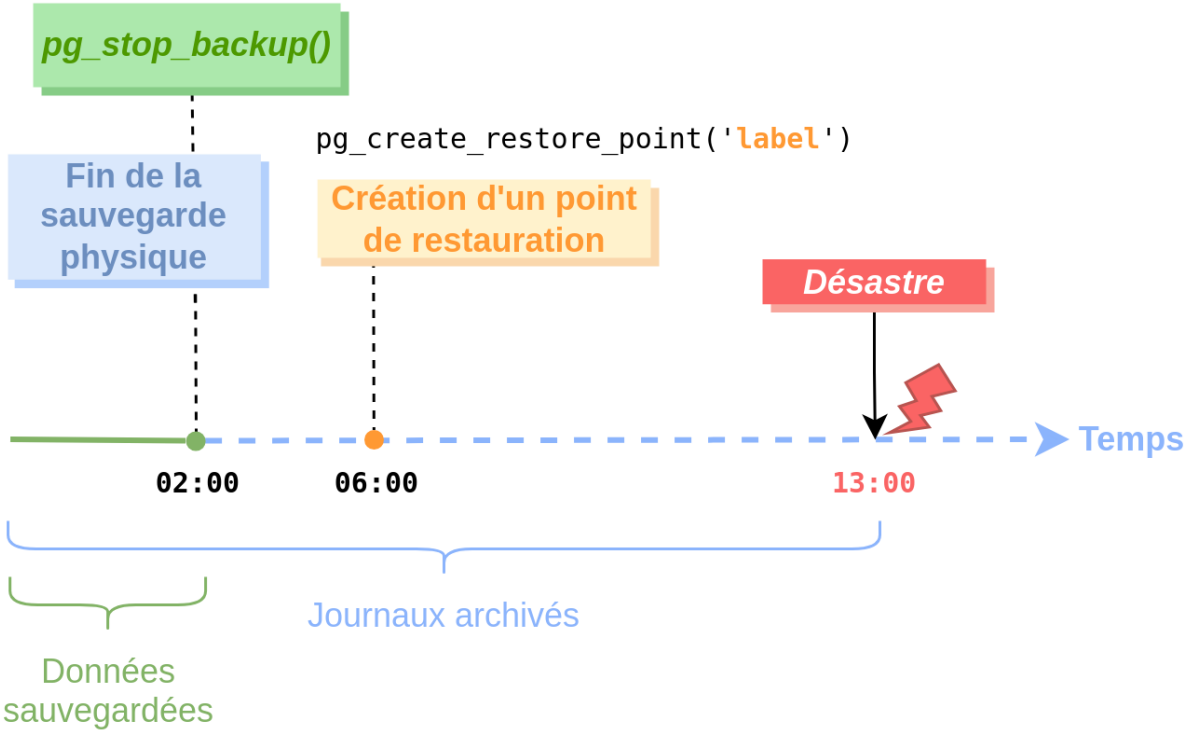
```
SELECT *, pg_size_pretty (backup_total) AS total,
round(100.0*backup_streamed/backup_total::numeric,2) AS "%"
FROM  pg_stat_progress_basebackup  \gx

-[ RECORD 1 ]--------+-------------------------
pid                  | 3608155
phase                | streaming database files
backup_total         | 925114368
backup_streamed      | 197094400
tablespaces_total    | 1
tablespaces_streamed | 0
total                | 882 MB
%                    | 21.30
```

La vue `pg_stat_progress_basebackup` permet de suivre la progression de la sauvegarde de base, quelque soit l'outil utilisé, à condition qu'il passe par le protocole de réplication. Cela permet ainsi de savoir à quelle phase la sauvegarde se trouve, quelle volumétrie a été envoyée, celle à envoyer, etc.

14.8 RESTAURER UNE SAUVEGARDE PITR

Simple, mais à appliquer rigoureusement

14.8.1 Exemple de scénario : sauvegarde



Dans cet exemple, la sauvegarde a fini à 02 h du matin (le moment où une fonction `pg_backup_stop()` est appelée par un outil ou un script). La sauvegarde des fichiers de données s'est effectuée en parallèle de l'archivage des journaux, qui continue indéfiniment ensuite.

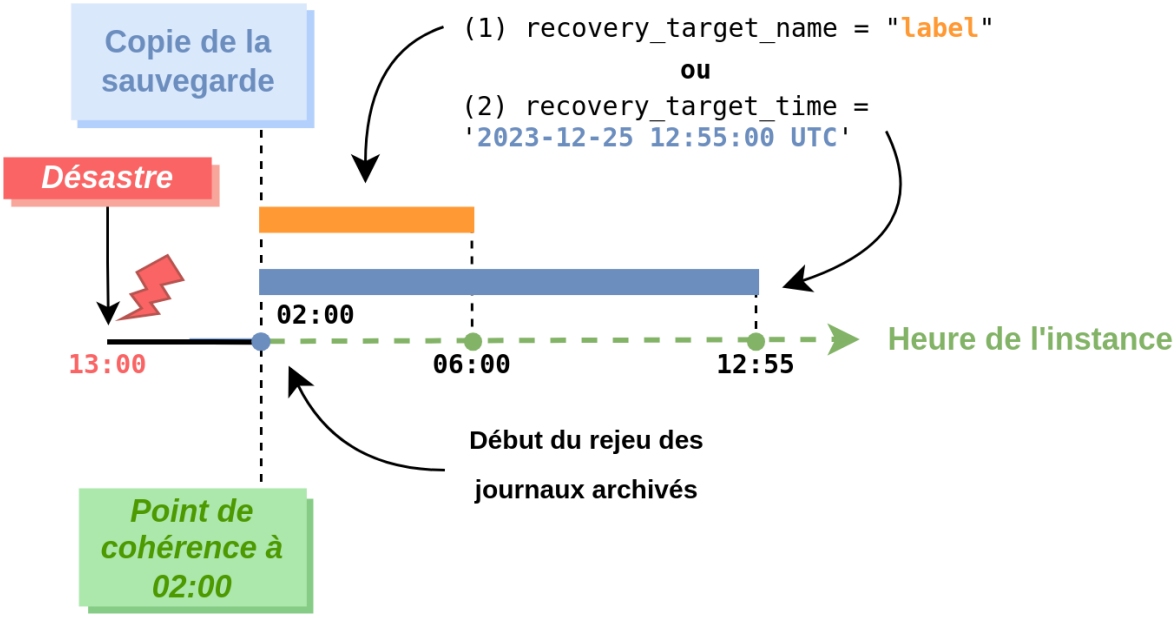
À 06 h, le DBA a créé un « point de restauration », ainsi (le nom est arbitraire) :

```
SELECT pg_create_restore_point ('label');

 pg_create_restore_point
-------------------------
 26/9B000090
```

Ces points de restauration sont totalement optionnels, et peuvent être créés avant certaines opérations (par exemple un batch ou une mise en production), ou périodiquement.

Une catastrophe quelconque frappe à 13 h et il faut restaurer.

14.8.2 Exemple de scénario : restauration



La ligne de temps de ce schéma correspond aux heures des transactions originales.

Pour restaurer, le DBA copie la sauvegarde de base, modifie la configuration et démarre l'instance qui commence à rejouer les journaux. Elle atteint le point de cohérence (correspondant à la fin de la sauvegarde), et est donc dans l'état correspondant à la fin de la sauvegarde, donc comme à 02 h.

Deux possibilités sont montrées ici :

1. le DBA demande à redérouler les journaux jusqu'au point de restauration `label`, pour avoir une image de la base à 06 h ;
2. le DBA demande à redérouler les journaux jusque 12 h 55, juste avant la catastrophe.

Nous verrons qu'il lui suffira de choisir les bons paramètres (ici `recovery_target_name` ou `recovery_target_time`).

14.8.3 Restaurer une sauvegarde PITR (1/5)

- S'il s'agit du même serveur
  - arrêter PostgreSQL
- Nettoyer les répertoires des données
  - y compris les tablespaces
  - sauf outil travaillant en mode delta

La restauration se déroule en trois, voire quatre étapes suivant qu'elle est effectuée sur le même serveur ou sur un autre serveur. Dans le cas où la restauration a lieu sur le même serveur, les étapes préliminaires suivantes sont à effectuer.

Il faut arrêter PostgreSQL s'il n'est pas arrêté. Cela arrive quand la restauration a pour but, par exemple, de récupérer des données qui ont été supprimées par erreur.

Ensuite, il faut supprimer (ou archiver) l'ancien répertoire des données pour pouvoir y placer la sauvegarde des fichiers. Écraser l'ancien répertoire n'est pas suffisant, il faut le supprimer, ainsi que les répertoires des tablespaces au cas où l'instance en possède. (L'exception est l'utilisation d'outils capable de trouver les différence entre les fichiers à restaurer et ceux présents, pour gagner du temps, comme `rsync` ou `pgbackrest restore --delta`.) Cela est valable aussi pour chaque tablespace. Une exception : on peut vouloir mettre de côté le dernier WAL (incomplet) d'une instance que l'on restaure pour ne perdre aucune transaction (voir plus bas).

14.8.4 Restaurer une sauvegarde PITR (2/5)

- Restaurer les fichiers de la sauvegarde
- Peut-être besoin de nettoyer les fichiers restaurés
  - ex : `pg_wal`, `postmaster.pid`, `log/`
  - un bon outil ne les a pas copiés
- Si restauration après crash :
  - récupérer le dernier journal de transactions connu (si disponible)

La sauvegarde des fichiers peut enfin être restaurée. Il faut bien porter attention à ce que les fichiers

soient restaurés au même emplacement, tablespaces compris.

Une fois cette étape effectuée, il peut être intéressant de faire un peu de ménage. Des outils comme pgBackRest ou Barman, ou un bon script, rendent cette étape inutile, car ils n'auront pas copié les fichiers inutiles.

Par exemple, le fichier `postmaster.pid` peut poser un problème au démarrage. On peut supprimer les traces, si elles sont dans `$PGDATA/log/`, pour éviter toute confusion entre l'ancienne et la nouvelle incarnation de l'instance, surtout si on restaure sur une nouvelle machine.

Si des journaux de transactions sont compris dans la sauvegarde (`$PGDATA/pg_wal/`), il est préférable de les supprimer. De toute façon, PostgreSQL les ignorera s'ils sont dans les archives. La commande sera similaire à celle-ci :

```
$ rm postmaster.pid log/* pg_wal/[0-9A-F]*
```

Enfin, si on restaure après un crash, on peut chercher à récupérer le dernier journal en cours lors de l'arrêt, incomplet et qui n'a pu être archivé. S'il n'a pas disparu avec le disque par exemple, le récupérer peut permettre de sauver les transactions qui y figurent. Ce dernier journal sera pris en compte après le rejeu des journaux archivés, après l'échec de la `restore_command` sur ce journal.

14.8.5 Restaurer une sauvegarde PITR (3/5)

- Indiquer qu'on est en restauration
  - fichier vide `recovery.signal`
- Commande de restauration
  - `restore_command = '… une commande …'`
  - directement dans `pg_wal/`
  - dans `postgresql.[auto.]conf`

Quand PostgreSQL démarre après avoir subi un arrêt brutal, il ne restaure que les journaux en place dans `pg_wal/`, puis il s'ouvre en écriture. Pour une restauration, il faut lui indiquer qu'il doit plutôt demander les journaux quelque part, et les rejouer tous jusqu'à épuisement, avant de s'ouvrir. Pour cela, il suffit de créer un fichier vide `recovery.signal` dans le répertoire des données.

Pour la récupération des journaux, le paramètre essentiel est `restore_command`. Il contient une commande symétrique des paramètres `archive_command` (ou `archive_library`) pour l'archivage. Il s'agit d'une commande copiant un journal dans le répertoire des journaux `pg_wal/`. Cette commande est souvent fournie par l'outil de sauvegarde PITR s'il y en a un. Si nous poursuivons notre exemple, ce paramètre pourrait être :

```
restore_command = 'cp /mnt/nfs1/archivage/%f %p'
```

Cette commande est appelée après la restauration de chaque journal pour récupérer le suivant, qui est restauré, et ainsi de suite.

Techniquement, la commande est lancée depuis le PGDATA, en remplaçant `%f` par le nom du journal attendu (par exemple `000000010000000980000000E0`) et `%p` par `pg_wal/RECOVERYXLOG`. Ce dernier fichier sera ensuite renommé avec le nom du journal. On peut également constater la recherche d'un fichier d'historique des *timelines*, par exemple `00000002.history`, sauvé temporairement sous le nom `pg_wal/RECOVERYHISTORY`. PostgreSQL cherche ces fichiers d'historique pour savoir quelle chaîne de journaux suivre quand il y a eu des restaurations ou bascules sur un secondaire (voir plus loin).

Il n'y a aucune parallélisation prévue, mais des outils de sauvegarde PITR peuvent en faire en arrière-plan pendant l'exécution de la commande (par exemple pgBackRest en mode asynchrone).

Si le but est de restaurer tous les journaux archivés, il n'est pas nécessaire d'aller plus loin dans la configuration. La restauration se poursuivra jusqu'à ce que `restore_command` tombe en erreur, ce qui signifie l'épuisement de tous les journaux disponibles, et la fin de la restauration.

Au cas où vous rencontreriez une instance en version 11 ou antérieure : il faut savoir que la restauration se paramétrait dans un fichier texte nommé `recovery.conf`, dans le PGDATA, contenant `recovery_command` et éventuellement les options de restauration.

14.8.6 Restaurer une sauvegarde PITR (4/5)

- Jusqu'où restaurer :
  - `recovery_target_name`, `recovery_target_time`
  - `recovery_target_xid`, `recovery_target_lsn`
  - `recovery_target_inclusive`
- Le backup de base doit être antérieur !
- Suivi de timeline :
  - `recovery_target_timeline` : `latest` (en général)
- Et on fait quoi ?
  - `recovery_target_action` : `pause`
  - `pg_wal_replay_resume` pour ouvrir immédiatement
  - ou modifier & redémarrer

Si l'on ne veut pas simplement restaurer tous les journaux, par exemple pour s'arrêter avant une fausse manipulation désastreuse, plusieurs paramètres permettent de préciser le point d'arrêt :

- jusqu'à un certain nom, grâce au paramètre `recovery_target_name` (le nom correspond à un label enregistré précédemment dans les journaux de transactions grâce à la fonction `pg_create_restore_point()`) ;
- jusqu'à une certaine heure, grâce au paramètre `recovery_target_time` ;
- jusqu'à un certain identifiant de transaction, grâce au paramètre `recovery_target_xid`, numéro de transaction qu'il est possible de chercher dans les journaux eux-mêmes grâce à l'utilitaire `pg_waldump` (voir cet article[10]) ;
- jusqu'à un certain LSN (*Log Sequence Number*[11]), grâce au paramètre `recovery_target_lsn`, que là aussi on doit aller chercher dans les journaux eux-mêmes.

Évidemment, il ne faudra choisir qu'un paramètre parmi ceux-là.

Avec le paramètre `recovery_target_inclusive` (par défaut à `true`), il est possible de préciser si la restauration se fait en incluant les transactions au nom, à l'heure ou à l'identifiant de transaction demandé, ou en les excluant.

Dans les cas complexes, nous verrons plus tard que choisir la *timeline* peut être utile (avec `recovery_target_timeline`, en général à `latest`).

Exemples de paramétrage :

```
recovery_target_name = 'label';

recovery_target_time = '2022-12-31 12:45:00 UTC'

recovery_target_lsn = '0/2000060'

recovery_target_xid = '1100842'
```

Ces restaurations à un moment précis ne sont possibles que si elles correspondent à un état cohérent d'**après** la fin du *base backup*, soit après le moment du `pg_stop_backup`. Si l'on a un historique de plusieurs sauvegardes, il faudra en choisir une antérieure au point de restauration voulu. Ce n'est pas forcément la dernière. Les outils ne sont pas forcément capables de deviner la bonne sauvegarde à restaurer.

Il est possible de demander à la restauration de s'arrêter une fois arrivée au stade voulu avec :

```
recovery_target_action = pause
```

C'est même l'action par défaut si une des options d'arrêt ci-dessus a été choisie : cela permet à l'utilisateur de vérifier que le serveur est bien arrivé au point qu'il désirait. Les alternatives sont `promote` (ouverture en écriture après le rejeu) et `shutdown`.

[10] https://blog.hagander.net/locating-the-recovery-point-just-before-a-dropped-table-230/
[11] https://docs.postgresql.fr/current/datatype-pg-lsn.html

Si la cible est atteinte mais que l'on décide de continuer la restauration jusqu'à un autre point (évidemment postérieur), il faut modifier la cible de restauration dans le fichier de configuration, et **redémarrer** PostgreSQL. C'est le seul moyen de rejouer d'autres journaux sans ouvrir l'instance en écriture.

Si l'on est arrivé au point de restauration voulu, un message de ce genre apparaît :

```
LOG:  recovery stopping before commit of transaction 8693270, time 2021-09-02
↪  11:46:35.394345+02
LOG:  pausing at the end of recovery
HINT:  Execute pg_wal_replay_resume() to promote.
```

(Le terme *promote* pour une restauration est un peu abusif.) `pg_wal_replay_resume()` — malgré ce que pourrait laisser croire son nom ! — provoque ici l'arrêt immédiat de la restauration, donc ignore les opérations contenues dans les WALs que l'on n'a pas souhaités restaurer, puis le serveur s'ouvre en écriture sur une nouvelle timeline.

Attention : jusque PostgreSQL 12 inclus, si un `recovery_target` était spécifié mais n'était toujours *pas* atteint à la fin du rejeu des archives, alors le mode *recovery* se terminait et le serveur était promu sans erreur, et ce, même si `recovery_target_action` avait la valeur `pause` ! (À condition, bien sûr, que le point de cohérence ait tout de même été dépassé.) Il faut donc être vigilant quant aux messages dans le fichier de trace de PostgreSQL !
À partir de PostgreSQL 13, l'instance détecte le problème et s'arrête avec un message `FATAL` : la restauration ne s'est pas déroulée comme attendu. S'il manque juste certains journaux de transactions, cela permet de relancer PostgreSQL après correction de l'oubli.

La documentation officielle complète sur le sujet est sur le site du projet[12].

14.8.7 Restaurer une sauvegarde PITR (5/5)

- Démarrer PostgreSQL
- Rejeu des journaux
- Vérifier que le point de cohérence est atteint !
- Ne jamais effacer `recovery.signal` volontairement

La dernière étape est particulièrement simple. Il suffit de démarrer PostgreSQL. PostgreSQL va comprendre qu'il doit rejouer les journaux de transactions.

[12] https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET

Les éventuels journaux présents sont rejoués, puis `restore_command` est appelé pour fournir d'autres journaux, jusqu'à ce que la commande ne trouve plus rien dans les archives.

Les journaux doivent se dérouler au moins jusqu'à rencontrer le « point de cohérence », c'est-à-dire la mention insérée par `pg_backup_stop()`. Avant ce point, il n'est pas possible de savoir si les fichiers issus du *base backup* sont à jour ou pas, et il est impossible de démarrer l'instance. Le message apparaît dans les traces et, dans le doute, on doit vérifier sa présence :

```
2020-01-17 16:08:37.285 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000031"…
2020-01-17 16:08:37.789 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000032"…
2020-01-17 16:08:37.949 UTC [15221] LOG: consistent recovery state reached
                                         at 1/32BFDD88
2020-01-17 16:08:37.949 UTC [15217] LOG: database system is ready to accept
                                         read only connections
2020-01-17 16:08:38.009 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000033"…
```

Si le message apparaît, le rejeu n'est pas terminé, mais on a au moins Au moment où ce message apparaît, le rejeu n'est pas terminé, mais il a atteint un stade où l'instance est cohérente et utilisable.

PostgreSQL continue ensuite jusqu'à arriver à la limite fixée, jusqu'à ce qu'il ne trouve plus de journal à rejouer (`restore_command` tombe en erreur), ou que le bloc de journal lu soit incohérent (ce qui indique qu'on est arrivé à la fin d'un journal qui n'a pas été terminé, le journal courant au moment du crash par exemple). PostgreSQL vérifie qu'il n'existe pas une *timeline* supérieure sur laquelle basculer (par exemple s'il s'agit de la deuxième restauration depuis la sauvegarde du PGDATA).

Puis il va s'ouvrir en écriture (sauf si vous avez demandé `recovery_target_action = pause`).

```
2020-01-17 16:08:45.938 UTC [15221] LOG: restored log file "000000010000000100000003C"
                                         from archive
2020-01-17 16:08:46.116 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000003D"…
2020-01-17 16:08:46.547 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000003E"…
2020-01-17 16:08:47.262 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000003F"…
2020-01-17 16:08:47.842 UTC [15221] LOG: invalid record length at 1/3F0000A0:
                                         wanted 24, got 0
2020-01-17 16:08:47.842 UTC [15221] LOG: redo done at 1/3F000028
2020-01-17 16:08:47.842 UTC [15221] LOG: last completed transaction was
                                         at log time 2020-01-17 14:59:30.093491+00
2020-01-17 16:08:47.860 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000003F"…
cp: cannot stat '/opt/pgsql/archives/00000002.history': No such file or directory
2020-01-17 16:08:47.966 UTC [15221] LOG:  selected new timeline ID: 2
2020-01-17 16:08:48.179 UTC [15221] LOG:  archive recovery complete
cp: cannot stat '/opt/pgsql/archives/00000001.history': No such file or directory
2020-01-17 16:08:51.613 UTC [15217] LOG:  database system is ready
                                          to accept connections
```

Le fichier `recovery.signal` est effacé pour ne pas poser problème en cas de crash immédiat. (Ne l'effacez jamais manuellement !)

Le fichier `backup_label` d'une sauvegarde exclusive est renommé en `backup_label.old`.

14.8.8 Restauration PITR : durée

- Durée dépendante du nombre de journaux
  - rejeu séquentiel des WAL
- Accéléré en version 15 (*prefetch*)

La durée de la restauration est fortement dépendante du nombre de journaux. Ils sont rejoués séquentiellement. Mais avant cela, un fichier journal peut devoir être récupéré, décompressé, et restauré dans `pg_wal`.

Il est donc préférable qu'il n'y ait pas trop de journaux à rejouer, et donc qu'il n'y ait pas trop d'espaces entre sauvegardes complètes successives.

La version 15 a optimisé le rejeu en permettant l'activation du *prefetch* des blocs de données lors du rejeu des journaux.

Un outil comme pgBackRest en mode asynchrone permet de paralléliser la récupération des journaux, ce qui permet de les récupérer via le réseau et de les décompresser par avance pendant que PostgreSQL traite les journaux précédents.

14.8.9 Restauration PITR : différentes timelines

- Fin de *recovery* => changement de *timeline* :
  - l'historique des données prend une autre voie
  - le nom des WAL change (ex : `000000020000000000000000C`)
  - fichiers `.history`
- Permet plusieurs restaurations PITR à partir du même *basebackup*
- Choix : `recovery_target_timeline`
  - défaut : `latest`

Lorsque le mode *recovery* s'arrête, au point dans le temps demandé ou faute d'archives disponibles, l'instance accepte les écritures. De nouvelles transactions se produisent alors sur les différentes bases

de données de l'instance. Dans ce cas, l'historique des données prend un chemin différent par rapport aux archives de journaux de transactions produites avant la restauration. Par exemple, dans ce nouvel historique, il n'y a pas le `DROP TABLE` malencontreux qui a imposé de restaurer les données. Cependant, cette transaction existe bien dans les archives des journaux de transactions.

On a alors plusieurs historiques des transactions, avec des « bifurcations » aux moments où on a réalisé des restaurations. PostgreSQL permet de garder ces historiques grâce à la notion de *timeline*. Une *timeline* est donc l'un de ces historiques. Elle est identifiée par un numéro et se matérialise par un ensemble de journaux de transactions. Le numéro de la *timeline* est le premier nombre hexadécimal du nom des segments de journaux de transactions, en 8e position (le second est le numéro du journal, et le troisième, à la fin, le numéro du segment). Ainsi, lorsqu'une instance s'ouvre après une restauration PITR, elle peut archiver immédiatement ses journaux de transactions au même endroit, les fichiers ne seront pas écrasés vu qu'ils seront nommés différemment. Par exemple, après une restauration PITR s'arrêtant à un point situé dans le segment `000000010000000000000009` :

```
$ ls -1 /backup/postgresql/archived_wal/
000000010000000000000007
000000010000000000000008
000000010000000000000009
00000001000000000000000A
00000001000000000000000B
00000001000000000000000C
00000001000000000000000D
00000001000000000000000E
00000001000000000000000F
000000010000000000000010
000000010000000000000011
000000020000000000000009
00000002000000000000000A
00000002000000000000000B
00000002000000000000000C
00000002.history
```

Noter les timelines `1` et `2` en 8e position des noms des fichiers. Il y a deux fichiers finissant par `09` : le premier `000000010000000000000009` contient des informations communes aux deux *timelines* mais sa fin ne figure pas dans la *timeline* 2. Les fichiers `00000001000000000000000A` à `000000010000000000000011` contiennent des informations qui ne figurent que dans la *timeline* 1. Les fichiers `00000002000000000000000A` jusque `00000002000000000000000C` sont uniquement dans la *timeline* 2. Le fichier `00000002.history` contient l'information sur la transition entre les deux *timelines*.

Ce fichier sert pendant le *recovery*, quand l'instance doit choisir les *timelines* à suivre et les fichiers à restaurer. Les *timelines* connues avec leur point de départ sont suivies grâce aux fichiers d'historique, nommés d'après le numéro hexadécimal sur huit caractères de la *timeline* et le suffixe `.history`, et archivés avec les journaux. En partant de la *timeline* qu'elle quitte, l'instance restaure les fichiers historiques des *timelines* suivantes pour choisir la première disponible. Une fois la restauration finie, avant de s'ouvrir en écriture, l'instance archive un nouveau fichier `.history` pour la nouvelle *timeline* sélectionnée. Il contient l'adresse du point de départ dans la *timeline* qu'elle quitte, c'est-à-dire le point de bifurcation entre la 1 et la 2 :

```
$ cat 00000002.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
```

Puis l'instance continue normalement, et archive ses journaux commençant par `00000002`.

Après une seconde restauration, repartant de la *timeline* 2, l'instance choisit la *timeline* 3 et écrit un nouveau fichier :

```
$ cat 00000003.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
2   0/105AF7D0  before 2015-10-22 10:25:56.614316+02
```

Ce fichier reprend les *timelines* précédemment suivies par l'instance. En effet, l'enchaînement peut être complexe s'il y a eu plusieurs retours en arrière ou restauration.

À la restauration, on peut choisir la *timeline* cible en configurant le paramètre `recovery_target_timeline`. Il vaut par défaut `latest`, et la restauration suit donc les changements de *timeline* depuis le moment de la sauvegarde.

Pour choisir une autre *timeline* que la dernière, il faut donner le numéro de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. Les timelines permettent ainsi d'effectuer plusieurs restaurations successives à partir du même *base backup*, avec des retours en arrière, et d'archiver vers le même dépôt sans mélanger les journaux.

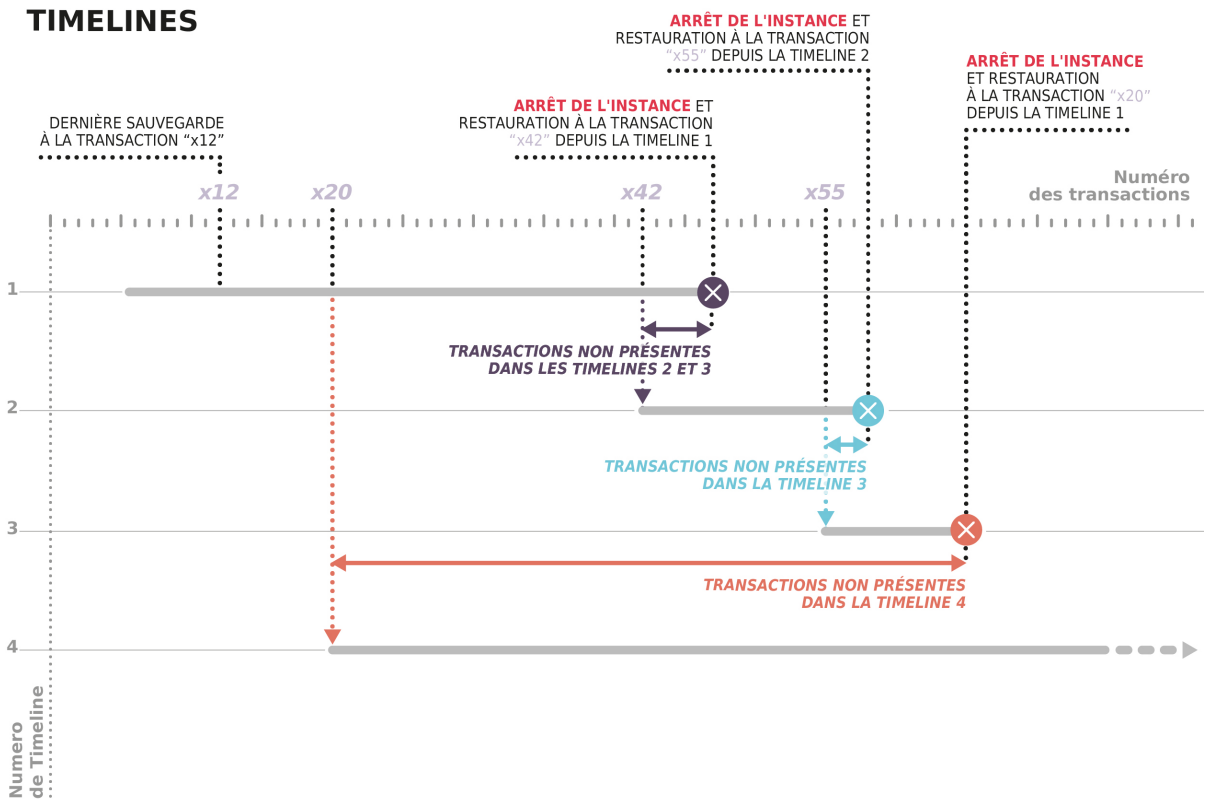
Bien sûr, pour restaurer dans une *timeline* précise, il faut que le fichier `.history` correspondant soit encore présent dans les archives, sous peine d'erreur.

Un changement de *timeline* ne se produit que lors d'une restauration explicite, et pas en cas de *recovery* après crash, notamment. (Cela arrive aussi quand un serveur secondaire est promu : il crée une nouvelle *timeline*.)

Il y a quelques pièges :

- Le numéro de timeline dans les traces, ou affiché par `pg_controldata`, est en décimal. Mais les fichiers `.history` portent un numéro en hexadécimal (par exemple `00000014.history` pour la timeline 20). On peut fournir les deux à `recovery_target_timeline` (`20` ou `'0x14'`). Attention, il n'y a pas de contrôle !
- **Attention sur les anciennes versions** : jusque PostgreSQL 11 compris, la valeur par défaut de `recovery_target_timeline` était `current` : la restauration se faisait donc dans la même *timeline* que le *base backup*. Si entre-temps il y avait eu une bascule ou une précédente restauration, la nouvelle timeline n'était pas automatiquement suivie !

14.8.10 Restauration PITR : illustration des timelines



Ce schéma illustre ce processus de plusieurs restaurations successives, et la création de différentes *timelines* qui en résulte.

On observe ici les éléments suivants avant la première restauration :

- la fin de la dernière sauvegarde se situe en haut à gauche sur l'axe des transactions, à la transaction `x12` ;
- cette sauvegarde a été effectuée alors que l'instance était en activité sur la *timeline* 1.

On décide d'arrêter l'instance alors qu'elle est arrivée à la transaction `x47`, par exemple parce qu'une nouvelle livraison de l'application a introduit un bug qui provoque des pertes de données. L'objectif est de restaurer l'instance avant l'apparition du problème afin de récupérer les données dans un état cohérent, et de relancer la production à partir de cet état. Pour cela, on restaure les fichiers de l'instance à partir de la dernière sauvegarde, puis on modifie le fichier de configuration pour que l'instance, lors de sa phase de *recovery* :

- restaure les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaure les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x42`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la

timeline 2, la bifurcation s'effectuant à la transaction `x42`. L'instance étant de nouveau ouverte en écriture, elle va générer de nouveaux WAL, qui seront associés à la nouvelle *timeline* : ils n'écrasent pas les fichiers WAL archivés de la *timeline* 1, ce qui permet de les réutiliser pour une autre restauration en cas de besoin (par exemple si la transaction `x42` utilisée comme point d'arrêt était trop loin dans le passé, et que l'on désire restaurer de nouveau jusqu'à un point plus récent).

Un peu plus tard, on a de nouveau besoin d'effectuer une restauration dans le passé - par exemple, une nouvelle livraison applicative a été effectuée, mais le bug rencontré précédemment n'était toujours pas corrigé. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, puis on configure PostgreSQL pour suivre la *timeline* 2 (paramètre `recovery_target_timeline = 2`) jusqu'à la transaction `x55`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point de la bifurcation (transaction `x42`) ;
- suivre la *timeline* indiquée (2) et rejouer les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x55`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 3, la bifurcation s'effectuant cette fois à la transaction `x55`.

Enfin, on se rend compte qu'un problème bien plus ancien et subtil a été introduit précédemment aux deux restaurations effectuées. On décide alors de restaurer l'instance jusqu'à un point dans le temps situé bien avant, jusqu'à la transaction `x20`. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, et on configure le serveur pour restaurer jusqu'à la transaction `x20`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point précédant immédiatement l' apparition du bug applicatif (transaction `x20`).

Comme la création des deux *timelines* précédentes est archivée dans les fichiers *history*, l'ouverture de l'instance en écriture va basculer sur une nouvelle *timeline* (4). Suite à cette restauration, toutes les modifications de données provoquées par des transactions effectuées sur la *timeline* 1 après la transaction `x20`, ainsi que celles effectuées sur les *timelines* 2 et 3, ne sont donc pas présentes dans l'instance.

14.8.11 Après la restauration

- Bien vérifier que l'archivage a repris
  - et que les archives des journaux sont complètes
- Ne pas hésiter à reprendre une sauvegarde complète
- Bien vérifier que les secondaires ont suivi

Une fois le nouveau primaire en place, la production peut reprendre, mais il faut vérifier que la sauvegarde PITR est elle aussi fonctionnelle.

Ce nouveau primaire a généralement commencé à archiver ses journaux à partir du dernier journal récupéré de l'ancien primaire, renommé avec l'extension `.partial`, juste avant la bascule sur la nouvelle *timeline*. Il faut bien sûr vérifier que l'archivage des nouveaux journaux fonctionne.

Sur l'ancien primaire, les derniers journaux générés juste avant l'incident n'ont pas forcément été archivés. Ceux-ci possèdent un fichier témoin `.ready` dans `pg_wal/archive_status`. Même s'ils ont été copiés manuellement vers le nouveau primaire avant sa promotion, celui-ci ne les a pas archivés.

Rappelons qu'un « trou » dans le flux des journaux dans le dépôt des archives empêchera la restauration d'aller au-delà de ce point !

Il est possible de forcer l'archivage des fichiers `.ready` depuis l'ancien primaire, avant la bascule, en exécutant à la main les `archive_command` que PostgreSQL aurait générées, mais la facilité pour le faire dépend de l'outil. La copie de journaux à la main est donc risquée.

De plus, s'il y a eu plusieurs restaurations successives, qui ont provoqué quelques archivages et des apparitions de *timelines* dans le même dépôt d'archives, avant d'être abandonnées, il faut faire attention au paramètre `recovery_target_timeline` (`latest` ne convient plus), ce qui complique une future restauration.

Pour faciliter des restaurations ultérieures, il est recommandé de procéder au plus tôt à une sauvegarde complète du nouveau primaire.

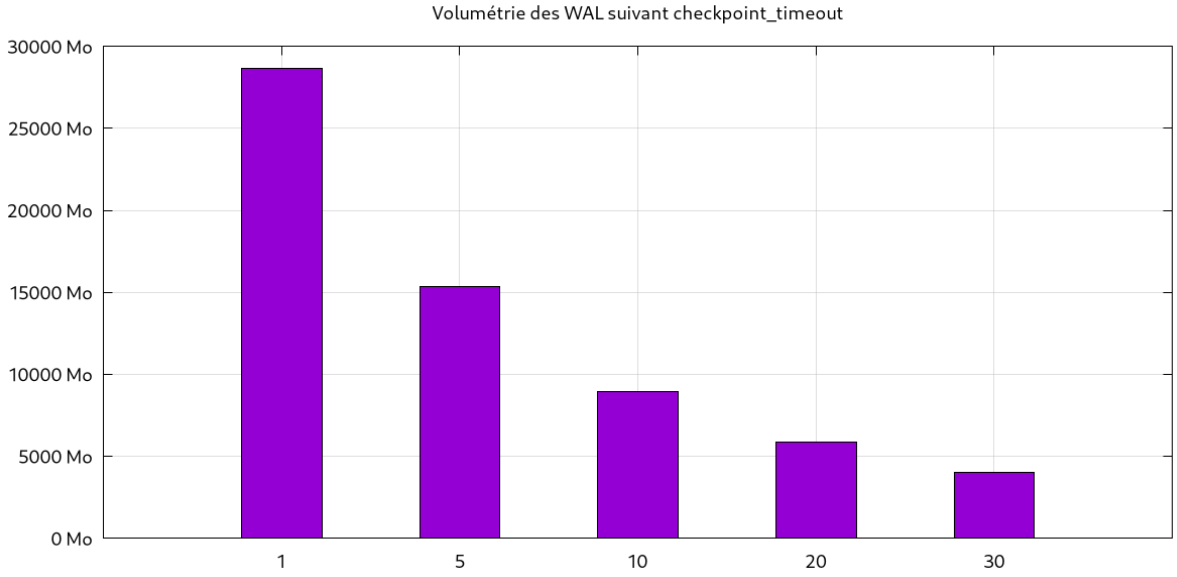
Quant aux éventuelles instances secondaires, il est vraisemblable qu'elles doivent être reconstruites suite à la restauration de l'instance primaire. (Si elles ont appliqué des journaux qui ont été perdus et

n'ont pas été repris par le primaire restauré, ces secondaires ne pourront se raccrocher. Consulter les traces.)

14.9 POUR ALLER PLUS LOIN

- Limiter la volumétrie des journaux sauvegardés
- Quels sont les outils PITR ?

14.9.1 Réduire le nombre de journaux sauvegardés



- Monter
  - `checkpoint_timeout`
  - `max_wal_size`

L'un des problèmes de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Si un journal de 16 Mo (par défaut) est généré toutes les minutes, le total est de 23 Go de journaux par jour, et parfois beaucoup plus. Il n'est pas forcément possible de conserver autant de journaux.

Un premier moyen est de reduire la volumétrie à la source en espaçant les checkpoints. Le graphique ci-dessus représente la volumétrie générée par un simple test avec `pgbench` (OLTP classique donc) avec `checkpoint_timeout` variant entre 1 et 30 minutes : les écarts sont énormes.

La raison est que, pour des raisons de fiabilité, un bloc modifié est intégralement écrit (8 ko) dans les journaux à sa première modification après un checkpoint. Par la suite, seules les modifications de ce bloc, souvent beaucoup plus petites, sont journalisées. (Ce comportement dépend du paramètre `full_page_writes`[13], activé par défaut et qu'il est impératif de laisser tel quel, sauf peut-être sur ZFS.)

Espacer les checkpoints permet d'économiser des écritures de blocs complets, si l'activité s'y prête (en OLTP surtout). Il y a un intérêt en performances, mais surtout en place disque économisée quand les journaux sont archivés, aussi accessoirement en CPU s'ils sont compressés, et en trafic réseau s'ils sont répliqués. Un exemple figure dans ce billet du blog Dalibo[14].

Par cohérence, si l'on monte `checkpoint_timeout`, il faut penser à augmenter aussi `max_wal_size`, et vice-versa. Des valeurs courantes sont respectivement 15 minutes, parfois plus, et plusieurs giga-octets.

Il y a cependant un inconvénient : un écart plus grand entre checkpoints peut allonger la restauration après un arrêt brutal, car il y aura plus de journaux à rejouer, parfois des centaines ou des milliers.

14.9.2 Compresser les journaux de transactions

- `wal_compression = on`
  - moins de journaux
  - un peu plus de CPU
  - à activer : `pglz` (`on`), `lz4`, `zstd` (v15+)
- Outils de compression standards : `gzip`, `bzip2`, `lzma`…
  - attention à ne pas ralentir l'archivage

PostgreSQL peut compresser les journaux à la source, si le paramètre `wal_compression` (désactivé par défaut) est passé à `on`. La compression est opérée par PostgreSQL au niveau de la page, avec un gros gain en volumétrie (souvent plus de 50 % !). Les journaux font toujours 16 Mo (par défaut), mais comme ils sont moins nombreux, leur rejeu est plus rapide, ce qui accélère la réplication et la reprise après un crash. Cette compression des journaux est totalement transparente pour l'archivage ou la restauration. Le prix est une augmentation de la consommation en CPU, souvent négligeable.

Depuis PostgreSQL 15, on peut même choisir l'algorithme de compression : `pglz`, `lz4` ou `zstd`. `on` est le synonyme de `pglz`… qui est sans doute le moins bon des trois (voir ce petit test[15]), surtout en terme de consommation CPU.

[13] https://wiki.postgresql.org/wiki/Full_page_writes
[14] https://blog.dalibo.com/2024/01/05/cambouis.html
[15] https://www.postgresql.org/message-id/YMmlvyVyAFlxZ%2B/H%40paquier.xyz

Une autre solution est la compression à la volée des journaux archivés dans l'`archive_command`. Les outils classiques comme `gzip`, `bzip2`, `lzma`, `xz`, etc. conviennent. Tous les outils PITR incluent plusieurs de ces algorithmes. Un fichier de 16 Mo aura généralement une taille compressée comprise entre 3 et 6 Mo.

Cependant, attention au temps de compression des journaux : en cas d'écritures lourdes, une compression élevée mais lente peut mener à un retard conséquent de l'archivage par rapport à l'écriture des journaux, jusque saturation de `pg_wal`, et arrêt de l'instance. Il est courant de se contenter de `gzip -1` ou `lz4 -1` pour les journaux, et de ne compresser agressivement que les sauvegardes des fichiers de la base.

14.9.3 Outils de sauvegarde PITR dédiés

- Se faciliter la vie avec différents outils
  - pgBackRest
  - barman
- Fournissent :
  - un outil pour les backups, les purges…
  - une commande pour l'archivage

Il n'est pas conseillé de réinventer la roue et d'écrire soi-même des scripts de sauvegarde, qui doivent prévoir de nombreux cas et bien gérer les erreurs. La sauvegarde concurrente est également difficile à manier. Des outils reconnus existent, dont nous évoquerons brièvement les plus connus. Il en existe d'autres. Ils ne font pas partie du projet PostgreSQL à proprement parler et doivent être installés séparément.

Les outils décrits succinctement plus bas fournissent :

- un outil pour procéder aux sauvegardes, gérer la péremption des archives… ;
- un outil qui sera appelé par `archive_command`.

Leur philosophie peut différer, notamment en terme de centralisation ou de compromis entre simplicité et fonctionnalités. Ces dernières s'enrichissent d'ailleurs au fil du temps.

Voir https://dali.bo/i4_html pour une description plus complète.

14.9.4 pgBackRest

- Gère la sauvegarde et la restauration
  - *pull* ou *push*, multidépôts
  - mono- ou multiserveur
- Indépendant des commandes système
  - protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multithread, sauvegarde depuis un secondaire, archivage asynchrone…
- Projet mature

pgBackRest[16] est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multithread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de `pg_wal` ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser, ou avoir plusieurs niveaux d'archives) ;
- le support intégré de dépôts S3 ou Azure ;
- le support d'un accès TLS géré par pgBackRest en alternative à SSH ;
- la sauvegarde depuis un serveur secondaire ;
- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé ;
- la reprise d'une sauvegarde échouée.

pgBackRest n'utilise pas `pg_receivewal` pour garantir la sauvegarde du dernier journal (non terminé) avant un sinistre. Les auteurs considèrent que dans ce cas un secondaire synchrone est plus adapté et plus fiable.

Le projet est très actif et considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

[16] https://pgbackrest.org/

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest[17].

14.9.5 barman

- Gère la sauvegarde et la restauration
  - mode *pull*
  - multiserveurs
- Une seule commande (`barman`)
- Et de nombreuses actions
  - `list-server`, `backup`, `list-backup`, `recover`...
- Spécificité : gestion de `pg_receivewal`

barman est un outil créé par 2ndQuadrant (racheté depuis par EDB). Il a pour but de faciliter la mise en place de sauvegardes PITR. Il gère à la fois la sauvegarde et la restauration.

La commande `barman` dispose de plusieurs actions :

- `list-server`, pour connaître la liste des serveurs configurés ;
- `backup`, pour lancer une sauvegarde de base ;
- `list-backup`, pour connaître la liste des sauvegardes de base ;
- `show-backup`, pour afficher des informations sur une sauvegarde ;
- `delete`, pour supprimer une sauvegarde ;
- `recover`, pour restaurer une sauvegarde (la restauration peut se faire à distance).

Contrairement aux autre outils présentés ici, barman permet d'utiliser `pg_receivewal`.

Il supporte aussi les dépôts S3 ou blob Azure.

Site web de barman[18]

[17] https://github.com/pgstef/check_pgbackrest/
[18] https://www.pgbarman.org/

14.10 CONCLUSION

- Une sauvegarde
  - fiable
  - éprouvée
  - rapide
  - continue
- Mais
  - plus complexe à mettre en place que `pg_dump`
  - qui restaure toute l'instance

Cette méthode de sauvegarde est la seule utilisable dès que les besoins de performance de sauvegarde et de restauration augmentent (*Recovery Time Objective* ou RTO), ou que le volume de perte de données doit être drastiquement réduit (*Recovery Point Objective* ou RPO).

14.10.1 Questions

N'hésitez pas, c'est le moment !

14.11 QUIZ

https://dali.bo/i2_quiz

14.12 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/i2_solutions.

14.12.1 pg_basebackup : sauvegarde ponctuelle & restauration

But : Créer une sauvegarde physique à chaud à un moment précis de la base avec `pg_basebackup`, et la restaurer.

Configurer la réplication dans `postgresql.conf` et `pg_hba.conf` :

- désactiver l'archivage s'il est actif
- autoriser des connexions de réplication en streaming en local.

Pour insérer des données :

- générer de l'activité avec `pgbench` en tant qu'utilisateur **postgres** :

```
$ createdb bench
$ /usr/pgsql-17/bin/pgbench -i -s 100 bench
$ /usr/pgsql-17/bin/pgbench bench -n -P 3 -R 50 -T 1800
```

- laisser tourner en arrière-plan
- surveiller l'évolution de l'activité sur la table `pgbench_history`, par exemple ainsi :

```
$ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
```

En parallèle, sauvegarder l'instance avec :

- `pg_basebackup` au format tar, compressé avec gzip ;
- sans oublier les journaux ;
- avec l'option `--max-rate=30M` pour ralentir la sauvegarde ;
- le répertoire de sauvegarde sera `/var/lib/pgsql/17/backups/basebackup` ;
- surveillez la progression dans une autre session avec la vue système adéquate.

Une fois la sauvegarde terminée :

- arrêter la session `pgbench` ;
- regarder les fichiers générés ; Afficher la date de dernière modification dans `pgbench_history`.

- Arrêter l'instance.
- Faire une copie à froid des données (par exemple avec `cp -rfp`) vers `/var/lib/pgsql/17/data.old` (cette copie resservira plus tard).

- Vider le répertoire des données.
- Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.
- Redémarrer l'instance.

Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

Quelle est la dernière donnée restaurée ?

Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction. Que se passe-t-il ?

14.12.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants

But : Coupler une sauvegarde à chaud avec `pg_basebackup` et l'archivage

Si le TP précédent a été déroulé et que l'instance n'est pas fonctionnelle, remettre en place la copie à froid de l'instance depuis `/var/lib/pgsql/17/data.old`.

Configurer l'archivage vers un répertoire `/archives`, par exemple avec `rsync`. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

Générer à nouveau de l'activité avec `pgbench` : en tant qu'utilisateur **postgres** :

```
$ createdb bench                # si pas déjà fait précédemment
$ /usr/pgsql-17/bin/pgbench -i -s 100 bench             # idem
$ /usr/pgsql-17/bin/pgbench bench -n -P 3 -R 50 -T 1800
```

Vérifier que l'archivage fonctionne dans le répertoire `/archives`, dans les traces et dans la vue `pg_stat_archiver`.

> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup` au format plain.

> Utiliser `pg_verify_backup` pour contrôler l'intégrité de la sauvegarde.

> À quoi correspond le fichier finissant par `.backup` dans les archives ?

> Arrêter `pgbench` et noter la date des dernières données insérées.

> Effacer le PGDATA. Restaurer la sauvegarde précédente *sans* les journaux. Configurer la `restore_command`. Créer le fichier `recovery.signal`. Démarrer PostgreSQL.

> Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

> Vérifier quelles données ont été restaurées.

14.13 TRAVAUX PRATIQUES (SOLUTIONS)

14.13.1 pg_basebackup : sauvegarde ponctuelle & restauration

NB : Ce TP a été mis à jour pour la version 17 de PostgreSQL. Adapter les chemins pour les autres versions au besoin.

Dans ce qui suit, la plupart des commandes seront à lancer en tant que **postgres**, les ordres `sudo` nécessitant un utilisateur privilégié.

> Configurer la réplication dans `postgresql.conf` et `pg_hba.conf` :
>
> - désactiver l'archivage s'il est actif
> - autoriser des connexions de réplication en streaming en local.

On n'aura pas encore besoin de l'archivage ici. S'il est déjà actif, on peut se contenter d'inhiber ainsi la commande d'archivage (cela permet d'épargner le redémarrage à chaque modification de `archive_mode`) :

```
archive_mode = on
archive_command = '/bin/true'
```

dans `/var/lib/pgsql/17/data/postgresql.conf` (pour les paquets RPM du PGDG ; sur Debian/Ubuntu, le `postgresql.conf` est dans `/etc/postgresql/17/main/`).

Vérifier l'autorisation de connexion en réplication dans `pg_hba.conf`. Si besoin, mettre à jour la ligne en fin de fichier :

```
local   replication     all                                     peer
```

Cela va ouvrir l'accès sans mot de passe depuis l'utilisateur système **postgres**.

Redémarrer PostgreSQL :

```
sudo systemctl restart postgresql-17
```

(Sous Debian, le nom du service est plutôt `postgresql@17-main`.)

> Pour insérer des données :
>
> - générer de l'activité avec `pgbench` en tant qu'utilisateur **postgres** :
>
> ```
> $ createdb bench
> $ /usr/pgsql-17/bin/pgbench -i -s 100 bench
> $ /usr/pgsql-17/bin/pgbench bench -n -P 3 -R 50 -T 1800
> ```
>
> - laisser tourner en arrière-plan
> - surveiller l'évolution de l'activité sur la table `pgbench_history`, par exemple ainsi :
>
> ```
> $ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
> ```

> En parallèle, sauvegarder l'instance avec :
>
> - `pg_basebackup` au format tar, compressé avec gzip ;
> - sans oublier les journaux ;
> - avec l'option `--max-rate=30M` pour ralentir la sauvegarde ;
> - le répertoire de sauvegarde sera `/var/lib/pgsql/17/backups/basebackup` ;
> - surveillez la progression dans une autre session avec la vue système adéquate.

(Le répertoire de sauvegarde correspond à une installation par défaut sur un Linux Red Hat ou similaire. Adapter le chemin sur un système Debian/Ubuntu.)

En tant que **postgres** :

```
pg_basebackup -D /var/lib/pgsql/17/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=30M
```

```
1583675/1583675 kB (100%), 1/1 tablespace
```

La progression peut se suivre depuis psql avec :

```
\x on
SELECT * FROM pg_stat_progress_basebackup ;
\watch
```

```
Thu Nov 11 17:58:05 2024 (every 2s)

-[ RECORD 1 ]--------+---------------------------------
pid                 | 19763
phase                | waiting for checkpoint to finish
backup_total         |
backup_streamed      | 0
tablespaces_total    | 0
tablespaces_streamed | 0

Thu Nov 11 17:58:07 2024 (every 2s)

-[ RECORD 1 ]--------+-------------------------
pid                  | 19763
phase                | streaming database files
backup_total         | 1711215360
backup_streamed      | 29354496
tablespaces_total    | 1
tablespaces_streamed | 0
…
```

Évidemment, en production, il ne faut pas sauvegarder en local.

> Une fois la sauvegarde terminée :
>
> - arrêter la session `pgbench` ;
> - regarder les fichiers générés ; Afficher la date de dernière modification dans `pgbench_history`.

L'activité avec `pgbench` s'arrête avec un simple **Ctrl-C**. L'heure de dernière modification est :

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
  2024-11-05 17:01:51.595414
```

Les fichiers générés sont :

```
$ ls -lha /var/lib/pgsql/17/backups/basebackup
```

```
…
-rw-------. 1 postgres postgres 180K Nov 11 17:00 backup_manifest
-rw-------. 1 postgres postgres  91M Nov 11 17:00 base.tar.gz
-rw-------. 1 postgres postgres  23M Nov 11 17:00 pg_wal.tar.gz
```

On obtient donc :

- une archive de la sauvegarde « de base » ;
- une archive des journaux nécessaires ;
- un fichier manifeste au format texte contenant les sommes de contrôles des fichiers archivés.

> - Arrêter l'instance.
> - Faire une copie à froid des données (par exemple avec `cp -rfp`) vers `/var/lib/pgsql/17/data.old` (cette copie resservira plus tard).

En tant qu'utilisateur privilégié :

```
sudo systemctl stop postgresql-17
```

En tant que **postgres** :

```
cp -rfp /var/lib/pgsql/17/data /var/lib/pgsql/17/data.old
```

(Ce répertoire de sauvegarde correspond à une installation par défaut sur un Linux Red Hat ou similaire. Adapter le chemin sur un système Debian/Ubuntu, où les données sont par défaut dans `/var/lib/postgresql/17/main/`.)

> - Vider le répertoire des données.
> - Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.
> - Redémarrer l'instance.

On restaure dans le répertoire de données l'archive de base, puis les journaux dans leur sous-répertoire. La suppression des traces est optionnelle, mais elle nous permettra de ne pas mélanger celles d'avant et d'après la restauration.

En tant que **postgres** :

```
rm -rf /var/lib/pgsql/17/data/*
tar -C /var/lib/pgsql/17/data \
    -xzf /var/lib/pgsql/17/backups/basebackup/base.tar.gz
tar -C /var/lib/pgsql/17/data/pg_wal \
    -xzf /var/lib/pgsql/17/backups/basebackup/pg_wal.tar.gz
rm -rf /var/lib/pgsql/17/data/log/*
```

```
sudo systemctl start postgresql-17
```

> Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

En tant **postgres** ou **root** :

```
tail -F /var/lib/pgsql/17/data/log/postgresql-*.log
```

```
…
… LOG:  database system was interrupted; last known up at 2024-11-05 17:59:03 UTC
… LOG:  redo starts at 0/830000B0
… LOG:  consistent recovery state reached at 0/8E8450F0
… LOG:  redo done at 0/8E8450F0 system usage: CPU: user: 0.28 s, system: 0.24 s,
 ↪  elapsed: 0.59 s
… LOG:  checkpoint starting: end-of-recovery immediate wait
… LOG:  checkpoint complete: wrote 17008 buffers (97.7%); …
… LOG:  database system is ready to accept connections
```

PostgreSQL considère qu'il a été interrompu brutalement et part en *recovery*. Noter en particulier la mention *consistent recovery state reached* : la sauvegarde est bien cohérente.

> Quelle est la dernière donnée restaurée ?

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
 2024-11-05 17:00:40.936925
```

Grâce aux journaux (`pg_wal`) restaurés, l'ensemble des modifications survenues **pendant** la sauvegarde ont bien été récupérées. Par contre, les données générées après la sauvegarde n'ont, elles, pas été récupérées.

> Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction. Que se passe-t-il ?

En tant que **root** :

```
systemctl stop postgresql-17
```

```
rm -rf /var/lib/pgsql/17/data/*
tar -C /var/lib/pgsql/17/data \
    -xzf /var/lib/pgsql/17/backups/basebackup/base.tar.gz
rm -rf /var/lib/pgsql/17/data/log/*
systemctl start postgresql-17
```

Et le redémarrage donne :

```
systemctl start postgresql-17
```

```
Job for postgresql-17.service failed because the control process exited with error
 ↪  code.
See "systemctl status postgresql-17.service" and "journalctl -xe" for details.
```

Pour trouver la cause, il faut aller chercher dans les traces :

```
tail -F /var/lib/pgsql/17/data/log/postgresql-*.log
```

```
… LOG:  database system was interrupted; last known up at 2025-01-13 18:04:26 CET
… LOG:  starting backup recovery with redo LSN 0/BB000028, checkpoint LSN
 ↪  0/BB020CF8, on timeline ID 1
… LOG:  invalid checkpoint record
… FATAL:  could not locate required checkpoint record at 0/BB020CF8
… HINT:  If you are restoring from a backup, touch
 ↪  "/var/lib/pgsql/17/data/recovery.signal" or
 ↪  "/var/lib/pgsql/17/data/standby.signal" and add required recovery options.
        If you are not restoring from a backup, try removing the file
 ↪  "/var/lib/pgsql/17/data/backup_label".
        Be careful: removing "/var/lib/pgsql/17/data/backup_label" will result in a
 ↪  corrupt cluster if restoring from a backup.
… LOG:  startup process (PID 114562) exited with exit code 1
… LOG:  aborting startup due to startup process failure
… LOG:  database system is shut down
```

PostgreSQL ne trouve pas les journaux nécessaires à sa restauration à un état cohérent, le service refuse de démarrer. Il a trouvé un checkpoint dans le fichier `backup_label` créé au début de la sauvegarde, mais aucun checkpoint postérieur dans les journaux (et pour cause).

Les traces contiennent ensuite des suggestions qui peuvent être utiles.

Cependant, un fichier `recovery.signal` ne sert à rien sans `restore_command`, et nous n'en avons pas encore paramétré ici.

Quant au fichier `backup_label`, le supprimer permettrait peut-être de démarrer l'instance mais celle-ci serait alors dans un état incohérent ! Il y a de bonnes chances que le démarrage s'achève par :

```
PANIC:  could not locate a valid checkpoint record
```

En résumé : la restauration des journaux n'est pas optionnelle !

14.13.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants

> Si le TP précédent a été déroulé et que l'instance n'est pas fonctionnelle, remettre en place la copie à froid de l'instance depuis `/var/lib/pgsql/17/data.old`.

```
sudo systemctl stop postgresql-17  # si nécessaire
```

```
rm -rf /var/lib/pgsql/17/data
cp -rfp /var/lib/pgsql/17/data.old /var/lib/pgsql/17/data
```

```
sudo systemctl start postgresql-17
```

Vérifier qu'il est possible de se connecter par psql.

> Configurer l'archivage vers un répertoire `/archives`, par exemple avec `rsync`. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

Créer un répertoire d'archivage, s'il n'existe pas déjà, et vérifier que **postgres** a les droits nécessaires pour l'utiliser :

```
sudo mkdir /archives
sudo chown postgres: /archives
sudo chmod 700 /archives
```

Là encore, en production, ce sera plutôt un partage distant. L'utilisateur système **postgres** doit avoir le droit d'y écrire.

La commande d'archivage se définit dans `postgresql.conf` :

```
archive_mode = on
archive_command = 'rsync %p /archives/%f'
```

et on peut y définir aussi tout de suite la commande de restauration :

```
restore_command = 'rsync /archives/%f %p'
```

```
sudo systemctl start postgresql-17
```

(Sous Debian, le nom du service est plutôt `postgresql@17-main`.)

Vérifier le bon démarrage :

```
systemctl status postgresql-17
```

> Générer à nouveau de l'activité avec `pgbench` : en tant qu'utilisateur **postgres** :
>
> ```
> $ createdb bench                # si pas déjà fait précédemment
> $ /usr/pgsql-17/bin/pgbench -i -s 100 bench              # idem
> $ /usr/pgsql-17/bin/pgbench bench -n -P 3 -R 50 -T 1800
> ```

> Vérifier que l'archivage fonctionne dans le répertoire `/archives`, dans les traces et dans la vue `pg_stat_archiver`.

```
ls -lha /archives
```

```
…
-rw-------. 1 postgres postgres 30M Jan  5 18:32 0000000100000000000000BB
-rw-------. 1 postgres postgres 30M Jan  5 18:32 0000000100000000000000BC
-rw-------. 1 postgres postgres 30M Jan  5 18:32 0000000100000000000000BD
…
```

Si cela ne fonctionne pas, il faut aller voir les traces dans le dernier fichier en date sous `/var/lib/pgsql/17/data/log/postgresql-*.log` (paquets RPM du PGDG), ou dans `/var/log/postgresql/postg` (paquets Debian). Par exemple, une erreur de frappe dans `archive_command` peut donner ces messages :

```
…ERROR:  invalid value for parameter "archive_command": "rsync % p /archives/%f"
…DETAIL:  String contains unexpected placeholder "% ".
…WARNING:  archiving write-ahead log file "000000010000000000000C5" failed too many
↪  times, will try again later
```

La vue `pg_stat_archiver` doit indiquer que les erreurs d'archivage sont plus anciennes que les succès, comme ci-dessous. `pg_switch_wal()` peut aider à forcer un archivage si l'activité est trop faible.

```
SELECT pg_switch_wal() ; TABLE pg_stat_archiver \gx
```

```
 pg_switch_wal
---------------
 0/CC1232A8
(1 row)

-[ RECORD 1 ]------+------------------------------
archived_count     | 16
last_archived_wal  | 000000010000000000000CB
last_archived_time | 2024-11-11 18:32:34.427199+01
failed_count       | 36
last_failed_wal    | 000000010000000000000C5
last_failed_time   | 2024-11-11 18:32:08.631785+01
stats_reset        | 2024-11-10 17:55:27.850351+01
```

> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup` au format plain.

En tant que **postgres** :

```
rm -rf /var/lib/pgsql/17/backups/basebackup
```

```
pg_basebackup -D /var/lib/pgsql/17/backups/basebackup -Fp \
--checkpoint=fast --progress --max-rate=30M
```

```
1586078/1586078 kB (100%), 1/1 tablespace
```

Le répertoire cible devra avoir été vidé.

La taille de la sauvegarde sera bien sûr nettement plus grosse qu'en tar compressé.

> Utiliser `pg_verify_backup` pour contrôler l'intégrité de la sauvegarde.

Si tout va bien, le message sera lapidaire :

```
/usr/pgsql-17/bin/pg_verifybackup  /var/lib/pgsql/17/backups/basebackup
```

```
backup successfully verified
```

S'il y a un problème, des messages de ce genre apparaîtront :

```
pg_verifybackup: error: "global/TEST" is present on disk but not in the manifest
pg_verifybackup: error: "global/2671" is present in the manifest but not on disk
pg_verifybackup: error: "postgresql.conf" has size 29507 on disk but size 29506 in
↪  the manifest
```

> À quoi correspond le fichier finissant par `.backup` dans les archives ?

En effet, parmi les journaux archivés, figure ce fichier :

```
ls -1 /archives
```

```
…
000000010000000000000BE
000000010000000000000BE.00003E00.backup
000000010000000000000BF
…
```

Son contenu correspond au futur `backup_label` :

```
START WAL LOCATION: 0/BE003E00 (file 000000010000000000000BE)
STOP WAL LOCATION: 0/C864D0F8 (file 000000010000000000000C8)
CHECKPOINT LOCATION: 0/BE0AB340
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2024-11-05 18:32:52 UTC
LABEL: pg_basebackup base backup
START TIMELINE: 1
STOP TIME: 2024-11-05 18:34:29 UTC
STOP TIMELINE: 1
```

> Arrêter `pgbench` et noter la date des dernières données insérées.

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
 2024-11-05 18:41:23.068948
```

> Effacer le PGDATA. Restaurer la sauvegarde précédente *sans* les journaux. Configurer la `restore_command`. Créer le fichier `recovery.signal`. Démarrer PostgreSQL.

```
sudo systemctl stop postgresql-17
```

La sauvegarde étant au format *plain*, il s'agit d'une simple copie de fichiers. En tant que **postgres** :

```
rm -rf /var/lib/pgsql/17/data/*
rsync -a --exclude 'pg_wal/*' --exclude 'log/*' \
 /var/lib/pgsql/17/backups/basebackup/ \
 /var/lib/pgsql/17/data/
```

Créer le fichier `recovery.signal` :

```
touch /var/lib/pgsql/17/data/recovery.signal
```

Démarrer le service :

```
sudo systemctl start postgresql-17
```

> Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

Les traces sont plus complexes à cause de la restauration depuis les archives :

```
# attention, le fichier à lire dépend du jour
tail -F /var/lib/pgsql/17/data/log/postgresql-Mon.log
```

```
…
… LOG:  database system was interrupted; last known up at 2024-11-05 18:32:52 UTC
rsync: link_stat "/archives/00000002.history" failed: No such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) [sender=3.1.3]
… LOG:  starting archive recovery
… LOG:  restored log file "0000000100000000000000BE" from archive
… LOG:  redo starts at 0/BE003E00
… LOG:  restored log file "0000000100000000000000BF" from archive
… LOG:  restored log file "0000000100000000000000C0" from archive
… LOG:  restored log file "0000000100000000000000C1" from archive
…
… LOG:  restored log file "0000000100000000000000C8" from archive
… LOG:  restored log file "0000000100000000000000C9" from archive
… LOG:  consistent recovery state reached at 0/C864D0F8
… LOG:  database system is ready to accept read-only connections
… LOG:  restored log file "0000000100000000000000CA" from archive
… LOG:  restored log file "0000000100000000000000CB" from archive
…
… LOG:  restored log file "0000000100000000000000E0" from archive
… LOG:  restored log file "0000000100000000000000E1" from archive
… LOG:  redo in progress, elapsed time: 10.25 s, current LSN: 0/E0FF3438
… LOG:  restored log file "0000000100000000000000E2" from archive
… LOG:  restored log file "0000000100000000000000E3" from archive
…
… LOG:  restored log file "0000000100000000000000EF" from archive
… LOG:  restored log file "0000000100000000000000F0" from archive
rsync: link_stat "/archives/0000000100000000000000F1" failed: No such file or
 ↪  directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
rsync: link_stat "/archives/0000000100000000000000F1" failed: No such file or
 ↪  directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
… LOG:  redo done at 0/F0A6C9E0 system usage:
                                          CPU: user: 2.51 s, system: 2.28 s, elapsed:
 ↪  15.77 s
… LOG:  last completed transaction
                                        was at log time 2024-11-05 18:41:23.077219+00
… LOG:  restored log file "0000000100000000000000F0" from archive
rsync: link_stat "/archives/00000002.history" failed: No such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
… LOG:  selected new timeline ID: 2
rsync: link_stat "/archives/00000001.history" failed: No such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
```

```
… LOG:  archive recovery complete
… LOG:  checkpoint starting: end-of-recovery immediate wait
… LOG:  checkpoint complete: wrote 17012 buffers (97.7%); …
… LOG:  database system is ready to accept connections
```

Les messages d'erreur de `rsync` ne sont pas inquiétants : celui-ci ne trouve simplement pas les fichiers demandés à la `restore_command`. PostgreSQL sait ainsi qu'il n'y a pas de fichier `00000002.history` et donc pas de timeline de ce numéro. Il devine aussi qu'il a restauré tous les journaux quand la récupération de l'un d'entre eux échoue.

Les erreurs sur les fichiers `00000001.history` et `00000002.history` sont normales. PostgreSQL cherche ces fichiers pour voir quel est l'enchaînement des *timelines* et quelle est la dernière.

La progression de la restauration peut être suivie grâce aux différents messages, repris ci-dessous, de démarrage, d'atteinte du point de cohérence, de statut… jusqu'à l'heure exacte de restauration. Enfin, il y a bascule sur une nouvelle *timeline*, et un checkpoint.

```
LOG:  starting archive recovery
LOG:  redo starts at 0/BE003E00
LOG:  consistent recovery state reached at 0/C864D0F8
LOG:  redo in progress, elapsed time: 10.25 s, current LSN: 0/E0FF3438
LOG:  redo done at 0/F0A6C9E0 …
LOG:  last completed transaction was at log time 2024-11-05 18:41:23.077219+00
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  checkpoint complete:
```

Noter que les journaux portent une nouvelle *timeline* numérotée 2 :

```
ls -l /var/lib/pgsql/17/data/pg_wal/
```

```
…
-rw-------. 1 postgres postgres 17777217 Jan  5 18:43 000000020000000100000023
-rw-------. 1 postgres postgres 17777217 Jan  5 18:43 000000020000000100000024
-rw-------. 1 postgres postgres       42 Jan  5 18:43 00000002.history
drwx------. 2 postgres postgres       35 Jan  5 18:43 archive_status
```

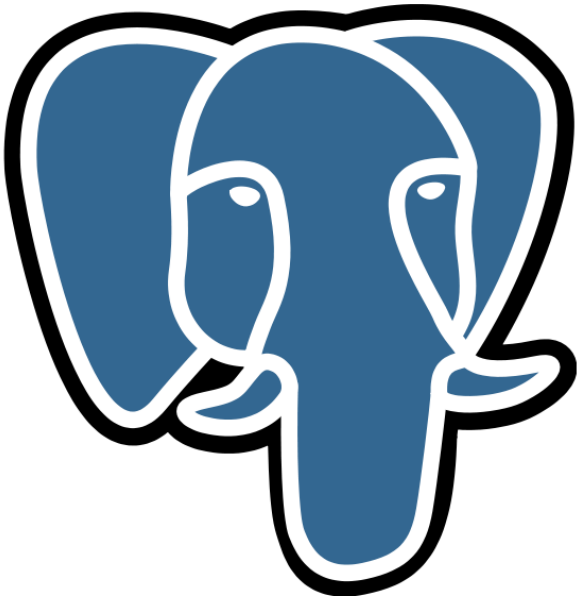
> Vérifier quelles données ont été restaurées.

Cette fois, toutes les données générées après la sauvegarde ont bien été récupérées :

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
 2024-11-05 18:41:23.068948
```

15/ Supervision



15.1 INTRODUCTION

- Deux types de supervision
  - occasionnelle
  - automatique
- Superviser PostgreSQL et le système
- Pour PostgreSQL, statistiques et traces

Superviser un serveur de bases de données consiste à superviser le SGBD lui-même, mais aussi le système d'exploitation et le matériel. Ces deux derniers sont importants pour connaître la charge système, l'utilisation des disques ou du réseau, qui pourraient expliquer des lenteurs au niveau du SGBD. PostgreSQL propose lui aussi des informations qu'il est important de surveiller pour détecter des problèmes au niveau de l'utilisation du SGBD ou de sa configuration.

Ce module a pour but de montrer comment effectuer une supervision occasionnelle (au cas où un problème survient, savoir comment interpréter les informations fournies par le système et par PostgreSQL) et comment mettre en place une supervision automatique (pour des alertes ou la supervision à long terme).

15.1.1 Menu

- Politique de supervision
- Supervision de PostgreSQL
- Traces : configuration & analyses
- Statistiques d'activité

15.2 POLITIQUE DE SUPERVISION

- Pour quoi ?
- Pour qui ?
- Quels critères ?
- Quels outils

Il n'existe pas qu'une seule supervision. Suivant la personne concernée par la supervision et son objectif, les critères de la supervision seront différents.

Lors de la mise en place de la supervision, il est important de se demander l'objectif de cette supervision, à qui elle va servir, les critères qui importent à cette personne.

Répondre à ces questions permettra de mieux choisir l'outil de supervision à mettre en place, ainsi que sa configuration.

15.2.1 Objectifs de la supervision

- Améliorer/mesurer les performances
- Améliorer l'applicatif
- Anticiper/prévenir les incidents
- Réagir vite en cas de crash

Généralement, les administrateurs mettant en place la supervision veulent pouvoir anticiper les problèmes, qu'ils soient matériels, de performance, de qualité de service, etc.

Améliorer les performances du SGBD sans connaître les performances globales du système est très difficile. Si un utilisateur se plaint d'une perte de performance, pouvoir corroborer ses dires avec des informations provenant du système de supervision aide à s'assurer qu'il y a bien un problème de performances et peut fréquemment aider à résoudre ce problème. De plus, il est important de pouvoir mesurer les gains de performances.

Une supervision des traces de PostgreSQL permet aussi d'améliorer les applications qui utilisent une base de données. Toute requête en erreur est tracée dans les journaux applicatifs, ce qui permet de trouver rapidement les problèmes que les utilisateurs rencontrent.

Un suivi régulier de la volumétrie ou du nombre de connexions permet de prévoir les évolutions nécessaires du matériel ou de la configuration : achat de matériel, création d'index, amélioration de la configuration.

Prévenir les incidents peut se faire en ayant une sonde de supervision des erreurs disques par exemple. La supervision permet aussi d'anticiper les problèmes de configuration. Par exemple, surveiller le nombre de sessions ouvertes sur PostgreSQL permet de s'assurer que ce nombre n'approche pas trop du nombre maximum de sessions configuré avec le paramètre `max_connections` dans le fichier `postgresql.conf`.

Enfin, une bonne configuration de la supervision implique d'avoir configuré finement la gestion des traces de PostgreSQL. Avoir un bon niveau de trace (autrement dit : ni trop, ni pas assez) permet de réagir rapidement après un crash.

15.2.2 Acteurs concernés

- Développeur
  - correction et optimisation de requêtes
- Administrateur de bases de données
  - surveillance, performance
  - mise à jour
- Administrateur système
  - surveillance, qualité de service

Il y a trois types d'acteurs concernés par la supervision.

Le développeur doit pouvoir visualiser l'activité de la base de données. Il peut ainsi comprendre l'impact du code applicatif sur la base. De plus, le développeur est intéressé par la qualité des requêtes que son code exécute. Donc des traces qui ramènent les requêtes en erreur et celles qui ne sont pas performantes sont essentielles pour ce profil.

L'administrateur de bases de données a besoin de surveiller les bases pour s'assurer de la qualité de service, pour garantir les performances et pour réagir rapidement en cas de problème. Il doit aussi faire les mises à jours mineures dès qu'elles sont disponibles.

Enfin, l'administrateur système doit s'assurer de la présence du service. Il doit aussi s'assurer que le service dispose des ressources nécessaires, en terme de processeur (donc de puissance de calcul), de mémoire et de disque (notamment pour la place disponible).

15.2.3 Exemples d'indicateurs - système d'exploitation

- Charge CPU
- Entrées/sorties disque
- Espace disque
- Sur-activité et non-activité du serveur
- Temps de réponse
- Outils Unix habituels :
  - `top`, `atop`, `free`, `df`, `vmstat`, `sar`, `iotop`

Voici quelques exemples d'indicateurs intéressants à superviser pour la partie du système d'exploitation.

La charge CPU (processeur) est importante. Elle peut expliquer pourquoi des requêtes, auparavant rapides, deviennent lentes. Cependant, la suractivité comme la non-activité sont un problème. En fait, si le service est tombé, le serveur sera en sous-activité, ce qui est un excellent indice.

Les entrées/sorties disque permettent de montrer un souci au niveau du système disque. Par exemple, PostgreSQL peut écrire trop à cause d'une mauvaise configuration des journaux de transactions, ou des fichiers temporaires issus de requêtes lourdes ou mal écrites ; ou il peut lire trop de données par manque de cache en RAM, ou de requêtes mal écrites.

L'espace disque est essentiel à surveiller. PostgreSQL ne propose rien pour cela, il faut donc le faire au niveau système. L'espace disque peut poser problème s'il manque, surtout si cela concerne la partition des journaux de transactions.

Unix possède de nombreux outils pour surveiller les différents éléments du système. Les grands classiques sont `top` et ses innombrables clones comme `atop` pour le CPU, `free` pour la RAM, `df` pour l'espace disque, `vmstat` pour la mémoire virtuelle, `iotop` pour les entrées/sorties, ou `sar` (généraliste). Sous Windows, les premiers outils sont bien sûr le Gestionnaire des tâches, et Process Monitor[1] des outils Sysinternals.

Il est conseillé d'avoir une requête étalon dont la durée d'exécution sera testée de temps à autre pour détecter les moments problématiques sur le serveur.

[1] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

15.2.4 Exemples d'indicateurs - base de données

- Nombre de connexions
- Requêtes lentes et/ou fréquentes
- Ratio d'utilisation du cache
- Verrous
- Volumétries
- …

Il existe de nombreux indicateurs intéressant sur les bases : nombre de connexions (en faisant par exemple la différence entre connexions inactives, actives, en attente de verrous), nombre de requêtes lentes et/ou fréquentes, volumétrie (en taille, en nombre de lignes), des ratios (utilisation du cache par exemple)…

15.3 SUPERVISION DE POSTGRESQL

- Supervision occasionnelle
  - sur incident…
- Supervision automatique
  - remonter des informations rapidement
  - archiver, suivre les tendances

La supervision occasionnelle est la conséquence d'une plainte d'un utilisateur : on se contente de réagir à un problème. C'est généralement insuffisant.

Il est important de mettre en place une solution de supervision automatique. Le but est de récupérer périodiquement des données statistiques sur les objets et sur l'utilisation du serveur pour avoir des graphes de tendance, et recevoir des alertes quand des seuils sont dépassés.

15.3.1 Informations internes

- PostgreSQL propose :
  - statistiques d'activité
  - traces
- …mais rien pour les historiser

PostgreSQL propose deux canaux d'informations : les statistiques d'activité (à ne pas confondre avec les statistiques sur les données, à destination de l'optimiseur de requêtes) et les traces applicatives (ou « logs »), souvent dans un fichier comme `postgresql.log` (le nom exact varie avec la distribution et l'installation).

PostgreSQL stocke un ensemble d'informations (métadonnées des schémas, informations sur les tables et les colonnes, données de suivi interne, etc.) dans des tables systèmes qui peuvent être consultées par les administrateurs. PostgreSQL fournit également des vues combinant des informations puisées dans différentes tables systèmes. Ces vues simplifient le suivi de l'activité de la base.

PostgreSQL est aussi capable de tracer un grand nombre d'informations qui peuvent être exploitées pour surveiller l'activité de la base de données.

Pour pouvoir mettre en place un système de supervision automatique, il est essentiel de s'assurer que les statistiques d'activité et les traces applicatives sont bien configurées et il faut aussi leur associer un outil permettant de sauvegarder les données, les alertes et de les historiser.

15.3.2 Outils externes

- Pour conserver les informations
- …et exécuter automatiquement des actions
  - graphiques (Munin, Zabbix…)
  - envoi d'alertes (Nagios, tail_n_mail)

Pour récupérer et enregistrer les informations statistiques, les historiser, envoyer des alertes, il faut faire appel à un outil externe. Cela peut être un outil très simple comme munin ou un outil très complexe comme Nagios ou Zabbix.

15.3.3 check_pgactivity

- Script de monitoring PostgreSQL pour Nagios
  - nombreuses sondes spécifiques à PostgreSQL
  - nombreuses métriques remontées
- Développé au départ par Dalibo
  - utilisable indépendamment
- https://github.com/OPMDG/check_pgactivity

Le script de monitoring `check_pgactivity` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios (entre autres).

Au départ, Dalibo utilisait une sonde nommmée `check_postgres` et participait activement à son développement, avec même un commiter dans le projet. Cependant, rapidement, nous nous sommes aperçus que nous ne pouvions pas aller aussi loin que nous le souhaitions. C'est à ce moment que, dans le cadre de sa R&D, Dalibo a conçu `check_pgactivity`.

La plupart des sondes de `check_postgres` y ont été réimplémentées. Le script corrige certaines sondes existantes et en fournit de nouvelles répondant mieux aux besoins de notre supervision, avec notamment un nombre plus important de métriques de performances. Il renvoie directement des ratios par rapport à la valeur précédente plutôt que des valeurs fixes. Pour les besoins les plus simples, le script peut être utilisé de façon autonome, sans nécessité d'installer toute l'infrastructure d'un outil comme Nagios, Icinga ou Grafana.

La supervision d'un serveur PostgreSQL passe par la surveillance de sa disponibilité, des indicateurs sur son activité, l'identification des besoins de maintenance, et le suivi de la réplication le cas échéant. Ci-dessous figurent les sondes `check_pgactivity` à mettre en place sur ces différents aspects. Le site du projet contient toute la documentation de chaque sonde.

Disponibilité :

- `connection` : réalise un test de connexion pour vérifier que le serveur est accessible ;
- `backends` : compte le nombre de connexions au serveur comparé au paramètre `max_connections` ;
- `backends_status` : permet d'obtenir des statistiques plus précises sur l'état des connexions clientes et d'être alerté lorsqu'un certain nombre de connexions clientes sont dans un état donné (*waiting*, *idle in transaction*…) ;
- `uptime` : détecte un redémarrage du serveur ou du rechargement de la configuration.

Vacuum :

- `autovacuum` : suit le fonctionnement de l'autovacuum et des tâches en cours (`VACUUM`, `ANALYZE`, `FREEZE`…) ;
- `table_bloat` : vérifie le volume de données « mortes » et la fragmentation des tables ;
- `btree_bloat` : vérifie le volume de données « mortes » et la fragmentation des index - par rapport à `check_postgres`, le calcul est séparé entre tables et index ;
- `last_analyze` : vérifie si le dernier analyze (relevé des statistiques relatives aux calculs des plans d'exécution) est trop ancien ;
- `last_vacuum` : vérifie si le dernier vacuum (relevé des espaces réutilisables dans les tables) est trop ancien.

Activité :

- `locks` : permet d'obtenir des statistiques plus détaillées sur les verrous obtenus et tient notamment compte des spécificités des *predicate locks* du niveau d'isolation `SERIALIZABLE` ;
- `wal_files` : compte le nombre de segments du journal de transaction présents dans le répertoire `pg_wal` ;
- `longest_query` : permet d'être alerté si une requête est en cours d'exécution depuis plus d'un certain temps ;
- `oldest_xact` : permet d'être alerté si une transaction est ouverte depuis un certain temps sans être utilisée ;
- `oldest_2pc` : calcule l'âge de la plus ancienne transaction préparée (*two-phase commit transaction*) ;

- `oldest_xmin` : repère la plus ancienne transaction de chaque base, et ce à quoi elle est liée (requête, slot…) ;
- `bgwriter` : permet de collecter des données de performance des différents processus d'écritures de PostgreSQL ;
- `hit_ratio` : calcule le *hit ratio* (utilisation du cache de PostgreSQL) ;
- `commit_ratio` : calcule la proportion de `COMMIT` et `ROLLBACK` ;
- `checksum_errors` : détecte l'apparition d'erreurs de sommes de contrôle (à partir de PostgreSQL 12) ;
- `database_size` : suit la volumétrie des bases et leurs variations ;
- `max_freeze_age` : calcule l'âge des plus vieilles lignes stockées dans chaque base pour suivre le bon passage des `VACUUM FREEZE` ;
- `stat_snapshot_age` : calcule l'âge des statistiques d'activité pour repérer un blocage du collecteur ;
- `temp_files` : suivi des fichiers temporaires.

Configuration :

- `configuration` : permet de vérifier que les principaux paramètres mémoire n'ont pas leur valeur par défaut ;
- `minor_version` : détecte les instances n'ayant pas la dernière version mineure ;
- `settings` : repère un changement des paramètres ;
- `invalid_indexes` : repérer tout index invalide ;
- `pgdata_permission` : vérifie les droits sur `PGDATA` pour éviter un blocage au redémarrage ;
- `table_unlogged` : remonte le nombre de tables *unlogged* ;
- `extensions_versions` : détecte les extensions à mettre à jour.

Réplication & archivage :

- `archiver` : compte le nombre de segments du journal de transaction en attente d'archivage ;
- `archive_folder` : vérifie qu'il n'y a pas de journal manquant dans les archives de sauvegarde PITR ;
- `hot_standby_delta` : calcule le délai de réplication entre un serveur primaire et un serveur secondaire ;
- `is_master` / `is_hot_standby` : vérifie que l'instance est bien démarrée en lecture/écriture, ou une instance secondaire ;
- `is_replay_paused` : vérifie si la réplication est en pause ;
- `replication_slots` : calcule la volumétrie conservée pour chaque slot de réplication.

Sauvegarde physique et logique :

- `backup_label_age` : calcule l'âge du fichier `backup_label` (sauvegardes PITR exclusives) ;
- `pg_dump_backup` : contrôle l'âge et la variation de taille des sauvegardes logiques.

15.3.4 check_postgres

- Script de monitoring PostgreSQL pour Nagios ou MRTG
- Quelques autres sondes
- https://bucardo.org/wiki/Check_postgres

Le script de monitoring `check_postgres` est la première sonde à avoir été écrite pour PostgreSQL. Comme vu précédemment, `check_pgactivity` est un remplaçant plus fonctionnel, donnant plus de métriques, sur un nombre plus limité de sondes.

`check_postgres` évolue cependant toujours et est intéressant dans certains cas : alerte pour les triggers désactivés, taille des relations, estimation du retard en réplication logique (native et Slony), comparaison de schémas, détection de proximité du wraparound. Elle intègre aussi beaucoup de sondes pour l'outil de pooling pgBouncer.

15.4 TRACES

- Configuration
  - traces peu fournies par défaut
- Récupération
  - des problèmes importants
  - des requêtes lentes/fréquentes
- Outils externes de classement

La première information que fournit PostgreSQL sur son activité sort dans les traces. Chaque requête en erreur génère une trace indiquant la requête erronée et l'erreur. Chaque problème de lecture ou d'écriture de fichier génère une trace. En fait, tout problème détecté par PostgreSQL fait l'objet d'un message dans les traces. PostgreSQL peut aussi y envoyer d'autres messages suivant certains événements, comme les connexions, l'activité de processus système en tâche de fond, etc.

Nous allons donc aborder la configuration des traces (où tracer, quoi tracer, quel niveau d'informations). Nous verrons au passage nombre d'informations intéressantes à récupérer. Enfin, nous verrons quelques outils permettant de traiter automatiquement les fichiers de trace.

15.4.1 Configuration des traces : principes

- Où tracer ?
- Quel niveau de traces ?
- Tracer les requêtes
  - durée, fichiers temporaires…
- Tracer certains comportements
  - erreurs

Il est essentiel de bien configurer PostgreSQL pour que les traces ne soient pas à la fois trop lourdes (pour ne pas être submergé par les informations) et incomplètes (il manque des informations). Un bon dosage du niveau des traces est important. Savoir où envoyer les traces est tout aussi important.

Suivant la configuration réalisée, les journaux applicatifs peuvent contenir quantité d'informations importantes. La plus fréquemment recherchée est la durée d'exécution des requêtes. L'intérêt principal est de récupérer les requêtes les plus lentes. L'autre information importante concerne les messages d'erreur, de niveau PANIC en premier lieu. Ces messages indiquent un état anormal du serveur qui s'est soldé par un arrêt brutal. Ce genre de problème est anormal et doit être surveillé.

15.4.2 Événements exceptionnels tracés

- Crash de PostgreSQL :

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

- Rechargement de la configuration :

```
LOG:  received SIGHUP, reloading configuration files
```

- Envoi immédiat d'une alerte
- Outil : tail_n_mail

Les messages `PANIC` sont très importants. Généralement, vous ne les verrez pas au moment où ils se produisent. Un crash va survenir et vous allez chercher à comprendre ce qui s'est passé. Il est possible à ce moment-là que vous trouviez dans les traces des messages `PANIC`, comme celui-ci :

```
PANIC:  could not write to file "pg_wal/xlogtemp.9109":
        No space left on device
```

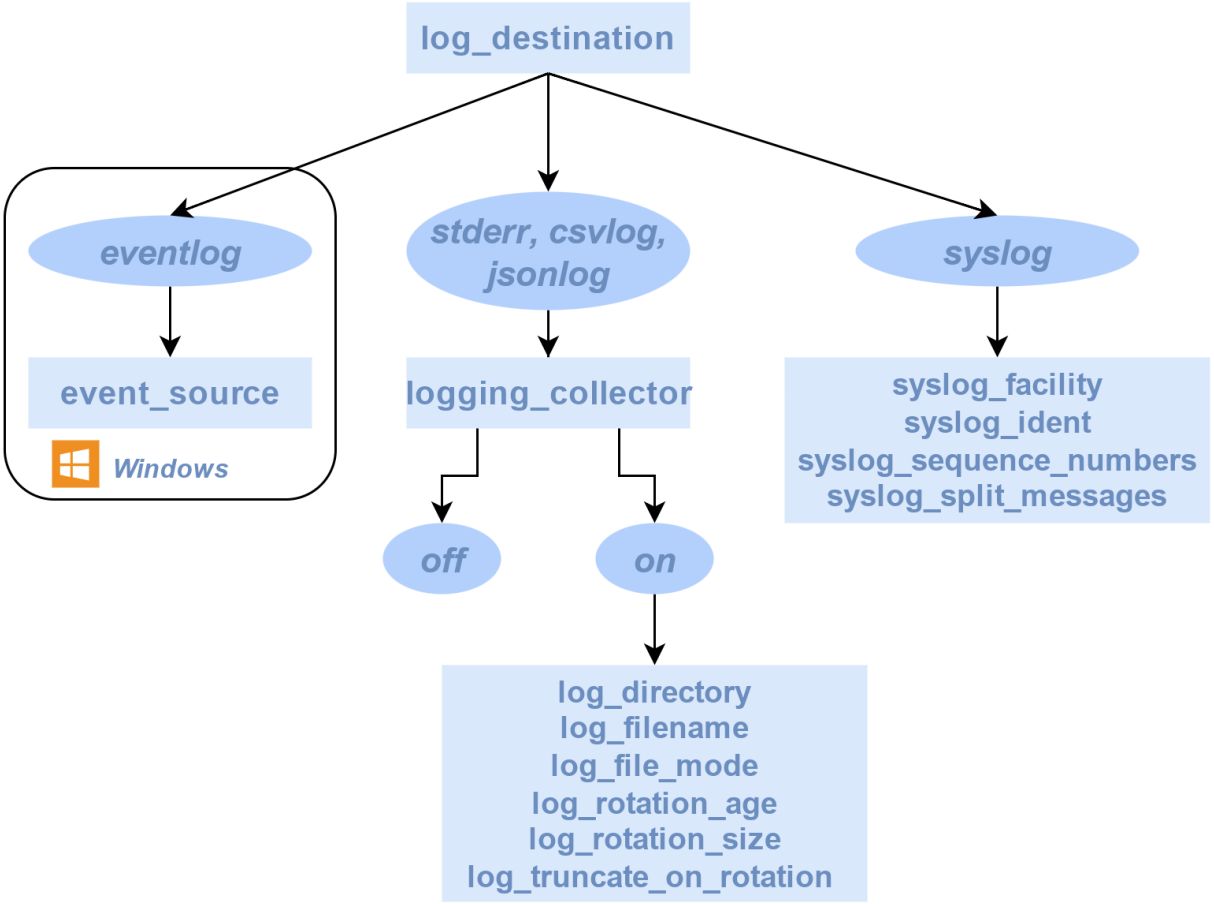
Là, le problème est très simple : PostgreSQL n'arrive pas à créer un journal de transactions à cause d'un manque d'espace sur le disque. Du coup, le système ne peut plus fonctionner, il panique et s'arrête.

Un outil comme tail_n_mail peut aider à détecter automatiquement ce genre de problème et à envoyer un mail à la personne d'astreinte. Il n'est d'ailleurs pas si rare que PostgreSQL, après un problème grave, redémarre si vite qu'il n'y a aucune conséquence visible sérieuse, et que ce genre de détection automatique soit le seul symptôme d'un problème.

Un autre événement à suivre est le changement de la configuration du serveur, et surtout la valeur des paramètres modifiés. PostgreSQL envoie un message niveau `LOG` lorsque la configuration est relue. Il indique aussi les nouvelles valeurs des paramètres, ainsi que les paramètres modifiés qu'il n'a pas pu prendre en compte (cela peut arriver pour tous les paramètres exigeant un redémarrage du serveur).

Là-aussi, tail_n_mail est l'outil adapté pour être prévenu dès que la configuration du serveur est relue.

15.4.3 Où tracer ?



15.4.4 Configuration de la destination des traces

- `log_destination` :
  - `stderr` / `csvlog` / `jsonlog` (v15)
  - `syslog` / `eventlog`
- `logging_collector` : géré par PostgreSQL (Red Hat)
  - `log_directory`, `log_filename`, `log_file_mode`
  - `log_rotation_age`, `log_rotation_size`, `log_truncate_on_rotation`
- Sinon : si `off`, penser à `logrotate` (Debian)
- `syslog` (Unix)
  - `syslog_facility`, `syslog_ident`
  - `syslog_sequence_numbers`, `syslog_split_messages`
- `eventlog` (Windows) : `event_source`

PostgreSQL peut envoyer les traces sur plusieurs destinations selon la valeur de `log_destination` :

stderr, csvlog et jsonlog

`stderr` (valeur par défaut, y compris sur Debian & Red Hat), `csvlog` et `jsonlog`, disponibles sur toutes les plateformes, correspondent à la sortie des erreurs.

La différence entre les trois réside dans le format des traces (respectivement texte simple ou CSV ou JSON). La sortie JSON n'existe que depuis la version 15 mais il est à noter qu'il existe une extension jsonlog[2], par Michaël Paquier, qui offre en plus le format JSON pour les versions antérieures.

Les paramètres suivants sont spécifiques à `stderr`, `csvlog` et `jsonlog`.

`logging_collector` indique ensuite si PostgreSQL doit s'occuper lui-même de la gestion des fichiers de logs.

S'il est à `on` (défaut sous Red Hat/CentOS/Rocky Linux) :

- `log_directory` désigne l'emplacement des journaux applicatifs. Par défaut, il est dans le répertoire de l'instance (sous-répertoire `log`, ou `pg_log` jusqu'en version 9.6 comprise) ;
- `log_filename` indique le nom des fichiers. Sa valeur varie suivant la distribution :

[2] https://github.com/michaelpq/pg_plugins/tree/master/jsonlog

  - `postgresql-%Y-%m-%d_%H%M%S.log` est le défaut des versions compilées (avec rotation quotidienne) ;
  - `postgresql-%a.log` est le défaut sur Red Hat/CentOS/Rocky Linux : on obtient une rotation quotidienne sur une semaine (`postgresql-Mon.log` , `postgresql-Tue.log` , etc.) ;

- `log_file_mode` précise les droits sur les fichiers (`0600` par défaut, les réservant à l'utilisateur sous lequel l'instance tourne). Une rotation est configurable suivant la taille (`log_rotation_size`) et la durée de vie (`log_rotation_age` , souvent `1d` , à garder en cohérence avec `log_filename`) ;
- `log_truncate_on_rotation` à `on` entraîne l'effacement de tout fichier dont le nom serait réutilisé, ce qui est en général une bonne idée si les mêmes noms de fichiers sont réutilisés.

Par contre, sous Debian, `logging_collector` est par défaut à `off` , les paramètres ci-dessus sont ignorés et la gestion des logs est gérée par le système d'exploitation :

- les traces vont dans `/var/log/postgresql/` , donc hors du PGDATA ;
- elles sont nommées en fonction de l'instance, de son nom et du nom fourni lors de la création avec l'outil `pg_ctlcluster` (donc pour l'instance installée par défaut, en version 14, le fichier sera `postgresql-14-main.log`), sauf à modifier le `pg_ctl.conf` de l'instance ;
- la rotation des fichiers est gérée par `logrotate` (comme les autres fichiers de log), et paramétrée dans `/etc/logrotate.d/postgresql-common` avec par défaut une rotation sur 10 jours.

Une instance compilée n'utilise pas non plus le *logging collector*.

syslog

`syslog` fonctionne uniquement sur un serveur Unix et est intéressant pour centraliser la configuration des traces.

Dans cette configuration, il reste à définir le niveau avec `syslog_facility` , et l'identification du programme avec `syslog_ident` . Les valeurs par défaut de ces deux paramètres sont généralement bonnes. Il est intéressant de modifier ces valeurs surtout si plusieurs instances de PostgreSQL sont installées sur le même serveur car cela permet de différencier leur traces.

`syslog_sequence_numbers` préfixe chaque message d'un numéro de séquence incrémenté automatiquement, pour éviter le message `--- last message repeated N times ---` , utilisé par un grand nombre d'implémentations de syslog. Ce comportement est activé par défaut. Quant à `syslog_split_messages` , s'il est activé, les messages envoyés à syslog sont divisés par lignes, elles-mêmes divisées pour tenir sur 1024 octets. Attention à ce que `syslog` soit configuré pour accepter des débits élevés quand on tient à tout tracer.

eventlog

`eventlog` est disponible uniquement sur Windows et alimente le journal des événements. Pour identifier les messages de PostgreSQL, il faut aussi renseigner `event_source` [3], qui par défaut est à « PostgreSQL ».

[3] https://docs.postgresql.fr/current/event-log-registration.html

15.4.5 Niveau des traces

- `log_min_messages`
  - défaut : `panic` / `fatal` / `log` / `error` / `warning`
- `log_min_error_statement`
  - défaut : `error` (ou `warning`)
- `log_error_verbosity`
  - `default` / `terse` / `verbose`

`log_min_messages` est le paramètre à configurer pour avoir plus ou moins de traces. Par défaut, PostgreSQL enregistre tous les messages de niveau `panic`, `fatal`, `log`, `error` et `warning`. Cela peut sembler beaucoup mais, dans les faits, c'est assez discret. Cependant, il est possible de descendre le niveau ou de l'augmenter.

`log_min_error_statement` indique à partir de quel niveau la requête est elle-aussi tracée. Par défaut, la requête n'est tracée que si une erreur est détectée. Généralement, ce paramètre n'est pas modifié, sauf dans un cas précis. Les messages d'avertissement (niveau `warning`) n'indiquent pas la requête qui a généré l'affichage du message. Cela est assez important, notamment dans le cadre de l'utilisation d'antislash dans les chaînes de caractères. On verra donc parfois un abaissement au niveau `warning` pour cette raison.

`log_error_verbosity` vaut `default`, qui convient généralement. Si une requête est tracée, plusieurs lignes peuvent apparaître, chacune de niveau `DETAIL`, `HINT`, `QUERY` ou `CONTEXT`. La valeur `terse` les masque. À l'inverse, `verbose` rajoute encore d'autres informations sur le code source d'origine.

15.4.6 Tracer les requêtes et leur durée

- Toutes les requêtes :
  - `log_min_duration_statement` (ex : `1s`)
  - ou `log_statement` + `log_duration`
- Extrait aléatoire :
  - `log_transaction_sample_rate`
  - `log_statement_sample_rate` + `log_min_duration_sample`

Pour répérer les problèmes de performances, il est intéressant de pouvoir tracer les requêtes et leur durée d'exécution. PostgreSQL propose deux solutions à cela.

log_statement & log_duration :

La première solution disponible concerne les paramètres `log_statement` et `log_duration`. Le premier permet de tracer toute requête exécutée si la requête correspond au filtre indiqué par le paramètre :

- `none` : aucune requête n'est tracée ;
- `ddl` : seules les requêtes DDL (autrement dit de changement de structure) sont tracées ;
- `mod` : seules les requêtes de changement de structure et de données sont tracées ;
- `all` : toutes les requêtes sont tracées.

Le paramètre `log_duration` est un simple booléen. S'il vaut `true` ou `on`, chaque requête exécutée envoie en plus un message dans les traces indiquant la durée d'exécution de la requête. Évidemment, il vaut mieux alors configurer `log_statement` à `all`, ou il sera impossible de dire à quelles requêtes les temps correspondent.

Donc pour tracer toutes les requêtes et leur durée d'exécution, une solution serait de réaliser la configuration suivante :

```
log_statement = 'all'
log_duration = on
```

Une requête générera deux entrées dans les traces, de cette façon :

```
2019-01-28 15:43:27.993 CET [25575] LOG:  statement: SELECT * FROM pg_stat_activity;
2019-01-28 15:43:27.999 CET [25575] LOG:  duration: 7.093 ms
```

log_min_duration_statement :

Il est préférable de désactiver ces deux paramètres et de préférer `log_min_duration_statement`. Son but est d'abord de cibler les requêtes lentes, par exemple celles qui prennent plus de deux secondes à s'exécuter :

```
log_min_duration_statement = '2s'
```

La requête et la durée d'exécution seront alors tracées dans le même message :

```
2019-01-28 15:49:56.193 CET [32067] LOG:
  duration: 2906.270 ms
  statement: insert into t1 select i, i from generate_series(1, 200000) as i;
```

En plus de la trace par `log_min_duration_statement`, rien n'interdit de tracer des requêtes sensibles, notamment le DDL :

```
log_statement = 'ddl'
```

Échantillonnage :

Quelle que soit la méthode, tracer toutes les requêtes peut poser problème pour de simples raisons de volumétrie du fichier de traces. Même s'il est possible de configurer finement la durée à partir de laquelle une requête est tracée, il faut bien comprendre que plus la durée minimale est importante, plus la vision des performances est partielle. Passeront ainsi « sous le radar » des requêtes relativement rapides mais très nombreuses qui, ensemble, peuvent représenter l'essentiel de la charge.

Cela étant dit, laisser 0 en permanence n'est pas recommandé. Il est préférable de configurer ce paramètre à une valeur plus importante en temps normal pour détecter seulement les requêtes longues et, lorsqu'un audit de la plateforme est nécessaire, passer temporairement ce paramètre à une valeur très basse (0 étant le mieux).

Une nouvelle fonctionnalité a donc été ajoutée : tracer une certaine proportion des requêtes ou des transactions.

`log_transaction_sample_rate`, à partir de PostgreSQL 12, indique une proportion de **transactions** à tracer. Par exemple, en le configurant à `0.01`, toutes les requêtes d'un centième des transactions, choisies au hasard, seront tracées.

De manière similaire, à partir de PostgreSQL 13, `log_statement_sample_rate` indique la proportion de **requêtes** à tracer, parmi celles durant plus d'une certaine durée, à indiquer dans `log_min_duration_sample` :

```
log_min_duration_sample = '10ms'
log_statement_sample_rate = 0.01
```

Évidemment, une requête dépassant la durée de `log_min_duration_statement` sera toujours tracée.

15.4.7 Configuration : tracer certains comportements

- `log_connections`, `log_disconnections`
- `log_autovacuum_min_duration`
- `log_checkpoints`
- `log_lock_waits` (mini 1s)
- `log_recovery_conflict_waits` (v14+)

En dehors des erreurs et des durées des requêtes, il est aussi possible de tracer certaines activités ou comportements. Le paramétrage par défaut est peu bavard, et il est généralement conseillé d'activer tous les paramètres qui suivent.

`log_connections` et son pendant `log_disconnections`, à `on`, permettent de suivre qui se (dé)connecte, depuis où, et durant combien de temps :

```
2019-01-28 13:34:32 CEST LOG:  connection received: host=[local]
2019-01-28 13:34:32 CEST LOG:  connection authorized: user=u1 database=b1
…
2016-09-01 13:34:35 CEST LOG:  disconnection: session time: 0:01:04.634
                               user=u1 database=b1 host=[local]
```

Jusque PostgreSQL 16 inclus, l'« authentification » par la méthode `trust` n'est pas tracée, au contraire des autres méthodes (depuis PostgreSQL 14), il y a juste la ligne de connexion. L'utilisation de cette méthode dangereuse (et à procrire) est donc difficile à auditer avant PostgreSQL 17.

Il est possible de récupérer cette durée de session pour calculer leur durée moyenne. Cette information est importante pour savoir si un outil de pooling de connexions a un intérêt.

`log_autovacuum_min_duration` équivaut à `log_min_duration_statement`, mais pour l'autovacuum. Le but est de tracer son activité, au-delà d'une certaine durée, de vérifier qu'il passe suffisamment fréquemment et rapidement.

`log_checkpoints` à `on` ajoute un message dans les traces pour indiquer qu'un checkpoint commence ou se termine, auquel cas s'ajoutent des statistiques :

```
2019-01-28 13:34:17 CEST LOG: checkpoint starting: xlog
2019-01-28 13:34:20 CEST LOG: checkpoint complete:
  wrote 13115 buffers (80.0%);
  0 WAL file(s) added, 0 removed, 0 recycled;
  write=3.007 s, sync=0.324 s, total=3.400 s;
```

```
sync files=16, longest=0.285 s, average=0.020 s;
distance=404207 kB, estimate=404207 kB
```

Le message indique donc en plus le nombre de blocs écrits sur disque, le nombre de journaux de transactions ajoutés, supprimés et recyclés. Il est rare que des journaux soient ajoutés, ils sont plutôt recyclés. Des journaux sont supprimés quand il y a eu une très grosse activité qui a généré plus de journaux que d'habitude. Les statistiques incluent aussi la durée des écritures, de la synchronisation sur disque, la durée totale, etc. Le plus important est de pouvoir vérifier que l'écriture des checkpoints est bien régulière (essentiellement périodique).

`log_lock_waits` à `on` permet de tracer les attentes de verrous (par exemple, un `UPDATE` bloqué par un autre `UPDATE`, un `SELECT` bloqué par `TRUNCATE` ou un `VACUUM FULL`, etc…) Lorsque l'attente dépasse la durée indiquée par le paramètre `deadlock_timeout` (1 seconde par défaut), un message est enregistré, comme dans cet exemple :

```
2019-01-28 13:38:40 CEST LOG:  process 15976 still waiting for
                               AccessExclusiveLock on relation 26160 of
                               database 16384 after 1000.123 ms
2019-01-28 13:38:40 CEST STATEMENT:  DROP TABLE t1;
```

Ici, un `DROP TABLE` attend depuis 1 seconde de pouvoir poser un verrou exclusif sur une relation.

Plus ce type de message apparaît dans les traces, plus des contentions ont lieu sur certains objets, ce qui peut diminuer fortement les performances. Ces messages peuvent permettre d'analyser la cause première d'une accumulation de verrous, à condition que les requêtes soient tracées.

En version 14 apparaît le paramètre `log_recovery_conflict_waits`. Ce dernier, une fois activé, permet de tracer toute attente due à un conflit de réplication. Il n'est donc valable et pris en compte que sur un serveur secondaire.

Ajoutons en passant qu'à partir de PostgreSQL 17, il est aussi possible de tracer les connexions avec un trigger sur événement, vers des tables. La documentation officielle fournit un exemple[4].

15.4.8 Repérer les fichiers temporaires

- Exemple :

```
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp9894.0",
      size 26927104
```

- `log_temp_files` à activer !
- Alerte : problème potentiel de performances

[4] https://docs.postgresql.fr/current/event-trigger-database-login-example.html

Quand PostgreSQL ne peut effectuer un tri en mémoire, il le fait sur disque dans un fichier temporaire, ce qui est beaucoup plus lent qu'en mémoire, même avec un SSD. Typiquement, cela concerne le tri de données et le hachage, quand la valeur du paramètre `work_mem` ne permet pas de tout faire en mémoire. Cela ne sera pas forcément gênant pour une grosse requête ponctuelle, mais, répétés, ces fichiers peuvent avoir un impact sur la performance du système. Ils sont parfois inévitables quand on brasse beaucoup de données.

Être averti lors de la création de ce type de fichiers peut être intéressant, mais ils sont parfois trop fréquents pour que ce soit réaliste. Il est préférable de faire analyser après coup un fichier de traces pour savoir combien de fichiers temporaires ont été créés, et de quelles tailles. Cela peut mener à vérifier les requêtes exécutées, les optimiser, vérifier la configuration, réviser la valeur de `work_mem`…

Le paramètre `log_temp_files` à 0 permet de tracer toutes les créations de fichiers temporaires, comme ici :

```
2019-01-28 13:41:11 CEST LOG:  temporary file: path
                               "base/pgsql_tmp/pgsql_tmp15617.1",
                               size 59645952
```

Pour le même tri, il peut y avoir de nombreux fichiers temporaires. De plus, la requête est aussi tracée, et si elle est longue et fréquente, le volume de traces peut être conséquent.

15.4.9 Configuration : divers

- `log_line_prefix`
  - Conseillé : `%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h`
- `lc_messages` = `C`
- `log_timezone` = `'Europe/Paris'`

Le paramètre `log_line_prefix` permet d'ajouter un préfixe à une trace. Le défaut (`'%m [%p] '`, soit horodatage et numéro de processus), est insuffisant :

```
2021-02-05 14:12:12.343 UTC [2917] LOG:  duration: 3.276 ms
                           statement: SELECT count(*) FROM pgbench_branches ;
```

Il est conseillé de rajouter le nom de l'application cliente, le nom de l'utilisateur, le nom de la base, etc. Une valeur habituellement conseillée pour pgBadger[5], pour une sortie vers `stderr`, est :

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
```

ce qui nous donnera ce genre de traces :

[5] https://pgbadger.darold.net/documentation.html#POSTGRESQL-CONFIGURATION

```
2021-02-05 14:30:01 UTC [3006]: user=durand,db=bench,app=test,client=[local]
    LOG:  duration: 0.184 ms  statement: SELECT count(*) FROM pgbench_branches ;
```

Pour une sortie vers `syslog`, l'horodatage est inutile :

```
log_line_prefix = 'user=%u,db=%d,app=%a,client=%h '
```

Par défaut, les traces sont enregistrées dans la locale par défaut du serveur. Des traces en français peuvent présenter certains intérêts pour des débutants, mais ont plusieurs gros inconvénients : un moteur de recherche renverra beaucoup moins de résultats avec des traces en français qu'en anglais, et les outils d'analyse automatique des traces se basent principalement sur des traces en anglais. Donc, il vaut mieux préciser systématiquement :

```
lc_messages = 'C'
```

Quant à `log_timezone`, il permet de choisir le fuseau horaire pour l'horodatage des traces. C'est inestimable quand on administre différents serveurs dispersés sur la planète.

```
log_timezone = 'UTC'
```

```
log_timezone = 'Europe/Paris'
```

15.5 OUTILS D'ANALYSE DES TRACES

- Beaucoup d'outils existent
  - en temps réel / rétro-analyse
  - généralistes / spécifiques PostgreSQL
- Exemples :
  - pgBadger
  - logwatch
  - tail_n_mail

Il existe de nombreux programmes qui analysent les traces. On peut distinguer deux catégories :

- ceux qui le font en temps réel ;
- ceux qui le font après coup (de la rétro-analyse en fait).

Mais également :

- ceux généralistes, connaissant plus ou moins bien beaucoup d'outils et logiciels ;
- ceux dédiés à PostgreSQL.

L'analyse en temps réel des traces permet de réagir rapidement à certains messages. Par exemple, il est important d'avoir une réaction rapide à l'archivage échoué d'un journal de transactions, ainsi qu'en cas de manque d'espace disque. Dans cette catégorie, il existe des outils généralistes comme logwatch[6], et des outils spécifiques pour PostgreSQL comme tail_n_mail[7].

L'analyse après coup permet une analyse plus fine, se terminant généralement par un rapport en HTML, parfois avec des graphes. Cette analyse plus fine nécessite des outils spécialisés. Il en a existé plusieurs qui ne sont plus maintenus. La référence dans le domaine est pgBadger[8].

[6] https://sourceforge.net/projects/logwatch/
[7] https://bucardo.org/tail_n_mail/
[8] https://pgbadger.darold.net/

15.5.1 pgBadger

- Site officiel : https://pgbadger.darold.net/
- Licence : PostgreSQL
- Analyse des traces de durée d'exécution des requêtes
- Analyse des traces du `VACUUM`, des connexions, des checkpoints
- Compatible syslog, stderr, csvlog

Gilles Darold a créé pgBadger, un analyseur des journaux applicatifs de PostgreSQL. Il permet de générer des rapports détaillés depuis ceux-ci. pgBadger est très souvent utilisé pour déterminer les requêtes à améliorer en priorité pour accélérer son application basée sur PostgreSQL. C'est certainement le meilleur outil actuel de rétro-analyse d'un fichier de traces PostgreSQL, au point qu'il est cité dans le manuel de PostgreSQL.

pgBadger est écrit en Perl et est facilement extensible si vous avez besoin de rapports spécifiques.

Il est conçu pour traiter rapidement de gros fichiers de traces avec une mémoire réduite, mais permet d'exploiter plusieurs CPU pour accélérer considérablement l'analyse.

15.5.2 pgBadger : exemple de rapport

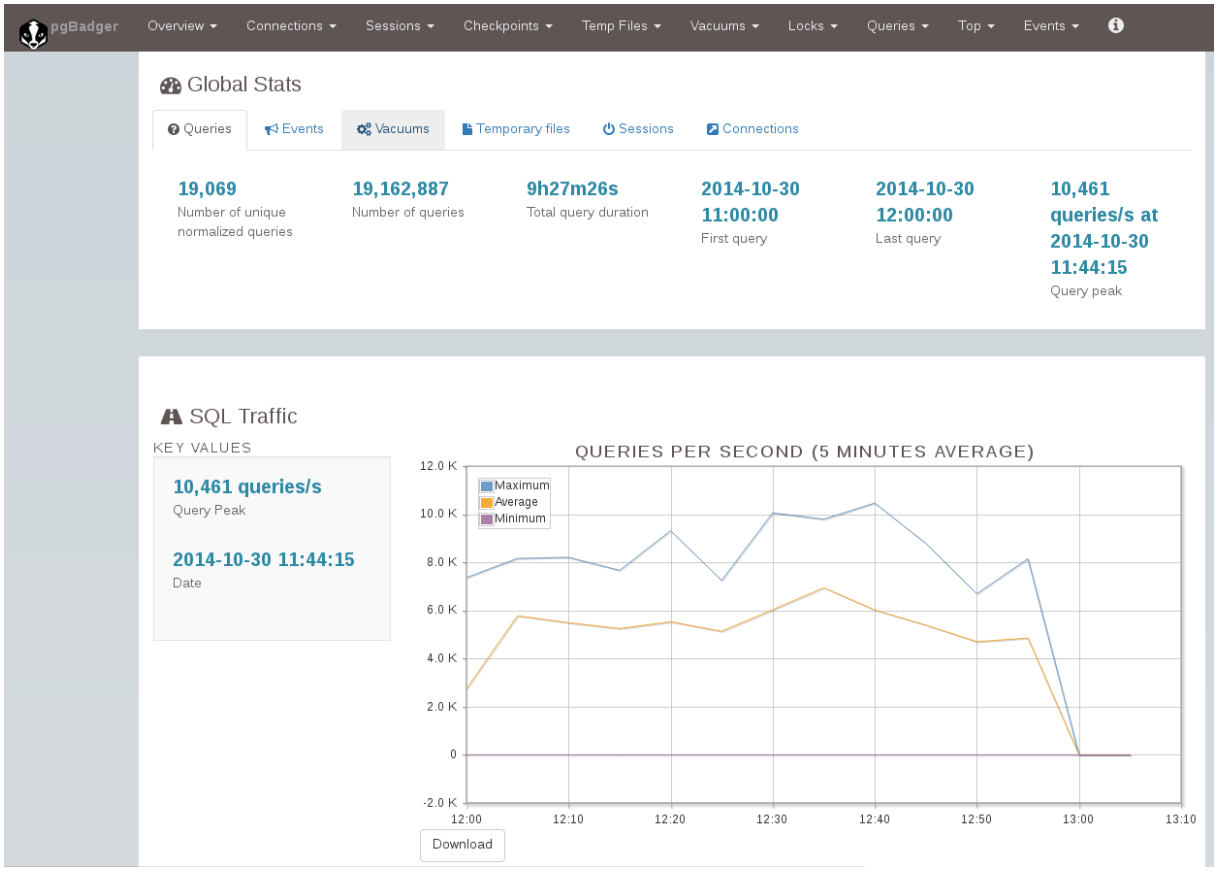


Figure 15/ .1: Capture pgBadger

15.5.3 Utiliser pgBadger

- Script Perl
- Traite les journaux applicatifs
- Recherche des informations
  - sur les requêtes (normalisées) et leur durée d'exécution
  - sur les connexions et sessions
  - sur les checkpoints
  - sur l'autovacuum
  - sur les verrous
  - etc.
- Génération d'un rapport HTML très détaillé

pgBadger s'utilise en ligne de commande : il suffit de lui fournir le ou les fichiers de trace à analyser et il rend un rapport HTML sur les requêtes exécutées, sur les connexions, sur les bases, etc. Le rapport est très complet. Les graphes sont zoomables. Les requêtes sont reformatées pour plus de lisibilité.

15.5.4 Configurer PostgreSQL pour pgBadger

- Minimum :
  - `log_destination`
  - `log_line_prefix`
  - `lc_messages` = `C`
- Base :
  - `log_connections`, `log_disconnections`
  - `log_checkpoints`
  - `log_lock_waits`
  - `log_temp_files`
  - `log_autovacuum_min_duration`
- Pour un audit :
  - `log_min_duration_statement = 0` (attention !)

pgBadger a besoin d'un minimum d'informations dans les traces : timestamp (`%t`), pid (`%p`) et numéro de ligne dans la session (`%l`). Il n'y a pas de conseil particulier sur la destination des traces (en dehors de `eventlog` que pgBadger ne sait pas traiter). De même, le préfixe des traces est laissé au choix de l'utilisateur. Dans certains cas, il faudra le préciser à pgBadger avec l'option `--prefix` (par exemple si la valeur de `log_line_prefix` a changé entre le début et la fin du fichier). Une configuration courante et vraiment informative est par exemple :

```
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

Noter que même sans cela, pgBadger essaie de récupérer les informations sur les adresses IP, les utilisateurs connectés, les bases de données à partir des traces sur les connexions. Ajouter le joker `%e` (code SQLState) permet d'obtenir un tableau sur les erreurs.

La langue des traces doit être l'anglais (`lc_messages` à `C`), ce qui de toute manière est la valeur conseillée.

Pour tracer les requêtes, il est préférable de passer par `log_min_duration_statement` plutôt que `log_statement` et `log_duration` : pgBadger fera plus facilement l'association entre chaque requête et sa durée :

```
log_min_duration_statement = 0
log_statement = none
log_duration = off
```

Comme déjà dit plus haut, `log_min_duration_statement = 0` peut générer un énorme volume de traces et n'est souvent configuré ainsi que le temps d'un audit. Avec une valeur supérieure, pgBadger ne verra absolument pas les requêtes les plus rapides.

Il est aussi conseillé de tirer parti d'autres informations dans les traces :

- `log_checkpoints` pour des statistiques sur les checkpoints ;
- `log_connections` et `log_disconnections` pour des informations sur les connexions et déconnexions ;
- `log_lock_waits` pour des statistiques sur les verrous en attente ;
- `log_temp_files` pour des statistiques sur les fichiers temporaires ;
- `log_autovacuum_min_duration` pour des statistiques sur l'activité de l'autovacuum.

15.5.5 Options de pgBadger

- Génération :

```
pgbadger … -o rapport.html postgresql-Mon.log postgresql-Tue.log …
```

- Très nombreuses options, dont :
  - `--outfile`
  - `--prefix`
  - `--begin` / `--end`
  - `--dbname`, `--dbuser`, `--dbclient`, `--appname`
  - `--jobs`

Pour l'utilisation la plus simple, il suffit de fournir au script en paramètre les noms des différents fichiers de traces, éventuellement compressés, pour obtenir le rapport sur tous les événements qu'il y trouvera.

Il existe énormément d'options[9], et Gilles Darold en rajoute fréquemment. L'aide fournie sur le site web officiel les cite intégralement.

Parmi les plus utiles, `--outfile` permet d'indiquer le nom du rapport (par défaut `out.html`).

`--prefix` permet de préciser une valeur de `log_line_prefix` si la détection automatique échoue.

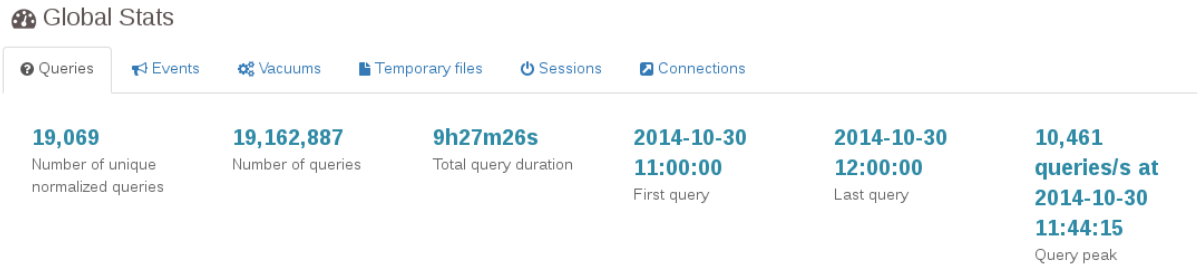
Le rapport peut être restreint à une tranche horaire précise avec `--begin` et `--end` (par exemple `--begin '2019-04-01 16:00:00'`).

Pour mieux cibler le rapport, il est possible de restreindre l'analyse à un utilisateur (`--dbuser`), une base de données (`--dbname`), une machine cliente (`--dbclient`), ou une application (`--appname`, si elle définit bien `application_name` à la connexion).

`--jobs` permet de paralléliser la lecture des traces sur plusieurs processeurs. Attention, de très gros fichiers les satureront probablement plusieurs minutes.

[9] https://pgbadger.darold.net/documentation.html#SYNOPSIS

15.5.6 pgBadger : exemple 1



Au tout début du rapport, pgBadger donne des statistiques générales sur les fichiers de traces.

Dans les informations importantes se trouve le nombre de requêtes normalisées. En fait, des requêtes telles que :

```
SELECT * FROM utilisateurs WHERE id = 1;
```

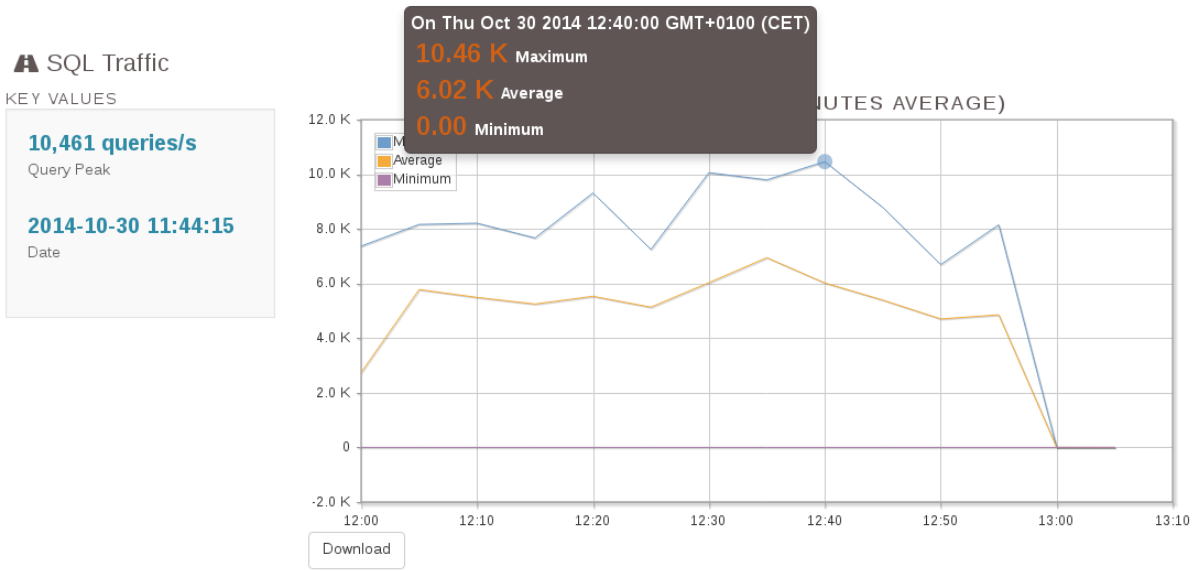
et

```
SELECT * FROM utilisateurs WHERE id = 2;
```

sont différentes car elles ne vont pas récupérer la même fiche utilisateur. Cependant, en enlevant la partie constante, les requêtes sont identiques. La seule différence est la ligne récupérée mais pas la requête. pgBadger est capable de faire cette différence. Toute constante, qu'elle soit de type numérique, textuelle, horodatage ou booléenne, peut être supprimée de la requête. Dans l'exemple ci-dessus, pgBadger a comptabilisé environ 19 millions de requêtes, mais seulement 19 069 requêtes différentes après normalisation. Ceci est important dans le fait où nous n'allons pas devoir travailler sur plusieurs millions de requêtes mais « seulement » sur 19 000.

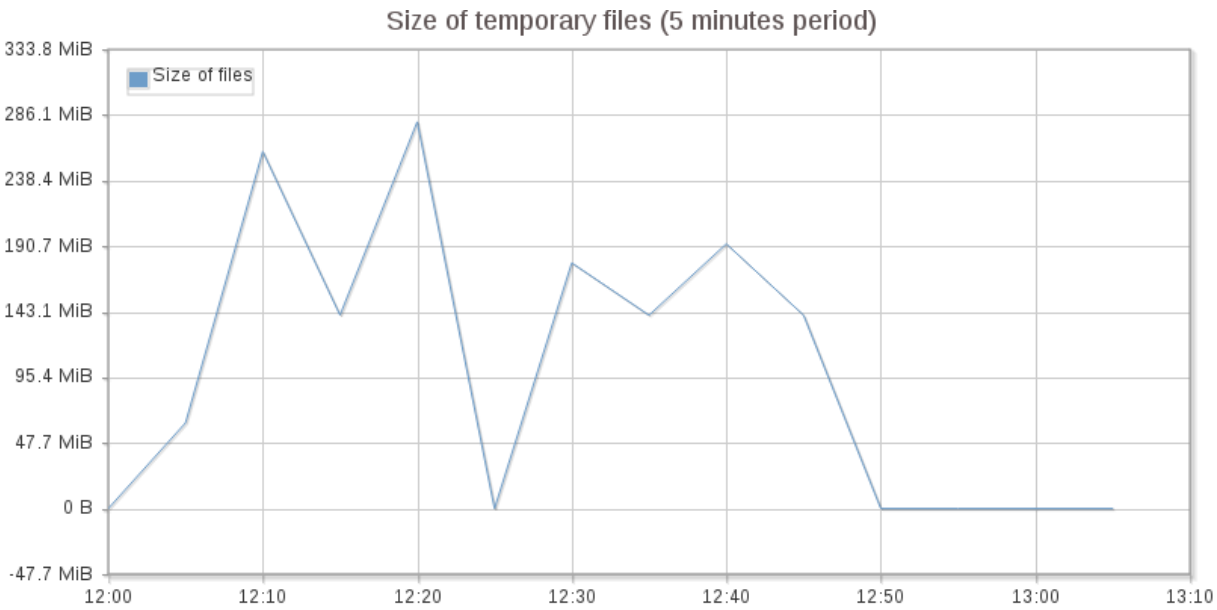
Autre information intéressante, la durée d'exécution totale des requêtes. Ici, nous avons 9 heures d'exécution de requêtes. Cependant, les traces ne couvrent que 11 h à 12 h, soit une heure. Cela indique que le serveur est assez sollicité. Il est fréquent que la durée d'exécution sérielle des requêtes soit plusieurs fois plus importantes que la durée des traces.

15.5.7 pgBadger : exemple 2



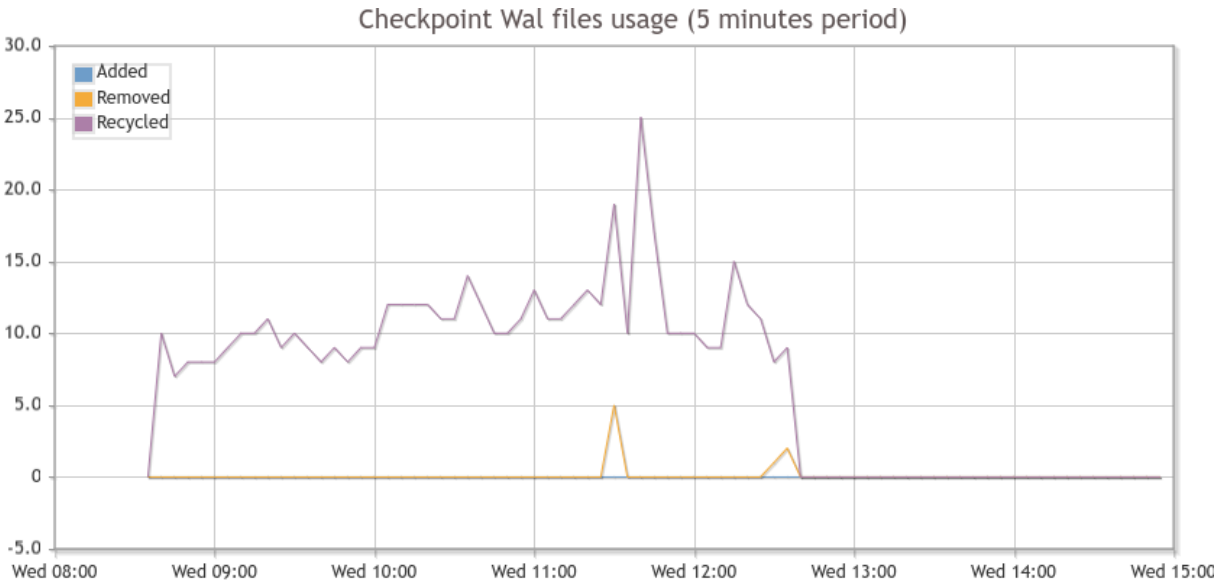
Ce graphe indique le nombre de requêtes par seconde : en fait, environ 33 requêtes/s en pointe.

15.5.8 pgBadger : exemple 3



Ce graphe affiche en vert le nombre de fichiers temporaires créés sur la durée des traces. La ligne bleue correspond à la taille des fichiers. Nous remarquons ainsi la création à peu près régulière de fichiers temporaires, à raison d'environ 200 Mo par tranche de 5 minutes.

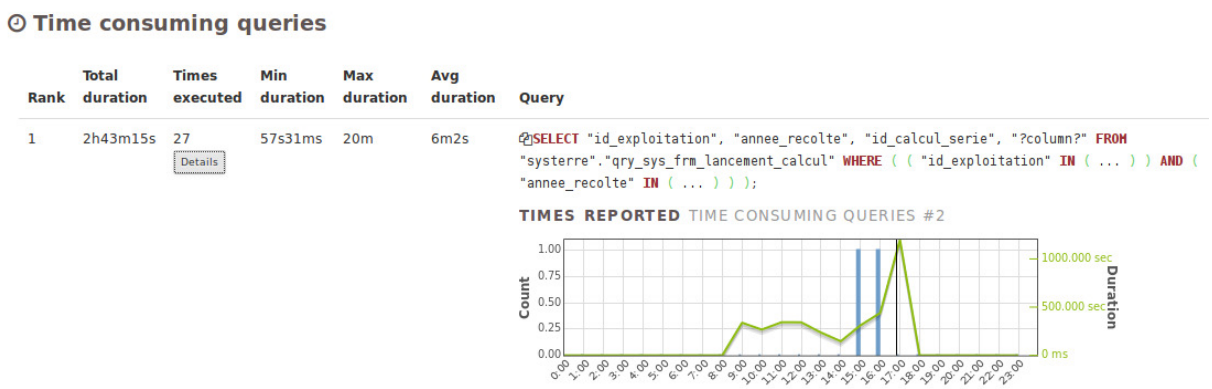
15.5.9 pgBadger : exemple 4



De grosses écritures peuvent mener à un nombre important de journaux. Cette courbe montre une création relativement régulière de journaux. En fait, il s'agit uniquement de recyclage de journaux existants : PostgreSQL n'a pas à en créer et en initialiser en masse. Le pic n'est que de 5 journaux de 16 Mo à la minute.

Plus bas dans l'onglet *Checkpoints* figurent des informations sur l'écart (en octets) entre deux checkpoints, la durée d'écriture, la durée de synchronisation sur le disque, et le nombre d'avertissements sur des checkpoints non périodiques.

15.5.10 pgBadger : exemple 5



Time consuming queries

Rank	Total duration	Times executed	Min duration	Max duration	Avg duration	Query
1	2h43m15s	27 Details	57s31ms	20m	6m2s	SELECT "id_exploitation", "annee_recolte", "id_calcul_serie", "?column?" FROM "systerre"."qry_sys_frm_lancement_calcul" WHERE (("id_exploitation" IN (...)) AND ("annee_recolte" IN (...)));

TIMES REPORTED TIME CONSUMING QUERIES #2

Le plus important est certainement l'onglet *Top/Time consuming queries*. Il liste les requêtes qui ont pris le plus de temps, que ce soit parce que les requêtes en question sont vraiment très lentes ou parce qu'elles sont exécutées un très grand nombre de fois.

Ci-dessus n'est affichée que la première requête de la liste. Le bouton *Details* permet d'afficher la courbe indiquant les heures d'occurrences et les durées.

Nous remarquons d'ailleurs dans cet exemple qu'avec la seule requête affichée, nous arrivons à un total de 2 h 43. Le premier exemple nous indique que l'exécution sérielle des requêtes aurait pris 9 heures. En ne travaillant que sur elle, nous travaillons en fait sur presque le tiers du temps total d'exécution des requêtes comprises dans les traces.

Les autres requêtes les plus consommatrices ne sont pas listées ici, mais il est peu probable que l'on ait à travailler sur les 19 000 requêtes normalisées. Il est fréquent que l'essentiel de la charge soit contenu dans les quelques premières requêtes du tableau. Ce sont évidemment les premières cibles pour une tentative d'optimisation : réécriture, modification du paramétrage, index dédiés…

15.5.11 logwatch

- Outil externe écrit en Perl
  - https://sourceforge.net/projects/logwatch/
  - Licence MIT
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs
- Exemple :

```
/usr/sbin/logwatch --detail Med --service postgresql --range All
```

Surveiller ses journaux applicatifs (ou fichiers de trace) est une activité nécessaire mais bien souvent rébarbative. De nombreux programmes existent pour nous faciliter la tâche, mais le propre de ces programmes est d'être exhaustif. De plus, ils demandent une action de la part de l'administrateur, à savoir : penser à aller les regarder.

logwatch est une petite application permettant d'analyser les journaux de nombreux services et de produire un rapport synthétique. Le nombre de services connus est impressionnant et il est simple d'en ajouter de nouveaux. logwatch est le plus souvent intégré à votre distribution Linux. Depuis la version 7.4.2 il sait analyser les traces de PostgreSQL.

Après son installation, il se lancera tous les jours grâce au fichier `/etc/cron.daily/00logwatch`. Vous recevrez tous les jours par mail les rapports des événements importants détectés dans les fichiers de traces. Vous pouvez aussi le lancer manuellement ainsi :

```
/usr/sbin/logwatch --detail Low --service postgresql --range All
```

`--range All` indique que l'on veut un rapport sur tous les fichiers de traces existants. La valeur par défaut est `Yesterday`. Pour n'avoir un rapport que sur la journée, choisir `Today`.

Avec le niveau de détail minimal (`--detail` à `Low`), seuls les messages de type `FATAL`, `PANIC` et `ERROR` seront remontés. Avec la valeur `Med`, apparaîtront aussi les messages de type `WARNING` et `HINTS`.

Exemple de résultat :

```
################### Logwatch 7.3.6 (05/19/07) ####################
       Processing Initiated: Tue Dec 13 12:28:46 2011
       Date Range Processed: all
       Detail Level of Output: 5
       Type of Output/Format: stdout / text
       Logfiles for Host: devel
 ##################################################################
--------------------- PostgreSQL Begin ------------------------
Fatals:
-------
9 times:
[2011-12-04 04:28:46 +/-9 day(s)]  password authentication failed for
                                   user "postgres"
8 times:
[2011-11-21 10:15:01 +/-11 day(s)]  terminating connection due to
                                    administrator command
…
Errors:
-------
7 times:
[2011-11-14 12:20:44 +/-58 minute(s)]  relation "COUNTRIES" does not exist
5 times:
[2011-11-14 12:21:24 +/-59 minute(s)]  syntax error at or near
                                       "role_permission_view"
…
Warnings:
---------
55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  terminating connection because of
                                   crash of another server process
Hints:
------
55 times:
[2011-11-18 12:26:39 +/-6 day(s)]  In a moment you should be able to
                                   reconnect to the database and repeat
                                   you command.
14 times:
[2011-12-08 09:47:16 +/-19 day(s)]  No function matches the given name and
                                    argument types. You might need to add
                                    explicit type casts.
---------------------- PostgreSQL End -------------------------
###################### Logwatch End #########################
```

logwatch dispose de nombreuses options, et la page de manuel est certainement la meilleure documentation à l'heure actuelle.

15.5.12 tail_n_mail

- Outil externe écrit en Perl
- À exécuter périodiquement
- Analyse le contenu des journaux applicatifs
- Envoie un mail s'il détecte certains motifs

tail_n_mail est un outil écrit par la société EndPointCorporation. Son but est d'analyser périodiquement le contenu des fichiers de traces et d'envoyer un mail en cas de la détection d'un motif d'intérêt. Par exemple, il peut envoyer un mail lorsqu'il rencontre un message de niveau `PANIC` ou `FATAL` dans les traces. Ainsi une personne d'astreinte sera prévenue rapidement et pourra agir en conséquence.

15.5.13 Configurer tail_n_mail

```
EMAIL: astreinte@dalibo.com
MAILSUBJECT: HOST Postgres fatal errors (FILE)
FILE: /var/log/postgresql-%Y-%m-%d.log
INCLUDE: PANIC:
INCLUDE: FATAL:
EXCLUDE: database ".+" does not exist
INCLUDE: temporary file
INCLUDE: reloading configuration files
```

Les clés `INCLUDE` et `EXCLUDE` permettent d'indiquer les motifs à inclure et à exclure respectivement. Tout motif non inclus ne sera pas pris en compte. Cette configuration permet donc d'envoyer un mail à l'adresse `astreinte@dalibo.com` à chaque fois qu'un message contenant les mots `PANIC`, `FATAL`, `temporary file` ou `reloading configuration files` sont enregistrés dans les traces. Par contre, tous les messages contenant la phrase `database ... does not exist` ne sont pas pris en compte.

15.5.14 tail_n_mail : exemple

Exemple:

```
[1] Between lines 123005 and 147976, occurs 39 times.
First:  Jan  1 00:00:01 rojogrande postgres[4306]
Last:   Jan  1 10:30:00 rojogrande postgres[16854]
Statement: user=root,db=rojogrande
        FATAL: password authentication failed for user "root"
```

Voici un exemple de mail envoyé:

```
Matches from /var/log/postgresql/postgresql-10-main.log: 42
Date: Fri Jan  1 10:34:00 2010
Host: pollo

[1] Between lines 123005 and 147976, occurs 39 times.
First:  Jan  1 00:00:01 rojogrande postgres[4306]
Last:   Jan  1 10:30:00 rojogrande postgres[16854]
Statement:  user=root,db=rojogrande FATAL:  password authentication failed
                                            for user "root"

[2] Between lines 147999 and 148213, occurs 2 times.
First:  Jan  1 10:31:01 rojogrande postgres[3561]
Last:   Jan  1 10:31:10 rojogrande postgres[15312]
Statement: FATAL  main: write to worker pipe failed -(9) Bad file descriptor

[3] (from line 152341)
PANIC:  could not locate a valid checkpoint record
```

15.6 STATISTIQUES D'ACTIVITÉ

- Configuration
- Liste des vues statistiques
- Outils externes de classement

Les statistiques sont certainement les informations les plus simples à récupérer par un système de supervision. Il faut dans un premier temps s'assurer que la configuration est adéquate. Ceci fait, il est possible de lire les statistiques disponibles dans les vues proposées par défaut. Enfin, il existe quelques outils capables de récupérer des informations provenant des tables statistiques de PostgreSQL. Leur mise en place permettra une supervision facilitée.

15.6.1 Statistiques d'activité - configuration 1

- Tracer l'activité :
  - `track_activities` = `on`
- `pg_stat_activity` affiche
  - les requêtes
  - d'autres processus
- S'assurer que les requêtes ne sont pas tronquées :
  - `track_activity_query_size` = `10000` ou +
  - Récupérer le `query_id`
    - `compute_query_id = on`
  - Consulter les *wait events*
    - et joindre à `pg_wait_events` (v17+)

`pg_stat_activity` est une des vues les plus utilisées et est souvent le point de départ d'une recherche. Elle donne la liste des processus en cours sur l'instance, en incluant entre autres :

- le numéro de processus sur le serveur (`pid`) ;
- la base de données, le nom dutilisateur, l'adresse et le port du client ;

- les dates de début d'ordre, de transaction ou de session ;
- son statut (active ou non) ;
- la requête en cours, ou la dernière requête si la session ne fait rien ;
- le nom de l'application s'il a été renseigné avec le paramètre `application_name` ;
- le type de processus : session d'un utilisateur (*client backend*), processus interne…

```
SELECT datname, pid, usename, application_name,
       backend_start, state, backend_type, query
FROM   pg_stat_activity \gx

-[ RECORD 1 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26378
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236776+02
state            | ¤
backend_type     | autovacuum launcher
query            |
-[ RECORD 2 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26380
usename          | postgres
application_name |
backend_start    | 2019-10-24 18:25:28.238157+02
state            | ¤
backend_type     | logical replication launcher
query            |
-[ RECORD 3 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22324
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.167611+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + -3810 WHERE…
-[ RECORD 4 ]----+-------------------------------------------------------------
datname          | postgres
pid              | 22429
usename          | postgres
application_name | psql
backend_start    | 2019-10-28 10:27:09.599426+01
state            | active
backend_type     | client backend
query            | select datname, pid, usename, application_name, backend_start…
-[ RECORD 5 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22325
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.172585+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 4360 WHERE…
```

```
-[ RECORD 6 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22326
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.178514+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 2865 WHERE…
-[ RECORD 7 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26376
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235574+02
state            | ¤
backend_type     | background writer
query            |
-[ RECORD 8 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26375
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235064+02
state            | ¤
backend_type     | checkpointer
query            |
-[ RECORD 9 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26377
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236239+02
state            | ¤
backend_type     | walwriter
query            |
```

Les textes des requêtes sont tronqués à 1024 caractères : c'est un problème courant. Il est conseillé de monter le paramètre `track_activity_query_size` à plusieurs kilooctets.

Cette vue fournit aussi les *wait events*, qui indiquent ce qu'une session est en train d'attendre. Cela peut être très divers et inclut la levée d'un verrou sur un objet, celle d'un verrou interne, la fin d'une entrée-sortie… L'absence de *wait event* indique que la requête s'exécute. À noter qu'une session avec un *wait event* peut rester en statut `active`.

Les détails sur les champs `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente) sont disponibles dans le tableau des événements d'attente[10]. de la documentation.

À partir de PostgreSQL 17, la vue `pg_wait_events` peut être directement jointe à `pg_stat_activity`, et son champ `description` évite d'aller voir la documentation :

[10] https://docs.postgresql.fr/current/monitoring-stats.html#WAIT-EVENT-TABLE

```
SELECT datname, application_name, pid,
       wait_event_type, wait_event, query, w.description
FROM pg_stat_activity a
  LEFT OUTER JOIN pg_wait_events w
  ON (a.wait_event_type = w.type AND a.wait_event = w.name)
WHERE backend_type='client backend'
AND   wait_event IS NOT NULL
ORDER BY wait_event DESC LIMIT 4 \gx
```

```
-[ RECORD 1 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786146
wait_event_type  | LWLock
wait_event       | WALWriteLock
query            | UPDATE pgbench_accounts SET abalance = abalance + 4055 WHERE…
description      | ø
-[ RECORD 2 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786190
wait_event_type  | IO
wait_event       | WalSync
query            | UPDATE pgbench_accounts SET abalance = abalance + -1859  WHERE…
description      | Waiting for a WAL file to reach durable storage
-[ RECORD 3 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786145
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 3553 WHERE…
description      | Waiting for a read from a relation data file
-[ RECORD 4 ]----+-------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786143
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 1929 WHERE…
description      | Waiting for a read from a relation data file
```

Le processus de la ligne 2 attend une synchronisation sur disque du journal de transaction (WAL), et les deux suivants une lecture d'un fichier de données. (La description vide en ligne 1 est un souci de la version 17.2).

Pour entrer dans le détail des champs liés aux connexions :

- `backend_type` est le type de processus : on filtrera généralement sur `client backend`, mais on y trouvera aussi des processus de tâche de fond comme `checkpointer`, `walwriter`, `autovacuum launcher` et autres processus de PostgreSQL, ou encore des *workers* lancés par des extensions ;
- `datname` est le nom de la base à laquelle la session est connectée, et `datid` est son identifiant (OID) ;

- `pid` est le processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client, qui durera le temps de la session (sauf parallélisation) ;
- `usename` est le nom de l'utilisateur connecté, et `usesysid` est son OID dans `pg_roles` ;
- `application_name` est un nom facultatif, et il est recommandé que l'application cliente le renseigne autant que possible avec `SET application_name TO 'nom_outil_client'` ;
- `client_addr` est l'adresse IP du client connecté (`NULL` si connexion sur socket Unix), et `client_hostname` est le nom associé à cette IP, renseigné uniquement si `log_hostname` a été passé à `on` (cela peut ralentir les connexions à cause de la résolution DNS) ;
- `client_port` est le numéro de port sur lequel le client est connecté, toujours s'il s'agit d'une connexion IP.

Une requête parallélisée occupe plusieurs processus, et apparaîtra sur plusieurs lignes de `pid` différents. Le champ `leader_pid` indique le processus principal. Les autres processus disparaîtront dès la requête terminée.

Pour les champs liés aux durées de session, transactions et requêtes :

- `backend_start` est le timestamp de l'établissement de la session ;
- `xact_start` est le timestamp de début de la transaction ;
- `query_start` est le timestamp de début de la requête en cours, ou de la dernière requête exécutée ;
- `status` vaut soit `active`, soit `idle` (la session ne fait rien) soit `idle in transaction` (en attente pendant une transaction) ;
- `backend_xid` est l'identifiant de la transaction en cours, s'il y en a une ;
- `backend_xmin` est l'horizon des transactions visibles, et dépend aussi des autres transactions en cours.

Rappelons qu'une session durablement en statut `idle in transaction` bloque le fonctionnement de l'autovacuum car `backend_xmin` est bloqué. Cela peut mener à des tables fragmentées et du gaspillage de place disque.

Depuis PostgreSQL 14, `pg_stat_activity` peut afficher un champ `query_id`, c'est-à-dire un identifiant de requête normalisée (dépouillée des valeurs de paramètres). Il faut que le paramètre `compute_query_id` soit à `on` ou `auto` (le défaut, et alors une extension peut l'activer). Ce champ est utile pour retrouver une requête dans la vue de l'extension `pg_stat_statements`, par exemple.

Certains champs de cette vue ne sont renseignés que si le paramètre `track_activities` est à `on` (valeur par défaut, qu'il est conseillé de laisser ainsi).

À noter qu'il ne faut pas interroger `pg_stat_activity` au sein d'une transaction, son contenu pourrait sembler figé.

15.6.2 Statistiques d'activité - configuration 2

- `track_counts` = `on`
- `track_io_timing` = `on`
- `track_functions` = `off` / `pl` / `all`

Par défaut, le paramètre `track_counts` est à `on`. Le collecteur de statistiques est alors capable de récupérer des informations sur des nombres de lignes lues, insérées, mises à jour, supprimées, vivantes, mortes, etc., et de blocs lus dans le cache de PostgreSQL (*hit*) ou en dehors (*read*).

`track_io_timing` réalise un chronométrage des opérations de lecture et écriture disque. Il complète les champs `blk_read_time` et `blk_write_time` dans les tables `pg_stat_database` et `pg_stat_statements` (si cette extension[11] est installée). Il ajoute des traces suite à un `VACUUM` ou un `ANALYZE` exécutés par le processus `autovacuum` Dans les plans d'exécutions (avec `EXPLAIN (ANALYZE,BUFFERS)`), il permet l'affichage du temps passé à lire hors du cache de PostgreSQL(sur disque ou dans le cache de l'OS) :

```
I/O Timings: read=2.062
```

Avant d'activer `track_io_timing` sur une machine peu performante, vérifiez avec l'outil `pg_test_timing`[12] que la quasi-totalité des appels dure moins d'une nanoseconde.

PostgreSQL sait aussi récupérer des statistiques sur les routines stockées. Il faut activer le paramètre `track_functions` qui a trois valeurs : `off` pour ne rien récupérer, `pl` pour récupérer les statistiques sur les procédures stockées en `PL/*` et `all` pour récupérer les statistiques des fonctions quelque soit leur langage (notamment celles en C). Les résultats (nombre et durée d'exécution) peuvent se suivre dans la vue `pg_stat_user_functions`.

[11] https://dali.bo/x2_html#pg_stat_statements
[12] https://docs.postgresql.fr/current/pgtesttiming.html

15.6.3 Statistiques d'activité - configuration 3

- `stats_temp_directory` (<v15)
  - répertoire contenant les fichiers temporaires des statistiques
  - copié vers `pg_stat` lors d'un arrêt propre
  - à monter sur du tmpfs

Avant la version 15, il existe un processus collecteur de statistiques, qui fonctionne ainsi :

- il est lancé au démarrage de PostgreSQL (il est d'ailleurs impossible de le désactiver complètement) ;
- il collecte ses statistiques dans le répertoire ciblé par le paramètre `stats_temp_directory` ;
- il met à jour les fichiers dès que les autres processus lui fournissent des statistiques ;
- à l'arrêt de PostgreSQL, il recopie les fichiers du répertoire temporaire dans le répertoire `$PGDATA/pg_stat`.

L'intérêt de ce fonctionnement est de pouvoir copier le fichier de statistiques sur un disque très rapide (comme un disque SSD), voire dans de la mémoire montée en disque comme `tmpfs` (c'est d'ailleurs le défaut sur Debian).

À partir de la version 15, les statistiques sont stockés en mémoire partagée et il n'y a plus de collecteur.

15.6.4 Statistiques d'activité : perte

- Les statistiques d'activité sont perdues en cas de crash ou restauration
  - `ANALYZE` (voire `VACUUM`)

En cas d'arrêt brutal de PostgreSQL ou de restauration physique, les statisiques d'activité sont perdues (pas les statistiques sur les données). Lancer un `ANALYZE` est préférable pour éviter que l'autovacuum tarde à nettoyer les tables ou rafraîchir les statistiques sur les données.

15.6.5 Informations intéressantes à récupérer

Sur :

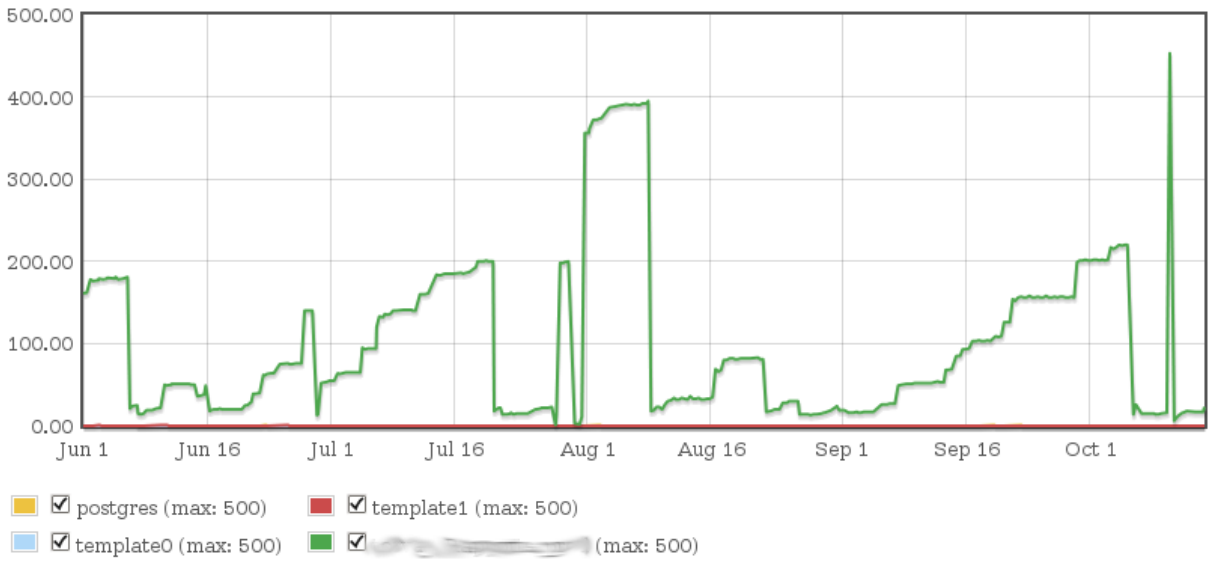
- l'activité
- l'instance
- les bases
- les tables
- les index
- les fonctions

Au niveau des statistiques, il existe un grand nombre d'informations intéressantes à récupérer. Cela va des informations sur l'activité en cours (nombre de connexions, noms des utilisateurs connectés, requêtes en cours d'exécution), à celles sur les bases de données (nombre d'écritures, nombre de lectures dans le cache), à celles sur les tables, index et fonctions.

Les connaître toutes présente peu d'intérêt car seulement certaines sont vraiment intéressantes à superviser. Nous allons voir ici quelques exemples.

15.6.6 Nombre de connexions par base

```
SELECT datname, numbackends FROM pg_stat_database;
SELECT datname, count(*) FROM pg_stat_activity
  WHERE datname IS NOT NULL
  GROUP BY datname;
```



Il est souvent intéressant de connaître le nombre de personnes connectées. Cela permet notamment de s'assurer que la configuration du paramètre `max_connections` est suffisamment élevée pour ne pas voir des demandes de connexion être refusées.

Il est aussi intéressant de pouvoir dénombrer le nombre de connexions par bases. Cela permet d'avoir une idée de la charge sur chaque base. Une base ayant de plus en plus d'utilisateurs et souffrant de performances devra peut-être être placée seule sur un serveur.

Il est aussi possible d'avoir le nombre de connexions par :

- utilisateur :

```
SELECT usename, count(*) FROM pg_stat_activity WHERE usename IS NOT NULL GROUP BY
↪  usename;
```

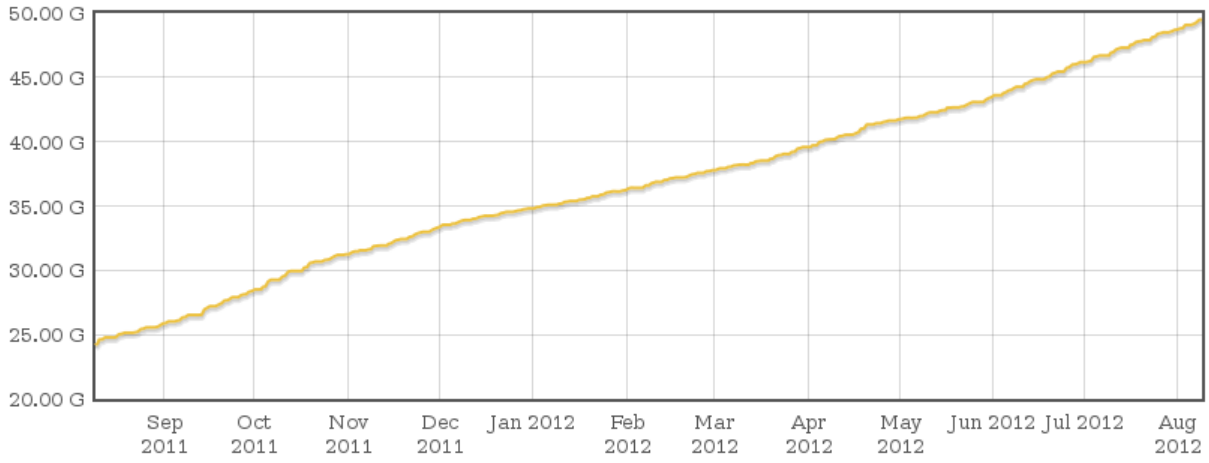
- application cliente :

```
SELECT application_name, count(*) FROM pg_stat_activity GROUP BY application_name;
```

Le graphe affiché ci-dessus montre l'utilisation des connexions sur différentes bases. Les bases systèmes ne sont pas du tout utilisées. Seule la base utilisateur reçoit un grand nombre de connexions. Il y a même eu deux pics, un à environ 400 connexions et un autre à 450 connexions.

15.6.7 Taille des bases

```
SELECT datname, pg_database_size(oid) FROM pg_database;
```

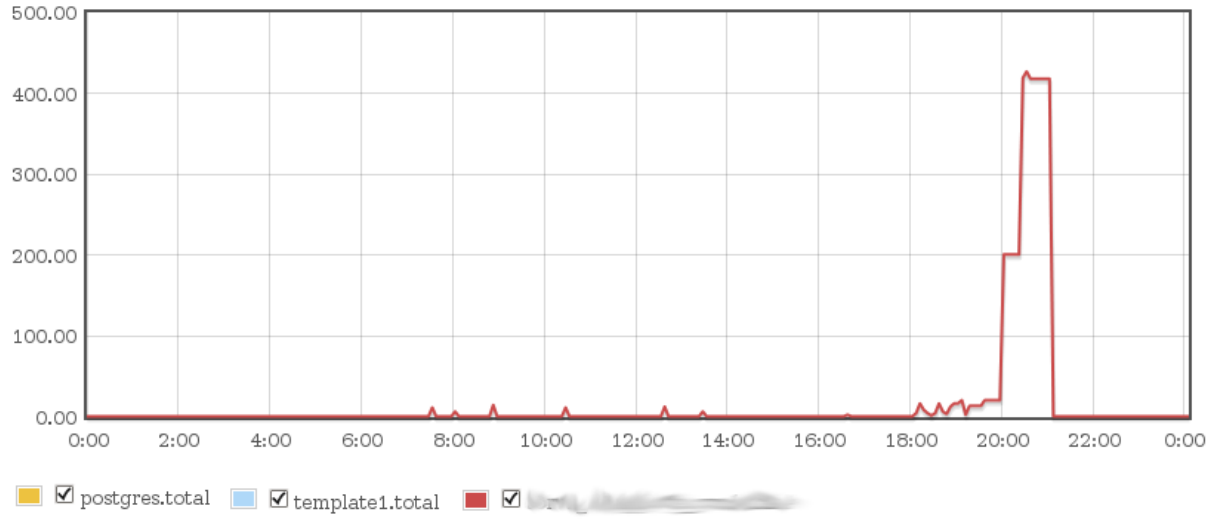


La volumétrie des bases est une autre information fréquemment demandée. L'exécution de cette requête toutes les cinq minutes permettra de suivre l'évolution de la taille des bases.

Le graphe ci-dessus montre une montée en volumétrie assez importante, la base ayant doublé en un an.

15.6.8 Nombre de verrous

```
SELECT d.datname, count(*) FROM pg_locks l
JOIN pg_database d ON l.database=d.oid
GROUP BY d.datname ORDER BY d.datname;
```



Autre demande fréquente : pouvoir suivre le nombre de verrous. La requête ci-dessus récupère le nombre de verrous posés par base. Il est aussi possible de récupérer les types de verrous, ou de ne prendre en compte que certains types de verrous.

Le graphe montre une utilisation très limitée des verrous. En fait, on observe surtout une grosse utilisation des verrous entre 20h et 21h30. Si le graphe montrait plusieurs jours d'affilé, on pourrait s'apercevoir que le pic est présent tous les jours à ce moment-là. En fait, il s'agit de la sauvegarde. La sauvegarde pose un grand nombre de verrous, des verrous très légers qui vont bloquer très peu de monde, mais néanmoins, ils sont présents.

15.6.9 Et un grand nombre d'autres informations

- Ratio de lecture du cache (souvent appelé *hit ratio*)
- Retard de réplication
- Nombre de transactions par seconde

Les trois exemples proposés ci-dessus ne sont que les exemples les plus marquants. Beaucoup d'autres informations sont récupérables. On peut citer par exemple :

- le ratio de lecture dans le cache de PostgreSQL ;
- le retard de la réplication interne de PostgreSQL (envoi, écriture, application) ;
- le nombre de transactions par seconde ;
- l'activité d'une table (en nombre de lectures, insertions, suppressions, modifications) ;
- le nombre de parcours séquentiels et de parcours d'index ;
- etc.

15.6.10 Outils

- Beaucoup d'outils existent pour exploiter les statistiques :
  - Munin, Nagios, Zabbix + sondes PG
- Dédiés à PostgreSQL :
  - pg_stat_statements : https://www.postgresql.org/docs/current/pgstatstatements.html
  - PoWA : https://powa.readthedocs.io/

Il existe de nombreux outils utilisables avec les statistiques. Les plus connus sont Munin (sondes déjà intégrées), Nagios et Zabbix. Pour ces deux derniers, il est nécessaire d'ajouter des sondes comme `check_postgres` et `check_pgactivity` évoquées plus haut.

`pg_stat_statements` est un module contrib de PostgreSQL, donc non installé par défaut. Il collecte des statistiques sur toutes les requêtes exécutées. Son contenu est souvent extrêmement instructif. Pour plus de détails sur les métriques relevées, voir https://dali.bo/h2_html#pg_stat_statements, et pour l'installation et des exemples de requêtes, voir https://dali.bo/x2_html#pg_stat_statements, ou encore la documentation officielle[13].

PoWA (*PostgreSQL Workload Analyzer*) est un outil communautaire historisant les informations collectées par `pg_stat_statements`, et fournissant une interface graphique permettant d'observer en temps réel les requêtes normalisées les plus consommatrices d'une instance selon plusieurs critères.

15.6.11 munin

- Scripts Perl
- Sondes PostgreSQL incluses
- Récupère les statistiques toutes les 5 min
- Crée des pages HTML statiques et des fichiers PNG
  - donc des graphes

Munin est à la base un outil de métrologie utilisé par les administrateurs systèmes. Il comprend un certain nombre de sondes capables de récupérer des informations telles que la charge système, l'utilisation de la mémoire et des interfaces réseau, l'espace libre d'un disque, etc. Des sondes lui ont été ajoutées pour pouvoir surveiller certains services comme Sendmail, Postfix, Apache, et même PostgreSQL.

Munin est composé de deux parties : les sondes et le générateur de rapports. Les sondes sont exécutées toutes les cinq minutes. Elles sont généralement écrites en Perl. Elles récupèrent les informations et les stockent dans des bases BerkeleyDB. Ensuite, le générateur de rapports lit les bases en question pour générer des graphes au format PNG. Des pages HTML sont créées pour faciliter l'accès aux graphes.

Munin est inclus dans les distributions habituelles.

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

15.6.12 Nagios

- Outil GPL, sur https://www.nagios.org/
- Nombreux concurrents et équivalents
- Sondes dédiées à PostgreSQL : `check_postgres` et `check_pgactivity`

Nagios est un outil très connu de supervision. Il dispose par défaut de quelques sondes, principalement système. Il existe aussi des outils concurrents (Naemon, Icinga 2…) ou des surcouches compatibles au niveau des sondes.

Pour PostgreSQL, il est possible de le coupler à des sondes dédiées, comme `check_postgres` ou `check_pgactivity`, déjà évoquées, pour qu'il puisse récupérer un certain nombre d'informations statistiques de PostgreSQL. Ne pas oublier de compléter par des sondes plus classiques pour les I/O, le CPU, la RAM, etc.

15.6.13 Outils - Zabbix

- Outil GPL, sur https://www.zabbix.com/
- Sonde `check_postgres.pl`
- Template pg-monz
  - https://pg-monz.github.io/pg_monz/index-en.html

Zabbix est certainement le deuxième outil opensource le plus utilisé pour la supervision. Son avantage par rapport à Nagios est qu'il est capable de faire des graphes directement et qu'il dispose d'une interface plus simple d'approche.

Là-aussi, la sonde `check_postgres.pl` lui permet de récupérer des informations sur un serveur PostgreSQL.

15.6.14 Outils - pg_stat_statements

- Module contrib de PostgreSQL
- Récupère et stocke des statistiques d'exécution des requêtes
- Les requêtes sont normalisées
- Pas d'historisation

`pg_stat_statements` est une extension issue des « contrib » de PostgreSQL, donc livrée avec lui, mais non installée par défaut. Sa mise en place nécessite le préchargement de bibliothèques dans la mémoire partagée (paramètre `shared_preload_libraries`), et donc le redémarrage de l'instance.

Une fois installé et configuré, des mesures (nombre de blocs lus dans le cache, hors cache, …) sont collectées sur toutes les requêtes exécutées, et elles sont stockées avec les requêtes normalisées. Ces données sont ensuite exploitables en interrogeant la vue `pg_stat_statements`. À noter que ces statistiques sont cumulées sans être historisées, il est donc souvent difficile d'identifier quelle requête est la plus consommatrice à un instant donné, à moins de réinitialiser les statistiques.

Voir aussi la documentation officielle : https://docs.postgresql.fr/current/pgstatstatements.html

15.6.15 Outils - PoWA

- Site officiel : https://github.com/powa-team
- Licence : PostgreSQL
- Surveillance de l'activité SQL
- Captures des statistiques collectées par `pg_stat_statements`
  - et d'autres extensions
- Interface graphique : activité des requêtes en temps réel
- Dépôt GitHub
  - archiveur : https://github.com/powa-team/powa-archivist
  - UI web : https://github.com/powa-team/powa-web

PoWA (*PostgreSQL Workload Analyzer*) est un outil communautaire, sous licence PostgreSQL.

Tout comme pour l'extension standard `pg_stat_statements`, sa mise en place nécessite la modification du paramètre `shared_preload_libraries`, et donc le redémarrage de l'instance. Il faut également créer une nouvelle base de données dans l'instance. Par ailleurs, PoWA repose sur les statistiques collectées par `pg_stat_statements`, celui-ci doit donc être également installé.

Une fois installé et configuré, l'outil va récupérer à intervalle régulier les statistiques collectées par `pg_stat_statements`, les stocker et les historiser.

Il tire aussi partie d'autres extensions comme HypoPG, pg_qualstats, pg_stat_kcache…

L'outil fournit également une interface graphique permettant d'exploiter ces données, et donc d'observer en temps réel l'activité de l'instance. Cette activité est présentée sous forme de graphiques interactifs et de tableaux permettant de trier selon divers critères (nombre d'exécution, blocs lus hors cache…) les différentes requêtes normalisées sur l'intervalle de temps sélectionné.

15.7 CONCLUSION

- Un système est pérenne s'il est bien supervisé
- Supervision automatique
  - configuration des traces
  - configuration des statistiques
  - mise en place d'outils d'historisation

Une bonne politique de supervision est la clef de voûte d'un système pérenne. Pour cela, il faut tout d'abord s'assurer que les traces et les statistiques soient bien configurées. Ensuite, l'installation d'un outil d'historisation, de création de graphes et de génération d'alertes, est obligatoire pour pouvoir tirer profit des informations fournies par PostgreSQL.

15.7.1 Questions

N'hésitez pas, c'est le moment !

15.8 QUIZ

https://dali.bo/h1_quiz

15.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/h1_solutions.
Analyse de traces avec pgBadger
But : Analyser un journal de traces avec pgBadger

S'assurer que la configuration suivante est en place, au moins le temps de l'audit :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages='C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
```

Si cela n'a pas déjà été fait aujourd'hui, lancer une session `pgbench` de 10 minutes dans la base `pgbench`.

Installer pgBadger, soit depuis les dépôts du PGDG, soit depuis le site de l'auteur https://pgbadger.darold.net/.

Repérer où sont vos fichiers de traces, et notamment ceux d'aujourd'hui.

Générer dans `/tmp` un rapport pgBadger sur toute la journée en cours. La documentation est entre autres sur le site de l'auteur : https://pgbadger.darold.net/documentation.html. Combien de requêtes a-t-il détecté ?

Ouvrir le rapport dans un navigateur. Dans l'onglet *Overview*, à quoi correspondent les pics de trafic en nombre de requêtes ?

Dans l'onglet *Top*, chercher l'histogramme de la répartition des durées des requêtes.

Quelles sont les requêtes les plus lentes, prises une à une ?

> Quelles sont les requêtes qui ont consommé le plus de temps au total ?

15.10 TRAVAUX PRATIQUES (SOLUTIONS)

Analyse de traces avec pgBadger

> S'assurer que la configuration suivante est en place, au moins le temps de l'audit :

```
log_min_duration_statement = 0
log_temp_files = 0
log_autovacuum_min_duration = 0
lc_messages='C'
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
```

On vérifie que la configuration est bien active avec :

```
postgres@pgbench=# show log_min_duration_statement ;

 log_min_duration_statement
----------------------------
 0
```

Au besoin, le paramétrage peut soit se faire dans `postgresql.conf` (ne pas oublier de recharger la configuration), soit au niveau de la base de données (`ALTER DATABASE pgbench SET log_min_duration_statement`

> Si cela n'a pas déjà été fait aujourd'hui, lancer une session `pgbench` de 10 minutes dans la base `pgbench`.

Pour créer la base si elle n'existe pas déjà :

```
$ /usr/pgsql-14/bin/pgbench -i --foreign-keys -d pgbench -U testperf
```

Pour lancer un traitement de 10 minutes avec 20 clients répartis sur 2 processeurs :

```
$ /usr/pgsql-14/bin/pgbench -U testperf pgbench \
    --client=20 --jobs=2 -T 600 --no-vacuum -P 10
```

> Installer pgBadger, soit depuis les dépôts du PGDG, soit depuis le site de l'auteur https://pgbadger.darold.net/.

Le plus simple reste le dépôt de la distribution :

```
# yum install pgbadger
```

Pour la version la plus à jour, comme Gilles Darold fait évoluer le produit régulièrement, il n'est pas rare que le dépôt Github soit plus à jour et l'on peut préférer cette source. La release 12.4 est la dernière au moment où ceci est écrit.

```
$ curl -LO https://github.com/darold/pgbadger/archive/refs/tags/v12.4.tar.gz
$ tar xvf v12.4.tar.gz
```

Dans le répertoire `pgbadger-12.4`, il n'y a guère que le script `pgbadger` dont on ait besoin, et que l'on placera par exemple dans `/usr/local/bin`.

> Repérer où sont vos fichiers de traces, et notamment ceux d'aujourd'hui.

Par défaut, les traces sur Red Hat/CentOS/Rocky Linux sont dans `/var/lib/pgsql/14/data/log`.

Elles ont pu être déplacées dans `/var/lib/pgsql/traces` au fil de ces TP.

> Générer dans `/tmp` un rapport pgBadger sur toute la journée en cours. La documentation est entre autres sur le site de l'auteur : https://pgbadger.darold.net/documentation.html. Combien de requêtes a-t-il détecté ?

Dans sa plus simple expression, la commande est la suivante, à exécuter avec un utilisateur ayant accès aux traces :

```
$ cd /var/lib/pgsql/14/data/log
$ pgbadger -o /tmp/pgabdger-aujourdhui.html postgresql-2022-05-02*.log
```

Il peut arriver que la détection du format échoue, auquel cas il faut préciser la valeur du paramètre `log_line_prefix` (`%m [%p]` par défaut sur RHEL, mais l'auteur préconise `'%t [%p]: db=%d,user=%u,app=%a,clie` que l'on a configuré initialement) avec l'option `-p`.

Le paramètre `-j` permet de paralléliser le travail sur plusieurs processeurs.

On peut vouloir des statistiques plus fines que la moyenne par défaut de 5 minutes (`-a`), ou cibler sur une page de dates précises (`-b` et `-e`). Enfin, préciser le fuseau horaire du serveur (`-Z`) est souvent utile pour éviter des décalages dans les heures.

```
$ pgbadger -o /tmp/pgabdger-aujourdhui.html -p '%m [%p] ' -j 2  -a 1  \
           -b '2022-05-02 09:00:00' -e '2022-05-02 18:00:00' -Z '+02' \
           postgresql-2022-05-02*.log
...
[=========================>] Parsed 170877924 bytes of 170877924 (100.00%),
                                          queries: 503580, events: 51
LOG: Ok, generating html report...
```

En l'occurence, pgBadger a détecté ici un demi-million de requêtes.

> Ouvrir le rapport dans un navigateur. Dans l'onglet *Overview*, à quoi correspondent les pics de trafic en nombre de requêtes ?

Il s'agit probablement des tests avec `pgbench`.

> Dans l'onglet *Top*, chercher l'histogramme de la répartition des durées des requêtes.

Le graphique indique que l'essentiel des requêtes doit figurer dans la tranche « 0ms-1ms ». La version tableau de ce graphique liste une poignée de requête autour de la seconde ou plus, que l'on voit également dans les *slowest queries* en-dessous.

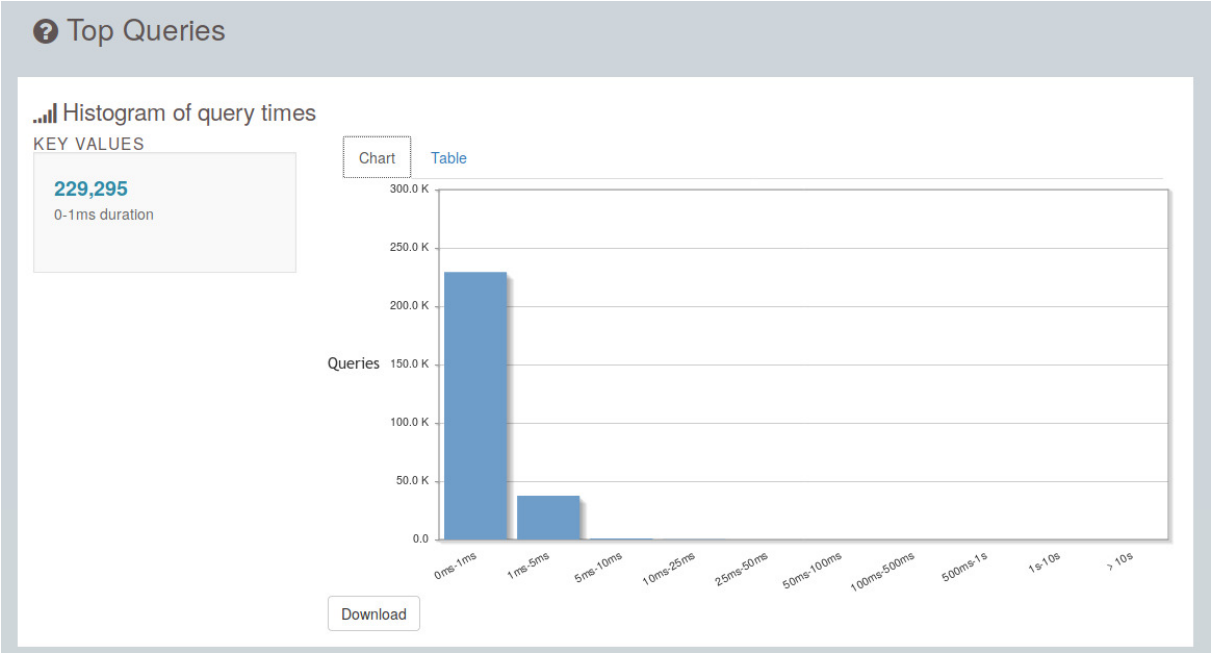


Figure 15/ .2: Nombre de requêtes par durée

> Quelles sont les requêtes les plus lentes, prises une à une ?

On les trouve dans *Top/Slowest individual queries*. Ce sont probablement les ordres `COPY` des différents essais avec `pg_restore`, et les ordres `CREATE DATABASE`.

> Quelles sont les requêtes qui ont consommé le plus de temps au total ?

On les trouve dans *Top/Time consuming queries*. Cet onglet est très intéressant pour repérer les requêtes relativement rapides, mais qui représentent la véritable charge de l'instance au quotidien.

Il s'agit très probablement des requêtes de `pgbench` :

Time consuming queries

Rank	Total duration	Times executed	Min duration	Max duration	Avg duration	Query
1	2h43m15s	27 Details	57s31ms	20m	6m2s	SELECT "id_exploitation", "annee_recolte", "id_calcul_serie", "?column?" FROM "systerre"."qry_sys_frm_lancement_calcul" WHERE (("id_exploitation" IN (...)) AND ("annee_recolte" IN (...)));

TIMES REPORTED TIME CONSUMING QUERIES #2

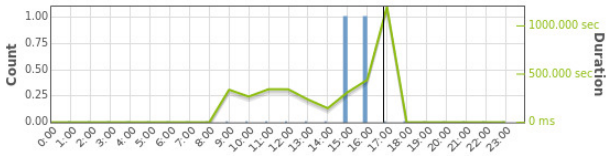
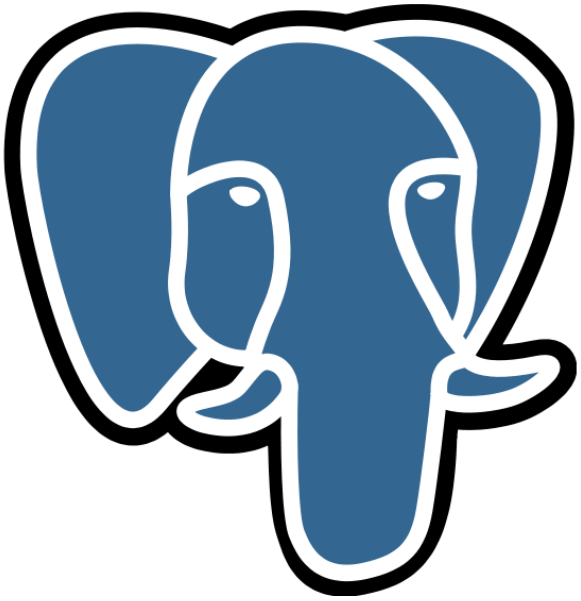


Figure 15/ .3: Requêtes les plus consommatrices en temps

16/ PostgreSQL : Gestion d’un sinistre



16.1 INTRODUCTION

- Une bonne politique de sauvegardes est cruciale
  - mais elle n'empêche pas les incidents
- Il faut être prêt à y faire face

Ce module se propose de faire une description des bonnes et mauvaises pratiques en cas de coup dur :

- crash de l'instance ;
- suppression / corruption de fichiers ;
- problèmes matériels ;
- sauvegardes corrompues…

Seront également présentées les situations classiques de désastres, ainsi que certaines méthodes et outils dangereux et déconseillés.

L'objectif est d'aider à convaincre de l'intérêt qu'il y a à anticiper les problèmes, à mettre en place une politique de sauvegarde pérenne, et à ne pas tenter de manipulation dangereuse sans comprendre précisément à quoi l'on s'expose.

Ce module est en grande partie inspiré de *The Worst Day of Your Life*, une présentation de Christophe Pettus au FOSDEM 2014[1]

16.1.1 Au menu

- Anticiper les désastres
- Réagir aux désastres
- Rechercher l'origine du problème
- Outils utiles
- Cas type de désastres

[1] http://thebuild.com/presentations/worst-day-fosdem-2014.pdf

16.2 ANTICIPER LES DÉSASTRES

- Un désastre peut toujours survenir
- Il faut savoir le détecter le plus tôt possible
  - et s'être préparé à y répondre

Il est impossible de parer à tous les cas de désastres imaginables.

Le matériel peut subir des pannes, une faille logicielle non connue peut être exploitée, une modification d'infrastructure ou de configuration peut avoir des conséquences imprévues à long terme, une erreur humaine est toujours possible.

Les principes de base de la haute disponibilité (redondance, surveillance…) permettent de mitiger le problème, mais jamais de l'éliminer complètement.

Il est donc extrêmement important de se préparer au mieux, de procéder à des simulations, de remettre en question chaque brique de l'infrastructure pour être capable de détecter une défaillance et d'y réagir rapidement.

16.2.1 Documentation

- Documentation complète et à jour
  - emplacement et fréquence des sauvegardes
  - emplacement des traces
  - procédures et scripts d'exploitation
- Sauvegarder et versionner la documentation

Par nature, les désastres arrivent de façon inattendue.

Il faut donc se préparer à devoir agir en urgence, sans préparation, dans un environnement perturbé et stressant — par exemple, en pleine nuit, la veille d'un jour particulièrement critique pour l'activité de la production.

Un des premiers points d'importance est donc de s'assurer de la présence d'une documentation claire, précise et à jour, afin de minimiser le risque d'erreurs humaines.

Cette documentation devrait détailler l'architecture dans son ensemble, et particulièrement la politique de sauvegarde choisie, l'emplacement de celles-ci, les procédures de restauration et éventuellement de bascule vers un environnement de secours.

Les procédures d'exploitation doivent y être expliquées, de façon détaillée mais claire, afin qu'il n'y ait pas de doute sur les actions à effectuer une fois la cause du problème identifié.

La méthode d'accès aux informations utiles (traces de l'instance, du système, supervision…) devrait également être soigneusement documentée afin que le diagnostic du problème soit aussi simple que possible.

Toutes ces informations doivent être organisées de façon claire, afin qu'elles soient immédiatement accessibles et exploitables aux intervenants lors d'un problème.

Il est évidemment tout aussi important de penser à versionner et sauvegarder cette documentation, afin que celle-ci soit toujours accessible même en cas de désastre majeur (perte d'un site).

16.2.2 Procédures et scripts

- Procédures détaillées de restauration / PRA
  - préparer des scripts / utiliser des outils
  - minimiser le nombre d'actions manuelles
- Tester les procédures régulièrement
  - bases de test, développement…
  - s'assurer que chacun les maîtrise
- Sauvegarder et versionner les scripts

La gestion d'un désastre est une situation particulièrement stressante, le risque d'erreur humaine est donc accru.

Un DBA devant restaurer d'urgence l'instance de production en pleine nuit courera plus de risques de faire une fausse manipulation s'il doit taper une vingtaine de commandes en suivant une procédure dans une autre fenêtre (voire un autre poste) que s'il n'a qu'un script à exécuter.

En conséquence, il est important de minimiser le nombre d'actions manuelles à effectuer dans les procédures, en privilégiant l'usage de scripts d'exploitation ou d'outils dédiés (comme pgBackRest ou barman pour restaurer une instance PostgreSQL).

Néanmoins, même cette pratique ne suffit pas à exclure tout risque.

L'utilisation de ces scripts ou de ces outils doit également être comprise, correctement documentée, et les procédures régulièrement testées. Le test idéal consiste à remonter fréquemment des environnements de développement et de test ; vos développeurs vous en seront d'ailleurs reconnaissants.

Dans le cas contraire, l'utilisation d'un script ou d'un outil peut aggraver le problème, parfois de façon dramatique — par exemple, l'écrasement d'un environnement sain lors d'une restauration parce que la procédure ne mentionne pas que le script doit être lancé depuis un serveur particulier.

L'aspect le plus important est de s'assurer par des tests réguliers **et manuels** que les procédures sont à jour, n'ont pas de comportement inattendu, et sont maîtrisées par toute l'équipe d'exploitation.

Tout comme pour la documentation, les scripts d'exploitation doivent également être sauvegardés et versionnés.

16.2.3 Supervision et historisation

- Tout doit être supervisé
  - réseau, matériel, système, logiciels…
  - les niveaux d'alerte doivent être significatifs
- Les métriques importantes doivent être historisées
  - cela permet de retrouver le moment où le problème est apparu
  - quand cela a un sens, faire des graphes

La supervision est un sujet vaste, qui touche plus au domaine de la haute disponibilité.

Un désastre sera d'autant plus difficile à gérer qu'il est détecté tard. La supervision en place doit donc être pensée pour détecter tout type de défaillance (penser également à superviser la supervision !).

Attention à bien calibrer les niveaux d'alerte, la présence de trop de messages augmente le risque que l'un d'eux passe inaperçu, et donc que l'incident ne soit détecté que tardivement.

Pour aider la phase de diagnostic de l'origine du problème, il faut prévoir d'historiser un maximum d'informations.

La présentation de celles-ci est également importante : il est plus facile de distinguer un pic brutal du nombre de connexions sur un graphique que dans un fichier de traces de plusieurs Go !

16.2.4 Automatisation

- Des outils existent
  - Patroni, PAF (Pacemaker)
- Automatiser une bascule est complexe
  - cela peut mener à davantage d'incidents
  - voire à des désastres (*split brain*)

Si on poursuit jusqu'au bout le raisonnement précédent sur le risque à faire effectuer de nombreuses opérations manuelles lors d'un incident, la conclusion logique est que la solution idéale serait de les éliminer complètement, et d'automatiser complètement le déclenchement et l'exécution de la procédure.

Un problème est que toute solution visant à automatiser une tâche se base sur un nombre limité de paramètres et sur une vision restreinte de l'architecture.

De plus, il est difficile à un outil de bascule automatique de diagnostiquer correctement certains types d'incident, par exemple une partition réseau. L'outil peut donc détecter à tort à un incident, surtout s'il est réglé de façon à être assez sensible, et ainsi provoquer lui-même une coupure de service inutile.

Dans le pire des cas, l'outil peut être amené à prendre une mauvaise décision amenant à une situation de désastre, comme un *split brain* (deux instances PostgreSQL se retrouvent ouvertes en écriture en même temps sur les mêmes données).

Il est donc fortement préférable de laisser un administrateur prendre les décisions potentiellement dangereuses, comme une bascule ou une restauration.

16.3 RÉAGIR AUX DÉSASTRES

- Savoir identifier un problème majeur
- Bons réflexes
- Mauvais réflexes

En dépit de toutes les précautions que l'on peut être amené à prendre, rien ne peut garantir qu'aucun problème ne surviendra.

Il faut donc être capable d'identifier le problème lorsqu'il survient, et être prêt à y répondre.

16.3.1 Symptômes d'un désastre

- Crash de l'instance
- Résultats de requêtes erronnés
- Messages d'erreurs dans les traces
- Dégradation importante des temps d'exécution
- Processus manquants
  - ou en court d'exécution depuis trop longtemps

De très nombreux éléments peuvent aider à identifier que l'on est en situation d'incident grave.

Le plus flagrant est évidemment le crash complet de l'instance PostgreSQL, ou du serveur l'hébergeant, et l'impossibilité pour PostgreSQL de redémarrer.

Les désastres les plus importants ne sont toutefois pas toujours aussi simples à détecter.

Les crash peuvent se produire uniquement de façon ponctuelle, et il existe des cas où l'instance redémarre immédiatement après (typiquement suite au `kill -9` d'un processus backend PostgreSQL).

Cas encore plus délicat, il peut également arriver que les résultats de requêtes soient erronés (par exemple en cas de corruption de fichiers d'index) sans qu'aucune erreur n'apparaisse.

Les symptômes classiques permettant de détecter un problème majeur sont :

- la présence de messages d'erreurs dans les traces de PostgreSQL (notamment des messages `PANIC` ou `FATAL`, mais les messages `ERROR` et `WARNING` sont également très significatifs, particulièrement s'ils apparaissent soudainement en très grand nombre) ;

- la présence de messages d'erreurs dans les traces du système d'exploitation (notamment concernant la mémoire ou le système de stockage) ;
- le constat d'une dégradation importante des temps d'exécution des requêtes sur l'instance ;
- l'absence de certains processus critiques de PostgreSQL ;
- la présence de processus présents depuis une durée inhabituelle (plusieurs semaines, mois…).

16.3.2 Bons réflexes 1

- Garder la tête froide
- Répartir les tâches clairement
- Minimiser les canaux de communication
- Garder des notes de chaque action entreprise

Une fois que l'incident est repéré, il est important de ne pas foncer tête baissée dans des manipulations.

Il faut bien sûr prendre en considération la criticité du problème, notamment pour définir la priorité des actions (par exemple, en cas de perte totale d'un site, quelles sont les applications à basculer en priorité ?), mais quelle que soit la criticité ou l'impact, il ne faut jamais effectuer une action sans en avoir parfaitement saisi l'impact et s'être assuré qu'elle répondait bien au problème rencontré.

Si le travail s'effectue en équipe, il faut bien faire attention à répartir les tâches clairement, afin d'éviter des manipulations concurrentes ou des oublis qui pourraient aggraver la situation.

Il faut également éviter de multiplier les canaux de communication, cela risque de favoriser la perte d'information, ce qui est critique dans une situation de crise.

Surtout, une règle majeure est de prendre le temps de noter systématiquement toutes les actions entreprises.

Les commandes passées, les options utilisées, l'heure d'exécution, toutes ces informations sont très importantes, déjà pour pouvoir agir efficacement en cas de fausse manipulation, mais également pour documenter la gestion de l'incident après coup, et ainsi en conserver une trace qui sera précieuse si celui-ci venait à se reproduire.

16.3.3 Bons réflexes 2

- Se prémunir contre une aggravation du problème
  - couper les accès applicatifs
- Si une corruption est suspectée
  - arrêter immédiatement l'instance
  - faire une sauvegarde immédiate des fichiers
  - travailler sur une copie

S'il y a suspicion de potentielle corruption de données, il est primordial de s'assurer au plus vite de couper tous les accès applicatifs vers l'instance afin de ne pas aggraver la situation.

Il est généralement préférable d'avoir une coupure de service plutôt qu'un grand volume de données irrécupérables.

Ensuite, il faut impérativement faire une sauvegarde complète de l'instance avant de procéder à toute manipulation. En fonction de la nature du problème rencontré, le type de sauvegarde pouvant être effectué peut varier (un export de données ne sera possible que si l'instance est démarrée et que les fichiers sont lisibles par exemple). En cas de doute, la sauvegarde la plus fiable qu'il est possible d'effectuer est une copie des fichiers à froid (instance arrêtée) - toute autre action (y compris un export de données) pourrait avoir des conséquences indésirables.

Si des manipulations doivent être tentées pour tenter de récupérer des données, il faut impérativement travailler sur une copie de l'instance, restaurée à partir de cette sauvegarde. Ne jamais travailler directement sur une instance de production corrompue, la moindre action (même en lecture) pourrait aggraver le problème !

Pour plus d'information, voir sur le wiki PostgreSQL[2].

[2] https://wiki.postgresql.org/wiki/Corruption

16.3.4 Bons réflexes 3

- Déterminer le moment de démarrage du désastre
- Adopter une vision générale plutôt que focalisée sur un détail
- Remettre en cause chaque élément de l'architecture
  - aussi stable (et/ou coûteux/complexe) soit-il
- Éliminer en priorité les causes possibles côté hardware, système
- Isoler le comportement précis du problème
  - identifier les requêtes / tables / index impliqués

La première chose à identifier est l'instant précis où le problème a commencé à se manifester. Cette information est en effet déterminante pour identifier la cause du problème, et le résoudre — notamment pour savoir à quel instant il faut restaurer l'instance si cela est nécessaire.

Il convient pour cela d'utiliser les outils de supervision et de traces (système, applicatif et PostgreSQL) pour remonter au moment d'apparition des premiers symptômes. Attention toutefois à ne pas confondre les symptômes avec le problème lui-même ! Les symptômes les plus visibles ne sont pas forcément apparus les premiers. Par exemple, la charge sur la machine est un symptôme, mais n'est jamais la cause du problème. Elle est liée à d'autres phénomènes, comme des problèmes avec les disques ou un grand nombre de connexions, qui peuvent avoir commencé à se manifester bien avant que la charge ne commence réellement à augmenter.

Si la nature du problème n'est pas évidente à ce stade, il faut examiner l'ensemble de l'architecture en cause, sans en exclure d'office certains composants (baie de stockage, progiciel…), quels que soient leur complexité / coût / stabilité supposés. Si le comportement observé côté PostgreSQL est difficile à expliquer (crashs plus ou moins aléatoires, nombreux messages d'erreur sans lien apparent…), il est préférable de commencer par s'assurer qu'il n'y a pas un problème de plus grande ampleur (système de stockage, virtualisation, réseau, système d'exploitation).

Un bon indicateur consiste à regarder si d'autres instances / applications / processus rencontrent des problèmes similaires.

Ensuite, une fois que l'ampleur du problème a été cernée, il faut procéder méthodiquement pour en déterminer la cause et les éléments affectés.

Pour cela, les informations les plus utiles se trouvent dans les traces, généralement de PostgreSQL ou du système, qui vont permettre d'identifier précisément les éventuels fichiers ou relations corrompus.

16.3.5 Bons réflexes 4

- En cas de défaillance matérielle
  - s'assurer de corriger sur du hardware sain et non affecté !
  - baies partagées…

Cette recommandation peut paraître aller de soi, mais si les problèmes sont provoqués par une défaillance matérielle, il est impératif de s'assurer que le travail de correction soit effectué sur un environnement non affecté.

Cela peut s'avérer problématique dans le cadre d'architecture mutualisant les ressources, comme des environnements virtualisés ou utilisant une baie de stockage.

Prendre également la précaution de vérifier que l'intégrité des sauvegardes n'est pas affectée par le problème.

16.3.6 Bons réflexes 5

- Communiquer, ne pas rester isolé
- Demander de l'aide si le problème est trop complexe
  - autres équipes
  - support
  - forums
  - listes

La communication est très importante dans la gestion d'un désastre.

Il est préférable de minimiser le nombre de canaux de communication plutôt que de les multiplier (téléphone, e-mail, chat, ticket…), ce qui pourrait amener à une perte d'informations et à des délais indésirables.

Il est primordial de rapidement cerner l'ampleur du problème, et pour cela il est généralement nécessaire de demander l'expertise d'autres administrateurs / équipes (applicatif, système, réseau, virtualisation, SAN…). Il ne faut pas rester isolé et risquer que la vision étroite que l'on a des symptômes (notamment en terme de supervision / accès aux traces) empêche l'identification de la nature réelle du problème.

Si la situation semble échapper à tout contrôle, et dépasser les compétences de l'équipe en cours d'intervention, il faut chercher de l'aide auprès de personnes compétentes, par exemple auprès d'autres équipes, du support.

En aucun cas, il ne faut se mettre à suivre des recommandations glanées sur Internet, qui ne se rapporteraient que très approximativement au problème rencontré, voire pas du tout. Si nécessaire, on trouve en ligne des forums et des listes de discussions spécialisées sur lesquels il est également possible d'obtenir des conseils — il est néanmoins indispensable de prendre en compte que les personnes intervenant sur ces médias le font de manière bénévole. Il est déraisonnable de s'attendre à une réaction immédiate, aussi urgent le problème soit-il, et les suggestions effectuées le sont sans aucune garantie.

16.3.7 Bons réflexes 6

- Dérouler les procédures comme prévu
- En cas de situation non prévue, s'arrêter pour faire le point
  - ne pas hésiter à remettre en cause l'analyse
  - ou la procédure elle-même

Dans l'idéal, des procédures détaillant les actions à effectuer ont été écrites pour le cas de figure rencontré. Dans ce cas, une fois que l'on s'est assuré d'avoir identifié la procédure appropriée, il faut la dérouler méthodiquement, point par point, et valider à chaque étape que tout se déroule comme prévu.

Si une étape de la procédure ne se passe pas comme prévu, il ne faut pas tenter de poursuivre tout de même son exécution sans avoir compris ce qui s'est passé et les conséquences. Cela pourrait être dangereux.

Il faut au contraire prendre le temps de comprendre le problème en procédant comme décrit précédemment, quitte à remettre en cause toute l'analyse menée auparavant, et la procédure ou les scripts utilisés.

C'est également pour parer à ce type de cas de figure qu'il est important de travailler sur une copie et non sur l'environnement de production directement.

16.3.8 Bons réflexes 7

- En cas de bug avéré
  - tenter de le cerner et de le reproduire au mieux
  - le signaler à la communauté de préférence (configuration, comment reproduire)

Ce n'est heureusement pas fréquent, mais il est possible que l'origine du problème soit liée à un bug de PostgreSQL lui-même.

Dans ce cas, la méthodologie appropriée consiste à essayer de reproduire le problème le plus fidèlement possible et de façon systématique, pour le cerner au mieux.

Il est ensuite très important de le signaler au plus vite à la communauté, généralement sur la liste pgsql-bugs@postgresql.org (cela nécessite une inscription préalable), en respectant les règles définies dans la documentation[3].

Notamment (liste non exhaustive) :

- indiquer la version précise de PostgreSQL installée, et la méthode d'installation utilisée ;
- préciser la plate-forme utilisée, notamment la version du système d'exploitation utilisé et la configuration des ressources du serveur ;
- signaler uniquement les faits observés, éviter les spéculations sur l'origine du problème ;
- joindre le détail des messages d'erreurs observés (augmenter la verbosité des erreurs avec le paramètre `log_error_verbosity`) ;
- joindre un cas complet permettant de reproduire le problème de façon aussi simple que possible.

Pour les problèmes relevant du domaine de la sécurité (découverte d'une faille), la liste adéquate est security@postgresql.org.

[3] https://www.postgresql.org/docs/current/static/bug-reporting.html

16.3.9 Bons réflexes 8

- Après correction
- Tester complètement l'intégrité des données
  - pour détecter tous les problèmes
- Validation avec export logique complet

```
pg_dumpall > /dev/null
```

- ou sauvegarde physique
  - `pg_basebackup`...
- Reconstruction dans une autre instance (vérification de cohérence)

```
pg_dumpall | psql -h autre serveur
```

Une fois les actions correctives réalisées (restauration, recréation d'objets, mise à jour des données…), il faut tester intensivement pour s'assurer que le problème est bien complètement résolu.

Il est donc extrêmement important d'avoir préparé des cas de tests permettant de reproduire le problème de façon certaine, afin de valider la solution appliquée.

En cas de suspicion de corruption de données, il est également important de tenter de procéder à la lecture de la totalité des données depuis PostgreSQL.

Un premier outil pour cela est une sauvegarde physique avec `pg_basebackup` ou un autre outil (voir plus loin).

Alternativement, la commande suivante, exécutée avec l'utilisateur système propriétaire de l'instance (généralement `postgres`) effectue une lecture complète de toutes les tables (et uniquement les tables), sans nécessiter de place sur disque supplémentaire :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell, la commande est :

```
PS C:\ pg_dumpall > $null
```

Cette commande ne devrait renvoyer aucune erreur. En cas de problème, notamment une somme de contrôle qui échoue, une erreur apparaîtra :

```
pg_dump: WARNING:  page verification failed, calculated checksum 20565 but expected
 ↪  17796
pg_dump: erreur : Sauvegarde du contenu de la table « corrompue » échouée :
                  échec de PQgetResult().
pg_dump: erreur : Message d'erreur du serveur :
```

```
        ERROR:  invalid page in block 0 of relation base/104818/104828
pg_dump: erreur : La commande était : COPY public.corrompue (i) TO stdout;
pg_dumpall: erreur : échec de pg_dump sur la base de données « corruption », quitte
```

Attention, les fichiers associés aux index ou aux vues matérialisées ne sont pas parcourus lors de la lecture des données par `pg_dumpall` ou `pg_dump`. Une corruption physique d'index ne sera donc pas remontée. Une vérification exhaustive implique d'autres outils comme `pg_checksums` ou `pg_basebackup`.
De plus, l'absence d'erreur physique ne garantit en aucun cas l'intégrité fonctionnelle des données : les corruptions peuvent très bien être logiques.

Dans les situations les plus extrêmes (problème de stockage, fichiers corrompus), il est important de tester la validité des données dans une nouvelle instance en effectuant un export/import complet des données.

Par exemple, initialiser une nouvelle instance avec `initdb`, sur un autre système de stockage, voire sur un autre serveur, puis lancer la commande suivante (l'application doit être coupée, ce qui est normalement le cas depuis la détection de l'incident si les conseils précédents ont été suivis) pour exporter et importer à la volée :

```
$ pg_dumpall -h <serveur_corrompu> -U postgres | psql -h <nouveau_serveur> \
                                                  -U postgres postgres
$ vacuumdb --analyze -h <nouveau_serveur> -U postgres postgres
```

D'éventuels problèmes peuvent être détectés lors de l'import des données, par exemple si des corruptions entraînent l'échec de la reconstruction de clés étrangères. Il faut alors procéder au cas par cas.

Enfin, même si cette étape s'est déroulée sans erreur, tout risque n'est pas écarté, il reste la possibilité de corruption de données silencieuses. Sauf si la fonctionnalité de checksum de PostgreSQL a été activée sur l'instance (ce n'est pas activé par défaut !), le seul moyen de détecter ce type de problème est de valider les données fonctionnellement.

Dans tous les cas, en cas de suspicion de corruption de données en profondeur, il est fortement préférable d'accepter une perte de données et de restaurer une sauvegarde d'avant le début de l'incident, plutôt que de continuer à travailler avec des données dont l'intégrité n'est pas assurée.

16.3.10 Mauvais réflexes 1

- Paniquer
- Prendre une décision hâtive
  - exemple, supprimer des fichiers du répertoire `pg_wal`
- Lancer une commande sans la comprendre, par exemple :
  - `pg_resetwal`
  - l'extension `pg_surgery`
  - DANGER, dernier espoir

Quelle que soit la criticité du problème rencontré, la panique peut en faire quelque chose de pire.

Il faut impérativement garder son calme, et résister au mieux au stress et aux pressions qu'une situation de désastre ne manque pas de provoquer.

Il est également préférable d'éviter de sauter immédiatement à la conclusion la plus évidente. Il ne faut pas hésiter à retirer les mains du clavier pour prendre de la distance par rapport aux conséquences du problème, réfléchir aux causes possibles, prendre le temps d'aller chercher de l'information pour réévaluer l'ampleur réelle du problème.

La plus mauvaise décision que l'on peut être amenée à prendre lors de la gestion d'un incident est celle que l'on prend dans la précipitation, sans avoir bien réfléchi et mesuré son impact. Cela peut provoquer des dégâts irrécupérables, et transformer une situation d'incident en situation de crise majeure.

Un exemple classique de ce type de comportement est le cas où PostgreSQL est arrêté suite au remplissage du système de fichiers contenant les fichiers WAL, `pg_wal`.

Le réflexe immédiat d'un administrateur non averti pourrait être de supprimer les plus vieux fichiers dans ce répertoire, ce qui répond bien aux symptômes observés mais reste une erreur dramatique qui va rendre le démarrage de l'instance impossible.

Quoi qu'il arrive, ne jamais exécuter une commande sans être certain qu'elle correspond bien à la situation rencontrée, et sans en maîtriser complètement les impacts. Même si cette commande provient d'un document mentionnant les mêmes messages d'erreur que ceux rencontrés (et tout particulièrement si le document a été trouvé via une recherche hâtive sur Internet) !

Là encore, nous disposons comme exemple d'une erreur malheureusement fréquente, l'exécution de la commande `pg_resetwal` sur une instance rencontrant un problème. Comme l'indique la documentation, « cette commande ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption_ » et « *il ne faut pas perdre de vue que la base de données peut contenir des données incohérentes du fait de transactions partiellement validées* » (documentation[4]). Nous

[4] https://docs.postgresql.fr/current/app-pgresetwal.html

reviendrons ultérieurement sur les (rares) cas d'usage réels de cette commande, mais dans l'immense majorité des cas, l'utiliser va aggraver le problème, en ajoutant des problématiques de corruption logique des données !

Il convient donc de bien s'assurer de comprendre les conséquences de l'exécution de chaque action effectuée.

16.3.11 Mauvais réflexes 2

- Arrêter le diagnostic quand les symptômes disparaissent
- Ne pas pousser l'analyse jusqu'au bout

Il est important de pousser la réflexion jusqu'à avoir complètement compris l'origine du problème et ses conséquences.

En premier lieu, même si les symptômes semblent avoir disparus, il est tout à fait possible que le problème soit toujours sous-jacent, ou qu'il ait eu des conséquences moins visibles mais tout aussi graves (par exemple, une corruption logique de données).

Ensuite, même si le problème est effectivement corrigé, prendre le temps de comprendre et de documenter l'origine du problème (rapport « post-mortem ») a une valeur inestimable pour prendre les mesures afin d'éviter que le problème ne se reproduise, et retrouver rapidement les informations utiles s'il venait à se reproduire malgré tout.

16.3.12 Mauvais réflexes 3

- Ne pas documenter
  - le résultat de l'investigation
  - les actions effectuées

Après s'être assuré d'avoir bien compris le problème rencontré, il est tout aussi important de le documenter soigneusement, avec les actions de diagnostic et de correction effectuées.

Ne pas le faire, c'est perdre une excellente occasion de gagner un temps précieux si le problème venait à se reproduire.

C'est également un risque supplémentaire dans le cas où les actions correctives menées n'auraient pas suffi à complètement corriger le problème ou auraient eu un effet de bord inattendu.

Dans ce cas, avoir pris le temps de noter le détail des actions effectuées fera là encore gagner un temps précieux.

16.4 RECHERCHER L'ORIGINE DU PROBLÈME

Quelques pistes (liste non exhaustive)

Les problèmes pouvant survenir sont trop nombreux pour pouvoir tous les lister, chaque élément matériel ou logiciel d'une architecture pouvant subir de nombreux types de défaillances.

Cette section liste quelques pistes classiques d'investigation à ne pas négliger pour s'efforcer de cerner au mieux l'étendue du problème, et en déterminer les conséquences.

16.4.1 Prérequis

- Avant de commencer à creuser
  - référencer les symptômes
  - identifier au mieux l'instant de démarrage du problème

La première étape est de déterminer aussi précisément que possible les symptômes observés, sans en négliger, et à partir de quel moment ils sont apparus.

Cela donne des informations précieuses sur l'étendue du problème, et permet d'éviter de se focaliser sur un symptôme particulier, parce que plus visible (par exemple l'arrêt brutal de l'instance), alors que la cause réelle est plus ancienne (par exemple des erreurs IO dans les traces système, ou une montée progressive de la charge sur le serveur).

16.4.2 Recherche d'historique

- Ces symptômes ont-ils déjà été rencontrés dans le passé ?
- Ces symptômes ont-ils déjà été rencontrés par d'autres ?
- Attention à ne pas prendre les informations trouvées pour argent comptant !

Une fois les principaux symptômes identifiés, il est utile de prendre un moment pour déterminer si ce problème est déjà connu.

Notamment, identifier dans la base de connaissances si ces symptômes ont déjà été rencontrés dans le passé (d'où l'importance de bien documenter les problèmes).

Au-delà de la documentation interne, il est également possible de rechercher si ces symptômes ont déjà été rencontrés par d'autres.

Pour ce type de recherche, il est préférable de privilégier les sources fiables (documentation officielle, listes de discussion, plate-forme de support…) plutôt qu'un quelconque document d'un auteur non identifié.

Dans tous les cas, il faut faire très attention à ne pas prendre les informations trouvées pour argent comptant, et ce même si elles proviennent de la documentation interne ou d'une source fiable !

Il est toujours possible que les symptômes soient similaires mais que la cause soit différente. Il s'agit donc ici de mettre en place une base de travail, qui doit être complétée par une observation directe et une analyse.

16.4.3 Matériel

- Vérifier le système disque (SAN, carte RAID, disques)
- Un `fsync` est-il bien honoré de l'OS au disque ? (batteries !)
- Rechercher toute erreur matérielle
- Firmwares pas à jour
  - ou récemment mis à jour
- Matériel récemment changé

Les défaillances du matériel, et notamment du système de stockage, sont de celles qui peuvent avoir les impacts les plus importants et les plus étendus sur une instance et sur les données qu'elle contient.

Ce type de problème peut également être difficile à diagnostiquer en se contentant d'observer les symptômes les plus visibles. Il est facile de sous-estimer l'ampleur des dégâts.

Parmi les bonnes pratiques, il convient de vérifier la configuration et l'état du système disque (SAN, carte RAID, disques).

Quelques éléments étant une source habituelle de problèmes :

- le système disque n'honore pas les ordres `fsync` ? (SAN ? virtualisation ?) ;
- quel est l'état de la batterie du cache en écriture ?

Il faut évidemment rechercher la présence de toute erreur matérielle, au niveau des disques, de la mémoire, des CPU…

Vérifier également la version des firmwares installés. Il est possible qu'une nouvelle version corrige le problème rencontré, ou à l'inverse que le déploiement d'une nouvelle version soit à l'origine du problème.

Dans le même esprit, il faut vérifier si du matériel a récemment été changé. Il arrive que de nouveaux éléments soient défaillants.

Il convient de noter que l'investigation à ce niveau peut être grandement complexifiée par l'utilisation de certaines technologies (virtualisation, baies de stockage), du fait de la mutualisation des ressources, et de la séparation des compétences et des informations de supervision entre différentes équipes.

16.4.4 Virtualisation

- Mutualisation excessive
- Configuration du stockage virtualisé
- Rechercher les erreurs aussi niveau superviseur
- Mises à jour non appliquées
  - ou appliquées récemment
- Modifications de configuration récentes

Tout comme pour les problèmes au niveau du matériel, les problèmes au niveau du système de virtualisation peuvent être complexes à détecter et à diagnostiquer correctement.

Le principal facteur de problème avec la virtualisation est lié à une mutualisation excessive des ressources.

Il est ainsi possible d'avoir un total de ressources allouées aux VM supérieur à celles disponibles sur l'hyperviseur, ce qui amène à des comportements de fort ralentissement, voire de blocage des systèmes virtualisés.

Si ce type d'architecture est couplé à un système de gestion de bascule automatique (Pacemaker, repmgr…), il est possible d'avoir des situations de bascules impromptues, voire des situations de *split brain*, qui peuvent provoquer des pertes de données importantes. Il est donc important de prêter une attention particulière à l'utilisation des ressources de l'hyperviseur, et d'éviter à tout prix la sur-allocation.

Par ailleurs, lorsque l'architecture inclut une brique de virtualisation, il est important de prendre en compte que certains problèmes ne peuvent être observés qu'à partir de l'hyperviseur, et pas à partir du système virtualisé. Par exemple, les erreurs matérielles ou système risquent d'être invisibles depuis une VM, il convient donc d'être vigilant, et de rechercher toute erreur sur l'hôte.

Il faut également vérifier si des modifications ont été effectuées peu avant l'incident, comme des modifications de configuration ou l'application de mises à jour.

Comme indiqué dans la partie traitant du matériel, l'investigation peut être grandement freinée par la séparation des compétences et des informations de supervision entre différentes équipes. Une bonne communication est alors la clé de la résolution rapide du problème.

16.4.5 Système d'exploitation 1

- Erreurs dans les traces
- Mises à jour système non appliquées
- Modifications de configuration récentes

Après avoir vérifié les couches matérielles et la virtualisation, il faut ensuite s'assurer de l'intégrité du système d'exploitation.

La première des vérifications à effectuer est de consulter les traces du système pour en extraire les éventuels messages d'erreur :

- sous Linux, on trouvera ce type d'informations en sortie de la commande `dmesg`, et dans les fichiers traces du système, généralement situés sous `/var/log` ;
- sous Windows, on consultera à cet effet le journal des événements (les `event logs`).

Tout comme pour les autres briques, il faut également voir s'il existe des mises à jour des paquets qui n'auraient pas été appliquées, ou à l'inverse si des mises à jour, installations ou modifications de configuration ont été effectuées récemment.

16.4.6 Système d'exploitation 2

- Opération d'IO impossible
  - FS plein ?
  - FS monté en lecture seule ?
- Tester l'écriture sur PGDATA
- Tester la lecture sur PGDATA

Parmi les problèmes fréquemment rencontrés se trouve l'impossibilité pour PostgreSQL d'accéder en lecture ou en écriture à un ou plusieurs fichiers.

La première chose à vérifier est de déterminer si le système de fichiers sous-jacent ne serait pas rempli à 100% (commande `df` sous Linux) ou monté en lecture seule (commande `mount` sous Linux).

On peut aussi tester les opérations d'écriture et de lecture sur le système de fichiers pour déterminer si le comportement y est global :

- pour tester une écriture dans le répertoire `PGDATA`, sous Linux :

```
$ touch $PGDATA/test_write
```

- pour tester une lecture dans le répertoire `PGDATA`, sous Linux :

```
$ cat $PGDATA/PGVERSION
```

Pour identifier précisément les fichiers présentant des problèmes, il est possible de tester la lecture complète des fichiers dans le point de montage :

```
$ tar cvf /dev/null $PGDATA
```

16.4.7 Système d'exploitation 3

- Consommation excessive des ressources
  - OOM killer (overcommit !)
- Après un crash, vérifier les processus actifs
  - ne pas tenter de redémarrer si des processus persistent
- Outils : `sar`, `atop`…

Sous Linux, l'installation d'outils d'aide au diagnostic sur les serveurs est très important pour mener une analyse efficace, particulièrement le paquet `systat` qui permet d'utiliser la commande `sar`.

La lecture des traces système et des traces PostgreSQL permettent également d'avancer dans le diagnostic.

Un problème de consommation excessive des ressources peut généralement être anticipée grâce à une supervision sur l'utilisation des ressources et des seuils d'alerte appropriés. Il arrive néanmoins parfois que la consommation soit très rapide et qu'il ne soit pas possible de réagir suffisamment rapidement.

Dans le cas d'une consommation mémoire d'un serveur Linux qui menacerait de dépasser la quantité totale de mémoire allouable, le comportement par défaut de Linux est d'autoriser par défaut la tentative d'allocation.

Si l'allocation dépasse effectivement la mémoire disponible, alors le système va déclencher un processus *Out Of Memory Killer* (OOM Killer) qui va se charger de tuer les processus les plus consommateurs.

Dans le cas d'un serveur dédié à une instance PostgreSQL, il y a de grandes chances que le processus en question appartienne à l'instance.

S'il s'agit d'un *OOM Killer* effectuant un arrêt brutal (`kill -9`) sur un backend, l'instance PostgreSQL va arrêter immédiatement tous les processus afin de prévenir une corruption de la mémoire et les redémarrer.

S'il s'agit du processus principal de l'instance (*postmaster*), les conséquences peuvent être bien plus dramatiques, surtout si une tentative est faite de redémarrer l'instance sans vérifier si des processus actifs existent encore.

Pour un serveur dédié à PostgreSQL, la recommendation est habituellement de désactiver la surallocation de la mémoire, empêchant ainsi le déclenchement de ce phénomène.

Voir pour cela les paramètres kernel `vm.overcommit_memory` et `vm.overcommit_ratio` (référence : https://kb.dalibo.com/overcommit_memory).

16.4.8 PostgreSQL

- Relever les erreurs dans les traces
  - ou messages inhabituels
- Vérifier les mises à jour mineures

Tout comme pour l'analyse autour du système d'exploitation, la première chose à faire est rechercher toute erreur ou message inhabituel dans les traces de l'instance. Ces messages sont habituellement assez informatifs, et permettent de cerner la nature du problème. Par exemple, si PostgreSQL ne parvient pas à écrire dans un fichier, il indiquera précisément de quel fichier il s'agit.

Si l'instance est arrêtée suite à un crash, et que les tentatives de redémarrage échouent avant qu'un message puisse être écrit dans les traces, il est possible de tenter de démarrer l'instance en exécutant directement le binaire `postgres` afin que les premiers messages soient envoyés vers la sortie standard.

Il convient également de vérifier si des mises à jour qui n'auraient pas été appliquées ne corrigeraient pas un problème similaire à celui rencontré.

Identifier les mises à jours appliquées récemment et les modifications de configuration peut également aider à comprendre la nature du problème.

16.4.9 Paramétrage de PostgreSQL : écriture des fichiers

- La désactivation de certains paramètres est dangereuse
  - `fsync`
  - `full_page_write`

Si des corruptions de données sont relevées suite à un crash de l'instance, il convient particulièrement de vérifier la valeur du paramètre `fsync`.

En effet, si celui-ci est désactivé, les écritures dans les journaux de transactions ne sont pas effectuées de façon synchrone, ce qui implique que l'ordre des écritures ne sera pas conservé en cas de crash. Le processus de *recovery* de PostgreSQL risque alors de provoquer des corruptions si l'instance est malgré tout redémarrée.

Ce paramètre ne devrait jamais être positionné à une autre valeur que `on`, sauf dans des cas extrêmement particuliers (en bref, si l'on peut se permettre de restaurer intégralement les données en cas de crash, par exemple dans un chargement de données initial).

Le paramètre `full_page_write` indique à PostgreSQL d'effectuer une écriture complète d'une page chaque fois qu'elle reçoit une nouvelle écriture après un checkpoint, pour éviter un éventuel mélange entre des anciennes et nouvelles données en cas d'écriture partielle.

La désactivation de `full_page_write` peut avoir le même type de conséquences catastrophiques que celle de `fsync` !

16.4.10 Paramétrage de PostgreSQL : les sommes de contrôle

- Activez les checksums !
  - pas (encore ?) par défaut
  - `initdb --data-checksums` (création)
  - `pg_checksums --enable` (à posteriori)
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Vérification :
  - `pg_checksums --check`
  - outils de sauvegarde et lecture

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Sans mécanisme de sécurité, il est donc possible de modifier un fichier sans que PostgreSQL s'en rende compte, ce qui aboutit à une corruption silencieuse.

Les sommes de contrôles (*checksums*) permettent de se prémunir contre des corruptions silencieuses de données. Hélas, elles ne sont pas actives par défaut (cela pourrait bientôt changer). Leur mise en place est donc fortement recommandée sur une nouvelle instance. L'activation se fait différemment suivant les environnements (voir les procédures pour les paquets RPM et Debian du PGDG[5] mais revient toujours à transmettre le paramètre `--data-checksums` à `initdb`.

Il est possible de mettre en place les *checksums* sur une instance existante avec l'utilitaire `pg_checksums`[6]. Les gros inconvénients d'une activation tardive sont :

- tous les fichiers de données et d'index seront réécrits (puisqu'il faudra calculer le *checksum* de chaque bloc, écrit dans le bloc même) ;
- l'instance doit être arrêtée pendant toute l'opération.

À titre d'exemple, créons une instance sans utiliser les *checksums*, et une autre qui les utilisera :

```
$ initdb -D /tmp/sans_checksums/
$ initdb -D /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacun des deux clusters :

```
CREATE TABLE test (name text);
INSERT INTO test (name) VALUES ('toto');
```

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans *checksums* est montré en exemple).

[5] https://dali.bo/b_html#installation-de-postgresql-depuis-les-paquets-communautaires
[6] https://docs.postgresql.fr/current/app-pgchecksums.html

```
SELECT pg_relation_filepath('test');
```

```
 pg_relation_filepath
----------------------
 base/12036/16317
```

Instance arrêtée (pour ne pas être gêné par le cache), on va s'attacher à corrompre ce fichier, en remplaçant la valeur « toto » par « goto » avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/12036/16317
$ hexedit /tmp/avec_checksums/base/12036/16399
```

(On remarquera au passage que la ligne est à la fin du bloc de 8 ko.) Enfin, on peut ensuite exécuter des requêtes sur ces deux clusters.

Sans *checksums* :

```
TABLE test;
```

```
 name
------
 qoto
```

Avec *checksums* :

```
TABLE test;
```

```
WARNING:  page verification failed, calculated checksum 16321
          but expected 21348
ERROR:  invalid page in block 0 of relation base/12036/16387
```

Les sommes de contrôle sont vérifiées lors des lectures par les requêtes, mais aussi lors des sauvegardes, notamment lors d'un appel à `pg_dump` (données de la base concernée uniquement), `pg_basebackup`, `pgbackrest`… En cas de corruption des données, l'opération sera interrompue. `pg_checksums --check` peut vérifier complètement une instance arrêtée.

En pratique, avec PostgreSQL 9.5 au moins et des processeurs supportant les instructions SSE 4.2 (voir dans `/proc/cpuinfo`, mais cela concerne au moins tous les processeurs Intel depuis 15 ans), il n'y aura pas d'impact notable en performances. Par contre, il y a aura un peu plus de journaux générés, pour des raisons techniques.

L'activation ou non des sommes de contrôle peut se faire indépendamment sur un serveur primaire et son secondaire, mais il est fortement conseillé de les activer simultanément des deux côtés pour éviter de gros problèmes dans certains scénarios de restauration.

16.4.11 Erreur de manipulation

- Traces système, traces PostgreSQL
- Revue des dernières manipulations effectuées
- Historique des commandes
- Danger : `kill -9`, `rm -rf`, `rsync`, `find … -exec` …

L'erreur humaine fait également partie des principales causes de désastre.

Une commande de suppression tapée trop rapidement, un oubli de clause `WHERE` dans une requête de mise à jour, nombreuses sont les opérations qui peuvent provoquer des pertes de données ou un crash de l'instance.

Il convient donc de revoir les dernières opérations effectuées sur le serveur, en commençant par les interventions planifiées, et si possible récupérer l'historique des commandes passées.

Des exemples de commandes particulièrement dangereuses :

- `kill -9`
- `rm -rf`
- `rsync`
- `find` (souvent couplé avec des commandes destructices comme `rm`, `mv`, `gzip`…)

16.5 OUTILS

- Quelques outils peuvent aider
  - à diagnostiquer la nature du problème
  - à valider la correction apportée
  - à appliquer un contournement
- ATTENTION
  - certains de ces outils peuvent corrompre les données !

16.5.1 Outils : pg_controldata

- Fournit des informations de contrôle sur l'instance
- Ne nécessite pas que l'instance soit démarrée

L'outil `pg_controldata` est livré avec PostgreSQL. Il lit les informations du fichier de contrôle d'une instance PostgreSQL. Cet outil ne se connecte pas à l'instance, il a juste besoin d'avoir un accès en lecture sur le répertoire `PGDATA` de l'instance.

Les informations qu'il récupère ne sont donc pas du temps réel, il s'agit d'une vision de l'instance telle qu'elle était la dernière fois que le fichier de contrôle a été mis à jour. L'avantage est qu'elle peut être utilisée même si l'instance est arrêtée.

`pg_controldata` affiche notamment les informations initialisées lors d'`initdb`, telles que la version du catalogue, ou la taille des blocs, qui peuvent être cruciales si l'on veut restaurer une instance sur un nouveau serveur à partir d'une copie des fichiers.

Il affiche également de nombreuses informations utiles sur le traitement des journaux de transactions et des checkpoints, par exemple :

- positions de l'avant-dernier checkpoint et du dernier checkpoint dans les WAL ;
- nom du WAL correspondant au dernier WAL ;
- timeline sur laquelle se situe le dernier checkpoint ;
- instant précis du dernier checkpoint.

Quelques informations de paramétrage sont également renvoyées, comme la configuration du niveau de WAL, ou le nombre maximal de connexions autorisées.

En complément, le dernier état connu de l'instance est également affiché. Les états potentiels sont :

- `in production` : l'instance est démarrée et est ouverte en écriture ;
- `shut down` : l'instance est arrêtée ;
- `in archive recovery` : l'instance est démarrée et est en mode `recovery` (restauration, `Warm` ou `Hot Standby`) ;
- `shut down in recovery` : l'instance s'est arrêtée alors qu'elle était en mode `recovery` ;
- `shutting down` : état transitoire, l'instance est en cours d'arrêt ;
- `in crash recovery` : état transitoire, l'instance est en cours de démarrage suite à un crash ;
- `starting up` : état transitoire, concrètement jamais utilisé.

Bien entendu, comme ces informations ne sont pas mises à jour en temps réel, elles peuvent être erronées.

Cet asynchronisme est intéressant pour diagnostiquer un problème, par exemple si `pg_controldata` renvoie l'état `in production` mais que l'instance est arrêtée, cela signifie que l'arrêt n'a pas été effectué proprement (*crash* de l'instance, qui sera donc suivi d'un `recovery` au démarrage).

Exemple de sortie de la commande :

```
$ /usr/pgsql-10/bin/pg_controldata /var/lib/pgsql/10/data

pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6451139765284827825
Database cluster state:               in production
pg_control last modified:             Mon 28 Aug 2017 03:40:30 PM CEST
Latest checkpoint location:           1/2B04EC0
Prior checkpoint location:            1/2B04DE8
Latest checkpoint's REDO location:    1/2B04E88
Latest checkpoint's REDO WAL file:    000000010000000100000002
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:1023
Latest checkpoint's NextOID:          41064
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        548
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  1022
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Mon 28 Aug 2017 03:40:30 PM CEST
Fake LSN counter for unlogged rels:   0/1
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
```

```
wal_level setting:                    replica
wal_log_hints setting:                off
max_connections setting:              100
max_worker_processes setting:         8
max_prepared_xacts setting:           0
max_locks_per_xact setting:           64
track_commit_timestamp setting:       off
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Data page checksum version:           0
Mock authentication nonce:            7fb23aca2465c69b2c0f54ccf03e0ece
                                      3c0933c5f0e5f2c096516099c9688173
```

16.5.2 Outils pour l'export/import de données

- `pg_dump`
- `pg_dumpall`
- `COPY`
- `psql` / `pg_restore`
  - `--section=pre-data` / `data` / `post-data`

Les outils `pg_dump` et `pg_dumpall` permettent d'exporter des données à partir d'une instance démarrée. Dans le cadre d'un incident grave, il est possible de les utiliser pour :

- extraire le contenu de l'instance suspecte ;
- extraire le contenu des bases de données ;
- tester si les données sont lisibles dans un format compréhensible par PostgreSQL.

Par exemple, un moyen rapide de s'assurer que tous les fichiers des tables de l'instance sont lisibles est de forcer leur lecture complète, notamment grâce à la commande suivante :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell :

```
pg_dumpall > $null
```

Rappelons que les fichiers associés aux index et vues matérialisées ne sont pas parcourus pendant cette opération.

Si `pg_dumpall` ou `pg_dump` renvoient des messages d'erreur et ne parviennent pas à exporter certaines tables, il est possible de contourner le problème à l'aide de la commande `COPY`, en sélectionnant exclusivement les données lisibles autour du bloc corrompu. Exemples :

```
-- si accès par index possible
COPY (SELECT * FROM pgbench_accounts WHERE aid BETWEEN 50000 AND 60000)
TO 'backuppartiel.dmp';
-- accès par références de lignes
COPY (SELECT * FROM pgbench_accounts WHERE ctid BETWEEN '(0,1)' AND '(1639,20)')
TO 'backuppartiel.dmp';
```

Il convient ensuite d'utiliser `psql` ou `pg_restore` pour importer les données dans une nouvelle instance, probablement sur un nouveau serveur, dans un environnement non affecté par le problème. Pour parer au cas où le réimport échoue à cause de contraintes non respectées, il est souvent préférable de faire le réimport par étapes :

```
$ pg_restore -1 --section=pre-data --verbose -d cible base.dump
$ pg_restore -1 --section=data --verbose -d cible base.dump
$ pg_restore -1 --section=post-data --exit-on-error --verbose -d cible base.dump
```

En cas de problème, on verra les contraintes posant problème.

Il peut être utile de générer les scripts en pur SQL avant de les appliquer, éventuellement par étape :

```
$ pg_restore --section=post-data -f postdata.sql base.dump
```

Pour rappel, même après un export/import de données réalisé avec succès, des corruptions logiques peuvent encore être présentes. Il faut donc être particulièrement vigilant et prendre le temps de valider l'intégrité fonctionnelle des données.

16.5.3 Outils : pageinspect

- Extension
- Vision du contenu d'un bloc
- Sans le dictionnaire, donc sans décodage des données
- Affichage brut
- Utilisé surtout en debug, ou dans les cas de corruption
- Fonctions de décodage pour les tables, les index (B-tree, hash, GIN, GiST), FSM
- Nécessite de connaître le code de PostgreSQL

Voici quelques exemples.

Contenu d'une page d'une table :

```
SELECT * FROM heap_page_items(get_raw_page('dspam_token_data',0)) LIMIT 2;
```

```
-[ RECORD 1 ]------
lp          | 1
lp_off      | 8152
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,7)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x01000000010000000100000001000000
-[ RECORD 2 ]------
lp          | 2
lp_off      | 8112
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,8)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x02000000010000000100000002000000
```

Et son entête :

```
SELECT * FROM page_header(get_raw_page('dspam_token_data',0));
```

```
-[ RECORD 1 ]--------
lsn       | F1A/5A6EAC40
checksum  | 0
flags     | 0
lower     | 56
upper     | 7872
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 839
```

Méta-données d'un index (contenu dans la première page) :

```
SELECT * FROM bt_metap('dspam_token_data_uid_key');
```

```
-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 243
level     | 2
fastroot  | 243
fastlevel | 2
```

La page racine est la 243. Allons la voir :

```
SELECT * FROM bt_page_items('dspam_token_data_uid_key',243) LIMIT 10;
```

```
 offset |   ctid     | len |nulls|vars|                 data
--------+-----------+-----+-----+----+-------------------------------------
      1 | (3,1)      |   8 | f   | f  |
      2 | (44565,1)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (242,1)    |  20 | f   | f  | 77 c6 0d 6f a6 92 db 81 28 00 00 00
      4 | (43569,1)  |  20 | f   | f  | 47 a6 aa be 29 e3 13 83 18 00 00 00
      5 | (481,1)    |  20 | f   | f  | 30 17 dd 8e d9 72 7d 84 0a 00 00 00
      6 | (43077,1)  |  20 | f   | f  | 5c 3c 7b c5 5b 7a 4e 85 0a 00 00 00
      7 | (719,1)    |  20 | f   | f  | 0d 91 d5 78 a9 72 88 86 26 00 00 00
      8 | (41209,1)  |  20 | f   | f  | a7 8a da 17 95 17 cd 87 0a 00 00 00
      9 | (957,1)    |  20 | f   | f  | 78 e9 64 e9 64 a9 52 89 26 00 00 00
     10 | (40849,1)  |  20 | f   | f  | 53 11 e9 64 e9 1b c3 8a 26 00 00 00
```

La première entrée de la page 243, correspondant à la donnée `f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00` est stockée dans la page 3 de notre index :

```
SELECT * FROM bt_page_stats('dspam_token_data_uid_key',3);
```

```
-[ RECORD 1 ]-+-----
blkno         | 3
type          | i
live_items    | 202
dead_items    | 0
avg_item_size | 19
page_size     | 8192
free_size     | 3312
btpo_prev     | 0
btpo_next     | 44565
btpo          | 1
btpo_flags    | 0
```

```
SELECT * FROM bt_page_items('dspam_token_data_uid_key',3) LIMIT 10;

 offset |   ctid     | len |nulls|vars|                data
--------+-----------+-----+-----+----+-----------------------------------
      1 | (38065,1) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      2 | (1,1)     |   8 | f   | f  |
      3 | (37361,1) |  20 | f   | f  | 30 fd 30 b8 70 c9 01 80 26 00 00 00
      4 | (2,1)     |  20 | f   | f  | 18 2c 37 36 27 03 03 80 27 00 00 00
      5 | (4,1)     |  20 | f   | f  | 36 61 f3 b6 c5 1b 03 80 0f 00 00 00
      6 | (43997,1) |  20 | f   | f  | 30 4a 32 58 c8 44 03 80 27 00 00 00
      7 | (5,1)     |  20 | f   | f  | 88 fe 97 6f 7e 5a 03 80 27 00 00 00
      8 | (51136,1) |  20 | f   | f  | 74 a8 5a 9b 15 5d 03 80 28 00 00 00
      9 | (6,1)     |  20 | f   | f  | 44 41 3c ee c8 fe 03 80 0a 00 00 00
     10 | (45317,1) |  20 | f   | f  | d4 b0 7c fd 5d 8d 05 80 26 00 00 00
```

Le type de la page est `i`, c'est-à-dire «internal», donc une page interne de l'arbre. Continuons notre descente, allons voir la page 38065 :

```
SELECT * FROM bt_page_stats('dspam_token_data_uid_key',38065);

-[ RECORD 1 ]-+-----
blkno         | 38065
type          | l
live_items    | 169
dead_items    | 21
avg_item_size | 20
page_size     | 8192
free_size     | 3588
btpo_prev     | 118
btpo_next     | 119
btpo          | 0
btpo_flags    | 65

SELECT * FROM bt_page_items('dspam_token_data_uid_key',38065) LIMIT 10;

 offset |    ctid     | len |nulls|vars|                data
--------+-------------+-----+-----+----+-----------------------------------
      1 | (11128,118) |  20 | f   | f  | 33 37 89 95 b9 23 cc 80 0a 00 00 00
      2 | (45713,181) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (45424,97)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 26 00 00 00
      4 | (45255,28)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 27 00 00 00
      5 | (15672,172) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 28 00 00 00
      6 | (5456,118)  |  20 | f   | f  | f3 bf 29 a2 39 a3 cb 80 0f 00 00 00
      7 | (8356,206)  |  20 | f   | f  | f3 bf 29 a2 39 a3 cb 80 28 00 00 00
      8 | (33895,272) |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 0a 00 00 00
      9 | (5176,108)  |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 0f 00 00 00
     10 | (5466,41)   |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 26 00 00 00
```

Nous avons trouvé une feuille (type `l`). Les ctid pointés sont maintenant les adresses dans la table :

```
SELECT * FROM dspam_token_data WHERE ctid = '(11128,118)';

 uid |        token         | spam_hits | innocent_hits |  last_hit
-----+----------------------+-----------+---------------+------------
  40 | -6317261189288392210 |         0 |             3 | 2014-11-10
```

16.5.4 Outils : pg_resetwal

- Efface les WAL courants
- Permet à l'instance de démarrer en cas de corruption d'un WAL
  - comme si elle était dans un état cohérent
  - …ce qui n'est pas le cas
- **Cet outil est dangereux et mène à des corruptions !!!**
- Pour récupérer ce qu'on peut, et réimporter ailleurs

`pg_resetwal` est un outil fourni avec PostgreSQL. Son objectif est de pouvoir démarrer une instance après un crash si des corruptions de fichiers (typiquement WAL ou fichier de contrôle) empêchent ce démarrage.

Cette action n'est pas une action de réparation ! La réinitialisation des journaux de transactions implique que des transactions qui n'étaient que partiellement validées ne seront pas détectées comme telles, et ne seront donc pas annulées lors du *recovery*.

La conséquence est que les **données de l'instance ne sont plus cohérentes**. Il est fort possible d'y trouver des violations de contraintes diverses (notamment clés étrangères), ou d'autres cas d'incohérences plus difficiles à détecter.

Il s'utilise manuellement, en ligne de commande. Sa fonctionnalité principale est d'effacer les fichiers WAL courants, et il se charge également de réinitialiser les informations correspondantes du fichier de contrôle. Il est possible de lui spécifier les valeurs à initialiser dans le fichier de contrôle si l'outil ne parvient pas à les déterminer (par exemple, si tous les WAL dans le répertoire `pg_wal` ont été supprimés).

Attention, `pg_resetwal` ne doit **jamais** être utilisé sur une instance démarrée. Avant d'exécuter l'outil, il faut toujours vérifier qu'il ne reste aucun processus de l'instance.

Après la réinitialisation des WAL, une fois que l'instance a démarré, **il ne faut surtout pas ouvrir les accès à l'application** ! Comme indiqué, les données présentent sans aucun doute des incohérences, et toute action en écriture à ce point ne ferait qu'aggraver le problème.

L'étape suivante est donc :

- de faire un export immédiat des données ;
- de les restaurer dans une nouvelle instance initialisée à cet effet (de préférence sur un nouveau serveur, surtout si l'origine de la corruption n'a pas été clairement identifiée) ;
- ensuite de procéder à une validation méthodique des données.

Il est probable que certaines données incohérentes puissent être identifiées à l'import, lors de la phase de recréation des contraintes : celles-ci échoueront si les données ne les respectent pas, ce qui permettra de les identifier.

En ce qui concerne les incohérences qui passeront au travers de ces tests, il faudra les trouver et les corriger manuellement, en procédant à une validation fonctionnelle des données.

Il faut donc bien retenir les points suivants :

- `pg_resetwal` n'est pas magique ;
- `pg_resetwal` rend les données incohérentes (ce qui est souvent pire qu'une simple perte d'une partie des données, comme on aurait eu en restaurant une sauvegarde) ;
- `pg_resetwal` n'est à utiliser qu'en dernier recours s'il n'y a aucun autre moyen de faire autrement pour récupérer les données ;
- il ne faut pas l'utiliser sur l'instance ayant subi le problème, mais sur une copie complète effectuée à froid ;
- il permet au mieux d'exporter les données pour les importer dans une nouvelle instance ;
- la validation soigneuse des données de la nouvelle instance est capitale.

16.5.5 Outils : pg_surgery

- Extension (v14+)
- Collection de fonctions permettant de modifier le statut des tuples d'une relation
- **Extrêmement dangereuse**

À partir de PostgreSQL 14, cette extension regroupe des fonctions qui permettent de modifier le statut d'un tuple dans une relation. Il est par exemple possible de rendre une ligne morte ou de rendre visible des tuples qui sont invisibles à cause des informations de visibilité.

Ces fonctions sont dangereuses et peuvent provoquer ou aggraver des corruptions. Elles peuvent par exemple rendre une table incohérente par rapport à ses index, ou provoquer une violation de contrainte d'unicité ou de clé étrangère. Il ne faut donc les utiliser qu'en dernier recours, sur une copie de votre instance.

Référence : Documentation officielle[7]

16.5.6 Outils pour vérifier les sommes de contrôle

- Base arrêtée :
  - `pg_checksums --check`
- À la lecture
  - dont `pg_dump`, `pg_dumpall`
- Lors d'une sauvegarde physique :
  - `pg_basebackup`, voire `pg_basebackup --target=blackhole`
  - pgBackRest, barman (selon configuration)

Une fois les sommes de contrôle en place, elles sont revérifiées à chaque lecture du bloc, mais il n'y a pas d'automatisme pour une vérification globale.

pg_checksums :

`pg_checksums --check` est la commande dédiée pour vérifier les sommes de contrôles existantes sur les bases de données **à froid** : l'instance doit être arrêtée proprement auparavant.

Par exemple, suite à une modification de deux blocs dans une table avec l'outil `hexedit`, on peut rencontrer ceci :

```
$ /usr/pgsql-12/bin/pg_checksums -D /var/lib/pgsql/12/data -c

pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 0:
  calculated checksum 5BF9 but block contains C55D
pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 4438:
  calculated checksum A3 but block contains B8AE
Checksum operation completed
```

[7] https://docs.postgresql.fr/current/pgsurgery.html

```
Files scanned:  1282
Blocks scanned: 28484
Bad checksums:  2
Data checksum version: 1
```

Rappelons que `pg_checksums` sait aussi ajouter ou supprimer les sommes de contrôle sur une instance existante arrêtée (donc après le `initdb`).

Sauvegarde logique :

`pg_dumpall > /dev/null` vérifie certaines sommes de contrôles car il lit tous les fichiers de données, mais pas les index. Si une sauvegarde logique fonctionne (et se restaure sans souci…), on a au moins sauvé l'essentiel.

pg_basebackup :

La meilleure alternative est d'effectuer une sauvegarde physique avec `pg_basebackup`. Par défaut, toutes les sommes de contrôle sont alors vérifiées. Cela n'oblige pas à arrêter la base. De plus, si l'outil est utilisé pour faire des sauvegardes, on obtient une vérification gratuite. S'il s'agit juste de forcer la lecture intégrale, il existe même une option dédiée `--target=blackhole` qui n'écrit pas la sauvegarde (à partir de PostgreSQL 15) :

```
$ pg_basebackup -h /var/run/postgresql -X none --target=blackhole
WARNING:  checksum verification failed in file "./base/5/1824979", block 1:
 ↪  calculated B15 but expected B41E
WARNING:  checksum verification failed in file "./base/5/1824979", block 3:
 ↪  calculated 601A but expected 4D7
WARNING:  checksum verification failed in file "./base/5/1824979", block 35:
 ↪  calculated 22A2 but expected 4DC
WARNING:  checksum verification failed in file "./base/5/1824979", block 67:
 ↪  calculated 633C but expected 4E5
WARNING:  checksum verification failed in file "./base/5/1824979", block 99:
 ↪  calculated 4CB9 but expected 4E2
WARNING:  further checksum verification failures in file "./base/5/1824979" will not
 ↪  be reported
WARNING:  file "./base/5/1824979" has a total of 142 checksum verification failures
WARNING:  could not verify checksum in file "./base/5/1824645", block 0: read buffer
 ↪  size 2 and page size 8192 differ
NOTICE:  all required WAL segments have been archived
WARNING:  142 total checksum verification failures
pg_basebackup: erreur : erreur de somme de contrôle
```

`pg_basebackup` peut désactiver la vérification des sommes de contrôle avec l'option `--no-verify-checksums` afin d'obtenir une copie, aussi corrompue que l'original, mais pouvant servir de base de travail.

Une sauvegarde physique périodique est un bon moyen de vérifier les sommes de contrôle régulièrement, tout en permettant de parer aux soucis qui pourraient survenir. L'outil barman, selon sa configuration, peut utiliser `pg_basebackup`. pgBackRest vérifie aussi les sommes de contrôle lors de ses sauvegardes.

16.5.7 Outils : amcheck & pg_amcheck

Pour les index :

- `amcheck` : vérification uniquement
  - fonction pour l'intégrité des tables
  - vérification de la cohérence index/table (probabiliste)
  - et de l'unicité
- `pg_amcheck` (v14+)
  - version ligne de commande
  - en masse
- Si problème : `REINDEX`

amcheck :

Plus finement, `amcheck` permet de vérifier la cohérence des tables et index et de leur structure interne, et ainsi détecter des corruptions. Il définit plusieurs fonctions :

- `verify_heapam` (PostgreSQL 14 et suivants) vérifie que chaque bloc est bien lisible, correctement construit ; cette fonction peut couvrir les TOAST et peut se limiter à certains blocs ;
- `bt_index_check` est destinée aux vérifications de routine des index, et ne pose qu'un verrou *AccessShareLock* peu gênant ;
- `bt_index_parent_check` est plus minutieuse, mais son exécution gêne les modifications dans la table (verrou *ShareLock* sur la table et l'index) et elle ne peut pas être exécutée sur un serveur secondaire.

Avec le paramètre `heapallindex` à `true`, les deux dernières fonctions effectuent une vérification supplémentaire en recréant temporairement une structure d'index et en la comparant avec l'index original. `bt_index_check` vérifie alors que chaque entrée de la table possède une entrée dans l'index. `bt_index_parent_check` vérifie en plus qu'à chaque entrée de l'index correspond une entrée dans la table.

Les verrous posés par les fonctions ne changent pas. Néanmoins, l'utilisation de ce mode a un impact sur la durée d'exécution des vérifications. Pour limiter l'impact, l'opération n'a lieu qu'en mémoire, et dans la limite du paramètre `maintenance_work_mem` (soit entre 256 Mo et 1 Go, parfois plus, sur les serveurs récents). C'est cette restriction mémoire qui implique que la détection de problèmes est probabiliste pour les plus grosses tables : selon la documentation, la probabilité de rater une incohérence est de 2 % si l'on peut consacrer 2 octets de mémoire à chaque ligne. Mais rien n'empêche de relancer les vérifications régulièrement, diminuant ainsi les chances de rater une erreur.

À partir de PostgreSQL 17, le paramètre `checkunique` à `true` permet de détecter des violations de contrainte d'unicité.

`amcheck` ne fournit aucun moyen de corriger une erreur, puisqu'il détecte des choses qui ne devraient jamais arriver. `REINDEX` sera souvent la solution la plus sûre, simple et facile, mais tout dépend de la cause du problème.

Exemple :

Soit `unetable_pkey`, un index de 10 Go sur un entier :

```
CREATE EXTENSION amcheck ;

SELECT bt_index_check('unetable_pkey');
Durée : 63753,257 ms (01:03,753)

SELECT bt_index_check('unetable_pkey', true);
Durée : 234200,678 ms (03:54,201)
```

Ici, la vérification exhaustive multiplie le temps de vérification par un facteur 4.

Voir la documentation d'amcheck[8] pour tous les détails et les options disponibles.

pg_amcheck :

À partir de la version 14, PostgreSQL dispose d'un nouvel outil en ligne de commande appelé `pg_amcheck`, basé sur `amcheck`.

Ses options de ligne de commande reprennent celles de l'extension, tout en permettant de lancer les vérifications en masse (choix des bases, des tables, parallélisation).

Voir sa documentation[9].

Exemple de recherche de corruption d'unicité :

Cet exemple, dérivé d'un autre de la liste pgsql-hackers, modifie les tables systèmes pour provoquer la corrution d'un index unique (à ne jamais faire en production, bien sûr !) :

```
CREATE TABLE junk (t text UNIQUE);
-- données
INSERT INTO junk (t) VALUES ('ba'), ('be'), ('bo'), ('by');
UPDATE pg_catalog.pg_index
   SET indisunique = false
 WHERE indrelid = 'junk'::regclass;
-- nouvelles données avec doublons
INSERT INTO junk (t) VALUES ('ba'), ('bi'), ('bu'), ('by');
-- remise en place de l'unicité
UPDATE pg_catalog.pg_index
   SET indisunique = true
 WHERE indrelid = 'junk'::regclass;
SELECT * FROM junk ORDER BY 1;
```

Tout semble fonctionner ensuite, si ce n'est que que l'unicité des données déjà présentes n'est pas là.

- Détection avec l'extension `amcheck` :

[8] https://docs.postgresql.fr/current/amcheck.html
[9] https://docs.postgresql.fr/current/app-pgamcheck.html

```
CREATE EXTENSION amcheck ;
SELECT bt_index_check('junk_t_key', false, true);
```

```
ERROR:  index uniqueness is violated for index "junk_t_key"
DÉTAIL : Index tid=(1,1) and tid=(1,2) (point to heap tid=(0,1) and tid=(0,5)) page
 ↪  lsn=65/EB31C810.
```

- Détection avec `pg_amcheck` (l'extension `amcheck` doit être installée) :

```
pg_amcheck --database=postgres --relation=junk --checkunique
```

```
vérification de l'index btree« postgres public.junk_t_key » :
    ERROR:  index uniqueness is violated for index "junk_t_key"
    DÉTAIL : Index tid=(1,1) and tid=(1,2) (point to heap tid=(0,1) and tid=(0,5))
 ↪  page lsn=65/EB31C810.
```

Les données responsables peuvent être trouvées dans la table (*heap*) grâce aux `tid` indiqués :

```
SELECT * FROM junk WHERE ctid='(0,1)' or ctid='(0,5)';
```

```
 t
----
 ba
 ba
```

Malheureusement, l'outil ne fournit que la première erreur rencontrée. Il faudra faire une requête pour les trouver :

```
SELECT t, count(*) FROM junk
GROUP BY 1  HAVING count(*) > 1  ORDER BY t ;
```

Sachant que cette requête risque de se baser sur l'index problématique, il faudra se méfier, ou plutôt agréger avec `t||''`.

16.6 CAS TYPE DE DÉSASTRES

16.7 CAS TYPE DE DÉSASTRES

- Les cas suivants sont assez rares
- Ils nécessitent généralement une restauration
- Certaines manipulations à haut risque sont possibles
  - mais complètement déconseillées !
- Généralement, il faudra restaurer une sauvegarde

Cette section décrit quelques-unes des pires situations de corruptions que l'on peut être amené à observer.

Dans la quasi-totalité des cas, la seule bonne réponse est la restauration de l'instance à partir d'une sauvegarde fiable.

16.7.1 Avertissement

- Privilégier une solution fiable (restauration, bascule)
- Les actions listées ici sont parfois destructrices
- La plupart peuvent (et vont) provoquer des incohérences
- Travailler sur une copie

La plupart des manipulations mentionnées dans cette partie sont destructives, et peuvent (et vont) provoquer des incohérences dans les données.

Tous les experts s'accordent pour dire que l'utilisation de telles méthodes pour récupérer une instance tend à aggraver le problème existant ou à en provoquer de nouveaux, plus graves. S'il est possible de l'éviter, ne pas les tenter (*ie* : préférer la restauration d'une sauvegarde) !

S'il n'est pas possible de faire autrement (ie : pas de sauvegarde utilisable, données vitales à extraire…), alors TRAVAILLER SUR UNE COPIE.

Il ne faut pas non plus oublier que chaque situation est unique, il faut prendre le temps de bien cerner l'origine du problème, documenter chaque action prise, s'assurer qu'un retour arrière est toujours possible.

16.7.2 Corruption de blocs dans des index

- Messages d'erreur lors des accès par l'index
- Requêtes incohérentes
- Données différentes entre un indexscan et un seqscan
- Supprimer et recréer l'index : `REINDEX`
- Est-ce juste cet index ?

Les index sont des objets de structure complexe, ils sont donc particulièrement vulnérables aux corruptions.

Lorsqu'un index est corrompu, on aura généralement des messages d'erreur de ce type :

```
ERROR: invalid page header in block 5869177 of relation base/17291/17420
```

Il peut arriver qu'un bloc corrompu ne renvoie pas de message d'erreur à l'accès, mais que les données elles-mêmes soient altérées, ou que des filtres ne renvoient pas les données attendues. Ce cas est néanmoins très rare dans un bloc d'index.

Dans la plupart des cas, si les données de la table sous-jacente ne sont pas affectées, il est possible de réparer l'index en le reconstruisant intégralement grâce à la commande `REINDEX`. Une corruption d'index se corrige donc facilement, mais le plus inquiétant est la présence même d'une corruption. Le contrôle complet de la base est fortement conseillé.

16.7.3 Corruption de blocs dans des tables 1

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

- Cas plus problématique
- Restauration probablement nécessaire

Les corruptions de blocs vont généralement déclencher des erreurs du type suivant :

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

Si la relation concernée est une table, tout ou partie des données contenues dans ces blocs est perdu.

L'apparition de ce type d'erreur est un signal fort qu'une restauration est certainement nécessaire.

16.7.4 Corruption de blocs dans des tables 2

```
SET zero_damaged_pages = true ;
VACUUM FULL tablecorrompue ;
```

- Des données vont certainement être perdues !

S'il est nécessaire de lire le maximum de données possibles de la table, il est possible d'utiliser l'option de PostgreSQL `zero_damaged_pages` pour demander au moteur de réinitialiser les blocs invalides à zéro lorsqu'ils sont lus au lieu de tomber en erreur. Il s'agit d'un des très rares paramètres absents de `postgresql.conf`.

Par exemple :

```
SET zero_damaged_pages = true ;
VACUUM FULL tablecorrompue ;

WARNING:  invalid page header in block 32570 of relation base/16390/2663; zeroing
 ↪  out page
```

Si l'opération continue et se termine sans erreur, les blocs invalides ont été réinitialisés.

Les données qu'ils contenaient sont évidemment perdues, mais la table peut désormais être accédée dans son intégralité en lecture, permettant ainsi par exemple de réaliser un export des données pour récupérer ce qui peut l'être.

Attention, du fait des données perdues, le résultat peut être incohérent (contraintes non respectées…).

Par ailleurs, sans les *checksums*, PostgreSQL ne détecte pas les corruptions logiques, c'est-à-dire n'affectant pas la structure des données mais corrompant le contenu. Il ne faut donc pas penser que la procédure d'export complet de données suivie d'un import sans erreur garantit l'absence de corruption.

16.7.5 Corruption de blocs dans des tables 3

- Si la corruption est importante, l'accès au bloc peut faire crasher l'instance
- Il est tout de même possible de réinitialiser le bloc
  - identifier le fichier à l'aide de `pg_relation_filepath()`
  - trouver le bloc avec `ctid` / `pageinspect`
  - réinitialiser le bloc avec `dd`
  - il faut vraiment ne pas avoir d'autre choix

Dans certains cas, il arrive que la corruption soit suffisamment importante pour que le simple accès au bloc fasse crasher l'instance. Dans ce cas, le seul moyen de réinitialiser le bloc est de le faire manuellement au niveau du fichier, **instance arrêtée**, par exemple avec la commande `dd`.

Pour identifier le fichier associé à la table corrompue, utiliser la fonction `pg_relation_filepath()` :

```
SELECT pg_relation_filepath('test_corruptindex') ;
```

```
 pg_relation_filepath
----------------------
 base/16390/40995
```

Le résultat donne le chemin vers le fichier principal de la table, relatif au `PGDATA` de l'instance.

Attention, une table peut contenir plusieurs fichiers. Par défaut une instance PostgreSQL sépare les fichiers en segments de 1 Go. Une table dépassant cette taille aura donc des fichiers supplémentaires (`base/16390/40995.1`, `base/16390/40995.2` ...).

Pour trouver le fichier contenant le bloc corrompu, il faudra donc prendre en compte le numéro du bloc trouvé dans le champ `ctid`, multiplier ce numéro par la taille du bloc (paramètre `block_size`, 8 ko par défaut), et diviser le tout par la taille du segment.

Cette manipulation est évidemment extrêmement risquée, la moindre erreur pouvant rendre irrécupérables de grandes portions de données. Il est donc fortement déconseillé de se lancer dans ce genre de manipulations à moins d'être absolument certain que c'est indispensable.

Encore une fois, ne pas oublier de travailler sur une copie, et pas directement sur l'instance de production.

16.7.6 Corruption des WAL 1

- Situés dans le répertoire `pg_wal`
- Les WAL sont nécessaires au *recovery*
- Démarrage impossible s'ils sont corrompus ou manquants
- Si les fichiers WAL ont été archivés, les récupérer
- Sinon, la restauration est la seule solution viable

Les fichiers WAL sont les journaux de transactions de PostgreSQL. Leur fonction est d'assurer que les transactions qui ont été effectuées depuis le dernier checkpoint ne seront pas perdues en cas de crash de l'instance.

Si certains sont corrompus ou manquants, alors PostgreSQL ne pourra pas redémarrer.

Il ne faut **jamais** supprimer les fichiers WAL, même si le système de fichiers est plein ! (Dans ce dernier cas, chercher la source : il s'agit souvent d'un archivage qui traîne ou d'un slot de réplication bloqué.)

Si l'archivage était activé et que les fichiers WAL perdus ont été archivés, alors il est possible de les restaurer avant de tenter un nouveau démarrage (copie manuelle, utilisation de `restore_command`...).

Si ce n'est pas possible, ou si les archives ont également été corrompues ou supprimées, l'instance ne pourra pas redémarrer.

Dans cette situation, comme dans la plupart des autres évoquées ici, la seule solution permettant de s'assurer que les données ne seront pas corrompues est de procéder à une restauration de l'instance depuis la dernière sauvegarde complète disponible.

16.7.7 Corruption des WAL 2

- `pg_resetwal` permet de forcer le démarrage
- ATTENTION !!!
  - cela va provoquer des pertes de données
  - des corruptions de données sont également probables
  - ce n'est pas une action corrective !

L'utilitaire `pg_resetwal` a comme fonction principale de supprimer les fichiers WAL courants et d'en créer un nouveau, avant de mettre à jour le fichier de contrôle pour permettre le redémarrage.

Au minimum, cette action va provoquer la perte de toutes les transactions validées effectuées depuis le dernier checkpoint. Il est également probable que des incohérences vont apparaître, certaines relativement simples à détecter via un export/import (incohérences dans les clés étrangères par exemple), certaines complètement invisibles.

L'utilisation de cet utilitaire est extrêmement dangereuse, n'est pas recommandée, et ne peut jamais être considérée comme une action corrective. Il faut toujours privilégier la restauration d'une sauvegarde plutôt que son exécution.

Si l'utilisation de `pg_resetwal` est néanmoins nécessaire (par exemple pour récupérer des données absentes de la sauvegarde), alors il faut travailler sur une copie des fichiers de l'instance, récupérer ce qui peut l'être à l'aide d'un export de données, et les importer dans une autre instance.

Les données récupérées de cette manière devraient également être soigneusement validées avant d'être importée de façon à s'assurer qu'il n'y a pas de corruption silencieuse.

Il ne faut en aucun cas remettre une instance en production après une réinitialisation des WAL.

16.7.8 Corruption du fichier de contrôle

- Fichier `global/pg_control`
- Contient les informations liées au dernier checkpoint
- Sans lui, l'instance ne peut pas démarrer
- Recréation avec `pg_resetwal`… parfois
- Restauration nécessaire

Le fichier de contrôle de l'instance contient de nombreuses informations liées à l'activité et au statut de l'instance, notamment l'instant du dernier checkpoint, la position correspondante dans les WAL, le numéro de transaction courant et le prochain à venir…

Ce fichier est le premier lu par l'instance. S'il est corrompu ou supprimé, l'instance ne pourra pas démarrer.

Il est possible de forcer la réinitialisation de ce fichier à l'aide de la commande `pg_resetwal`, qui va se baser par défaut sur les informations contenues dans les fichiers WAL présents pour tenter de « deviner » le contenu du fichier de contrôle.

Ces informations seront très certainement erronées, potentiellement à tel point que même l'accès aux bases de données par leur nom ne sera pas possible :

```
$ pg_isready
/var/run/postgresql:5432 - accepting connections

$ psql postgres
psql: FATAL:  database "postgres" does not exist
```

Encore une fois, utiliser `pg_resetwal` n'est en aucun cas une solution, mais doit uniquement être considéré comme un contournement temporaire à une situation désastreuse.

Une instance altérée par cet outil ne doit pas être considérée comme saine.

16.7.9 Corruption du CLOG

- Fichiers dans `pg_xact`
- Statut des différentes transactions
- Son altération risque de causer des incohérences majeures
- Préférer la restauration

Le fichier CLOG (*Commit Log*) dans `PGDATA/pg_xact/` contient le statut des différentes transactions, notamment si celles-ci sont en cours, validées ou annulées.

S'il est altéré ou supprimé, il est possible que des transactions qui avaient été marquées comme annulées soient désormais considérées comme valides, et donc que les modifications de données correspondantes deviennent visibles aux autres transactions.

C'est évidemment un problème d'incohérence majeur, tout problème avec ce fichier devrait donc être soigneusement analysé.

Il est préférable dans le doute de procéder à une restauration et d'accepter une perte de données plutôt que de risquer de maintenir des données incohérentes dans la base.

16.7.10 Corruption du catalogue système

- Le catalogue contient la définition du schéma
- Sans lui, les données sont inaccessibles
- Situation très délicate…

Le catalogue système contient la définition de toutes les relations, les méthodes d'accès, la correspondance entre un objet et un fichier sur disque, les types de données existantes… S'il est incomplet, corrompu ou inaccessible, l'accès aux données en SQL risque de ne pas être possible du tout.

Cette situation est très délicate, et appelle là encore une restauration.

Si le catalogue était complètement inaccessible, sans sauvegarde la seule solution restante serait de tenter d'extraire les données directement des fichiers data de l'instance, en oubliant toute notion de cohérence, de type de données, de relation… Personne ne veut faire ça.

16.8 CONCLUSION

- Les désastres peuvent arriver
- Il faut s'y être préparé
- Faites des sauvegardes !
  - et testez-les

16.9 QUIZ

https://dali.bo/i5_quiz

16.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/i5_solutions.

16.10.1 Corruption d'un bloc de données

But : Corrompre un bloc et voir certains impacts possibles.

Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

Créer une base **pgbench** et la remplir avec l'outil de même, avec un facteur d'échelle 10 et **avec les clés étrangères entre tables** ainsi :

```
/usr/pgsql-17/bin/pgbench -i -s 10 -d pgbench --foreign-keys
```

Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

Arrêter PostgreSQL.

Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

Tenter un `pg_dumpall > /dev/null`.

- Arrêter PostgreSQL.
- Voir ce que donne `pg_checksums` (`pg_verify_checksums` en v11).

- Faire une copie de travail à froid du PGDATA.
- Protéger en écriture le PGDATA original.
- Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause).

Démarrer le cluster sur la copie avec `pg_ctl`.

Que renvoie ceci ?

```
SELECT * FROM pgbench_accounts LIMIT 100 ;
```

Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

16.10.2 Corruption d'un bloc de données et incohérences

But : Corrompre une table portant une clé étrangère.

Nous continuons sur la copie de la base de travail, où les sommes de contrôle ont été désactivées.

Consulter le format et le contenu de la table `pgbench_branches`.

Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

Pour corrompre la table :

- Arrêter PostgreSQL.
- Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
- En toute fin de fichier, remplacer le dernier `01` par un `FF`.
- Redémarrer PostgreSQL.

- Compter le nombre de lignes dans `pgbench_branches`.
- Recompter après `SET enable_seqscan TO off ;`.
- Quelle est la bonne réponse ? Vérifier le contenu de la table.

> Qu'affiche `pageinspect` pour cette table ?

> Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

> Pour voir ce que donnerait une restauration :
>
> - Exporter `pgbench_accounts`, définition des index comprise.
> - Supprimer la table (il faudra supprimer `pgbench_history` aussi).
> - Tenter de la réimporter.

16.11 TRAVAUX PRATIQUES (SOLUTIONS)

16.11.1 Corruption d'un bloc de données

> Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

```
# SHOW data_checksums ;

 data_checksums
----------------
 on
```

Si la réponse est `off`, on peut (à partir de la v12) mettre les checksums en place :

```
$ /usr/pgsql-17/bin/pg_checksums -D /var/lib/pgsql/15/data.BACKUP/  --enable
↪  --progress
58/58 MB (100%) computed
Checksum operation completed
Files scanned:  964
Blocks scanned: 7524
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster
```

> Créer une base **pgbench** et la remplir avec l'outil de même, avec un facteur d'échelle 10 et **avec les clés étrangères entre tables** ainsi :
>
> `/usr/pgsql-17/bin/pgbench -i -s 10 -d pgbench --foreign-keys`

```
$ dropdb --if-exists pgbench ;
$ createdb pgbench ;

$ /usr/pgsql-17/bin/pgbench -i -s 10 -d pgbench --foreign-keys
...
creating tables...
generating data...
100000 of 1000000 tuples (10%) done (elapsed 0.15 s, remaining 1.31 s)
200000 of 1000000 tuples (20%) done (elapsed 0.35 s, remaining 1.39 s)
..
1000000 of 1000000 tuples (100%) done (elapsed 2.16 s, remaining 0.00 s)
vacuuming...
creating primary keys...
creating foreign keys...
done.
```

> Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

La table fait 128 Mo selon un `\d+`.

La clé `aid` va de 1 à 100000 :

```
# SELECT min(aid), max(aid) FROM pgbench_accounts ;
```

```
 min |   max
-----+---------
   1 | 1000000
```

Un `SELECT` montre que les valeurs sont triées mais c'est dû à l'initialisation.

> Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

```
 SELECT pg_relation_filepath('pgbench_accounts') ;
```

```
 pg_relation_filepath
----------------------
 base/16454/16489
```

> Arrêter PostgreSQL.

```
 # systemctl stop postgresql-15
```

Cela permet d'être sûr qu'il ne va pas écraser nos modifications lors d'un checkpoint.

> Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

```
 # dnf install hexedit
```

```
postgres$ hexedit /var/lib/pgsql/15/data/base/16454/16489
```

Aller par exemple sur la 2è ligne, modifier `80 9F` en `FF FF`. Sortir avec Ctrl-X, confirmer la sauvegarde.

> Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

```
 # systemctl start postgresql-15
```

```
# SELECT * FROM pgbench_accounts ;
```

```
WARNING:  page verification failed, calculated checksum 62947 but expected 57715
ERROR:  invalid page in block 0 of relation base/16454/16489
```

> Tenter un `pg_dumpall > /dev/null`.

```
$ pg_dumpall > /dev/null
pg_dump: WARNING:  page verification failed, calculated checksum 62947 but expected
 ↪  57715
pg_dump: error: Dumping the contents of table "pgbench_accounts" failed:
 ↪  PQgetResult() failed.
pg_dump: error: Error message from server:
         ERROR:  invalid page in block 0 of relation base/16454/16489
pg_dump: error: The command was:
         COPY public.pgbench_accounts (aid, bid, abalance, filler) TO stdout;
pg_dumpall: error: pg_dump failed on database "pgbench", exiting
```

> - Arrêter PostgreSQL.
> - Voir ce que donne `pg_checksums` (`pg_verify_checksums` en v11).

```
# systemctl stop postgresql-15
```

```
$ /usr/pgsql-17/bin/pg_checksums -D /var/lib/pgsql/15/data/  --check --progress
pg_checksums: error: checksum verification failed in file
"/var/lib/pgsql/15/data//base/16454/16489", block 0:
                                    calculated checksum F5E3 but block contains E173
216/216 MB (100%) computed
Checksum operation completed
Files scanned:  1280
Blocks scanned: 27699
Bad checksums:  1
Data checksum version: 1
```

> - Faire une copie de travail à froid du PGDATA.
> - Protéger en écriture le PGDATA original.
> - Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

Dans l'idéal, la copie devrait se faire vers un autre support, une corruption rend celui-ci suspect. Dans le cadre du TP, ceci suffira :

```
$ cp -upR /var/lib/pgsql/15/data/  /var/lib/pgsql/15/data.BACKUP/
$ chmod -R -w /var/lib/pgsql/15/data/
```

Dans `/var/lib/pgsql/15/data.BACKUP/pg_hba.conf` ne doit plus subsister que :

```
local   all             all                                     trust
```

> Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause).

```
$ /usr/pgsql-17/bin/pg_checksums -D /var/lib/pgsql/15/data.BACKUP/  --disable
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums disabled in cluster
```

> Démarrer le cluster sur la copie avec `pg_ctl`.

```
/usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/15/data.BACKUP/ start
```

> Que renvoie ceci ?
>
> ```
> SELECT * FROM pgbench_accounts LIMIT 100 ;
> ```

```
# SELECT *  FROM pgbench_accounts LIMIT 10;
```

```
ERROR:  out of memory
DÉTAIL : Failed on request of size 536888061 in memory context "printtup".
```

Ce ne sera pas forcément cette erreur, plus rien n'est sûr en cas de corruption. L'avantage des sommes de contrôle est justement d'avoir une erreur moins grave et plus ciblée.

Un `pg_dumpall` renverra le même message.

> Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

```
pgbench=# SET zero_damaged_pages TO on ;
SET
pgbench=# VACUUM FULL pgbench_accounts ;
VACUUM

pgbench=# SELECT *  FROM pgbench_accounts LIMIT 100;

 aid | bid | abalance |                                        filler
-----+-----+----------+----------------------------------------------------------------
 ↪  -------------------------
   2 |   1 |        0 |
   3 |   1 |        0 |
   4 |   1 |        0 |
[...]

pgbench=# SELECT min(aid), max(aid), count(aid) FROM pgbench_accounts ;

 min |   max   | count
-----+---------+--------
   2 | 1000000 | 999999
```

Apparemment une ligne a disparu, celle portant la valeur 1 pour la clé. Il est rare que la perte soit aussi évidente !

16.11.2 Corruption d'un bloc de données et incohérences

> Consulter le format et le contenu de la table `pgbench_branches`.

Cette petite table ne contient que 10 valeurs :

```
# SELECT * FROM pgbench_branches ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
   2 |        0 |
   3 |        0 |
   4 |        0 |
   5 |        0 |
   6 |        0 |
   7 |        0 |
   8 |        0 |
   9 |        0 |
  10 |        0 |
(10 lignes)
```

> Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

```
# SELECT pg_relation_filepath('pgbench_branches') ;

 pg_relation_filepath
----------------------
 base/16454/16490
```

> Pour corrompre la table :
>
> - Arrêter PostgreSQL.
> - Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
> - En toute fin de fichier, remplacer le dernier `01` par un `FF`.
> - Redémarrer PostgreSQL.

```
$ /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/15/data.BACKUP/ stop

$ hexedit /var/lib/pgsql/15/data.BACKUP/base/16454/16490

$ /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/15/data.BACKUP/ start
```

> - Compter le nombre de lignes dans `pgbench_branches`.
> - Recompter après `SET enable_seqscan TO off ;`.
> - Quelle est la bonne réponse ? Vérifier le contenu de la table.

Les deux décomptes sont contradictoires :

```
pgbench=# SELECT count(*) FROM pgbench_branches ;
 count
-------
     9

pgbench=# SET enable_seqscan TO off ;
SET

pgbench=# SELECT count(*) FROM pgbench_branches ;
 count
-------
    10
```

En effet, le premier lit la (petite) table directement, le second passe par l'index, comme un EXPLAIN le montrerait. Les deux objets diffèrent.

Et le contenu de la table est devenu :

```
# SELECT * FROM pgbench_branches ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
   2 |        0 |
```

```
   3 |         0 |
   4 |         0 |
   5 |         0 |
   6 |         0 |
   7 |         0 |
   8 |         0 |
   9 |         0 |
(9 lignes)
```

Le 1 est devenu 255 (c'est notre première modification) mais la ligne 10 a disparu !

Les requêtes peuvent renvoyer un résultat incohérent avec leur critère :

```
pgbench=# SET enable_seqscan TO off;
SET
pgbench=# SELECT * FROM pgbench_branches
          WHERE    bid = 1 ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
```

> Qu'affiche `pageinspect` pour cette table ?

```
pgbench=# CREATE EXTENSION pageinspect ;

pgbench=# SELECT t_ctid, lp_off, lp_len, t_xmin, t_xmax, t_data
          FROM heap_page_items(get_raw_page('pgbench_branches',0));

 t_ctid | lp_off | lp_len | t_xmin | t_xmax |       t_data
--------+--------+--------+--------+--------+--------------------
 (0,1)  |   8160 |     32 |  63726 |      0 | \xff00000000000000
 (0,2)  |   8128 |     32 |  63726 |      0 | \x0200000000000000
 (0,3)  |   8096 |     32 |  63726 |      0 | \x0300000000000000
 (0,4)  |   8064 |     32 |  63726 |      0 | \x0400000000000000
 (0,5)  |   8032 |     32 |  63726 |      0 | \x0500000000000000
 (0,6)  |   8000 |     32 |  63726 |      0 | \x0600000000000000
 (0,7)  |   7968 |     32 |  63726 |      0 | \x0700000000000000
 (0,8)  |   7936 |     32 |  63726 |      0 | \x0800000000000000
 (0,9)  |   7904 |     32 |  63726 |      0 | \x0900000000000000
        |  32767 |    127 |        |        |
(10 lignes)
```

La première ligne indique bien que le 1 est devenu un 255.

La dernière ligne porte sur la première modification, qui a détruit les informations sur le `ctid`. Celle-ci est à présent inaccessible.

> Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

La documentation est sur https://docs.postgresql.fr/current/amcheck.html.

Une vérification complète se fait ainsi :

```
pgbench=# CREATE EXTENSION amcheck ;

pgbench=# SELECT bt_index_check (index => 'pgbench_branches_pkey',
                                 heapallindexed => true);
 bt_index_check
----------------

(1 ligne)

pgbench=# SELECT bt_index_parent_check (index => 'pgbench_branches_pkey',
                                        heapallindexed => true, rootdescend => true);
ERROR:  heap tuple (0,1) from table "pgbench_branches"
        lacks matching index tuple within index "pgbench_branches_pkey"
```

Un seul des problèmes a été repéré.

Un `REINDEX` serait ici une mauvaise idée : c'est la table qui est corrompue ! Les sommes de contrôle, là encore, auraient permis de cibler le problème très tôt.

> Pour voir ce que donnerait une restauration :
>
> - Exporter `pgbench_accounts`, définition des index comprise.
> - Supprimer la table (il faudra supprimer `pgbench_history` aussi).
> - Tenter de la réimporter.

```
$ pg_dump -d pgbench  -t pgbench_accounts -f /tmp/pgbench_accounts.dmp

$ psql pgbench -c 'DROP TABLE pgbench_accounts CASCADE'
NOTICE:  drop cascades to constraint pgbench_history_aid_fkey on table
 ↪  pgbench_history
DROP TABLE

$ psql pgbench < /tmp/pgbench_accounts.dmp
SET
SET
SET
SET
SET
 set_config
------------

(1 ligne)

SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
COPY 999999
ALTER TABLE
CREATE INDEX
ERROR:  insert or update on table "pgbench_accounts"
```

```
        violates foreign key constraint "pgbench_accounts_bid_fkey"
DÉTAIL : Key (bid)=(1) is not present in table "pgbench_branches".
```

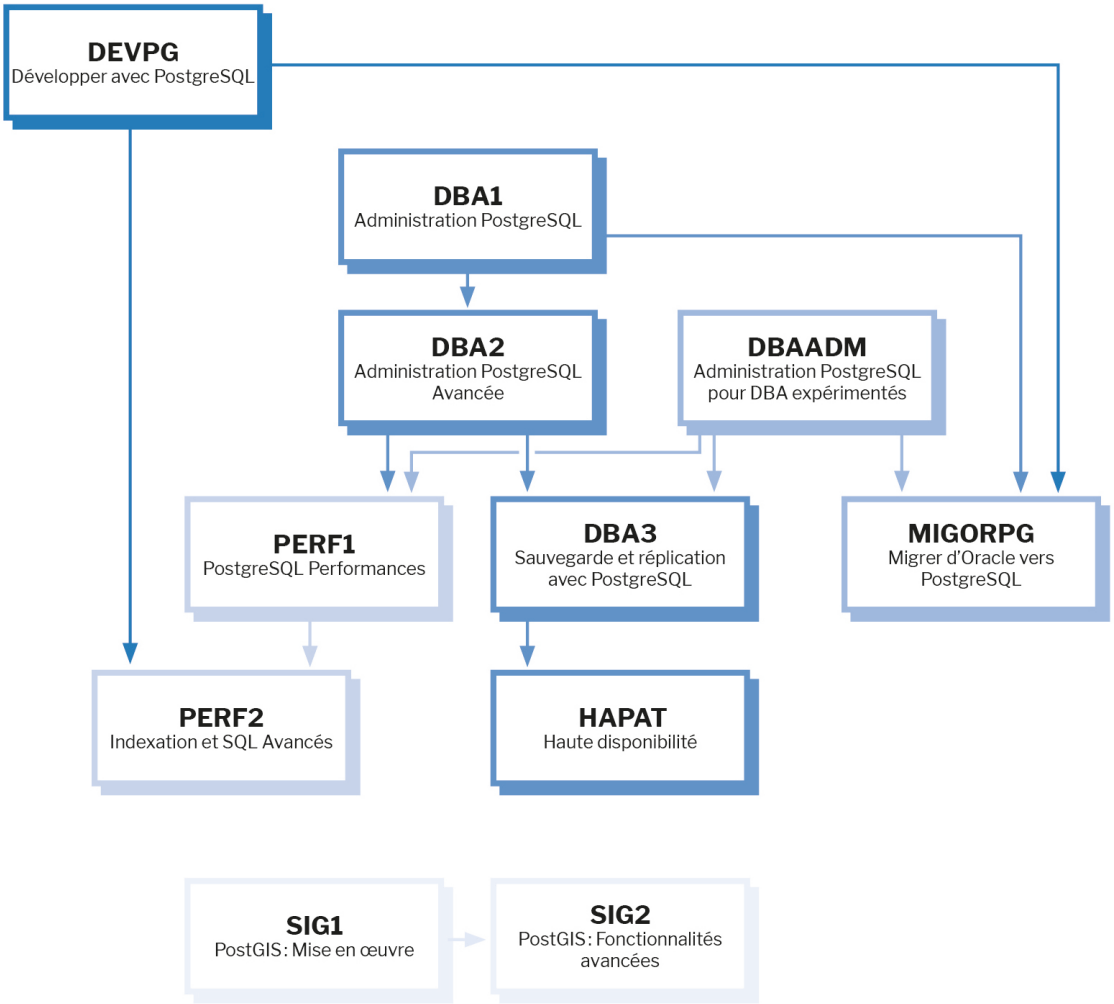
La contrainte de clé étrangère entre les deux tables ne peut être respectée : `bid` est à 1 sur de nombreuses lignes de `pgbench_accounts` mais n'existe plus dans la table `pgbench_branches` ! Ce genre d'incohérence doit être recherchée très tôt pour ne pas surgir bien plus tard, quand on doit restaurer pour d'autres raisons.

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.