Formation DBA2

PostgreSQL Avancé

17.12

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL Avancé

Formation DBA2

TITRE : PostgreSQL Avancé
SOUS-TITRE : Formation DBA2

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-01-2

Remerciements : Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, Sharon Bonan, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale: Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note: Ceci est un résumé de la licence.

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

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

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

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

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).

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.

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 à cette adresse: http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 RICHESSES DE L'ÉCOSYSTÈME POSTGRESQL

2.1 PRÉAMBULE

- Projet horizontal & décentralisé
- La « biodiversité » est une force
- Le meilleur SGBD du marché ?

La communauté est structurée de manière décentralisée et ouverte. Tout le monde peut participer, fournir des patchs, exprimer son opinion. Évidemment, tous les patchs ne seront pas forcément intégrés. Toutes les opinions n'ont pas forcément la même valeur. Le projet est basé sur la méritocratie. Chaque contributeur a un potentiel confiance dépendant de ses précédentes contributions.

Le fait qu'un grand nombre de personnes peut participer fait que des opinions diverses et variées sont prises en compte dans l'évolution de PostgreSQL. Cela ne peut être qu'une bonne chose, même si parfois les discussions sont longues et les décisions moins simples

à prendre.

2.1.1 AU MENU

- Projets satellites
- Comparatifs
- Communauté
- Avenir

Après un tour d'horizon des logiciels gravitant autour de PostgreSQL et un comparatif avec les autres SGBD dominants, nous verrons le fonctionnement de la communauté et l'avenir du projet.

2.1.2 OBJECTIFS

- Connaître les logiciels connexes
- Exploiter toute la puissance du SGBD
- Participer !

Ce module est destiné aux utilisateurs qui souhaitent repousser les limites d'une utilisation classique.

À l'issue de ce chapitre, vous aurez une vision claire des projets complémentaires qui vous simplifieront la gestion quotidienne de vos bases. Vous connaîtrez les différences entre PostgreSQL et ses concurrents. Enfin, vous serez en mesure de contribuer au projet, si le cœur vous en dit !

2.2 LES PROJETS SATELLITES

- Administration
- Supervision et monitoring
- Migrations
- SIG
- Modélisation

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.

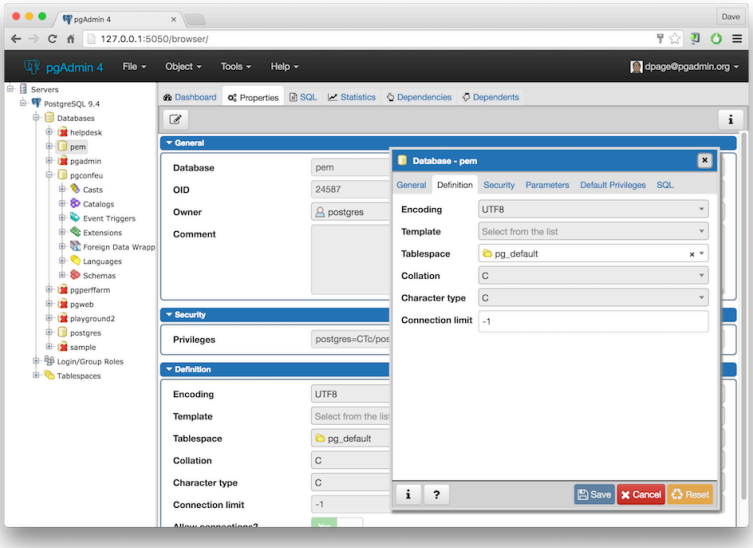
Du fait de ce manque, certaines personnes ont décidé de développer ces outils graphiques. Ceci a abouti à une grande richesse grâce à la grande variété de projets « satellites » qui gravitent autour du projet principal.

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

2.2.1 PGADMIN IV

- Site officiel : http://www.pgadmin.org/
- Version : 2.0
- Licence : PostgreSQL
- Gestion graphique de l'administration des bases
- Éditeur de requêtes
- Supervision

Logiciel libre d'administration de la base de données PostgreSQL, pgAdmin comprend une interface graphique d'administration, un outil de requêtage SQL, un éditeur de code procédural, un débogueur PL/psql, un éditeur des fichiers de configuration, une fenêtre de statut du serveur et bien plus encore.



pgAdmin est conçu pour répondre à la plupart des besoins, depuis l'écriture de simples requêtes SQL jusqu'au développement de bases de données complexes. L'interface graphique supporte les fonctionnalités de PostgreSQL les plus récentes et facilite l'administration.

Il supporte toutes les versions maintenues de PostgreSQL (il peut supporter des versions qui ne sont plus maintenues, mais ceci dépend fortement de la version de pgAdmin et du bon vouloir des développeurs), ainsi que les versions commerciales de PostgreSQL comme EnterpriseDB et Greenplum.

Il est disponible dans plusieurs langues et est utilisable sur différents systèmes d'exploitation.

Ce logiciel existe actuellement en deux versions :

- pgAdmin III ;
- pgAdmin IV.

Les développeurs de pgAdmin III ont abandonné cette version pour plusieurs raisons :

- le socle technique sur lequel elle reposait n'avançait pas avec son temps ;
- il y avait peu de développeurs connaissant ce socle et prêts à travailler dessus.

Dave Page, leader du projet, a donc décidé de ré-écrire ce projet dans un langage plus connu, plus apprécié, et où il serait possible de trouver plus de développeurs. Il a fini

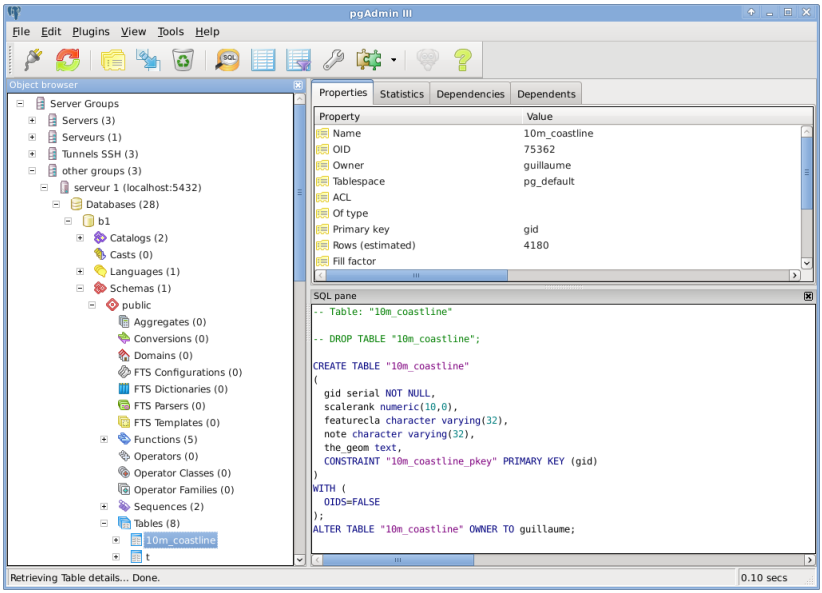
par sélectionner le langage Python et a réécrit, avec son équipe, une grande partie de pgAdmin III sous le nom de pgAdmin IV.

Les versions 1.x de pgAdmin IV ont beaucoup été décriées, par leur manque de fonctionnalités et par leur lenteur. Un grand nombre de ces plaintes ont été entendues et des corrections effectuées. Le résultat est une version 2.0 qui semble tenir ses promesses jusqu'à maintenant.

2.2.2 PGADMIN III

- Site officieux : https://www.openscg.com/bigsql/pgadmin3/
- Version : 1.23
- Licence : PostgreSQL
- Gestion graphique de l'administration des bases
- Éditeur de requêtes / `EXPLAIN` graphique
- Gestion de Slony

pgAdmin III est codé en C++ et utilise la bibliothèque wxWidgets pour être utilisable sur différents systèmes d'exploitation, et donc différents systèmes graphiques.



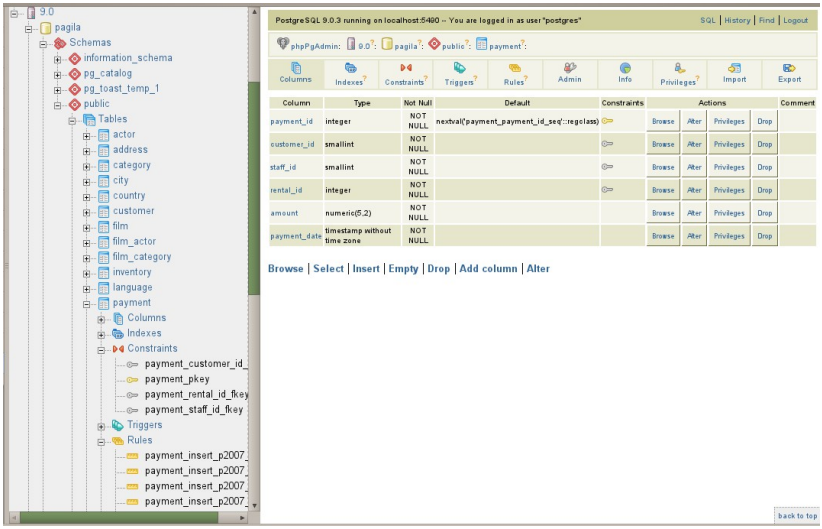
Cette version n'est plus maintenue par la communauté.

La société OpenSCG propose sa propre version, qu'elle appelle pgAdmin 3 LTS. D'après les commits visibles dans le dépôt officiel (https://bitbucket.org/openscg/pgadmin3-lts/commits/all), la seule amélioration est le support de la version 10. Par support, ils entendent que le test vérifiant le numéro de version accepte aussi la version 10. Il n'y a donc aucun support des nouvelles fonctionnalités de la version 10, aucune modification pour améliorer la stabilité, etc. Cette version de pgAdmin a vraisemblablement peu d'avenir devant elle.

2.2.3 PHPPGADMIN

- Site officiel : http://phppgadmin.sourceforge.net/
- Version : 5.1
- Licence : GPL
- Gestion graphique de l'administration des bases
- Interface web
- Mais ne semble plus maintenu
  - pas de nouvelles versions depuis avril 2013
  - pas de commit depuis avril 2016

- Utiliser plutôt pgAdmin IV sur un serveur web



PhpPgAdmin est une application web déportée, multilingue, simple d'emploi permettant la maintenance, la création, l'import-export de bases de données, la gestion des utilisateurs, ainsi que l'exécution de SQL et de scripts.

Cependant, il n'est plus maintenu. La dernière version, 5.1, date d'avril 2013, le dernier commit a été réalisé en avril 2016. Autrement dit, il y a un gros risque qu'il ne soit pas compatible avec PostgreSQL 10 et les versions suivantes, et il n'en supportera de toute façon pas les nouvelles fonctionnalités.

Notre conseil est d'utiliser plutôt pgAdmin IV sur un serveur web. La documentation de pgAdmin indique comment configurer un serveur web Apache.

2.2.4 PGBADGER

- Site officiel : http://projects.dalibo.org/pgbadger/
- Version : 9.2
- Licence : PostgreSQL
- Analyse des traces de durée d'exécution des requêtes
- Analyse des traces du `VACUUM`, des connexions, des checkpoints
- Compatible syslog, stderr, csvlog

pgBadger est un analyseur des journaux applicatifs de PostgreSQL. Il permet de créer

des rapports détaillés depuis ceux-ci. pgBadger est utilisé pour déterminer les requêtes à améliorer en priorité pour accélérer son application basée sur PostgreSQL.

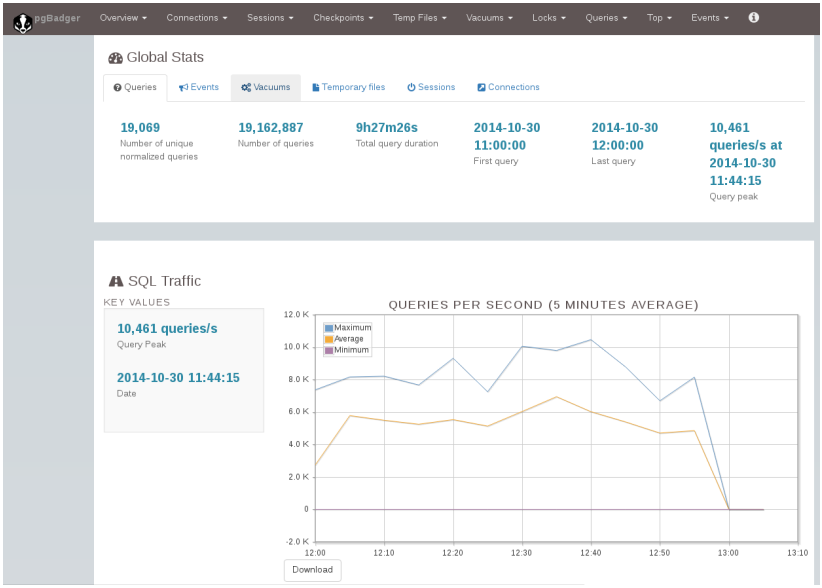


FIGURE 1: CAPTURE PGBADGER

pgBadger est écrit en Perl et est facilement extensible si vous avez besoin de rapports spécifiques.

pgBadger est conçu pour traiter de gros fichiers de logs avec une mémoire réduite.

Pour plus de détails : L'un des principaux développeurs de pgBadger est Gilles Darold, membre de l'équipe dalibo[1] . Le développement de l'outil se poursuit sur github[2] .

---

[1] http://www.dalibo.com
[2] https://github.com/dalibo/pgbadger/

2.2.5 OPM

- *Open PostgreSQL Monitoring*
- Site officiel : http://opm.io/
- Version : 2.4
- Licence : PostgreSQL
- Suite de supervision lancée par Dalibo en septembre 2014
- Projet indépendant mené par OPMDG (*OPM Development Group*)

Open PostgreSQL Monitoring est une suite de supervision lancée par Dalibo. L'objectif est d'avoir un outil permettant de surveiller un grand nombre d'instances PostgreSQL, et à terme de les administrer, de façon similaire à ce que permettent les outils *Oracle Enterprise Manager* ou *SQL Server Management Studio*.

Le projet est publié sous licence PostgreSQL, et un comité a été créé (OPMDG, *OPM Development Group*), à la manière du PGDG, de façon à assurer l'indépendance du projet.

Le cœur d'OPM est une base de données PostgreSQL stockant de nombreuses statistiques concernant les instances et le système d'exploitation des serveurs les hébergeant. Ces données sont ensuite utilisées par l'interface graphique pour les afficher sous forme de graphiques interactifs et personnalisables.

À ce jour, la collecte des statistiques nécessite la configuration de Nagios avec le script `check_pgactivity`, mais d'autres systèmes de collecte pourront être ajoutés à l'avenir.

2.2.6 POWA

- Site officiel : http://dalibo.github.io/powa/
- Version : 3.1
- Licence : PostgreSQL
- Surveillance de l'activité SQL
- Développé par Dalibo
- Dépôt GitHub
  - archiveur : https://github.com/dalibo/powa-archivist
  - UI web : https://github.com/dalibo/powa-web

PoWA (*PostgreSQL Workload Analyzer*) est un outil développé par Dalibo, 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.

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.

PoWA est compatible avec les versions 9.4 et supérieures de PostgreSQL. Pour utiliser PoWA avec PostgreSQL 9.3, il faut utiliser la version 1.2.

17.12

2.2.7 ORA2PG

- Site officiel : http://ora2pg.darold.net/
- Version : 18.2
- Licence : GPL
- Migration de la structure d'une base Oracle...
- ... des procédures stockées et des données

Ora2pg est un outil facilitant la migration d'une base Oracle vers une base PostgreSQL. Il s'occupe du schéma, des procédures stockées comme des données, suivant un fichier de configuration.

Pour plus de détails : Le développeur d'ora2pg est Gilles Darold, membre de l'équipe dalibo[3] . Le support s'effectue à travers github[4] .

2.2.8 SQLSERVER2PG

- Site officiel : http://dalibo.github.io/sqlserver2pgsql/
- Version : sans
- Licence : GPL v3
- Convertit un schéma SQL Server en un schéma PostgreSQL
- Produit en option un job Pentaho Data Integrator (Kettle) pour migrer toutes les données de SQL Server vers PostgreSQL

sqlserver2pgsql est un script Perl facilitant la migration d'une base SQL Server vers une base PostgreSQL.

2.2.9 DB2TOPG

- Site officiel : https://github.com/dalibo/db2topg
- Version : sans
- Licence : GPL v3
- Convertit une base DB2 en une base PostgreSQL

sqlserver2pgsql est un ensemble de scripts Perl facilitant la migration d'une base DB2 vers une base PostgreSQL.

[3] http://www.dalibo.com
[4] https://github.com/darold/ora2pg

Il faut partir d'une sauvegarde SQL de la base DB2. Le script db2topg le convertit en un schéma PostgreSQL. Vous pouvez demander en plus un script pour sauvegarder toutes les données de cette base DB2. Pour migrer les données, il suffit d'exécuter le script produit. Il récupère les fichiers CSV de DB2 qu'il faut ensuite importer dans PostgreSQL (le script deltocopy.pl aide à le faire.

2.2.10 POSTGIS

- Site officiel : http://postgis.net/
- Version : 2.4
- Licence : BSD
- Module spatial pour PostgreSQL
- Conforme aux spécifications de l'OpenGIS Consortium
- Compatible avec MapServer

PostGIS ajoute le support d'objets géographiques à PostgreSQL. En fait, PostGIS transforme un serveur 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 a été développé par la société Refractions Research comme une technologie Open-Source de base de données spatiale. Cette société continue à développer PostGIS, soutenue par une communauté active de contributeurs.

La version 2.0 apporte de nombreuses nouveautés attendues par les utilisateurs comme le support des fonctionnalités *raster* et les surfaces tri- dimensionnelles.

2.2.11 PGMODELER

- Site officiel : http://pgmodeler.com.br/
- Version : 0.9
- Licence : GPLv3
- Modélisation de base de données
- Fonctionnalité d'import export

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 open-source, son installation par les sources peut être laborieuse. L'équipe de développement propose des paquets binaires à prix modique.

2.3 COMPARATIFS

- Pas de SGBD universel
- S'inspirer des concurrents plutôt que de les combattre
- PostgreSQL vs.
  - MySQL
  - SQL Server

  - Oracle
  - Informix
  - NoSQL
- Attention aux benchmarks !

Il est toujours difficile de comparer différents SGBD, car ce sont des logiciels complexes et orientés différement.

Avant toute chose, il parait donc nécessaire de rappeler quelques points :

- **Il n'y a pas de SGBD universel !** Aucun logiciel de base de données ne peut couvrir parfaitement l'ensemble des besoins en matière de stockage d'information. On peut reprendre l'image suivante : si l'on pose la question « Quel est le véhicule le plus rapide ? une Ferrari ou un camion de 3,5 tonnes ? », la réponse dépend du contexte ! Si vous devez transporter une personne, alors la Ferrari est le meilleur choix. Si vous devez transporter une tonne de marchandises, alors vous opterez pour le camion !
- **Pour être réellement utiles, les benchmarks doivent être conduits en conditions réelles**. Si vous souhaitez tester différents SGBD, assurez- vous que les tests correspondent à votre cas d'utilisation et à votre matériel.
- **PostgreSQL s'inspire des projets concurrents.** Les autres SGBD sont vus comme des compétiteurs plutôt que comme des ennemis à abattre. La réciproque n'est pas toujours vraie !

2.3.1 POSTGRESQL VS. MYSQL

- Différents moteurs
  - MyISAM ou InnoDB ?
- Points forts de PostgreSQL
  - DDL transactionnel
- Points faibles de PostgreSQL
  - lenteur du `SELECT count(*)` (amélioration en 9.2)
- Quel avenir pour MySQL ?

Pendant des années, MySQL était perçu comme plus rapide et plus simple à utiliser que PostgreSQL. PostgreSQL était vu comme puissant, stable et respectueux des standards mais également lent et compliqué. Comme beaucoup de perceptions héritées du passé, cette vision des choses est complètement fausse. Les deux projets ont évolué et la comparaison est désormais plus complexe que cela.

17.12

L'un des principaux soucis de MySQL est son utilisation de différents moteurs qui activent/désactivent certaines fonctionnalités. La liste des fonctionnalités de MySQL est impressionnante mais il faut savoir que, pour bénéficier de certaines fonctionnalités, il faut en abandonner d'autres. Ce n'est pas le cas avec PostgreSQL où toutes les fonctionnalités sont disponibles en permanence, quelle que soit la configuration. Il est par exemple impossible d'utiliser l'indexation Full Text ou les extensions géographiques de MySQL sur autre chose que MyISAM, ce qui empêche l'utilisation de transactions.

Un des manques importants du moteur InnoDB est son incapacité à faire du transactionnel sur les requêtes de modification de schémas (DDL).

Le `SELECT count(*)` est généralement très coûteux avec PostgreSQL. MySQL, avec le moteur MyISAM, conserve sur ce point des performances que PostgreSQL ne peut pas concurrencer : MyISAM stocke un compteur d'enregistrements de la table en entête de celle-ci. Ceci n'est évidemment possible que pour un moteur non transactionnel, puisque deux transactions peuvent « voir » un nombre d'enregistrement différents dans une table suivant ce qui est validé ou non pour chacune d'entre elles. Néanmoins, cette requête est moins coûteuse depuis la version 9.2 de PostgreSQL.

Mais, malgré quelques atouts techniques, MySQL a actuellement un gros problème. Le rachat de MySQL AB par Sun puis par Oracle laisse de grosses incertitudes sur l'avenir de MySQL. Étant donné le comportement d'Oracle avec les outils libres, beaucoup de développeurs ont préféré démissionner. Beaucoup ont lancé leur propre fork (version dérivée) de MySQL (Drizzle, MariaDB, Falcon…). Ces forks sont généralement des versions comprenant moins de fonctionnalités que la version 5.5, ou une réimplémentation différente des nouveautés de la version 5.5.

2.3.2 POSTGRESQL VS. SQL SERVER

- Des performances difficile à battre sous Windows
  - … mais PostgreSQL reste plus efficace sous Linux
- Intégration à l'écosystème Microsoft
  - avantage, excellents outils graphiques (Studio)
  - inconvénient, peu de choix dans les outils
- Points faibles de PostgreSQL
  - un partitionnement perfectible (amélioration en 10)
  - pas de vues matérialisées (amélioration en 9.3)

Difficile de battre Microsoft pour fournir un serveur de bases de données plus rapide que SQL Server (créé par Microsoft) sur Windows (créé aussi par Microsoft). Du coup,

PostgreSQL a de fortes chances d'être moins performant que SQL Server sous Windows. Par contre, une comparaison de SQL Server sur Windows avec PostgreSQL sous Linux donne souvent un résultat inverse. En 2010, Red Hat a publié une étude de performances détaillées montrant la supériorité du couple « PostgreSQL + Linux » face à « SQL Server + Windows » : http://www.redhat.com/pdf/rhel/bmsql-postgres-sqlsrvr-v1.0-1.pdf

SQL Server et PostgreSQL s'opposent aussi sur la philosophie. Le premier propose de très nombreux outils périphériques (console, ETL...), et vous impose tout l'écosystème de Microsoft. PostgreSQL se concentre sur son rôle de SGBD et ne vous impose rien d'autre, ni en terme d'outils, ni en terme de société de support.

Au niveau des fonctionnalités, PostgreSQL ne dispose des vues matérialisées qu'à partir de la version 9.3 alors que SQL Server en dispose depuis bien longtemps. De plus, elles ne sont pas aussi avancées que ce que propose SQL Server. Le partitionnement de PostgreSQL était manuel et très rudimentaire par rapport à ce qui est disponible dans SQL Server. Cependant, la version 10 améliore bien la situation pour PostgreSQL.

2.3.3 POSTGRESQL VS. ORACLE

- PostgreSQL est le SGBD le plus proche d'Oracle
- Points forts de PostgreSQL
  - respect des standards
  - DDL transactionnel
  - pas de gestion et de coût de licence
- Points faibles de PostgreSQL
  - manque certains objets (synonymes, packages)
  - un parallélisme perfectible (amélioration en 9.6)
  - un partitionnement perfectible (amélioration en 10)
  - pas de vues matérialisées (amélioration en 9.3)
  - lenteur du `SELECT count(*)` (amélioration en 9.2)
  - pas d'équivalent de RAC
- Deux produits à présent dans la même catégorie

En terme de fonctionnalités et de fiabilité, PostgreSQL a surtout Oracle comme concurrent, et non pas MySQL (ce qui semblerait le concurrent le plus logique du fait de leur licence libre).

PostgreSQL ne dispose pas de l'intégralité de la pléthore de fonctionnalités d'Oracle. Par exemple, les vues matérialisées ne sont pas disponibles sur PostgreSQL avant la 9.3. Le

partitionnement est possible mais est assez complexe à mettre en place pour les versions antérieures à la 10.

Pour le `SELECT count(*)`, Oracle passe par une lecture de l'index, ce qui améliore fortement les performances. PostgreSQL passe aussi par une lecture d'index à partir de PostgreSQL 9.2, mais les versions précédentes ne pouvaient pas le faire. PostgreSQL ne disposait pas d'un mode d'exécution en parallèle avant la version 9.6 (ce qui ne peut poser problème qu'en contexte d'infocentre).

RAC n'a tout simplement pas d'équivalent dans le monde PostgreSQL.

PostgreSQL ne dispose pas d'équivalent complet à Oracle Enterprise Manager ou à Grid Control, les consoles de supervision d'Oracle.

PostgreSQL rattrape son retard sur beaucoup de fonctionnalités comme le partitionnement, les vues matérialisées, le parallélisme. Les fonctionnalités de Streaming Replication et Hot Standby sont l'équivalent de Oracle Data Guard. Les fonctions de fenêtrage sont très ressemblantes. Les performances sont très proches pour la plupart des cas d'utilisation.

PostgreSQL possède de l'avance sur certaines fonctionnalités, comme l'implémentation de SSI (Serializable Snapshot Isolation) ou comme le support du DDL transactionnel. Une transaction commencée sera automatiquement terminée avec un `COMMIT` au premier ordre DDL arrivé dans cette transaction.

Oracle pâtit également de la politique commerciale de son éditeur quant aux coûts de licence, et aux audits. La tendance du marché est de migrer des bases Oracle vers PostgreSQL, pas l'inverse.

2.3.4 POSTGRESQL VS. INFORMIX

- Informix est un lointain cousin de PostgreSQL
- Points forts de PostgreSQL
  - un vrai `CREATER USER`
  - pas de double quote (`"`) pour les chaînes
- Points faibles de PostgreSQL
  - pas de `synonym`

En 1995, la version « universitaire » de Postgres fut commercialisée sous le nom d'Illustra par une société éponyme dirigée notamment par Michael Stonebraker. Les deux logiciels évoluèrent de manière différente et Illustra a fini par être racheté en 1997 par Informix (maintenant détenu par IBM).

Il n'y a pas d'équivalent au concept de synonyme dans PostgreSQL. On utilise le `search_path` mais ce n'est qu'un contournement.

Par contre, pour ajouter un utilisateur de bases dans Informix, il faut créer un nouvel utilisateur système. Ce n'est pas le cas dans PostgreSQL, ce qui le sépare de la gestion du système d'exploitation.

La gestion des chaînes de caractères est différente. PostgreSQL n'accepte pas les double guillemets pour entourer les chaînes, conformément à la norme SQL (les double guillemets étant réservés pour spécifier des noms d'objets).

2.3.5 POSTGRESQL VS. NOSQL

- 4 technologies majeures dans le monde NoSQL
  - stockage clé->valeur (Redis, Apache Cassandra, Riak, MongoDB)
  - stockage documents (Apache CouchDB, MongoDB)
  - stockage colonnes (Apache Hbase, Google BigTable)
  - stockage graphes (Neo4j)
- PostgreSQL réunit le monde relationnel et le monde NoSQL
  - stockage clé->valeur : `hstore`
  - stockage documents : `xml`, `json` et `jsonb` (plus performant que MongoDB)
  - procédure stockée en Javascript : PL/V8
  - stockage colonnes : `cstore_fdw`

PostgreSQL n'est pas en reste vis à vis des bases de données NoSQL. PostgreSQL permet de stocker des données au format clé->valeur. Couplé aux index GiST et GIN, ce mode de stockage s'avère comparable à MongoDB (https://wiki.postgresql.org/images/b/b4/Pg-as-nosql-pgday-fosdem-2013.pdf) ou à Redis.

PostgreSQL peut stocker des données au format JSON (depuis la version 9.2, avec de nombreuses fonctions ajoutées en 9.3). L'avantage est que les données seront validées : si une donnée n'est pas conforme au format JSON, elle sera rejetée. Avec le type natif `jsonb` (en 9.4) il est possible d'indexer le JSON, et les performances de PostgreSQL avec ce nouveau type de stockage de documents sont très supérieures à MongoDB (http://blogs.enterprisedb.com/2014/09/24/postgres-outperforms-mongodb-and-ushers-in-new-developer-reality/).

Couplé au langage PL/V8, le type de données JSON permet de traiter efficacement ce type de données. PL/V8 permet d'écrire des fonctions en langage Javascript, elles seront exécutées avec l'interpréteur V8 écrit par Google. Ces fonctions sont bien entendus indexables par PostgreSQL si elles respectent un certain nombre de pré-requis.

17.12

Le stockage colonne pour PostgreSQL consiste actuellement en une extension *Foreign Data Wrapper* nommée `cstore_fdw`. Elle est développée par la société CitusData (http://www.citusdata.com/blog/76-postgresql-columnar-store-for-analytics).

Malgré cela, PostgreSQL conserve les fonctionnalités qui ont fait sa réputation et que les moteurs NoSQL ont dû abandonner :

- un langage de requête puissant ;
- un optimiseur de requêtes sophistiqué ;
- la normalisation des données ;
- les jointures ;
- l'intégrité référentielle ;
- la durabilité.

Voici enfin un lien vers une excellente présentation sur les différences entre PostgreSQL et les solutions NoSQL : http://fr.slideshare.net/EnterpriseDB/the-nosql-way-in-postgres

ainsi que l'avis de Bruce Momjian sur le choix du NoSQL pour de nouveaux projets[5]

2.4 À LA RENCONTRE DE LA COMMUNAUTÉ

- Cartographie du projet
- Pourquoi participer
- Comment participer

2.4.1 POSTGRESQL, UN PROJET MONDIAL

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

Quelques faits :

- Le projet est principalement anglophone.
- Il existe une très grande communauté au Japon.
- La communauté francophone est très dynamique mais on trouve très peu de développeurs francophones.
- La communauté hispanophone est naissante.
- Les développeurs du noyau (*core hackers*) vivent en Europe, au Japon, en Russie et en Amérique du Nord.

[5] https://momjian.us/main/blogs/pgblog/2017.html#June_12_2017

FIGURE 2: CARTE DES HACKERS

2.4.2 POSTGRESQL CORE TEAM

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

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

La « Core Team » est un ensemble de personnes doté d'un pouvoir assez limité. Ils peuvent décider de la sortie d'une version. Ce sont les personnes qui sont immédiatement au courant des failles de sécurité du serveur PostgreSQL. Tout le reste des décisions est pris par la communauté dans son ensemble après discussion, généralement sur la liste pgsql-hackers.

Détails sur les membres actuels de la *core team* :

- **Peter Eisentraut**, 2nd Quadrant, Philadelphie (États-Unis) - développement du moteur (internationalisation, SQL/Med, etc.)
- **Tom Lane**, Crunchy Data, Pittsburgh (USA) - certainement le développeur le plus aguerri

- **Bruce Momjian**, EnterpriseDB, Philadelphia (USA) - a lancé le projet, fait principalement de la promotion
- **Dave Page**, EnterpriseDB, Oxfordshire (Angleterre) - leader du projet pgAdmin, administration des serveurs
- **Magnus Hagander**, Redpill Linpro, Stockholm (Suède) - développeur PostgreSQL, administration des serveurs, président de PostgreSQL Europe

---

2.4.3 CONTRIBUTEURS

Actuellement, PostgreSQL compte une centaine de « contributeurs » qui se répartissent quotidiennement les tâches suivantes :

- développement des projets satellites (Slony, pgAdmin, ...) ;
- promotion du logiciel ;
- administration des serveurs ;
- rédaction de documentation ;
- conférences ;
- 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.

17.12

PGCon2009 a réuni 180 membres actifs à Ottawa.

Voir la liste des contributeurs officiels[6] .

2.4.4 UTILISATEURS

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

Il est impossible de connaître précisément le nombre d'utlisateurs de PostgreSQL. On sait toutefois que ce nombre est en constante augmentation.

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.

2.4.5 POURQUOI PARTICIPER

- Rapidité des corrections de bugs
- Préparer les migrations
- 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 ! 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.

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

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'approndir ses compétences.

2.4.6 SERVEURS

- Site officiel : http://www.postgresql.org/
- La doc : http://www.postgresql.org/docs/
- Actualité : http://planet.postgresql.org/
- Des extensions : http://pgxn.org/

Le site officiel de la communauté se trouve sur http://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. Il remplace petit à petit le site pgfoundry.org. Ce dernier est de plus en plus délaissé. Il s'agissait d'un SourceForge dédié à PostgreSQL, mais le manque d'administration fait que les développeurs l'abandonnent de plus en plus (ils vont généralement sur github ou sur le vrai SourceForge). Actuellement, il est déconseillé d'ouvrir un projet sur pgfoundry.org.

2.4.7 SERVEURS FRANCOPHONES

- Site officiel : http://www.postgresql.fr/
- La doc : http://docs.postgresql.fr/
- Forum : http://forums.postgresql.fr/
- Actualité : http://planete.postgresql.fr/
- Wiki : http://wiki.postgresql.fr/

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.

2.4.8 LISTES DE DISCUSSIONS / LISTES D'ANNONCES

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

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.

Pour s'inscrire ou consulter les archives : http://www.postgresql.org/community/lists/

Si vous avez une question ou un problème, la réponse se trouve probablement dans les archives ! Pourquoi ne pas utiliser un moteur de recherche spécifique ?

- http://www.nabble.com/
- http://markmail.org/

N'hésitez pas à rejoindre ces listes.

Les listes de diffusion sont régies par des règles de politesse et de bonne conduite. Avant de poser une question, nous vous conseillons de consulter le guide suivant : http://www.linux-france.org/article/these/smart-questions/smart-questions-fr.html

---

2.4.9 IRC

- Réseau Freenode
- IRC anglophone
  - `#postgresql`
  - `#postgresql-eu`
- IRC francophone
  - `#postgresqlfr`

Le point d'entrée principal pour le réseau Freenode est le serveur : `irc.freenode.net`. La majorité des développeurs sont disponibles sur IRC et peuvent répondre à vos questions.

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.

2.4.10 WIKI

- http://wiki.postgresql.org/
- http://wiki.postgresql.org/wiki/Fran%C3%A7ais

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

Au départ, le wiki postgresql.org 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.

Il existe une partie spécifiquement en français, indiquant la liste des documents écrits en français sur PostgreSQL. Elle n'est pas exhaustive et souffre fréquemment d'un manque de mises à jour.

2.4.11 L'AVENIR DE POSTGRESQL

- PostgreSQL 10 est sortie en septembre 2017
- Grandes orientations :
  - réplication logique
  - meilleur parallélisme
  - gros volumes
- Prochaine version, la 11
- Stabilité économique
- 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

17.12

- etc.

PostgreSQL est là pour durer. 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.

2.5 CONCLUSION

- Beaucoup de projets complémentaires
- Une communauté active
- Concurrent solide face aux SGBD propriétaires
- De nombreuses conversions en cours vers PostgreSQL

2.5.1 BIBLIOGRAPHIE

- Organisation du projet PostgreSQL, Guillaume Lelarge, 2010[7]

Iconographie :

La photo initiale est sous licence CC-BY-SA[8] : http://www.flickr.com/photos/st3f4n/675708572/

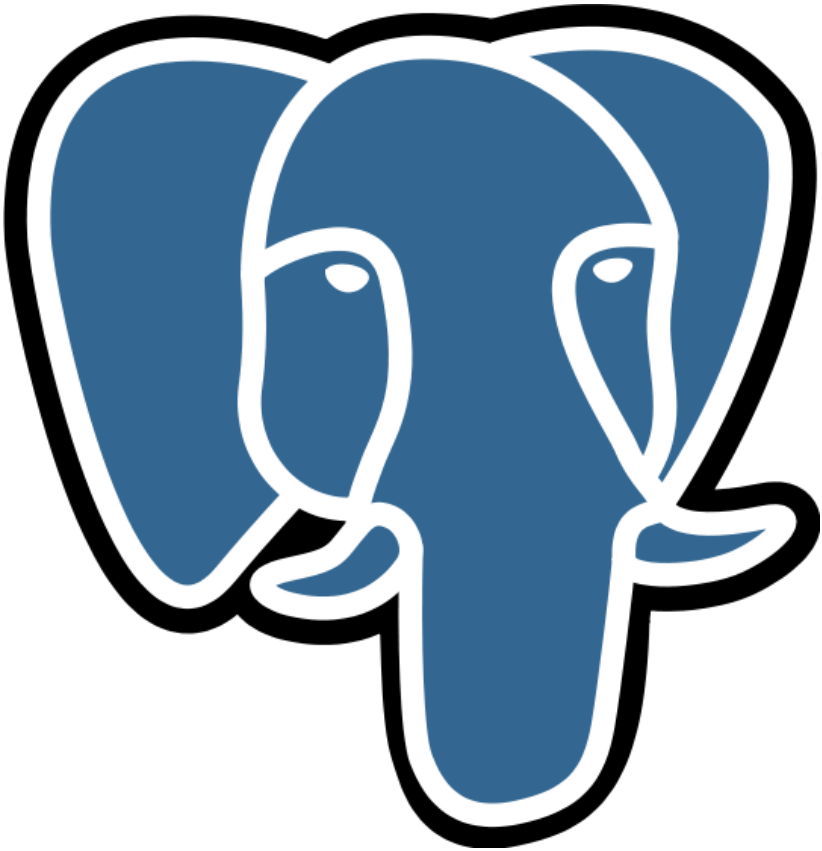
2.5.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[7] http://www.dalibo.org/organisation_du_projet_postgresql
[8] http://creativecommons.org/licenses/by-sa/2.0/deed.fr

3 FONCTIONNEMENT INTERNE



3.1 INTRODUCTION

Présenter le fonctionnement de PostgreSQL en profondeur :

- Comprendre :
  - Le paramétrage
  - Les choix d'architecture
  - Et ce que cela implique

17.12

- Deux modules (une journée) :
  - Fonctionnement interne
  - Transactions

3.1.1 AU MENU

- les processus
- gestion de la mémoire
- les fichiers
- *shared buffers*
- journalisation
- statistiques
- optimiseur de requête
- gestion des connexions

3.1.2 OBJECTIFS

- Premier module : fonctionnement interne
- Second module : implémentation MVCC
- Tous les paramètres ne sont pas abordés

La présentation se fait par module fonctionnel du noyau PostgreSQL. Bien sûr, tous les modules interagissent ensemble, et les frontières ne sont pas toujours parfaitement nettes. N'hésitez pas à poser des questions !

Tous les paramètres du moteur ne sont pas abordés, même si ces modules permettent d'en présenter une forte majorité. La compréhension théorique apportée par ces deux modules doit permettre la compréhension de tous les paramètres.

3.2 LES PROCESSUS

- PostgreSQL est :
  - multi-processus et non multi-thread
  - à mémoire partagée
  - client-serveur

L'architecture PostgreSQL est une architecture multi-processus et non multi-thread.

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 multi-thread, où l'ensemble du moteur s'exécute dans un seul processus, dans plusieurs threads (contextes) d'exécution, où tout est partagé par défaut.

Le principal avantage de cette architecture multi-processus 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.

Pour comparatif :

Multi-processus : Oracle, DB2

Multi-thread : Oracle sous Windows, SQL Server, MySQL

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. On ne se connecte à PostgreSQL qu'à travers un protocole bien défini, on n'accède jamais aux fichiers de données. Certains moteurs – de conception assez ancienne le plus souvent –, au contraire, permettent l'accès direct à ces fichiers de données.

3.2.1 PROCESSUS D'ARRIÈRE-PLAN (1/2)

```
# ps f -e --format=pid,command | grep postgres
 7771 /usr/local/pgsql/bin/postgres -D /var/lib/postgresql/10/data
 7773  \_ postgres: checkpointer process
 7774  \_ postgres: writer process
 7775  \_ postgres: wal writer process
 7776  \_ postgres: autovacuum launcher process
 7777  \_ postgres: stats collector process
 7778  \_ postgres: bgworker: logical replication launcher
```

On constate que plusieurs processus sont présents dès le démarrage de PostgreSQL. Nous allons les détailler.

17.12

Il est à noter que la commande `ps` affiche un grand nombre d'informations sur le processus seulement si le paramètre `update_process_title` est activé.

NB : sur Debian, le postmaster est nommé *postgres* comme ses processus fils.

3.2.2 PROCESSUS D'ARRIÈRE PLAN (2/2)

- Les processus présents au démarrage :
  - Un processus père, appelé le `postmaster`
  - Un `writer` ou `background writer`
  - Un `checkpointer`
  - Un `wal writer`
  - Un `autovacuum launcher`
  - Un `stats collector`
  - Un `bgwriter`
- 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. Nous y reviendrons dans la partie « Gestion de la mémoire ».
- Le `wal writer` écrit le journal de transactions de façon anticipée, afin de limiter le travail de l'opération `COMMIT`. Nous y reviendrons dans la partie « Journalisation ».
- L'`autovacuum launcher` pilote les opérations d'« autovacuum ». Ceci sera expliqué en détail dans le module « Mécanique du moteur transactionnel ».
- Le `stats collector` collecte les statistiques d'exécution du serveur. Nous y reviendrons dans la partie « Statistiques ».
- Le `bgwriter` est un *worker* dédié à la réplication logique, activé par défaut à partir de la version 10.

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.

Des processus supplémentaires peuvent apparaître, comme un `walsender` dans le cas où la base est un maître de réplication, un `logger process` si PostgreSQL doit gérer lui-même les fichiers de traces (par défaut sous RedHat, mais pas sous Debian), ou un `archiver process` si l'instance est paramétrée pour générer des archives de ses journaux de transactions.

3.2.3 PROCESSUS PAR CLIENT

- 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
  - mais peut être aidé par d'autres processus (>= 9.6)
- Le nombre de processus est régi par les paramètres :
  - `max_connections`
  - `superuser_reserved_connections`

Pour chaque nouvelle session à l'instance, le processus `postmaster` crée un processus fils qui s'occupe de gérer cette session.

Ce processus reçoit les ordres SQL, les interprète, exécute les requêtes, trie les données, et enfin retourne les résultats. À partir de la version 9.6, dans certains cas, il peut demander le lancement d'autres processus pour l' aider dans l'exécution d'une requête en lecture seule.

Il y a un processus dédié à chaque connexion cliente, et ce processus est détruit à fin de cette connexion.

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 possibles à la base est régi par le paramètre `max_connections`. Afin de permettre à l'administrateur de se connecter à l'instance si cette limite était atteinte, `superuser_reserved_connections` sont réservées aux super-utilisateurs de l'instance. Une prise en compte de la modification de ces deux paramètres impose un redémarrage complet de l'instance, puisqu'ils ont un impact sur la taille de la mémoire partagée entre les processus PostgreSQL.

3.3 GESTION DE LA MÉMOIRE

Structure de la mémoire sous PostgreSQL

- Zone de mémoire partagée :
  - `shared_buffers`
  - `wal_buffers`

  - Données de session
  - Verrous
- Par processus :
  - `work_mem`
  - `maintenance_work_mem`
  - `temp_buffers`

3.3.1 MÉMOIRE PARTAGÉE

- Zone de mémoire partagée :
  - `shared_buffers`
  - `wal_buffers`
  - Données de session (paramètres `max_connections` et `track_activity_query_size`)
  - Verrous (paramètres `max_connections` et `max_locks_per_transaction`)

La zone de mémoire partagée est allouée statiquement au démarrage de l'instance. Elle est calculée en fonction du dimensionnement des différentes zones :

- `shared_buffers` : le cache de blocs partagé entre les différents processus.
- `wal_buffers` : le tampon de journalisation.
- Les données de sessions. Les principaux paramètres dont cette zone dépend sont `max_connections` et `track_activity_query_size`
- Les verrous. Les paramètres dont cette zone dépend sont `max_connections` et `max_locks_per_transaction`.

Toute modification des paramètres régissant la mémoire partagée imposent un redémarrage de l'instance.

Nous verrons en détail l'utilité de chacune de ces zones dans les chapitres suivants.

3.3.2 MÉMOIRE PAR PROCESSUS

- Par processus :
  - `work_mem`
  - `maintenance_work_mem`
  - `temp_buffers`
- Pas de limite stricte à la consommation mémoire d'une session

Chaque processus, en plus de la mémoire partagée à laquelle il accède en permanence, peut allouer de la mémoire pour ses besoins propres. L'allocation de cette mémoire est temporaire (elle est libérée dès qu'elle n'est plus utile). Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Cette mémoire est utilisée dans plusieurs contextes :

- Pour des tris et hachages lors de l'exécution de requêtes : un `ORDER BY`, par exemple, peut nécessiter un tri. Ce tri sera effectué à hauteur de `work_mem` en mémoire, puis sera poursuivi sur le disque au besoin.
- Pour des opérations de maintenance : un `CREATE INDEX` ou un `VACUUM` par exemple. Ces besoins étant plus rares, mais plus gourmands en mémoire, on dispose d'un second paramètre `maintenance_work_mem`, habituellement plus grand que `work_mem`.
- Pour la gestion de l'accès à des tables temporaires : afin de minimiser les appels systèmes dans le cas d'accès à des tables temporaires (locales à chaque session), chaque session peut allouer `temp_buffers` de cache dédié à ces tables.

Comme elle n'est pas partagée, cette mémoire est totalement dynamique.

Il n'y a pas de limite globale de la mémoire pouvant être utilisée par ces paramètres. Par exemple, il est possible, potentiellement, que tous les processus allouent simultanément plusieurs fois `work_mem` (si la requête en cours d'exécution nécessite plusieurs tris par exemple ou si le processus qui a reçu la requête a demandé l'aide à d'autres processus). Il faut donc rester prudent sur les valeurs de ces paramètres, `work_mem` tout particulièrement, et superviser les conséquences d'une modification de celui-ci.

3.4 LES FICHIERS

- Une instance est composée de fichiers :
  - Répertoire de données
  - Fichiers de configuration
  - Fichier PID
  - Tablespaces
  - Statistiques
  - Fichiers de trace

Une instance est composée de :

- Un répertoire de données. Il contient les fichiers obligatoires au bon fonctionnement de l'instance (fichiers de données, journaux de transaction…)

- Des fichiers de configuration. Selon la distribution ils sont stockés dans le répertoire de données (Red Hat, CentOS...) ou dans `/etc/postgresql` (Debian et dérivés).
- Un fichier `PID`, qui 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 donnée.
- Optionnellement, des tablespaces. Ce sont des espaces de stockage supplémentaires, stockés habituellement dans d'autres systèmes de fichiers.
- Un fichier de statistiques d'exécution.
- Un ou des fichiers de trace (journaux) de la base, si celle-ci n'utilise pas le mécanisme du système d'exploitation (syslog sous Unix, journal des évènements sous Windows).

3.4.1 RÉPERTOIRE DE DONNÉES

```
postgres$ ls $PGDATA
# ls $PGDATA
base            pg_ident.conf  pg_serial     pg_tblspc    postgresql.auto.conf
global          pg_logical     pg_snapshots  pg_twophase  postgresql.conf
pg_commit_ts    pg_multixact   pg_stat       PG_VERSION   postmaster.opts
pg_dynshmem     pg_notify      pg_stat_tmp   pg_wal       postmaster.pid
pg_hba.conf     pg_replslot    pg_subtrans   pg_xact
```

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. On peut aussi le connaître, étant connecté à la base, en interrogeant le paramètre `data_directory`.

```
postgres=# SHOW data_directory ;
      data_directory
---------------------------
 /var/lib/pgsql/10/data
(1 row)
```

Vous pouvez trouver une description de tous les fichiers et répertoires dans la documentation officielle[9] .

---

[9] https://www.postgresql.org/docs/current/static/storage-file-layout.html

3.4.2 FICHIERS DE DONNÉES

- Contient de quoi démarrer l'instance :
  - **base/** : contient les fichiers de données (un sous-répertoire par base)
  - **global/** : contient les objets globaux à toute l'instance

`base/` contient les fichiers de données (tables, index, séquences). Il contient un sous-répertoire par base, le nom du répertoire étant l' `OID` de la base dans `pg_database`. Dans ces répertoires, on trouve 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`. Il peut changer dans la vie de l'objet (par exemple lors d'un `VACUUM FULL`, un `TRUNCATE`...)
- Si le nom est postfixé 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 postfixé par `_fsm`, il s'agit du fichier stockant la `Free Space Map` (liste des blocs réutilisables).
- Si le nom est postfixé 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 fichier de l'objet ayant `relfilenode=14356` dans `pg_class`, dans la base pour laquelle `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 :

```
postgres=# SELECT pg_table_size('test');
 pg_table_size
---------------
     181346304
(1 row)
```

Néanmoins, il existe un utilitaire appelé `oid2name` dont le but est de faire la liaison entre nom de fichier et nom de l'objet PostgreSQL.

Le répertoire `global` contient les objets qui sont globaux à toute une instance, comme la table des bases de données, celle des rôles et celle des tablespaces.

Avant la 9.3, il contient aussi le fichier des statistiques d'activité (`pgstat.stat`).

3.4.3 GESTION DES TRANSACTIONS

- **pg_wal/** : contient les journaux de transactions
  - **pg_xlog/** avant la v10
- **pg_xact/** : contient l'état des transactions
  - **pg_clog/** avant la v10
- **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 `logs` (journaux) contenus dans `pg_wal` ne doivent **jamais** être effacés. Ces fichiers sont cruciaux au bon fonctionnement de la base.

Les répertoires `pg_xact`, `pg_commit_ts`, `pg_multixact`, `pg_serial`, `pg_snapshots`, `pg_subtrans` et `pg_twophase` contiennent des fichiers essentiels à la gestion des transactions.

- `pg_xact` contient l'état de toutes les transactions passées ou présentes sur la base (validées, annulées ou en cours).
- `pg_commit_ts` contient l'horodatage de la validation de chaque transaction.
- `pg_multixact` est utilisé dans l'implémentation des verrous partagés (`SELECT xxx FOR SHARE`).
- `pg_serial` est utilisé dans l'implémentation de SSI (`Serializable Snapshot Isolation`).
- `pg_snapshots` est utilisé pour stocker les snapshots exportées de transactions.
- `pg_subtrans` est utilisé pour l'implémentation des sous-transactions (les `SAVEPOINTS`). Les fichiers qu'il contient permettent de stocker l'imbrication de transactions.
- `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.

L'utilité de `pg_xact` sera détaillée plus avant dans le module « Mécanique du moteur transactionnel ».

Les *logs* (journaux) contenus dans `pg_xact` ne doivent **jamais** être effacés. Ces fichiers sont cruciaux au bon fonctionnement de la base.

La version 10 a été l'occasion du changement de nom de quelques répertoires. `pg_wal` s'appelait auparavant `pg_xlog`, `pg_xact` s'appelait `pg_clog`.

3.4.4 GESTION DE 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.

Ces répertoires n'existent pas avant la 9.4.

3.4.5 LE RÉPERTOIRE DES TABLESPACES

- **pg_tblspc/** : contient des liens symboliques vers les répertoires contenant des tablespaces

`pg_tblspc` contient des liens symboliques vers les répertoires contenant des tablespaces. Chaque lien symbolique a comme nom l'`OID` du tablespace dans `pg_tablespace`.

Sous Windows, il ne s'agit pas de liens symboliques comme sous Unix, mais de `Reparse Points`, qu'on trouve parfois aussi nommés `Junction Points` dans la documentation de Microsoft.

3.4.6 STATISTIQUES D'ACTIVITÉ

- **pg_stat/**
- **pg_stat_tmp/**

`pg_stat_tmp` est le répertoire par défaut de stockage des statistiques d'exécution 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`. Il est modifiable à chaud par édition du fichier de configuration puis demande de rechargement de la configuration au serveur PostgreSQL. À l'arrêt, les fichiers sont copiés dans le répertoire `global/` jusqu'à la 9.3. À partir de la 9.3, ils sont stockés dans le répertoire `pg_stat/`.

Exemple d'un répertoire de stockage des statistiques déplacé en `tmpfs` (défaut sous Debian) :

```
postgres=# SHOW stats_temp_directory ;
           stats_temp_directory
----------------------------------------
 /var/run/postgresql/10-main.pg_stat_tmp
(1 row)
```

3.4.7 AUTRES RÉPERTOIRES

- **pg_dynshmem/**
- **pg_notify/**

`pg_dynshmem` est utilisé par les extensions utilisant de la mémoire partagée dynamique. Il apparaît en version 9.4.

`pg_notify` est utilisé par le mécanisme de gestion de notification de PostgreSQL (`LISTEN` et `NOTIFY`) qui permettent de passer des messages de notification entre sessions.

3.4.8 FICHIERS DE CONFIGURATION

- `pg_hba.conf`
- `pg_ident.conf`
- `postgresql.conf`
- `postgresql.auto.conf`

Les fichiers de configuration sont habituellement les 3 suivants :

- `postgresql.conf` : il contient une liste de paramètres, sous la forme `paramètre=valeur`. Tous les paramètres énoncés précédemment sont modifiables (et présents) dans ce fichier.
- `pg_hba.conf` : il contient les règles d'authentification à la base.

- `pg_ident.conf` : il complète `pg_hba.conf`, quand on décide de se 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`.

3.4.9 AUTRES FICHIERS

- `PG_VERSION` : fichier contenant la version majeure de l'instance
- `postmaster.pid`
  - contient de nombreuses informations sur le processus maître
  - fichier externe possible, paramètre `external_pid_file`
- `postmaster.opts`

`PG_VERSION` est un fichier. Il contient en texte lisible la version majeure devant être utilisée pour accéder au répertoire (par exemple `9.6`). 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 maître sur la première ligne, l'emplacement du répertoire des données sur la deuxième ligne et des informations sur le segment de mémoire partagée sur la troisième ligne. Par exemple :

```
postgres@pegase:~/10/data$ cat /var/lib/postgresql/10/data/postmaster.pid
7771
/var/lib/postgresql/10/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/10/data
pos 7773 0 42486  4656   0 16:26 00:00  postgres: checkpointer process
```

```
pos 7774 0 42486  5044   1 16:26 00:00  postgres: writer process
pos 7775 0 42486  8224   1 16:26 00:00  postgres: wal writer process
pos 7776 0 42850  5640   1 16:26 00:00  postgres: autovacuum launcher process
pos 7777 0  6227  2328   3 16:26 00:00  postgres: stats collector process
pos 7778 0 42559  3684   0 16:26 00:00  postgres: bgworker: logical replication launch

$ipcs -p |grep  7771
54919263   postgres   7771       10640

$ipcs  | grep 54919263
0x0052e2c1 54919263   postgres   600        56         6
```

Le processus maître 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/10/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.

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/10/data"
```

3.4.10 PARAMÈTRES EN LECTURE SEULE

- Dépendent d'options de compilation
- Quasiment jamais modifiés
- Tailles de bloc ou de fichier
- `block_size`
- `wal_block_size`
- `segment_size`
- `wal_segment_size`
- Ces paramètres sont en lecture seule, mais peuvent être consultés par la commande `SHOW`, ou en interrogeant la vue `pg_settings`. On peut aussi obtenir l'information

via la commande `pg_controldata`.

- Ils sont fixés à la compilation du moteur.
- `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, par défaut 16 Mo.

Un moteur compilé avec des options non-standard ne pourra pas ouvrir des fichiers n'ayant pas les mêmes valeurs pour ces options.

3.4.11 POSTGRESQL.CONF

- Le fichier principal de configuration :
  - format clé = valeur
  - valeur entre « ' » (single quote) si chaîne de caractère
  - classé par sections
  - commenté (*change requires restart*)
  - paramètre `config_file`
  - inclusion externe supportée avec les clauses `include` et `include_if_exists`

C'est le fichier le plus important. Il contient le paramétrage de l'instance. Le format est un paramètre par ligne, sous le format `clé = valeur`. Les commentaires commencent par « # » (croisillon).

Par exemple :

```
listen_addresses = 'localhost'
```

Les valeurs de ce fichier ne seront pas forcément les valeurs actives. Si des options sont passées en arguments à `pg_ctl`, elles seront prises en priorité par rapport à celles du fichier de configuration. On peut aussi surcharger les options modifiables à chaud par utilisateur, par base, et par combinaison « utilisateur+base » depuis la version 9.0 (*cf* la vue `pg_db_role_setting`).

Ainsi, l'ordre des surcharge est le suivant :

```
paramètre par défaut
  -> postgresql.conf
    -> option de pg_ctl / postmaster
      -> paramètre par base
```

```
        -> paramètre par rôle
          -> paramètre base+rôle
            -> paramètre de session
```

La meilleure source d'information est la vue `pg_settings` :

```
SELECT name,source,context,setting,boot_val,reset_val
FROM pg_settings
WHERE name IN ('client_min_messages', 'wal_keep_segments', 'wal_segment_size');
```

```
        name          | source  | context  | setting | boot_val | reset_val
----------------------+---------+----------+---------+----------+-----------
 client_min_messages  | session | user     | debug   | notice   | notice
 wal_keep_segments    | default | sighup   | 0       | 0        | 0
 wal_segment_size     | default | internal | 2048    | 2048     | 2048
(3 rows)
```

On constate par exemple que dans la session ayant effectué la requête, `client_min_messages` a été modifié à la valeur `debug`. On peut aussi voir le contexte dans lequel le paramètre est modifiable : le `client_min_messages` est modifiable par l'utilisateur dans sa session. Le `wal_keep_segments` seulement par `sighup`, c'est-à-dire par un `pg_ctl reload`, et le `wal_segment_size` n'est pas modifiable, c'est un paramètre interne.

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 proviennent le paramètre. Depuis la version 9.5, une nouvelle colonne est apparue, nommée `pending_restart`. Elle indique si un paramètre a été modifié mais nécessite un redémarrage pour être appliqué.

On peut aussi inclure d'autres fichiers dans le fichier `postgresql.conf`, par la syntaxe

```
include 'filename'
```

Ce fichier est alors inclus à l'endroit où la directive `include` est positionnée. Si le fichier n'existe pas, une erreur `FATAL` est levée. La clause `include_if_exists` ne fait que notifier l'absence du fichier, mais poursuit la lecture du fichier de base.

Parfois, il n'est pas facile de trouver l'emplacement de ce fichier. Le plus simple dans ce cas est de se connecter à une base et de regarder la valeur du paramètre `config_file` :

```
postgres=# SHOW config_file;
               config_file
----------------------------------------
 /var/lib/postgresql/10/data/postgresql.conf
(1 row)
```

À partir de la version 9.5, il existe aussi la vue pg_file_settings. Elle indique la configuration présente dans les fichiers de configuration. Elle peut être utile lorsque la configuration est réalisée 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 :

```
postgres=# ALTER SYSTEM SET work_mem TO '16MB' ;
ALTER SYSTEM

postgres=# ALTER SYSTEM SET max_connections TO 200 ;
ALTER SYSTEM

postgres=# select pg_reload_conf() ;
 pg_reload_conf
----------------
 t
(1 ligne)

postgres=# SELECT * FROM pg_file_settings
WHERE name IN ('work_mem','max_connections')
ORDER BY name ;

-[ RECORD 1 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/10/data/postgresql.conf
sourceline | 65
seqno      | 2
name       | max_connections
setting    | 100
applied    | f
error      |
-[ RECORD 2 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/10/data/postgresql.auto.conf
sourceline | 3
seqno      | 14
name       | max_connections
setting    | 200
applied    | f
error      | setting could not be applied
-[ RECORD 3 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/10/data/postgresql.auto.conf
```

```
sourceline | 4
seqno      | 15
name       | work_mem
setting    | 16MB
applied    | t
error      |
```

3.4.12 PG_HBA.CONF ET PG_IDENT.CONF

- Authentification multiple, suivant l'utilisateur, la base et la source de la connexion.
  - `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`.

3.4.13 LES TABLESPACES

- Espace de stockage d'objets
- Un simple répertoire
- Répartit la charge et la volumétrie sur plusieurs disques
- paramètres `default_tablespace` et `temp_tablespaces`

Un tablespace, vu de PostgreSQL, est un espace de stockage des objets (tables et indexes principalement).

Vu du système d'exploitation, il s'agit d'un répertoire. Il n'aura d'intérêt que s'il est placé sur un système de fichiers différent du système de fichiers contenant `PGDATA`.

Les tablespaces sont déclarés dans la table système `pg_tablespace`. Ils sont aussi stockés au niveau du système de fichiers, sous forme de liens symboliques (ou de *reparse points* sous Windows), dans le répertoire `PGDATA/pg_tblspc`.

Le paramètre `default_tablespace` permet d'utiliser un autre tablespace par défaut. On peut aussi définir un ou plusieurs tablespaces pour les opérations de tri, dans le paramètre `temp_tablespaces` (il faudra donner des droits dessus aux utilisateurs avec `GRANT CREATE ON TABLESPACE ... TO ...`).

Par ailleurs, on peut aussi modifier le tablespace par défaut par base avec les commandes `CREATE DATABASE` ou `ALTER DATABASE` :

```
CREATE DATABASE nom TABLESPACE 'tbl_nom';
-- ou
ALTER DATABASE nom SET default_tablespace TO 'tbl_nom';
```

3.4.14 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. Ils sont traités dans un prochain chapitre.

Si `logging_collector` est activé, c'est-à-dire que PostgreSQL collecte lui-même ses traces, l'emplacement de ces journaux se paramètre grâce à `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.

3.5 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
  - Importants pour les performances

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 pas écrite sur le disque par le processus effectuant la modification. 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 *pin 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. 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 qui y est fait. La redondance est donc habituellement limitée, et il est conseillé de restreindre `shared_buffers` à 1/4 de la mémoire totale du serveur au maximum.

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 n'effectuer que 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 si `shared_buffers` dépasse 8 Go sous Linux, et 1 Go sous Windows.

Ce paramétrage du cache est malgré tout moins critique que sur de nombreux autres SGBD : le cache système limite la plupart du temps l'impact d'un mauvais paramétrage de `shared_buffers`, et il est donc préférable de sous-dimensionner `shared_buffers` que de le sur-dimensionner.

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. On le paramètre habituellement aux alentours des 2/3 de la taille du cache du système d'exploitation, pour un serveur dédié.

3.5.1 NOTIONS ESSENTIELLES

- Buffer pin
- Buffer dirty/clean
- Compteur d'utilisation (usagecount)
- Clocksweep

Les principales notions à connaître pour comprendre le mécanisme de gestion du cache de PostgreSQL sont :

- Buffer pin : chaque processus voulant accéder à un buffer (un bloc du cache) doit d'abord en forcer le maintien en cache (*to pin* signifie *épingler*). Chaque processus accédant à un buffer incrémente ce compteur, et le décrémente quand il a fini. Un buffer dont le pin est différent de 0 ne peut donc pas être recyclé.
- Buffer dirty/clean : un buffer est soit propre (clean), soit sale (dirty). Il est sale si son contenu dans le cache ne correspond pas à son contenu sur disque (il a été modifié dans le cache, mais pas encore resynchronisé). La différence fondamentale

est qu'un buffer propre peut être supprimé du cache sans plus de formalité, alors qu'un buffer sale doit d'abord être resynchronisé, ce qui est bien plus coûteux.

- Compteur d'utilisation : 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). Un autre mécanisme décrémente ce compteur. Seuls un buffer dont ce compteur est à zéro peut voir son contenu remplacé par un nouveau bloc.
- *Clocksweep* : on parle aussi d'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 dont le compteur d'utilisation est à zéro. Tout buffer visité voit son compteur décrémenté de un. 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.

3.5.2 SYNCHRONISATION EN ARRIÈRE PLAN

- Le *Background Writer* synchronise les buffers :
  - de façon anticipée
  - une portion des pages à synchroniser
  - paramètres : `bgwriter_delay`, `bgwriter_lru_maxpages`, `bgwriter_lru_multiplier` et `bgwriter_flush_after`
- Le *checkpointer* synchronise les buffers :
  - lors des checkpoints
  - synchronise toutes les dirty pages

Afin de limiter les attentes des sessions interactives, PostgreSQL dispose de deux processus, le *Background Writer* et le *Checkpointer*, tous deux essayant d'effectuer de façon asynchrone les écritures des buffers sur le disque. Le but étant que les temps de traitement ressentis par les utilisateurs soient les plus courts possibles, et d'essayer de lisser les écritures sur de plus grandes plages de temps (pour ne pas saturer les disques).

Le *Background Writer* est donc responsable d'un mécanisme :

Il anticipe les besoins de buffers des sessions. À intervalle régulier, il se réveille et synchronise un nombre de buffers proportionnel à l'activité sur l'intervalle précédent, dans ceux qui seront examinés par les sessions pour les prochaines allocations. Trois paramètres

sont responsables de ce 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 ailleurs) : à partir de quelle quantité de données écrites une synchronisation sur disque est demandée.

Le *Checkpointer* est responsable d'un autre mécanisme :

Il synchronise tous les buffers *dirty* lors des checkpoints. Son rôle est d'effectuer cette synchronisation, en évitant de saturer les disques. Le paramètre `checkpoint_completion_target` (par défaut 0.5) est la cible en temps entre deux checkpoints à viser pour l'accomplissement de chaque checkpoint. Par exemple, à 0.5, un checkpoint devrait se terminer à la moitié du temps séparant habituellement deux checkpoints. À la fois `checkpoint_timeout` et `checkpoint_segments` sont pris en compte dans le calcul, et le *Background Writer* tente de se terminer avant que `checkpoint_timeout` x `checkpoint_completion_target` et que `checkpoint_segments` x `checkpoint_completion_target` ne soient atteints. La synchronisation sur disque ne se fait qu'à partir d'une certaine quantité de données écrites, dépendant du paramètre `checkpointer_flush_after` (par défaut 256 ko sous Linux, 0 ailleurs).

Les valeurs par défaut sont la plupart du temps satisfaisantes. Toutefois, passer `checkpoint_completion_target` à 0.9 (sa valeur maximale) améliore parfois les performances, sans ralentissement à craindre.

Pour les paramètres `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier`, *lru* signifie *Least Recently Used* que l'on pourrait traduire par « moins récemment utilisé ». Ainsi, pour ce mécanisme, le *Background Writer* synchronisera les pages qui ont été utilisées le moins récemment.

Les paramètres `bgwriter_flush_after` et `checkpointer_flush_after` permettent de configurer une synchronisation automatique sur disque plutôt que de laisser ce soin au système d'exploitation et à sa configuration.

Nous expliquerons plus en détail le fonctionnement d'un checkpoint dans le chapitre suiv-

17.12

ant.

Noter qu'avant la 9.2 les deux rôles étaient assurés par le seul *Background Writer*.

3.6 JOURNALISATION

- Garantir la durabilité des données
- Base encore cohérente après :
  - Un arrêt brutal des processus
  - Un crash machine
  - ...
- Appelée `Write Ahead Logging`
- Écriture des modifications dans un journal avant de les effectuer.

La journalisation, sous PostgreSQL, permet de garantir l'intégrité des fichiers, et la durabilité des opérations :

- L'intégrité : quoi qu'il arrive, excepté la perte des disques de stockage bien sûr, la base reste cohérente. Un arrêt d'urgence ne corrompra pas la base.
- Toute donnée validée (`COMMIT`) est écrite. Un arrêt d'urgence ne va pas la faire disparaître.

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 que dans les *shared buffers*. Elles seront écrites de façon asynchrone, soit par un processus recherchant un buffer libre, soit par le *Background Writer*.

Les écritures dans le journal, bien que synchrones, sont relativement performantes, car elles sont séquentielles (moins de déplacement de têtes pour les disques).

3.6.1 CHECKPOINT

- Point de reprise
- D'où doit-on rejouer le journal ?
- Données écrites au moins au niveau du checkpoint
- Deux moyens pour déclencher automatiquement et périodiquement un CHECKPOINT
  - `max_wal_size`
  - `checkpoint_timeout`

- Dilution des écritures
  - `checkpoint_completion_target`

PostgreSQL trace les modifications de données dans les journaux *WAL*. Si le système ou la base 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*.

- 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 de 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 on doit rejouer les journaux. Il faut donc pouvoir garantir que tous les buffers *dirty* **au démarrage du checkpoint** auront été synchronisés sur le disque quand le checkpoint sera terminé, et donc 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.

Les paramètres suivants ont une influence sur les checkpoints :

- `min_wal_size` : quantité de WAL conservés pour le recyclage. Par défaut 80 Mo ;
- `max_wal_size` : quantité maximale de WAL avant un checkpoint. Par défaut 1 Go (Attention : le terme peut porter à confusion, le volume de WAL peut dépasser `max_wal_size` en cas de forte activité, ce n'est pas une valeur plafond.) ;
- `checkpoint_timeout` : le temps maximum en secondes entre deux checkpoints. Par défaut, 300 secondes ;
- `checkpoint_completion_target` : la fraction de l'espace entre deux checkpoints que doit prendre l'écriture d'un checkpoint. Par défaut, 0.5, ce qui signifie qu'un checkpoint se termine quand la moitié de `max_wal_size` a été écrit ou la moitié de `checkpoint_timeout` est écoulé (c'est le premier atteint qui prime). La valeur préconisée est 0.9, car elle permet de lisser davantage les écritures dues aux checkpoints dans le temps ;
- `checkpoint_warning` : si deux checkpoints sont rapprochés d'un intervalle de

temps inférieur à celui-ci, un message d'avertissement sera écrit dans le journal. Par défaut, 30 secondes ;

- `checkpointer_flush_after` : quantité de données écrites à partir de laquelle une synchronisation sur disque est demandée.

Les paramètres `min_wal_size` et `max_wal_size` sont apparus avec la version 9.5. Auparavant existait le paramètre `checkpoint_segments`. Il avait pour valeur le nombre de segments de journaux maximum entre deux checkpoints. Sa valeur par défaut était de 3 mais une valeur comprise entre 10 et 20 était généralement recommandée.

Le paramétrage du système d'exploitation peut aussi avoir une influence sur les checkpoints. Le *Checkpointer* envoie ses écritures au système d'exploitation au fil de l'eau, mais doit effectuer un appel *fsync* (demande de synchronisation sur disque) pour les fichiers de la base, à la fin du checkpoint. Si le système a mis beaucoup de données en cache, cette dernière phase peut déclencher un fort pic d'activité. Ceci se paramètre sous Linux en abaissant les valeurs des *sysctl* `vm.dirty_*`.

3.6.2 WAL BUFFERS : JOURNALISATION EN MÉMOIRE

- Réduire les appels à fsync
- Mutualiser les écritures entre transactions
- Un processus d'arrière plan
- Paramètres importants :
  - `wal_buffers`
  - `wal_writer_delay`
  - `wal_writer_flush_after`
  - `synchronous_commit`
- Attention au paramètre `fsync`

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 ces buffers. Ces *buffers*, ou *WAL buffers*, sont vidés quand une session demande validation de son travail (`COMMIT`), ou quand il n'y a plus de buffer disponible.

Écrire un bloc ou plusieurs 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 *WAL Writer* écrit à intervalle régulier tous les buffers à synchroniser de *WAL buffers*.

Les paramètres relatifs à ceci sont :

- `wal_buffers` : la taille des *WAL buffers*. On conservera en général la valeur par défaut -1 qui lui affectera 1/32e de `shared_buffers` avec un maximum de 16 Mo (la taille d'un segment) ; sauf avant la 9.1 où on fournira cette valeur ; des valeurs supérieures peuvent être intéressantes pour les très grosses charges ;
- `wal_writer_delay` : l'intervalle auquel le *WAL Writer* se réveille pour écrire les buffers non synchronisés. Par défaut 200ms.
- `wal_sync_method` : l'appel système à utiliser pour demander l'écriture synchrone. Sauf très rare exception, PostgreSQL détecte tout seul le bon appel système à utiliser.
- `synchronous_commit` : la validation de la transaction en cours doit-elle déclencher une écriture synchrone dans le journal. C'est un paramètre de session, qui peut être modifié à la volée par une commande `SET`. Il est donc possible de le désactiver si on peut accepter une perte de données de `3 × wal_writer_delay` ou de `wal_writer_flush_after` octets écrits. La base restera, quoi qu'il arrive, cohérente.
- `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`.
- `wal_compression` (9.5+) : Active la compression des blocs complets enregistrés dans les journaux de transactions. Ce paramètre permet de réduire le volume de WAL et la charge en écriture sur les disques. Le rejeu des WAL est plus rapide, ce qui améliore la réplication et le processus de rejeu après un crash. Cette option entraîne aussi une augmentation de la consommation des ressources CPU pour la compression.
- `commit_delay`, `commit_siblings` : mécanisme de regroupement de transactions. Si on a au moins `commit_siblings` transactions en cours, attendre `commit_delay` (en microsecondes) au moment de valider une transaction pour permettre à d'autres transactions de s'y rattacher. Ce mécanisme est désactivé par défaut, et apporte un gain de performances minime.
- `fsync` : doit-on réellement effectuer les écritures synchrones ? S'il est passé à `off`, les performances sont très accélérées. Par contre, les données seront totalement corrompues si le serveur subit un arrêt d'urgence. Il n'est donc intéressant de le passer à `off` que très temporairement, pendant le chargement initial d'un cluster

par exemple.

3.6.3 ARCHIVAGE : CONSERVATION DES JOURNAUX

- Récupération à partir de vieille sauvegarde
- Sauvegarde à chaud
- Sauvegarde en continu
- Paramètres : `wal_level`, `archive_mode`, `archive_command` et `archive_timeout`

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 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**, bien plus ancienne. Le mécanisme d'archivage permet donc de repartir d'une sauvegarde binaire de la base (les fichiers, pas un `pg_dump`), et de réappliquer le contenu des journaux archivés.

Ce mécanisme permet par ailleurs d'effectuer une sauvegarde **base ouverte** (c'est-à-dire pendant que les fichiers de la base sont en cours de modification). Il suffit de rejouer tous les journaux depuis le checkpoint précédent la sauvegarde jusqu'à la fin de la sauvegarde. 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. En effet, rien n'oblige à rejouer tout ce qui se trouve dans l'archive. Lors de la récupération, on peut préciser le point exact (en temps ou en numéro de transaction) où l'on souhaite arrêter la récupération. Une base en archivage peut donc être restaurée **à n'importe quel point dans le temps**. On parle aussi de *Point In Time Recovery* (récupération à un point dans le temps) dans la documentation.

Ce mécanisme permet enfin de créer une copie de la base de production, en transférant les fichiers archivés et en les appliquant sur cette seconde base. Il suffit de restaurer une sauvegarde à chaud de la base de production sur le serveur dit « de standby », puis d'appliquer les journaux sur cette base de standby au fur et à mesure de leur génération.

Tout ceci est revu dans le module « Point In Time Recovery ».

Les paramètres associés sont :

- `wal_level` : `minimal` (défaut jusqu'en version 9.6 incluse), `replica` (défaut en 10) ou `logical`, suivant ce qu'on souhaite faire des journaux : juste récupérer suite à un

crash, les archiver ou alimenter une base de *Hot Standby*, ou avoir de la réplication logique.

- `archive_mode` : (`off` par défaut). L'archivage doit-il être activé.
- `archive_command` : la commande pour archiver un journal, accepte des « jokers », `rsync` très fortement recommandé.
- `archive_timeout` : au bout de combien de temps doit-on forcer un changement de journal, même s'il n'est pas fini, afin de l'archiver.

Avant la version 9.6, il existait deux niveaux intermédiaires pour le paramètre `wal_level` : `archive` et `hot_standby`. Le premier permettait seulement l'archivage, le second permettait en plus d'avoir un serveur secondaire en lecture seule. Ces deux valeurs ont été fusionnées en `replica` avec la version 9.6. Les anciennes valeurs sont toujours acceptées, et remplacées silencieusement par la nouvelle valeur.

3.6.4 STREAMING REPLICATION

- Appliquer les journaux :
  - Non plus fichier par fichier
  - Mais entrée par entrée (en flux continu)
  - Base de *Standby* très proche de la production
  - Paramètres : `max_wal_senders`, `wal_keep_segments`, `wal_sender_delay` et `wal_level`

Le mécanisme de *Streaming Replication* a été une des nouveautés majeures de la version 9.0. Il n'est plus nécessaire d'attendre qu'un fichier de journal soit généré entièrement sur le maître pour appliquer son contenu sur l'esclave. Celui-ci se connecte sur le maître, et lui demande à intervalle régulier de lui envoyer tout ce qui a été généré depuis la dernière interrogation.

Voici les paramètres concernés :

- `max_wal_senders` : le nombre maximum de processus d'envoi de journaux démarrés sur le maître (1 par esclave).
- `wal_keep_segments` : le nombre de journaux à garder en ligne pour pouvoir les envoyer aux esclaves, même si ils ne sont plus nécessaires à la récupération sur le maître (ils sont antérieurs au dernier checkpoint) ;
- `wal_sender_delay` : la fréquence à laquelle le maître se réveille pour envoyer les nouvelles entrées de journal aux esclaves ;
- `wal_level` : doit être au moins à archive, pour pouvoir être envoyé aux esclaves.

17.12

Et un fichier `recovery.conf` correctement paramétré sur l'esclave, indiquant comment récupérer les fichiers de l'archive, et comment se connecter au maître pour récupérer des journaux.

3.6.5 HOT STANDBY

- Base de *Standby* accessible en lecture
- Peut basculer en lecture/écriture sans redémarrage (sessions conservées)
- Peut fonctionner avec la *Streaming Replication*
- Paramètres
  - sur l'esclave :

`hot_standby`, `max_standby_archive_delay`, `max_standby_streaming_delay`

  - sur le maître :

`wal_level`

L'autre importante nouveauté de PostgreSQL 9.0 a été la possibilité pour une base de *Standby* d'être accessible en lecture seule, alors qu'elle applique les journaux provenant de la base maître. Il est possible que des modifications devant être appliquées par les journaux entrent en conflit avec les requêtes en cours. Le retard acceptable est déclaré par la configuration.

Voici les paramètres concernés :

- `hot_standby` : `on` ou `off` Active ou non la fonctionnalité lorsque la base est en récupération.
- `max_standby_archive_delay` : quel est le délai maximum acceptable dans l'application de journaux venant de l'archive? Passé ce délai, les requêtes gênantes sur l'esclave seront automatiquement annulées.
- `max_standby_streaming_delay` : quel est le délai maximum acceptable dans l'application de journaux venant par le mécanisme de Streaming Replication? Passé ce délai, les requêtes gênantes sur l'esclave seront automatiquement annulées.

Et sur le maître :

- `wal_level` : doit être à `replica` afin de générer quelques informations supplémentaires à destination de la base de *Standby*

Et un fichier `recovery.conf` correctement paramétré sur l'esclave.

3.7 STATISTIQUES

- Collecte de deux types de statistiques différents :
  - Statistiques d'activité
  - Statistiques sur les données

PostgreSQL collecte deux types de statistiques différentes :

- Les statistiques d'activité : elles permettent de mesurer l'activité de la base.
  - 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 du être demandés au système d'exploitation ?
  - Quelles sont les requêtes en cours d'exécution ?
  - Combien de buffers ont été écrits par le *Background Writer* ? Par les processus eux-mêmes ? durant un checkpoint?
- Les statistiques sur les données : elles sont utilisées par l'optimiseur de requêtes dans sa recherche du meilleur plan d'exécution. Elles contiennent entre autres les informations suivantes :
  - Taille des tables
  - Taille moyenne d'un enregistrement de table
  - Taille moyenne d'un attribut

3.7.1 STATISTIQUES SUR L'ACTIVITÉ

- Collectées par chaque session durant son travail
- Remontées au *Stats Collector*
- Stockées régulièrement dans un fichier, consultable par des vues systèmes
- Paramètres :
  - `track_activities`, `track_activity_query_size`, `track_counts`, `track_io_timing` et `track_functions`
  - `update_process_title` et `stats_temp_directory`

Chaque session collecte des statistiques, dès qu'elle effectue une opération.

Ces informations, si elles sont transitoires, comme la requête en cours, sont directement stockées dans la mémoire partagée de PostgreSQL.

Si elles doivent être agrégées et stockées, elles sont remontées au processus responsable de cette tâche, le *Stats Collector*.

Voici les paramètres concernés :

- `track_activities` : les processus doivent-ils mettre à jour leur activité dans `pg_stat_activity`? (`on` par défaut)
- `track_activity_query_size` : quelle est la taille maximum du texte de requête pouvant être stocké dans `pg_stat_activity` (1024 caractères par défaut, que l'on doit souvent monter vers 10000 si les requêtes sont longues) ; nécessite un redémarrage)
- `track_counts` : les processus doivent-ils collecter des informations sur leur activité ? (`on` par défaut)
- `track_io_timing` : les processus doivent-ils collecter des informations de chronométrage sur les lectures et écritures ? Ce paramètre complète les champs `blk_read_time` et `blk_write_time` dans `pg_stat_database`, `pg_stat_statements` (mêmes champs) et les plans d'exécutions appelés avec `EXPLAIN (ANALYZE,BUFFERS)` (`off` par défaut ; avant de l'activer sur une machine peu performante, vérifiez l'impact avec l'outil `pg_test_timing`)
- `track_functions` : les processus doivent-ils aussi collecter des informations sur l'exécution des procédures stockées (`none` par défaut, `pl` pour ne tracer que les procédures en langages procéduraux, `all` pour tracer aussi les procédures en `C` et en `SQL`)
- `update_process_title` : le titre du processus (visible par exemple avec `ps -ef` sous Unix) sera modifié si cette option est à `on` (défaut sous Unix ; mettre à `off` sous Windows pour des raisons de performance)
- `stats_temp_directory` : le fichier de statistiques pouvant être source de contention (il peut devenir gros, et est réécrit fréquemment), il peut être stocké ailleurs que dans le répertoire de l'instance PostgreSQL, par exemple sur un *ramdisk* ou *tmpfs* (défaut sous Debian).

3.7.2 STATISTIQUES COLLECTÉES

- 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

Pour les statistiques aux 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 statistiques accessibles sont :

- Les accès logiques aux objets (tables, index et fonctions). Ce sont les vues `pg_stat_xxx_tables`, `pg_stat_xxx_indexes` et `pg_stat_user_functions`.
- Les accès physiques aux objets (tables, index et séquences). Ce sont les vues `pg_statio_xxx_tables`, `pg_statio_xxx_indexes`, et `pg_statio_xxx_sequences`.
- `pg_stat_bgwriter` stocke les statistiques d'écriture des buffers : par le *Background Writer* en fonctionnement normal, par le *Background Writer* durant les checkpoints, ou par les sessions elles-mêmes.
- Des statistiques globales par base sont aussi disponibles, dans `pg_stat_database` : le nombre de transactions validées et annulées, le nombre de sessions en cours, et quelques statistiques sur les accès physiques et en cache, ainsi que sur les opérations logiques.
- Les statistiques sur les conflits entre application de la réplication et requêtes en lecture seule sont disponibles dans `pg_stat_database_conflicts`.
- `pg_stat_activity` donne des informations sur les processus en cours sur l'instance, que ce soit des processus en tâche de fond ou des processus backend : numéro de processus, adresse et port, date de début d'ordre, de transaction, de session, requête en cours, état, ordre SQL et nom de l'application si elle l'a renseigné (avant la version 10, cette vue n'affichait que les processus backend ; à partir de la version 10 apparaissent des workers, le checkpointer, le walwriter...).
- `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_replication` donne des informations sur les esclaves connectés.

3.7.3 STATISTIQUES SUR LES DONNÉES

- Statistiques sur les données :
  - Collectées par échantillonnage
  - Table par table (et pour certains index)
  - Colonne par colonne

- Pour de meilleurs plans d'exécution

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.

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

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`).

La vue `pg_stats` affiche les statistiques collectées :

```
postgres=# \d pg_stats
                        Vue « pg_catalog.pg_stats »
        Colonne         |   Type   | Collationnement | NULL-able | Par défaut
------------------------+----------+-----------------+-----------+------------
 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 nuls.
- `avg_width` : taille moyenne de cet attribut dans l'échantillon collecté.

- `n_distinct` : si positif, nombre de valeurs distinctes, si négatif, fraction de valeurs distinctes pour cette colonne dans la table. On peut forcer le nombre de valeurs distinctes, si on constate que la collecte des statistiques n'y arrive pas : `ALTER TABLE xxx ALTER COLUMN yyy SET (n_distinct = -0.5) ; ANALYZE xxx;` par exemple indique à l'optimiseur que chaque valeur apparaît statistiquement deux fois.
- `most_common_vals` et `most_common_freqs` : les valeurs les plus fréquentes de la table, et leur fréquence. Le nombre de valeurs collecté est au maximum celle indiquée par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target`.
- `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 on 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 collecté est au maximum celle indiquée 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.

3.8 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. On ne précise pas au moteur comment accéder aux tables `path` et `file` (par index ou parcours complet par exemple), ni comment effectuer la jointure (il existe plusieurs méthodes pour PostgreSQL). 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 sur file.pathid, sur path.pathid ?
- quelles sont les tailles des deux tables ?

On comprend bien que la stratégie la plus efficace ne sera pas la même suivant les informations retournées par toutes ces questions…

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 (c'est appelé 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 : on pourrait 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 appelé un *nested loop*).

3.8.1 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 :
  - `seq_page_cost`, `random_page_cost`, `cpu_tuple_cost`, `cpu_index_tuple_cost` et `cpu_operator_cost`
  - `parallel_setup_cost` et `parallel_tuple_cost`
  - `effective_cache_size`

Afin de choisir un bon plan, le moteur essaye 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 informations à effectuer. Ces informations sont les suivantes :
  - `seq_page_cost` : coût de la lecture d'une page disque de façon séquentielle (lors d'un parcours séquentiel de table par exemple). Par défaut `1.0`.
  - `random_page_cost` : coût de la lecture d'une page disque de façon aléatoire (lors d'un accès à une page d'index par exemple). Par défaut `4.0`.
  - `cpu_tuple_cost` : coût de traitement par le processeur d'un enregistrement de table. Par défaut `0.01`.
  - `cpu_index_tuple_cost` : coût de traitement par le processeur d'un enregistrement d'index. Par défaut `0.005`.
  - `cpu_operator_cost` : coût de traitement par le processeur de l'exécution d'un opérateur. Par défaut `0.0025`.

Ce sont les coûts relatifs de ces différentes opérations qui sont importants : l'accès à une page de façon aléatoire est par exemple 4 fois plus coûteuse que de façon séquentielle, du fait du déplacement des têtes de lecture sur un disque dur. `seq_page_cost`, `random_page_cost` et `effective_io_concurrency` peuvent être paramétrés par tablespace (depuis la version 9.0 de PostgreSQL pour les deux premiers, et depuis la version 9.6 pour le dernier), afin de refléter les caractéristiques de disques différents. Sur une base **fortement en cache**, on peut donc 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.

La mise en place du parallélisme 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 un autre processus a également un coût représenté par le paramètre `parallel_tuple_cost`.

Ainsi une lecture complète d'une grosse table peut être moins couteuse 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 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 du cache du système d'exploitation. 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, 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 exhaustive 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[10] issu de la base de connaissance Dalibo.

3.8.2 PARAMÈTRES SUPPLÉMENTAIRES DE L'OPTIMISEUR 1/2

- Pour le partitionnement : `constraint_exclusion`
- Pour limiter les réécritures : `from_collapse_limit` et `join_collapse_limit`
- Pour les curseurs : `cursor_tuple_fraction`
- Pour mutualiser les entrées-sorties : `synchronize_seqscans`

Tous les paramètres suivants peuvent être modifiés par session.

Pour partitionner une table sous PostgreSQL, on crée une table parente et des tables filles héritent de celle-ci. Sur ces tables filles, on définit des contraintes `CHECK`, que tous les enregistrements de la table fille doivent vérifier. Ce sont les 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 parcoure 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 limiter la complexité des plans d'exécution à étudier, il est possible de limiter la quantité de réécriture autorisée par l'optimiseur via les paramètres `from_collapse_limit` et `join_collapse_limit`. Le premier interdit que plus de 8 (par défaut) tables provenant d'une sous-requête ne soient déplacées dans la requête principale. Le second interdit que

[10] https://support.dalibo.com/kb/cout_planification

plus de 8 (par défaut) tables provenant de clauses `JOIN` ne soient déplacées vers la clause `FROM`. Ceci réduit la qualité du plan d'exécution généré, mais permet qu'il soit généré dans un temps fini.

Lors de l'utilisation de curseurs, le moteur n'a aucun moyen de connaître le nombre d'enregistrements que souhaite récupérer réellement l'utilisateur. Il est tout-à-fait possible que seuls les premiers enregistrements générés soient récupérés. Et 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.

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`.

3.8.3 PARAMÈTRES SUPPLÉMENTAIRES DE L'OPTIMISEUR 2/2

- GEQO :
  - Un optimiseur génétique
  - État initial, puis mutations aléatoires
  - Rapide, mais non optimal
  - Paramètes : `geqo` et `geqo_threshold`

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.

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe (voir la documentation dans le code source de PostgreSQL[11]).

[11] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f

Tant que le paramètre `geqo_seed` ainsi que les autres paramètres contrôlant GEQO restent inchangés, le plan obtenu pour une requête donnée restera inchangé. Il est possible de faire varier la valeur de `geqo_seed` pour obtenir d'autres plans (voir la documentation officielle[12]).

Il est configuré par les paramètres dont le nom commence par geqo dans le moteur. Excepté `geqo_threshold` et `geqo`, il est déconseillé de modifier les paramètres sans une grande connaissance des algorithmes génétiques.

- `geqo`, par défaut à `on`, permet de désactiver complètement 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.

3.8.4 DÉBOGGAGE 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 :

```
marc=# EXPLAIN ANALYZE SELECT * FROM test WHERE a<3;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..84641.51 rows=4999971 width=4)
                   (actual time=0.011..454.286 rows=5000003 loops=1)
   Filter: (a < 3)
   Rows Removed by Filter: 998
 Planning time: 0.248 ms
 Execution time: 601.176 ms
(5 rows)
```

Le moteur a choisi un parcours séquentiel de table. Si on veut vérifier qu'un parcours par l'index sur la colonne a n'est pas plus rentable :

```
marc=# SET enable_seqscan TO off;
SET
marc=# SET enable_indexonlyscan TO off;
```

[12] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116517

```
SET
marc=# SET enable_bitmapscan TO off;
SET
marc=# EXPLAIN ANALYZE SELECT * FROM test WHERE a<3;
                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using test_a_idx on test
        (cost=0.43..164479.92 rows=4999971 width=4)
        (actual time=0.297..669.051 rows=5000003 loops=1)
   Index Cond: (a < 3)
 Planning time: 0.065 ms
 Execution time: 816.566 ms
(4 rows)
```

Attention aux effets du cache : le parcours par index est ici relativement performant parce que les données ont été trouvées dans le cache disque. La requête, sinon, aurait été bien plus coûteuse. 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 on supprime l'index, on constate que le *sequential scan* n'a pas été désactivé. Il a juste été rendu très coûteux par ces options de déboggage :

```
marc=# DROP INDEX test_a_idx ;
DROP INDEX
marc=# EXPLAIN ANALYZE SELECT * FROM test WHERE a<3;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on test  (cost=10000000000.00..10000084641.51 rows=5000198 width=4)
                   (actual time=0.012..455.615 rows=5000003 loops=1)
   Filter: (a < 3)
   Rows Removed by Filter: 998
 Planning time: 0.046 ms
 Execution time: 603.004 ms
(5 rows)
```

Le « très coûteux » est un coût majoré de 10 000 000 000 pour l'exécution d'un nœud interdit.

Voici la liste des options de désactivation :

- enable_bitmapscan
- enable_hashagg
- enable_hashjoin
- enable_indexonlyscan
- enable_indexscan

- enable_material
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort
- enable_tidscan

3.9 GESTION DES CONNEXIONS

- L'accès à la base se fait par un protocole réseau clairement défini :
  - Sur des sockets TCP (IPV4 ou IPV6)
  - Sur des sockets Unix (sous Unix uniquement)
- Les demandes de connexion sont gérées par le *postmaster*.
- Paramètres : port, listen_adresses, unix_socket_directory, 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_directory : le répertoire où sera stockée 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.

3.9.1 PARAMÈTRES LIÉS AUX SOCKETS TCP

- Paramètres de keepalive TCP
  - tcp_keepalives_idle
  - tcp_keepalives_interval
  - tcp_keepalives_count
- Paramètres SSL
  - ssl
  - ssl_ciphers

  - `ssl_renegotiation_limit`
- Autres paramètres

On peut 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 session 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 acceptés 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.

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[13] .

De nombreux autres paramètres sont disponibles, qui configurent principalement l'authentification (les paramètres ldap ou GSSAPI par exemple), mais aussi le paramétrage de l'authentification proprement dite, pilotée par le fichier de configuration `pg_hba.conf`.

3.10 CONCLUSION

- PostgreSQL est un SGBD complet.
- Cela impose une conception complexe, et l'interaction de nombreux composants.

[13] http://docs.postgresql.fr/current/ssl-tcp.html

- Une bonne compréhension de cette architecture est la clé d'une bonne administration :
  - Le paramétrage est compris
  - La supervision est plus rigoureuse
- Le second module traite de la gestion des transactions (MVCC).

3.10.1 QUESTIONS

N'hésitez pas, c'est le moment !

3.11 TRAVAUX PRATIQUES

3.11.1 ÉNONCÉS

Processus

- Lancez PostgreSQL (si ce n'est pas déjà fait).
- Listez les processus du serveur PostgreSQL.
- Qu'observe-t-on ?
- Ouvrez une connexion
- Listez de nouveau les processus du serveur.
- Qu'observe-t-on ?
- Créez une table et ajoutez-y beaucoup de lignes.
- Pendant l'insertion, listez de nouveau les processus du serveur.
- Qu'observe-t-on ?
- Configurez max_connections à 11.
- Redémarrez PostgreSQL.
- Connectez-vous cinq fois à PostgreSQL.
- Essayez une sixième connexion
- Qu'observe-t-on ?

Mémoire partagée

- Créez une table avec une colonne `id` de type integer.
- Insérez 500 lignes (pensez à utiliser generate_series)
- Réinitialisez les statistiques pour t2 uniquement.
- Redémarrez PostgreSQL (pour vider le cache).
- Lisez la table entière
- Récupérez les statistiques IO pour cette table (pensez à utiliser pg_statio_user_tables)
- Qu'observe-t-on ?
- Lisez la table entière une deuxième fois et récupérez les statistiques IO pour cette table (pensez à utiliser pg_statio_user_tables).
- Qu'observe-t-on ?
- Lisez la table entière une troisième fois, à partir d'une autre session, et récupérez les statistiques IO pour cette table (pensez à utiliser pg_statio_user_tables).
- Qu'observe-t-on ?

Mémoire par processus

- Activez la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (vous pouvez le faire sur la session uniquement).
- Insérez un million de lignes dans la table précédente.
- Listez les données de la table en triant par la colonne.
- Qu'observe-t-on ?
- Augmentez la valeur du paramètre `work_mem`.
- Listez les données de la table en triant par la colonne.
- Qu'observe-t-on ?

Fichiers

- Allez dans le répertoire des données.
- Listez les fichiers.
- Allez dans base.
- Listez les fichiers.
- À quelle base est lié chaque répertoire de base ? (oid2name ou pg_database)

17.12

- Créez une nouvelle base.
- Qu'est-il survenu dans le répertoire base ?
- Connectez-vous sur cette nouvelle base et créez une table avec une seule colonne.
- Récupérer le chemin vers le fichier correspond à cette table.
- Regardez la taille du fichier.
- Pourquoi est-il vide ?
- Ajoutons une ligne.
- Quelle taille fait le fichier ?
- Ajoutons 500 lignes.
- Quelle taille fait le fichier ?
- Pourquoi cette taille pour simplement 501 fois un entier (ie, 4 octets) ?.
- Ajoutez un tablespace.
- Ajoutez-y une table.
- Récupérer le chemin vers le fichier correspond à cette table.
- Insérer dix millions de lignes dans la table t2.
- Que se passe-t-il au niveau du répertoire pg_wal ?
- Exécutez un CHECKPOINT.
- Que se passe-t-il au niveau du répertoire pg_wal ?

Cache disque de PostgreSQL

- Installez l'extension de pg_buffercache
- Redémarrez PostgreSQL
- Videz le cache système
- Que contient le cache de PostgreSQL ?
- Lisez complètement t2, en récupérant la durée d'exécution de la requête
- Que contient le cache de PostgreSQL ?
- Extrayez de nouveau toutes les données de la table t2
- Que contient le cache de PostgreSQL ?
- Changez shared_buffers, puis redémarrez PostgreSQL.

- Extrayez de nouveau toutes les données de la table t2
- Que contient le cache de PostgreSQL ?
- Faire une mise à jour.
- Que contient le cache de PostgreSQL ?
- Faites un CHECKPOINT
- Que contient le cache de PostgreSQL ?

Statistiques d'activités

- Créer une table
- Insérer des données
- Lire ses statistiques d'activité

Statistiques sur les données

- Créer une table avec une seule colonne de type integer
- Empêcher autovacuum d'analyser automatiquement la table
- Mettre des données différentes sur c1
- Lire la table avec un filtre sur c1
- Exécuter la commande ANALYZE
- Lire la table avec un filtre sur c1. Que constatez-vous
- Ajouter un index sur c1
- Lire la table avec un filtre sur c1
- Modifier la colonne c1 avec la valeur 1 pour toutes les lignes
- Lire la table avec un filtre sur c1
- Exécuter la commande ANALYZE
- Lire la table avec un filtre sur c1

3.11.2 SOLUTIONS

Processus

Lancez PostgreSQL (si ce n'est pas déjà fait).

```
/etc/init.d/postgresql start
```

17.12

Listez les processus du serveur PostgreSQL.

```
$ ps -o pid,cmd fx
  PID CMD
 1792 -bash
 2324  \_ ps -o pid,cmd fx
 1992 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
 1994  \_ postgres: logger process
 1996  \_ postgres: checkpointer process
 1997  \_ postgres: writer process
 1998  \_ postgres: wal writer process
 1999  \_ postgres: autovacuum launcher process
 2000  \_ postgres: stats collector process
 2001  \_ postgres: bgworker: logical replication launcher
```

Qu'observe-t-on ?

Les processus de gestion du serveur PostgreSQL.

Ouvrez une connexion

```
$ psql postgres
psql (10)
Type "help" for help.

postgres=#
```

Listez de nouveau les processus du serveur.

```
$ ps -o pid,cmd fx
  PID CMD
 2031 -bash
 2326  \_ psql postgres
 1792 -bash
 2328  \_ ps -o pid,cmd fx
 1992 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
 1994  \_ postgres: logger process
 1996  \_ postgres: checkpointer process
 1997  \_ postgres: writer process
 1998  \_ postgres: wal writer process
 1999  \_ postgres: autovacuum launcher process
 2000  \_ postgres: stats collector process
 2001  \_ postgres: bgworker: logical replication launcher
```

```
 2327  \_ postgres: postgres postgres [local] idle
```

Qu'observe-t-on ?

Il y a un nouveau processus (PID 457) qui va gérer l'exécution des requêtes du client psql.

Créez une table et ajoutez-y beaucoup de lignes.

```
postgres=# CREATE TABLE t1 (id integer);
CREATE TABLE
postgres=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
```

Pendant l'insertion, listez de nouveau les processus du serveur.

```
$ ps -o pid,cmd fx    PID CMD  2031 -bash  2326  \_ psql postgres  1792
-bash  2363  \_ ps -o pid,cmd fx  1992 /usr/pgsql-10/bin/postmaster -D
/var/lib/pgsql/10/data  1994  \_ postgres: logger process
\_ postgres: checkpointer process                          1997  \_
postgres: writer process                                1998  \_ postgres:
wal writer process                              1999  \_ postgres: autovacuum
launcher process                   2000  \_ postgres: stats collector
process                       2001  \_ postgres: bgworker: logical
replication launcher        2327  \_ postgres: postgres postgres [local]
INSERT
```

Qu'observe-t-on ?

Le processus serveur exécute l'INSERT, ce qui se voit au niveau du nom du processus. Seul l'ordre SQL est affiché (ie, le mot INSERT et non pas la requête complète).

Configurez `max_connections` à 11.

Pour cela, il faut ouvrir le fichier de configuration `postgresql.conf` et modifier la valeur du paramètre `max_connections` à 11.

Redémarrez PostgreSQL.

```
# service postgresql-10 restart
```

Connectez-vous cinq fois à PostgreSQL.

Très simple avec ce petit script shell :

```
$ for i in $(seq 1 11); do psql -c "SELECT pg_sleep(1000);" postgres & done
[1] 998
[2] 999
[3] 1000
[4] 1001
```

```
[5] 1002
[6] 1003
[7] 1004
[8] 1005
[9] 1006
[10] 1007
[11] 1008
```

Essayez une douzième connexion

```
$ psql postgres
psql: FATAL:  sorry, too many clients already
```

Qu'observe-t-on ?

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.

Mémoire partagée

Créez une table avec une colonne `id` de type integer.

```
$ psql postgres
psql (10)
Type "help" for help.

postgres=# \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | t1   | table | postgres
(1 row)

postgres=# CREATE TABLE t2 (id integer);
CREATE TABLE
```

Insérez 500 lignes (pensez à utiliser generate_series)

```
postgres=# INSERT INTO t2 SELECT generate_series(1, 500);
INSERT 0 500
```

Réinitialisez les statistiques pour t2 uniquement.

```
postgres=# SELECT pg_stat_reset_single_table_counters(oid) FROM pg_class
WHERE relname='t2';
 pg_stat_reset_single_table_counters
```

```
-----------------------------------
```

```
(1 row)
```

Redémarrez PostgreSQL (pour vider le cache).

```
# service postgresql-10 restart
```

Lisez la table entière

```
postgres=# SELECT * FROM t2;
[...]
```

Récupérez les statistiques IO pour cette table (pensez à utiliser pg_statio_user_tables)

```
postgres=# \x
Expanded display is on.
postgres=# 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   |
```

Qu'observe-t-on ?

3 blocs ont été lus en dehors du cache de PostgreSQL (colonne `heap_blks_read`).

Lisez la table entière une deuxième fois et récupérez les statistiques IO pour cette table (pensez à utiliser pg_statio_user_tables).

```
postgres=# \x
Expanded display is off.
postgres=# SELECT * FROM t2;
[...]
postgres=# 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
idx_blks_read   |
```

```
idx_blks_hit    |
toast_blks_read |
toast_blks_hit  |
tidx_blks_read  |
tidx_blks_hit   |
```

Qu'observe-t-on ?

Les trois blocs sont maintenant lus à partir du cache de PostgreSQL.

Lisez la table entière une troisième fois, à partir d'une autre session, et récupérez les statistiques IO pour cette table (pensez à utiliser pg_statio_user_tables).

```
postgres=# \x
Expanded display is off.
postgres=# SELECT * FROM t2;
[...]
postgres=# 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
idx_blks_read   |
idx_blks_hit    |
toast_blks_read |
toast_blks_hit  |
tidx_blks_read  |
tidx_blks_hit   |
```

Qu'observe-t-on ?

Quelque soit la session, le cache étant partagé, tout le monde profite des données en cache.

Mémoire par processus

Activez la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (vous pouvez le faire sur la session uniquement).

```
postgres=# SET client_min_messages TO log;
SET
postgres=# SET log_temp_files TO 0;
SET
```

Insérez un million de lignes dans la table précédente.

```
postgres=# INSERT INTO t2 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Listez les données de la table en triant par la colonne.

```
postgres=# SELECT * FROM t2 ORDER BY id;
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp1197.0", size 14032896
   id
---------
       1
       1
       2
       2
       3
[...]
```

Qu'observe-t-on ?

PostgreSQL a dû créer un fichier temporaire pour stocker le résultat temporaire du tri. Ce fichier s'appelle `/pgsql_tmp/pgsql_tmp11331.1`. Il est spécifique à la session et sera détruit dès qu'il ne sera plus utile. Il fait 14 Mo.

Augmentez la valeur du paramètre work_mem.

```
postgres=# SET work_mem TO '100MB';
SET
```

Listez les données de la table en triant par la colonne.

```
postgres=# SELECT * FROM t2 ORDER BY id;
   id
---------
       1
       1
       2
       2
       3
       3
[...]
```

Qu'observe-t-on ?

Il n'y a plus de fichier temporaire généré. La durée d'exécution est bien moindre.

Fichiers

Allez dans le répertoire des données.

```
$ cd $PGDATA
```

Listez les fichiers.

```
$ ll
total 140
```

```
drwx------ 9 postgres postgres  4096  8 sept. 09:52 base
-rw------- 1 postgres postgres    30 19 sept. 05:06 current_logfiles
drwx------ 2 postgres postgres  4096 19 sept. 05:06 global
drwx------ 2 postgres postgres  4096 15 sept. 05:00 log
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_commit_ts
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_dynshmem
-rw------- 1 postgres postgres  4381  8 sept. 07:24 pg_hba.conf
-rw------- 1 postgres postgres  1636  8 sept. 06:15 pg_ident.conf
drwx------ 4 postgres postgres  4096 19 sept. 05:06 pg_logical
drwx------ 4 postgres postgres  4096  8 sept. 06:12 pg_multixact
drwx------ 2 postgres postgres  4096 19 sept. 05:06 pg_notify
drwx------ 2 postgres postgres  4096  8 sept. 06:15 pg_replslot
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_serial
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_snapshots
drwx------ 2 postgres postgres  4096 19 sept. 05:06 pg_stat
drwx------ 2 postgres postgres  4096 19 sept. 05:10 pg_stat_tmp
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_subtrans
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_tblspc
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_twophase
-rw------- 1 postgres postgres     3  8 sept. 06:12 PG_VERSION
drwx------ 3 postgres postgres  4096  8 sept. 10:52 pg_wal
drwx------ 2 postgres postgres  4096  8 sept. 06:12 pg_xact
-rw------- 1 postgres postgres    88  8 sept. 06:15 postgresql.auto.conf
-rw------- 1 postgres postgres 22773 19 sept. 05:03 postgresql.conf
-rw------- 1 postgres postgres    57 19 sept. 05:06 postmaster.opts
-rw------- 1 postgres postgres   103 19 sept. 05:06 postmaster.pid
```

Allez dans base.

```
$ cd base
```

Listez les fichiers.

```
$ ll
total 44
drwx------ 2 postgres postgres  4096  8 sept. 06:12 1
drwx------ 2 postgres postgres 12288 19 sept. 05:08 13451
drwx------ 2 postgres postgres 12288 19 sept. 05:06 16384
drwx------ 2 postgres postgres  4096 19 sept. 05:08 pgsql_tmp
```

À quelle base est lié chaque répertoire de base ? (oid2name ou pg_database)

Chaque répertoire correspond à une base de données. Le numéro indiqué est un identifi-

ant système (OID). Il existe deux moyens pour récupérer cette information :

- directement dans le catalogue système `pg_database`

```
$ psql postgres
psql (10)
Type "help" for help.

postgres=# SELECT oid, datname FROM pg_database;
  oid  |  datname
-------+-----------
     1 | template1
 13451 | template0
 16384 | postgres
(3 rows)
```

- avec l'outil `oid2name`

```
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  13451       postgres  pg_default
  13450      template0  pg_default
      1      template1  pg_default
```

Done le répertoire `1` correspond à la base template1, le répertoire `13450` à la base template0 et le répertoire `13451` à la base postgres.

Créez une nouvelle base.

```
$ createdb b1
```

Qu'est-il survenu dans le répertoire base ?

```
$ ll
total 44
drwx------ 2 postgres postgres  4096  8 sept. 06:12 1
drwx------ 2 postgres postgres  4096  8 sept. 06:12 13450
drwx------ 2 postgres postgres 12288 19 sept. 05:13 13451
drwx------ 2 postgres postgres 12288 19 sept. 05:06 16384
drwx------ 2 postgres postgres  4096 19 sept. 05:08 pgsql_tmp
$ oid2name | grep b1
  16384             b1  pg_default
```

Un nouveau sous-répertoire est apparu, nommé `16384`. Il correspond bien à la base b1 d'après `oid2name`.

Connectez-vous sur cette nouvelle base et créez une table avec une seule colonne.

```
$ psql b1
psql (10)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);
CREATE TABLE
```

Récupérer le chemin vers le fichier correspond à cette table.

```
b1=# SELECT current_setting('data_directory')||'/'||pg_relation_filepath('t1');
                      ?column?
-----------------------------------------------------
 /var/lib/pgsql/10/data/base/16384/24724
(1 row)
```

Regardez la taille du fichier.

```
$ ll /var/lib/pgsql/10/data/base/16384/24724
-rw------- 1 postgres postgres 0  8 sept. 10:34 /var/lib/pgsql/10/data/base/16384/2472
```

Pourquoi est-il vide ?

La table vient d'être créée. Aucune donnée n'a encore été ajoutée. Les méta-données se trouvent sur d'autres tables (des catalogues systèmes). Donc il est logique que le fichier soit vide.

Ajoutons une ligne.

```
$ psql b1
psql (10)
Type "help" for help.

b1=# INSERT INTO t1 VALUES (1);
INSERT 0 1
```

Quelle taille fait le fichier ?

```
$ ll /var/lib/pgsql/10/data/base/16384/24724
-rw------- 1 postgres postgres 8192 19 sept. 05:24 /var/lib/pgsql/10/data/base/16384/2
```

Il fait 8 Ko. En fait, PostgreSQL travaille par bloc de 8 Ko. On ajoute une nouvelle ligne, il crée un nouveau bloc et y place la ligne. Les prochaines lignes iront dans le bloc, jusqu'à ce que ce dernier soit plein. Une fois que le bloc en cours est plein, il ajoute un nouveau bloc et y intègre les nouvelles lignes.

Ajoutons 500 lignes.

```
b1=# INSERT INTO t1 SELECT generate_series(1, 500);
INSERT 0 500
```

Quelle taille fait le fichier ?

```
[gui@localhost base]$ ll /var/lib/pgsql/10/data/base/16384/24724
-rw------- 1 postgres postgres 24576 19 sept. 05:25 /var/lib/pgsql/10/data/base/16384/
```

Le fichier fait maintenant 24 Ko, soit 3 blocs de 8 Ko.

Pourquoi cette taille pour simplement 501 fois un entier (ie, 4 octets) ?.

On a 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 systèmes. La requête suivante permet d'en savoir plus :

```
b1=# SELECT CASE WHEN attnum<0 THEN 'systeme' ELSE 'utilisateur' END AS type,
         attname, attnum, typname, typlen,
         sum(typlen) OVER (PARTITION BY attnum<0)
  FROM pg_attribute a
  JOIN pg_type t ON t.oid=a.atttypid
  WHERE attrelid IN (SELECT oid FROM pg_class WHERE relname='t1')
  ORDER BY attnum;
    type     | attname  | attnum | typname | typlen | sum
-------------+----------+--------+---------+--------+-----
 systeme     | tableoid |     -7 | oid     |      4 |  26
 systeme     | cmax     |     -6 | cid     |      4 |  26
 systeme     | xmax     |     -5 | xid     |      4 |  26
 systeme     | cmin     |     -4 | cid     |      4 |  26
 systeme     | xmin     |     -3 | xid     |      4 |  26
 systeme     | ctid     |     -1 | tid     |      6 |  26
 utilisateur | c1       |      1 | int4    |      4 |   3
 utilisateur | c2       |      2 | text    |     -1 |   3
(8 rows)
```

L'en-tête de chaque ligne pèse 26 octets dans le meilleur des cas. Il peut peser 30 si on demande à avoir un OID généré pour chaque ligne de la table. Dans notre cas très particulier avec une seule petite colonne, c'est très défavorable mais ce n'est généralement pas le cas.

Donc 30 octets par lignes, 501 lignes, on obtient 15 Ko. Avec l'entête de bloc et les pointeurs, on dépasse facilement 16 Ko, ce qui explique pourquoi nous en sommes à 24 Ko.

Ajoutez un tablespace.

```
# mkdir /opt/ts1
# chown postgres: /opt/ts1
$ psql b1
psql (10)
Type "help" for help.

b1=# CREATE TABLESPACE ts1 LOCATION '/opt/ts1';
CREATE TABLESPACE
```

Ajoutez-y une table.

```
b1=# CREATE TABLE t2 (id integer) TABLESPACE ts1;
CREATE TABLE
```

Récupérer le chemin vers le fichier correspond à cette table.

```
b1=# SELECT current_setting('data_directory')||'/'||pg_relation_filepath('t2');
                            ?column?
-----------------------------------------------------------------
 /var/lib/pgsql/10/data/pg_tblspc/24764/PG_10_201707211/16384/24765
```

Le fichier n'a pas été créée dans un sous-répertoire du répertoire base. Il est mis dans le tablespace indiqué par la commande `CREATE TABLE`. Il s'agit là-aussi d'un fichier :

```
$ ll /var/lib/pgsql/10/data/pg_tblspc/24764/PG_10_201707211/16384/24765
-rw------- 1 postgres postgres 0 19 sept. 05:28 /var/lib/pgsql/10/data/pg_tblspc/24764
$ ll /opt/ts1/PG_10_201707211/16384/24765
-rw------- 1 postgres postgres 0 19 sept. 05:28 /opt/ts1/PG_10_201707211/16384/24765
$ ll /var/lib/pgsql/10/data/pg_tblspc/
total 0
lrwxrwxrwx 1 postgres postgres 8 19 sept. 05:28 24764 -> /opt/ts1
```

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.

Insérer dix millions de lignes dans la table t2.

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Que se passe-t-il au niveau du répertoire pg_wal ?

```
$ ll pg_wal
total 131076
-rw-------. 1 gui gui 16777216 Feb 12 16:39 00000001000000000000002B
-rw-------. 1 gui gui 16777216 Feb 12 16:39 00000001000000000000002C
-rw-------. 1 gui gui 16777216 Feb 12 16:39 00000001000000000000002D
-rw-------. 1 gui gui 16777216 Feb 12 16:39 00000001000000000000002E
```

```
-rw-------. 1 gui gui 16777216 Feb 12 16:39 00000001000000000000002F
-rw-------. 1 gui gui 16777216 Feb 12 15:14 000000010000000000000030
-rw-------. 1 gui gui 16777216 Feb 12 15:14 000000010000000000000031
-rw-------. 1 gui gui 16777216 Feb 12 15:14 000000010000000000000032
drwx------. 2 gui gui     4096 Feb 12 13:59 archive_status
```

Des journaux de transactions sont écrits lors des écritures dans la base.

Exécutez un CHECKPOINT.

```
b1=# CHECKPOINT;
CHECKPOINT
```

Que se passe-t-il au niveau du répertoire pg_wal ?

```
$ ll pg_wal
total 131076
-rw-------. 1 gui gui 16777216 19 sept. 16:39 00000001000000000000002E
-rw-------. 1 gui gui 16777216 19 sept. 16:39 00000001000000000000002F
-rw-------. 1 gui gui 16777216 19 sept. 15:14 000000010000000000000030
-rw-------. 1 gui gui 16777216 19 sept. 15:14 000000010000000000000031
-rw-------. 1 gui gui 16777216 19 sept. 15:14 000000010000000000000032
-rw-------. 1 gui gui 16777216 19 sept. 16:39 000000010000000000000033
-rw-------. 1 gui gui 16777216 19 sept. 16:39 000000010000000000000034
-rw-------. 1 gui gui 16777216 19 sept. 16:39 000000010000000000000035
drwx------. 2 gui gui     4096 19 sept. 13:59 archive_status
```

Les anciens journaux devenus obsolètes sont recyclés.

Cache disque de PostgreSQL

Installez l'extension de pg_buffercache

```
b1=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

Redémarrez PostgreSQL

```
# service postgresql-10 restart
```

Videz le cache système

```
# sync
# echo 3 > /proc/sys/vm/drop_caches
```

Que contient le cache de PostgreSQL ?

```
b1=# SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1;
 relfilenode | count
-------------+-------
```

```
            | 16282
       12808 |     1
       12690 |     2
       12709 |     5
       12774 |     1
[...]
```

La colonne relfilenode correspond à l'identifiant système de la table. La deuxième colonne indique le nombre de blocs. Il y a 16282 blocs non utilisés pour l'instant dans le cache, ce qui est logique vu qu'on vient de redémarrer PostgreSQL. Il y a quelques blocs utilisés par des tables systèmes, mais aucune table utilisateur (ie, celle dont l'OID est supérieur à 16384).

Lisez complètement t2, en récupérant la durée d'exécution de la requête

```
b1=# \timing
Timing is on.
b1=# SELECT * FROM t2;
   id
---------
       1
       2
       3
       4
       5
[...]
Time: 356.927 ms
```

Que contient le cache de PostgreSQL ?

```
b1=# SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 1;
 relfilenode | count
-------------+-------
       12682 |     1
       12687 |     3
[...]
       12947 |     2
       12948 |     2
       24765 |  4425
             | 16190
(42 rows)

b1=# SELECT pg_table_size('t2');
 pg_table_size
---------------
      36282368
(1 row)
```

32 blocs ont été alloués pour la lecture de la table t2. 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)
```

Extrayez de nouveau toutes les données de la table t2

```
b1=# SELECT * FROM t2;
   id
---------
       1
       2
       3
       4
       5
[...]
Time: 184.529 ms
```

La lecture est bien plus rapide car la table est en cache, en partie au niveau PostgreSQL, mais surtout au niveau système d'exploitation.

Que contient le cache de PostgreSQL ?

```
b1=# SELECT relfilenode, count(*) FROM pg_buffercache
  WHERE relfilenode=24765 GROUP BY 1 ORDER BY 1;
 relfilenode | count
-------------+-------
       24585 |    64
(1 row)
```

On en a un peu plus dans le cache. En fait, plus vous l'exécutez, et plus le nombre de blocs présents en cache augmentera.

Changez shared_buffers, puis redémarrez PostgreSQL.

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.

Ensuite, il faut redémarrer PostgreSQL.

Extrayez de nouveau toutes les données de la table t2

```
b1=# SELECT * FROM t2;
   id
---------
       1
       2
```

```
       3
       4
       5
[...]
Time: 203.189 ms
```

Le temps d'exécution est un peu moins rapide que précédemment. En effet, le cache PostgreSQL a été vidé mais pas le cache du système d'exploitation.

Que contient le cache de PostgreSQL ?

```
b1=# SELECT relfilenode, count(*) FROM pg_buffercache
  WHERE relfilenode=24765 GROUP BY 1 ORDER BY 1;
 relfilenode | count
-------------+-------
       24585 |  4425
(1 row)
```

On se retrouve avec énormément plus de blocs directement dans le cache de PostgreSQL, et ce dès la première exécution. PostgreSQL est optimisé principalement pour du multi-utilisateurs. Dans ce cadre, il faut pouvoir exécuter plusieurs requêtes en même temps et donc chaque requête ne peut pas monopoliser tout le cache. De ce fait, chaque requête ne peut prendre qu'une partie réduite du cache. Mais plus le cache est gros, plus la partie est grosse.

Faire une mise à jour.

```
b1=# UPDATE t2 SET id=0 WHERE id<1000;
UPDATE 999
```

Que contient le cache de PostgreSQL ?

```
b1=# SELECT isdirty, count(*) FROM pg_buffercache
  WHERE relfilenode=24765 GROUP BY 1 ORDER BY 1;
 isdirty | count
---------+-------
 f       |  4421
 t       |    11
(2 rows)
```

Faites un CHECKPOINT

```
b1=# CHECKPOINT;
CHECKPOINT
```

Que contient le cache de PostgreSQL ?

```
b1=# SELECT isdirty, count(*) FROM pg_buffercache
b1=# WHERE relfilenode=24585 GROUP BY 1 ORDER BY 1;
 isdirty | count
```

```
---------+-------
 f       |  4432
(1 row)
```

Statistiques d'activités

Créer une table

```
b1=# CREATE TABLE t3 (id integer);
CREATE TABLE
```

Insérer des données

```
b1=# INSERT INTO t3 SELECT generate_series(1, 1000);
INSERT 0 1000
```

Lire ses statistiques d'activité

```
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.

Statistiques sur les données

Créer une table avec une seule colonne de type integer.

```
b1=# CREATE TABLE t4 (c1 integer);
CREATE TABLE
```

Empêcher autovacuum d'analyser automatiquement la table.

```
b1=# ALTER TABLE t4 SET (autovacuum_enabled=false);
ALTER TABLE
```

Mettre des données différentes sur c1.

```
b1=# INSERT INTO t4 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Lire la table avec un filtre sur c1.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE c1=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: (c1 = 100000)
(4 rows)
```

Exécuter la commande ANALYZE.

```
b1=# ANALYZE t4;
ANALYZE
```

Lire la table avec un filtre sur c1. Que constatez-vous ?

```
b1=# EXPLAIN SELECT * FROM t4 WHERE c1=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: (c1 = 100000)
(4 rows)
```

Les statistiques sont beaucoup plus précises. Il sait 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 est intéressant.

Ajouter un index sur c1.

```
b1=# CREATE INDEX ON t4(c1);
CREATE INDEX
```

Lire la table avec un filtre sur c1.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE c1=100000;
                              QUERY PLAN
-----------------------------------------------------------------------
```

```
 Index Only Scan using t4_c1_idx on t4  (cost=0.42..8.44 rows=1 width=4)
   Index Cond: (c1 = 100000)
(2 rows)
```

Après création de l'index, on constate que PostgreSQL choisir un autre plan qui permet d'utiliser cet index.

Modifier la colonne c1 avec la valeur 1 pour toutes les lignes.

```
b1=# UPDATE t4 SET c1=100000;
UPDATE 1000000
```

Toutes les lignes ont la même valeur.

Lire la table avec un filtre sur c1.

```
b1=# EXPLAIN ANALYZE SELECT * FROM t4 WHERE c1=100000;
                  QUERY PLAN
-----------------------------------------------------------
 Index Only Scan using t4_c1_idx on t4
       (cost=0.43..8.45 rows=1 width=4)
       (actual time=0.040..265.573 rows=1000000 loops=1)
   Index Cond: (c1 = 100000)
   Heap Fetches: 1000001
 Planning time: 0.066 ms
 Execution time: 303.026 ms
(5 rows)
```

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.

```
b1=# ANALYZE t4;
ANALYZE
```

Lire la table avec un filtre sur c1.

```
b1=# EXPLAIN ANALYZE SELECT * FROM t4 WHERE c1=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: (c1 = 100000)
 Planning time: 0.122 ms
 Execution time: 223.357 ms
(4 rows)
```

Avec des statistiques à jour et malgré la presence de l'index, PostgreSQL va utiliser un parcours séquentiel qui, au final, sera plus performant.

4 MÉCANIQUE DU MOTEUR TRANSACTIONNEL

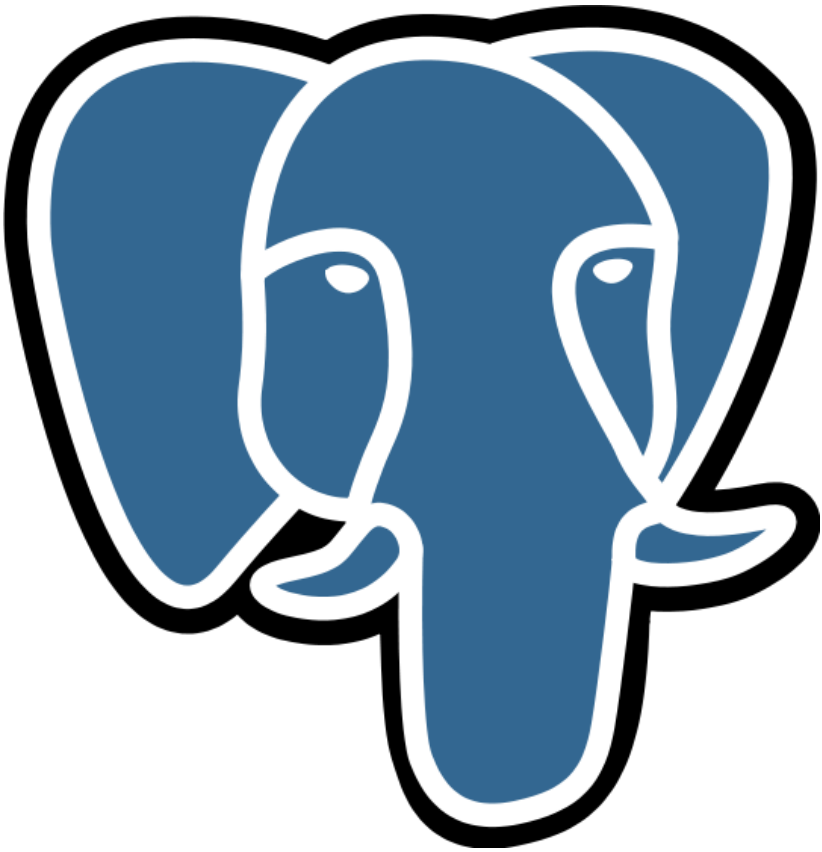


FIGURE 3: POSTGRESQL

4.1 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.

4.1.1 AU MENU

- Nous allons aborder :
  - Présentation de MVCC
  - Niveaux d'isolation
  - Implémentation de MVCC de PostgreSQL
  - Vacuum et son paramétrage
  - Autovacuum et son paramétrage
  - Verrouillage

4.2 PRÉSENTATION DE MVCC

- Que signifie MVCC
- Quelles solutions alternatives
- Implémentations possibles de MVCC

4.2.1 PRÉSENTATION DE MVCC

- MultiVersion Concurrency Control
- Contrôle de Concurrence Multi-Version
- Plusieurs versions du même enregistrement

MVCC est un acronyme signifiant « MultiVersion Concurrency Control », ou « contrôle de concurrence multi-version ».

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).

4.2.2 ALTERNATIVES À MVCC

- Une seule version de l'enregistrement en base
- Verrouillage (granularité ?)
- 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.

De nombreux moteurs de base de données fonctionnent sur ce mode. MVCC devient progressivement la norme, de nombreux autres moteurs l'adoptant.

Cette solution a l'avantage de la simplicité : il suffit d'un gestionnaire de verrou 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 ? On implémente habituellement des mécanismes de promotion de verrou. 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 (on ne veut pas 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. On pourrait 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.

4.2.3 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
  - L'annulation est lente

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 est habituellement très supérieur au temps de traitement initial devant être annulé.

4.2.4 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

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.

4.3 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
- On peut spécifier le niveau d'isolation au démarrage d'une transaction :
  - `BEGIN ISOLATION LEVEL xxx`;

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).

4.3.1 NIVEAU READ UNCOMMITTED

- Autorise la lecture de données modifiées mais non validées par d'autres transactions
- Aussi appelé `DIRTY READS` par d'autres moteurs
- Pas de blocage entre les sessions
- Inutile sous PostgreSQL en raison du MVCC
- Si demandé, la transaction s'exécute en `READ COMMITTED`

Ce niveau d'isolation n'est nécessaire que pour les SGBD non-MVCC. Il est très dangereux : on peut lire des données invalides, ou temporaires, puisqu'on lit tous les enregistrements de la table, quel que soit leur état. Il est utilisé dans certains cas où les performances sont cruciales, au détriment de la justesse des données.

Sous PostgreSQL, ce mode est totalement inutile. Une transaction qui demande le niveau d'isolation `READ UNCOMMITTED` s'exécute en fait en `READ COMMITTED`.

4.3.2 NIVEAU READ COMMITTED

- La transaction ne lit que les données validées en base
- Niveau d'isolation par défaut
- 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 pré-requis de `READ COMMITTED` dans la norme SQL). On ne souffre plus des lectures d'enregistrements non valides (`dirty reads`). On peut toutefois 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 transaction 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.

4.3.3 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 en cas de 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. En lecture seule, ces transactions ne peuvent pas échouer (elles sont entre autres utilisées pour réaliser des exports des données, par pg_dump).

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.

Ce niveau d'isolation souffre toujours des lectures fantômes, c'est-à-dire de lecture d'enregistrements qui ne satisfont plus la même clause `WHERE` entre deux exécutions de requêtes. Cependant, PostgreSQL est plus strict que la norme et ne permet pas ces lectures fantômes en `REPEATABLE READ`.

4.3.4 NIVEAU SERIALIZABLE

- Niveau d'isolation maximum
- Plus de lectures non répétables
- Plus de lectures fantômes
- Instantané au démarrage de la transaction
- Verrouillage informatif des enregistrements consultés (verrouillage des prédicats)
- Erreurs de sérialisation en cas d'incompatibilité

Le niveau `SERIALIZABLE` permet de développer comme si chaque transaction se déroulait seule sur la base. En cas d'incompatibilité entre les opérations réalisées par plusieurs transactions, PostgreSQL annule celle qui déclenchera le moins de perte de données. Tout comme dans le mode `REPEATABLE READ`, il est essentiel de pouvoir rejouer une transaction si on développe en mode `SERIALIZABLE`. Par contre, on simplifie énormément tous les autres points du développement.

Ce mode empêche les erreurs dues à une transaction effectuant un `SELECT` d'enregistrements, puis d'autres traitements, pendant qu'une autre transaction modifie les enregistrements vus par le `SELECT` : il est probable que le `SELECT` initial de notre transaction utilise les enregistrements récupérés, et que le reste du traitement réalisé par notre transaction dépende de ces enregistrements. Si ces enregistrements sont modifiés par une transaction concurrente, notre transaction ne s'est plus déroulée comme si elle était seule sur la base, et on a donc une violation de sérialisation.

Pour tous les détails (et des exemples)[14]

---

[14] http://wiki.postgresql.org/wiki/SSI/fr

4.4 L'IMPLÉMENTATION MVCC DE POSTGRESQL

- Colonnes `xmin`/`xmax`
- Fichiers `clog`
- Avantages/inconvénients
- Opération `VACUUM`
- Structure `Free Space Map` (`FSM`)
- `Wrap-Around`
- `Heap-Only Tuples` (`HOT`)
- `Visibility Map`

4.4.1 XMIN ET XMAX (1/4)

Table initiale :

xmin	xmax	Nom	Solde
100		M. Durand	1500
100		M. Dupond	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.

4.4.2 XMIN ET XMAX (2/4)

```
BEGIN;
UPDATE soldes SET solde=solde-200 WHERE nom = 'M. Durand';
```

xmin	xmax	Nom	Solde
100 100	**150**	M. Durand	1500
150		M. Dupond	2200
		M. Durand	1300

On décide d'enregistrer un virement de 200 € du compte de M. Durand vers celui de M. Dupond. 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';
```

4.4.3 XMIN ET XMAX (3/4)

```
UPDATE soldes SET solde=solde+200 WHERE nom = 'M. Dupond';
```

xmin	xmax	Nom	Solde
100 100	150 **150**	M. Durand	1500
150 **150**		M. Dupond	2200
		M. Durand	1300
		M. Dupond	2400

Puis nous effectuerons :

```
UPDATE soldes SET solde=solde+200 WHERE nom = 'M. Dupond';
```

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 est-elle *postérieure* à la nôtre (numéro supérieur au notre), et sommes-nous dans un niveau d'isolation (*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.

4.4.4 XMIN ET XMAX (4/4)

xmin	xmax	Nom	Solde
100 100	150 **150**	M. Durand	1500
150 **150**		M. Dupond	2200
		M. Durand	1300
		M. Dupond	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.
- Il n'y a rien à écrire dans les tables pour annuler une transaction. Il suffit de marquer la transaction comme étant annulée dans la `CLOG`.

4.4.5 CLOG

- La `CLOG` (Commit Log) enregistre l'état des transactions.
- Chaque transaction occupe 2 bits de `CLOG`

La `CLOG` est stockée 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` : transaction en cours, c'est l'état initial
- `TRANSACTION_STATUS_COMMITTED` : la transaction a été validée
- `TRANSACTION_STATUS_ABORTED` : la transaction a été annulée
- `TRANSACTION_STATUS_SUB_COMMITTED` : ceci est utilisé dans le cas où la transaction comporte des sous-transactions, afin de valider l'ensemble des sous-transactions de façon atomique.

On a donc un million d'états de transactions par fichier de 256 ko.

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.

Toute modification dans la CLOG, comme toute modification d'un fichier de données (table, index, séquence), est bien sûr enregistrée tout d'abord dans les journaux de transactions (fichiers XLOG dans le répertoire pg_wal).

4.4.6 AVANTAGES DU MVCC POSTGRESQL

- Avantages :
  - 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
- 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 dans la clog. Pas besoin de restaurer les valeurs précédentes, elles redeviennent automatiquement visibles.
- 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.

4.4.7 INCONVÉNIENTS DU MVCC POSTGRESQL

- Inconvénients :
  - Nettoyage des enregistrements (VACUUM)
  - Tables plus volumineuses

  - Pas de visibilité dans les index

Comme toute solution complexe, l'implémentation MVCC de PostgreSQL est un compromis. Les avantages cités précédemment sont obtenus au prix de concessions :

- Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande `VACUUM`. On peut aussi voir ce point comme un avantage : contrairement à la solution `UNDO`, ce travail de nettoyage n'est pas effectué par le client faisant des mises à jour (et créant donc des enregistrements morts). Le ressenti est donc meilleur.
- Les tables sont forcément plus volumineuses que dans l'implémentation par `UNDO`, pour deux raisons :
  - Les informations de visibilité qui y sont stockées. Il y a 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`.
- 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'exécutions de requêtes comme `SELECT count(*)` sur une table : il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles.

4.4.8 FONCTIONNEMENT DE VACUUM (1/3)

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.

L'enregistrement mort ne peut pas être supprimé immédiatement : des enregistrements d'index pointent vers lui et doivent aussi être nettoyés. Les adresses (`tid` ou `tuple id`) des enregistrements sont donc mémorisés par la session effectuant le vacuum, dans un espace mémoire dont la taille est à hauteur de `maintenance_work_mem`. Si `maintenance_work_mem` est trop petit pour contenir tous les enregistrements morts en une seule passe, vacuum effectue plusieurs séries de ces trois passes.

Un tid est composé du numéro de bloc et du numéro d'enregistrement dans le bloc.

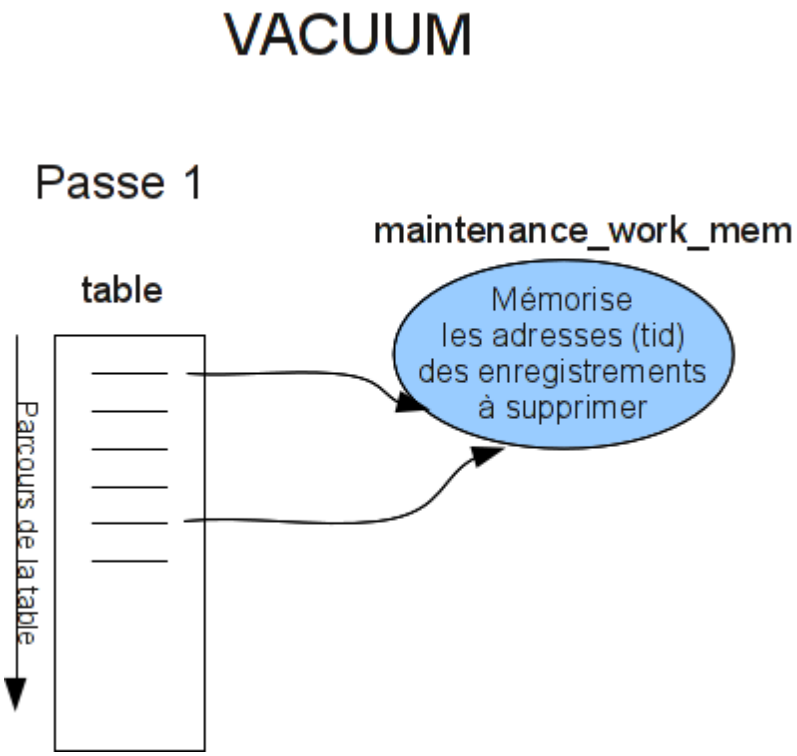


FIGURE 4: ALGORITHME DU VACUUM 1/3

4.4.9 FONCTIONNEMENT DE VACUUM (2/3)

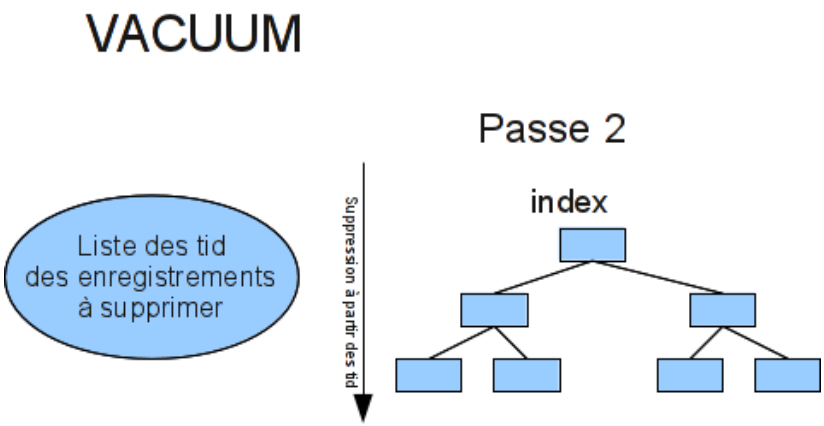


FIGURE 5: ALGORITHME DU VACUUM 2/3

La seconde passe se charge de nettoyer les entrées d'index. Vacuum possède une liste de tid 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`).

4.4.10 FONCTIONNEMENT DE VACUUM (3/3)

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements identifiés, nous pouvons supprimer les enregistrements de la table elle-même. C'est le rôle de cette passe, qui quant à elle, peut accéder directement aux enregistrements. Quand un enregistrement est supprimé d'un bloc, ce bloc est réorganisé afin de consolider l'espace libre, et cet espace libre est consolidé 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 1 (la `maintenance_work_mem` était pleine), le travail reprend où il en était du parcours de la table.

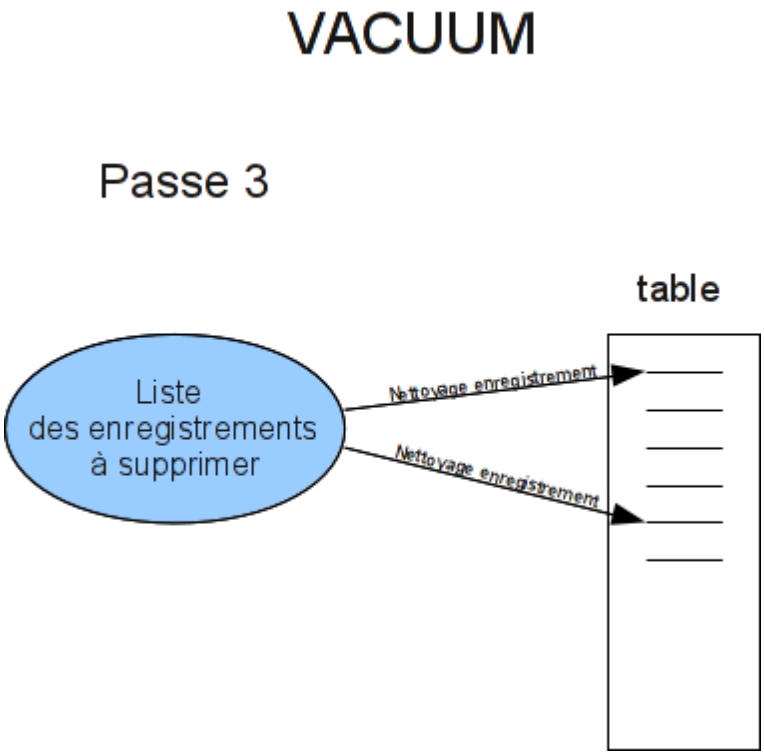


FIGURE 6: ALGORITHME DU VACUUM 3/3

4.4.11 PROGRESSION DU VACUUM

- Vue `pg_stat_progress_vacuum`
  - disponible dès la 9.6
- `heap_blks_scanned`, blocs parcourus
- `heap_blks_vacuumed`, blocs nettoyés
- `index_vacuum_count`, nombre de passes dans l'index

La version 9.6 intègre une nouvelle vue pour suivre la progression d'une commande `VACUUM`. Cette vue s'appelle `pg_stat_progress_vacuum` et contient une ligne par `VACUUM` en cours d'exécution.

Voici un exemple :

```
pid                 | 4299
datid               | 13356
datname             | postgres
relid               | 16384
phase               | scanning heap
heap_blks_total     | 127293
heap_blks_scanned   | 86665
heap_blks_vacuumed  | 86664
index_vacuum_count  | 0
max_dead_tuples     | 291
num_dead_tuples     | 53
```

Dans cet exemple, le `VACUUM` exécuté par le PID 4299 a parcouru 86665 blocs (soit 68 % de la table), et en a traité 86664.

4.4.12 OPTIMISATIONS DE MVCC

- MVCC a été affiné au fil des versions :
  - `Heap-Only Tuples`
  - `Free Space Map` dynamique
  - `Visibility Map`

Les améliorations suivantes ont été ajoutées au fil des versions :

- En 8.3, les *Heap-Only Tuples* (`HOT`). Il s'agit de pouvoir stocker, sous condition, plusieurs versions du même enregistrement dans le même bloc. Ceci permettant au fur et à mesure des mises à jour de supprimer automatiquement les anciennes

versions, sans besoin de VACUUM. Cela permet aussi de ne pas toucher aux index, qui pointent donc grâce à cela sur plusieurs versions du même enregistrement. Les conditions sont que :
  - Le bloc contienne assez de place pour la nouvelle version (on ne chaîne pas les enregistrements entre plusieurs blocs)
  - Aucune colonne indexée n'a été modifiée par l'opération.
- En 8.4, la Free Space Map est devenue dynamique. Jusqu'à ce moment-là, la Free Space Map était un segment de mémoire partagée statique. S'il était plein, les conséquences pouvaient être dramatiques (base qui enfle rapidement). Sa surveillance était donc une des tâches cruciales. Depuis la version 8.4, la Free Space Map est totalement dynamique, stockée dans des fichiers. Elle s'étend automatiquement au besoin.
- En 8.4, la Visibility Map a été introduite. Pour chaque bloc de chaque table, la Visibility Map permet de savoir que l'ensemble des enregistrements de ce bloc est visible. En cas de doute, ou d'enregistrement non visible, le bloc n'est pas marqué comme totalement visible. Cela permet à la phase 1 du traitement de VACUUM de ne plus parcourir toute la table, mais uniquement les enregistrements pour lesquels la Visibility Map est à *faux* (des données sont potentiellement obsolètes dans le bloc). VACUUM repositionne la Visibility Map à *vrai* après nettoyage d'un bloc, si tous les enregistrements sont visibles pour toutes les sessions.

Toutes ces optimisations visent le même but : rendre VACUUM le moins pénalisant possible, et simplifier la maintenance. La Visibility Map a permis en plus l'ajout des parcours d'index seuls en 9.2.

4.4.13 LE PROBLÈME DU WRAPAROUND

Wraparound : bouclage d'un compteur

- Le compteur de transactions : 32 bits
- 4 milliards de transactions
- Qu'arrive-t-il si on boucle ?
- Quelles protections ?

Le compteur de transactions de PostgreSQL est stocké sur 32 bits. Il peut donc, en théorie, y avoir un dépassement de ce compteur au bout de 4 milliards de transactions. En fait, le compteur est cyclique, et toute transaction considère que les 2 milliards de transactions supérieures à la sienne sont dans le futur, et les 2 milliards inférieures dans le passé. Le risque de bouclage est donc plus proche des 2 milliards.

17.12

En théorie, si on bouclait, de nombreux enregistrements deviendraient invisibles, car validés par des transactions futures. Heureusement PostgreSQL l'empêche. Au fil des versions, la protection est devenue plus efficace.

- Le moteur trace la plus vieille transaction d'une base, et refuse toute nouvelle transaction à partir du moment où le stock de transaction disponible est à 10 millions. Il suffit de lancer un `VACUUM` sur la base incriminée à ce point, qui débloquera la situation, en nettoyant les plus anciens xmin.
- Depuis l'arrivée d'`AUTOVACUUM`, celui-ci déclenche automatiquement un `VACUUM` quand le *Wraparound* se rapproche trop. Ceci se produit même si `AUTOVACUUM` est désactivé.

Si vous voulez en savoir plus, la documentation officielle[15] contient un paragraphe sur ce sujet.

Ce qu'il convient d'en retenir, c'est que le système empêchera le wraparound en se bloquant à partir de la version 8.1 et que la protection contre ce phénomène est automatique depuis la version 8.2, mais déclenche automatiquement un `VACUUM`, quel que soit le paramétrage d'`AUTOVACUUM`. Les versions précédentes imposent une surveillance régulière des journaux.

4.5 VACUUM ET SON PARAMÉTRAGE 1/2

- Mémoire
  - `maintenance_work_mem`
- Gestion du coût
  - `vacuum_cost_delay`
  - `vacuum_cost_page_hit`
  - `vacuum_cost_page_miss`
  - `vacuum_cost_page_dirty`
  - `vacuum_cost_limit`

4.6 VACUUM ET SON PARAMÉTRAGE 2/2

- Gel des lignes
  - `vacuum_freeze_min_age`

[15] http://docs.postgresql.fr/current/maintenance.html

- vacuum_freeze_table_age
- vacuum_multixact_freeze_min_age
- vacuum_multixact_freeze_table_age

4.6.1 VACUUM : MAINTENANCE_WORK_MEM

- Quantité de mémoire allouable
- Impact sur VACUUM
- Sur construction d'index (hors sujet)

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 limite est utilisée dans deux contextes :

- La construction d'index, afin d'effectuer les tris de données.
- VACUUM.

Dans le cas de VACUUM, cette mémoire est utilisée pour stocker les tid (adresses d'enregistrement) des enregistrements pouvant être recyclés. Cette mémoire est remplie pendant la phase 1 du processus de VACUUM tel qu'expliqué au chapitre précédent. La taille d'un tid est 6 octets.

Si tous les enregistrements morts d'une table ne tiennent pas dans maintenance_work_mem, VACUUM est obligé de faire plusieurs passes de nettoyage. Le VACUUM devient plus coûteux, car chaque passe impose un parcours complet de chaque index. Une valeur assez élevée (256 Mo ou plus) de maintenance_work_mem est souvent conseillée, cette mémoire n'étant utilisée que lors des VACUUM et reconstructions d'index. On peut stocker plusieurs dizaines de millions d'enregistrements à effacer dans 256 Mo.

4.6.2 VACUUM : VACUUM_COST_*

- vacuum_cost_page_hit
- vacuum_cost_page_miss
- vacuum_cost_page_dirty
- vacuum_cost_limit
- vacuum_cost_delay

Ces paramètres permettent de limiter l'agressivité disque de Vacuum.

`vacuum_cost_page_hit`, `vacuum_cost_page_miss`, `vacuum_cost_page_dirty` permettent d'affecter un coût arbitraire aux trois actions suivantes :

- `vacuum_cost_page_hit` : le coût d'accès à une page qui est dans le cache (défaut : 1)
- `vacuum_cost_page_miss` : le coût d'accès à une page qui n'est pas dans le cache (défaut : 10)
- `vacuum_cost_page_dirty` : le coût de modification d'une page, et donc d'avoir à l'écrire (défaut : 20)

Ces paramètres de coût permettent de « mesurer » l'activité de `VACUUM`, et le mettre en pause quand il aura atteint cette limite. Ce second point est gouverné par :

- `vacuum_cost_limit` : le coût à atteindre avant de déclencher une pause (défaut : 200)
- `vacuum_cost_delay` : le temps à attendre une fois que le coût est atteint (défaut : 0 ms, c'est à dire pas d'attente).

Ces paramètres permettent de limiter l'agressivité de `VACUUM`, en l'obligeant à faire des pauses à intervalle de travail régulier. Par défaut, `VACUUM` fonctionne aussi vite que possible.

Il est déconseillé de modifier les paramètres `vacuum_cost_page_*`. Habituellement, on modifie soit le délai, soit la limite, si on souhaite réduire la vitesse de `VACUUM`. Ce paramètre ne s'applique que pour les `VACUUM` lancés « manuellement » (en ligne de commande, *via* `vacuumdb`), pas pour des `VACUUM` lancés par le processus `autovacuum`.

Il est conseillé de laisser `vacuum_cost_limit` et `vacuum_cost_delay` aux valeurs par défaut (pas de limitation de débit) : quand `VACUUM` est lancé à la main, habituellement, c'est soit pour une procédure de traitement de nuit, soit en urgence. Dans les deux cas, il est la plupart du temps préférable que ces traitements s'exécutent aussi vite que possible.

On peut aussi utiliser ces deux paramètres pour la recherche des paramètres optimaux d'`autovacuum` (voir plus loin).

4.6.3 VACUUM FREEZE

- Principe de `FREEZE`
- `vacuum_freeze_min_age`
- `vacuum_freeze_table_age`

- vacuum_multixact_freeze_min_age
- vacuum_multixact_freeze_table_age

Afin d'éviter le « wraparound », VACUUM modifie le xmin des vieux enregistrements afin que ceux-ci ne se retrouvent pas brusquement dans le futur.

Il pourrait positionner le xmin à une valeur plus récente, ce qui serait suffisant. Toutefois, un identifiant spécial de transaction a été réservé à cet usage (l'identifiant frozenxid), identifiant qui est toujours dans le passé.

Il reste à déterminer :

- à partir de quel âge un enregistrement peut et doit être gelé si VACUUM l'aperçoit : c'est le rôle de vacuum_freeze_min_age ;
- à partir de quel moment VACUUM doit déclencher un traitement de nettoyage de toute la table (et non pas uniquement les blocs modifiés depuis le dernier VACUUM), afin de nettoyer tous les vieux enregistrements. C'est le rôle de vacuum_freeze_table_age.

Les deux valeurs par défaut sont satisfaisantes pour la plupart des installations.

4.7 AUTOVACUUM ET SON PARAMÉTRAGE

- Autovacuum :
  - Ne plus s'occuper de VACUUM
  - Automatique
  - Suit l'activité
  - S'occupe aussi des statistiques

Autovacuum est un processus de l'instance PostgreSQL. Il est disponible en module contrib à partir de la version 7.4, et intégré au moteur depuis la version 8.1, et activé par défaut (donc recommandé) depuis la version 8.3.

Le principe est le suivant :

- Le démon autovacuum se réveille à intervalle régulier, et inspecte les statistiques sur les tables des différentes bases (vous pouvez les consulter dans la vue pg_stat_all_tables). Ces statistiques sont liées au nombre d'INSERT, UPDATE et DELETE pour chaque table.
- Une fois qu'une table a dépassé la limite paramétrée de mises à jour, autovacuum démarre un « worker » (processus chargé d'effectuer un traitement), afin qu'il dé-

clenche le traitement sur la table. Le nombre de ces workers est limité, afin de ne pas engendrer de charge trop élevée.

- `autovacuum` ne se cantonne pas à exécuter des commandes `VACUUM`. Il s'occupe aussi de la collecte des statistiques sur les données. Suivant la quantité et le type de modifications, il déclenchera soit un `VACUUM ANALYZE` (vacuum et collecte des statistiques en une seule commande), soit seulement un `ANALYZE`.

4.7.1 AUTOVACUUM - PARAMÉTRAGE

- `autovacuum`
- `autovacuum_naptime`
- `autovacuum_max_workers`
- `autovacuum_work_mem`

- `autovacuum` : `on`/`off`. Détermine si autovacuum doit être activé. « on » par défaut depuis la version 8.3
- `autovacuum_naptime` : temps d'attente entre deux périodes de vérification. Depuis la version 8.3, il s'agit du temps entre deux passages sur la même base de l'instance. Précédemment, il fallait multiplier `naptime` par le nombre de bases de l'instance pour obtenir la fréquence de passage sur une base.
- `autovacuum_max_workers` : nombre maximum de « workers » qu'autovacuum pourra déclencher simultanément. La valeur par défaut (3) est généralement suffisante. Néanmoins, si vous constatez qu'il y a fréquemment trois autovacuum workers travaillant en même temps, il peut être nécessaire d'augmenter ce paramètre.
- `autovacuum_work_mem` : permet de surcharger `maintenance_work_mem` spécifiquement pour l'autovacuum (désactivé par défaut)

4.7.2 AUTOVACUUM - PARAMÉTRAGE

- `autovacuum_vacuum_threshold`
- `autovacuum_vacuum_scale_factor`
- `autovacuum_analyze_threshold`
- `autovacuum_analyze_scale_factor`

Ces paramètres sont les critères de déclenchement d'autovacuum sur une table.

- `VACUUM ANALYZE` :

  - autovacuum_vacuum_threshold : nombre minimum d'enregistrements devant être morts (mis à jour ou supprimés) dans la table avant de déclencher un VACUUM (50 par défaut).
  - autovacuum_vacuum_scale_factor : fraction du nombre d'enregistrements de la table à ajouter à autovacuum_vacuum_threshold avant de déclencher un VACUUM (0.2 par défaut).
  - Un VACUUM est donc déclenché si :

```
nb_enregistrements_morts (n_dead_tup) >=
    autovacuum_vacuum_threshold + nb_enregs × autovacuum_vacuum_scale_factor
```

- ANALYZE :
  - autovacuum_analyze_threshold : nombre minimum d'enregistrements devant être ajoutés, modifiés ou supprimés dans la table avant de déclencher un ANALYZE (25 par défaut).
  - autovacuum_analyze_scale_factor : fraction du nombre d'enregistrements de la table à ajouter à autovacuum_analyze_threshold avant de déclencher un ANALYZE (0.1 par défaut).
  - Un ANALYZE est donc déclenché si :

```
nb_insert+nb_updates+nb_delete >=
    autovacuum_analyze_threshold + nb_enregs × autovacuum_analyze_scale_factor
```

Attention : les INSERT sont pris en compte pour ANALYZE, puisqu'ils modifient le contenu de la table. Ils ne sont pas pris en compte pour VACUUM, puisqu'ils ne créent pas d'enregistrement mort.

4.7.3 AUTOVACUUM - PARAMÉTRAGE

- autovacuum_vacuum_cost_delay
- autovacuum_vacuum_cost_limit

Tout comme les VACUUM interactifs, les VACUUM d'autovacuum peuvent être limités en débit. Contrairement aux VACUUM interactifs, les VACUUM d'autovacuum sont par défaut limités en débit : autovacuum_vacuum_cost_delay vaut 20 ms.

La signification des paramètres est la même que pour vacuum_cost_limit et vacuum_cost_delay.

autovacuum_vacuum_cost_limit et autovacuum_vacuum_cost_delay peuvent en plus prendre la valeur « -1 », signifiant qu'ils héritent leur valeur de vacuum_cost_limit et vacuum_cost_delay.

4.7.4 AUTOVACUUM - PARAMÉTRAGE

- `autovacuum_freeze_max_age`
- `autovacuum_multixact_freeze_max_age`

`autovacuum_freeze_max_age` est l'âge maximum que peut avoir le plus vieil enregistrement d'une table avant qu'un `VACUUM` soit lancé sur cette table pour éviter un *Wraparound*. **Ce traitement est lancé même si `autovacuum` est désactivé (c'est-à-dire positionné à `off`).**

4.8 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

4.8.1 LE GESTIONNAIRE DE VERROUS

PostgreSQL possède un gestionnaire de verrous

- Verrous d'objet
- Niveaux de verrouillage
- Deadlock
- Vue pg_locks

PostgreSQL dispose d'un gestionnaire de verrous, comme tout SGBD.

Ce gestionnaire de verrous est capable de gérer des verrous sur des tables, sur des enregistrements, sur des ressources virtuelles. De nombreux types de verrous - 8 - sont disponibles, chacun entrant en conflit avec d'autres.

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler.

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.

On peut accéder aux verrous actuellement utilisés sur un cluster par la vue `pg_locks`.

4.8.2 VERROUS SUR ENREGISTREMENT

- Le gestionnaire de verrous possède des verrous sur enregistrements.
- Ils sont :
  - transitoires
  - pas utilisés pour prendre les verrous définitifs
- Utilisation de verrous sur disque.
- Pas de risque de pénurie de verrous.

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 :

- Chaque transaction verrouille son objet « identifiant de transaction » de façon exclusive.
- Une transaction voulant mettre à jour un enregistrement consulte le `xmax`. S'il constate que ce `xmax` est celui d'une transaction en cours, il demande un verrou exclusif sur l'objet « identifiant de transaction » de cette transaction. Qui ne lui est naturellement pas accordé. Il est donc placé en attente.
- Quand la 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.

17.12

Le mécanisme exposé ici est légèrement simplifié. Pour une explication approfondie, n'hésitez pas à consulter l'article suivant[16] issu de la base de connaissance Dalibo.

4.8.3 VERROUS - PARAMÈTRES

- `max_locks_per_transaction` et `max_pred_locks_per_transaction`
- `lock_timeout`
- `deadlock_timeout`
- `log_lock_waits`

- `max_locks_per_transaction` et `max_pred_locks_per_transaction` servent à dimensionner l'espace en mémoire partagée réservé aux verrous. Le nombre de verrous total est :

```
(max_locks_per_transaction + max_pred_locks_per_transaction) × max_connections
```

Le nombre maximum de verrous d'une session n'est pas limité à `max_locks_per_transaction`, une session peut acquérir autant de verrous qu'elle le souhaite. La valeur par défaut de 64 est largement suffisante la plupart du temps.

- Si une session attend un verrou depuis plus longtemps que `lock_timeout`, la requête est annulée.
- Si une session reste plus de `deadlock_timeout` en attente, le système vérifie que cette transaction n'est pas en *deadlock*. La valeur par défaut est 1 seconde, ce qui est largement suffisant la plupart du temps : les deadlocks sont assez rares, il n'est donc pas intéressant d'être trop agressif dans leur vérification.
- `log_lock_waits` : si une session reste plus de `deadlock_timeout` en attente de verrou et que ce paramètre est à `on`, et qu'elle n'est pas effectivement victime d'un verrou mortel (*deadlock*), le système trace cet événement dans le journal. Est aussi tracé le moment où la session obtient réellement son verrou. La valeur est `off` par défaut.

4.9 CONCLUSION

- PostgreSQL dispose d'une implémentation MVCC complète, permettant :
  - Que les lecteurs ne bloquent pas les écrivains

[16] https://support.dalibo.com/kb/verrouillage

  - 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.

4.9.1 QUESTIONS

N'hésitez pas, c'est le moment !

4.10 TRAVAUX PRATIQUES

4.10.1 ÉNONCÉS

Niveaux d'isolation

- Créez une nouvelle base.
- Créez une table avec deux colonnes.
- Ajoutez cinq lignes dans cette table.
- Ouvrez une transaction
- Lire la table t1.
- Avec une autre session, modifiez quelques lignes de la table t1.
- Revenez à la première session et lisez de nouveau toute la table.
- Fermer la transaction et ouvrez-en une nouvelle, cette fois-ci en REPEATABLE READ.
- Lisez la table.
- Avec une autre session, modifiez quelques lignes de la table t1.
- Revenez à la première session et lisez de nouveau toute la table.
- Que s'est-il passé ?

Effets de MVCC

- Créez une nouvelle table avec deux colonnes.
- Ajoutez cinq lignes dans cette table.

- Lisez la table.
- Commencez une transaction et modifiez une ligne.
- Lisez la table.
- Que remarquez-vous ?
- Ouvrez une autre session et lisez la table.
- Qu'observez-vous ?
- Récupérez quelques informations systèmes xmin et xmax pour les deux sessions.
- Récupérez maintenant en plus le CTID.
- Validez la transaction.
- Installez l'extension pageinspect.
- Décodez le bloc 0 de la table t2 à l'aide de cette extension.

Traiter la fragmentation

- Exécutez un VACUUM VERBOSE sur la table t2.
- Trouvez la ligne intéressante dans les traces de la commande, et indiquez pourquoi.
- Créez une autre table (une seule colonne de type integer est nécessaire).
- Désactivez l'autovacuum pour cette table.
- Insérez un million de lignes dans cette table.
- Récupérez la taille de la table.
- Supprimez les 500000 premières lignes.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Exécutez un VACUUM.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Exécutez un VACUUM FULL.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Créez encore une autre table (une seule colonne de type integer est nécessaire).
- Désactivez l'autovacuum pour cette table.
- Insérez un million de lignes dans cette table.
- Récupérez la taille de la table.

- Supprimez les 500000 dernières lignes.
- Récupérez la taille de la table. Qu'en déduisez-vous ?
- Exécutez un VACUUM.
- Récupérez la taille de la table. Qu'en déduisez-vous ?

Détecter la fragmentation

- Installez l'extension pg_freespacemap.
- Créer une autre table à deux colonnes (integer et text).
- Désactivez l'autovacuum pour cette table.
- Insérer un million de lignes dans cette table.
- Que rapporte pg_freespacemap quant à l'espace libre de la table ?
- Modifier des données (par exemple 200000 lignes).
- Que rapporte pg_freespacemap quant à l'espace libre de la table ?
- Exécutez un VACUUM sur la table.
- Que rapporte pg_freespacemap quant à l'espace libre de la table ? Qu'en déduisez-vous ?
- Récupérez la taille de la table.
- Exécutez un VACUUM FULL sur la table.
- Récupérez la taille de la table et l'espace libre rapporté par pg_freespacemap. Qu'en déduisez-vous ?

Gestion de l'autovacuum

- Créez une table avec une colonne de type integer.
- Insérer un million de lignes dans cette table.
- Que contient la vue pg_stat_user_tables pour cette table ?
- Modifiez 200000 lignes de cette table.
- Attendez une minute.
- Que contient la vue pg_stat_user_tables pour cette table ?
- Modifier 60 lignes supplémentaires de cette table.
- Attendez une minute.
- Que contient la vue pg_stat_user_tables pour cette table ? Qu'en déduisez-vous ?

- Descendez le facteur d'échelle de cette table à 10% pour le VACUUM.
- Modifiez 200000 lignes de cette table ?
- Attendez une minute.
- Que contient la vue pg_stat_user_tables pour cette table ? Qu'en déduisez-vous ?

Verrous

- Ouvrez une transaction et lisez la table t1.
- Ouvrez une autre transaction, et tentez de supprimer la table t1.
- Listez les processus du serveur PostgreSQL. Que remarquez-vous ?
- Récupérez la liste des sessions en attente d'un verrou avec la vue pg_stat_activity.
- Récupérez la liste des verrous en attente pour la requête bloquée.
- Récupérez le nom de l'objet dont on n'arrive pas à récupérer le verrou.
- Récupérez la liste des verrous sur cet objet. Quel processus a verrouillé la table t1 ?
- Retrouvez les informations sur la session bloquante.

4.10.2 SOLUTIONS

Niveaux d'isolation

Créez une nouvelle base.

```
$ createdb b2
```

Créez une table avec deux colonnes.

```
b2=# CREATE TABLE t1 (c1 integer, c2 text);
CREATE TABLE
```

Ajoutez cinq lignes dans cette table.

```
b2=# INSERT INTO t1 VALUES
  (1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
INSERT 0 5
```

Ouvrez une transaction

```
b2=# BEGIN;
BEGIN
```

Lire la table t1.

```
b2=# SELECT * FROM t1;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
(5 rows)
```

Avec une autre session, modifiez quelques lignes de la table t1.

```
$ psql b2
psql (10)
Type "help" for help.

b2=# UPDATE t1 SET c2=upper(c2) WHERE c1=3;
UPDATE 1
```

Revenez à la première session et lisez de nouveau toute la table.

```
b2=# SELECT * FROM t1;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
(5 rows)
```

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 ouvrez-en une nouvelle.

```
b2=# ROLLBACK;
ROLLBACK
b2=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
```

Lisez la table.

```
b2=# SELECT * FROM t1;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
```

```
  5 | cinq
  3 | TROIS
(5 rows)
```

Avec une autre session, modifiez quelques lignes de la table t1.

```
$ psql b2
psql (10)
Type "help" for help.

b2=# UPDATE t1 SET c2=upper(c2) WHERE c1=4;
UPDATE 1
```

Revenez à la première session et lisez de nouveau toute la table.

```
b2=# SELECT * FROM t1;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
(5 rows)
```

Que s'est-il passé ?

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).

Effets de MVCC

Créez une nouvelle table avec deux colonnes.

```
b2=# CREATE TABLE t2 (c1 integer, c2 text);
CREATE TABLE
```

Ajoutez cinq lignes dans cette table.

```
b2=# INSERT INTO t2 VALUES
  (1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
INSERT 0 5
```

Lisez la table.

```
b2=# SELECT * FROM t2;
 c1 |   c2
----+--------
  1 | un
  2 | deux
```

```
  3 | trois
  4 | quatre
  5 | cinq
(5 rows)
```

Commencez une transaction et modifiez une ligne.

```
b2=# BEGIN;
BEGIN
b2=# UPDATE t2 SET c2=upper(c2) WHERE c1=3;
UPDATE 1
```

Lisez la table.

```
b2=# SELECT * FROM t2;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
(5 rows)
```

Que remarquez-vous ?

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, les lignes sont affichées dans leur ordre de stockage dans les blocs de la table.

Ouvrez une autre session et lisez la table.

```
$ psql b2
psql (10)
Type "help" for help.

b2=# SELECT * FROM t2;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
(5 rows)
```

Qu'observez-vous ?

Les autres sessions voient toujours l'ancienne version de ligne, tant que la transaction n'a pas été validée. Et du coup, l'ordre des lignes en retour n'est pas le même vu que cette version de ligne était introduite avant.

Récupérez quelques informations systèmes xmin et xmax pour les deux sessions.

Voici ce que renvoie la session qui a fait la modification :

```
b2=# SELECT xmin, xmax, * FROM t2;
 xmin | xmax | c1 |   c2
------+------+----+--------
 1930 |    0 |  1 | un
 1930 |    0 |  2 | deux
 1930 |    0 |  4 | quatre
 1930 |    0 |  5 | cinq
 1931 |    0 |  3 | TROIS
(5 rows)
```

Et voici ce que renvoie l'autre session :

```
b2=# SELECT xmin, xmax, * FROM t2;
 xmin | xmax | c1 |   c2
------+------+----+--------
 1930 |    0 |  1 | un
 1930 |    0 |  2 | deux
 1930 | 1931 |  3 | trois
 1930 |    0 |  4 | quatre
 1930 |    0 |  5 | cinq
(5 rows)
```

La transaction 1931 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.

Récupérez maintenant en plus le CTID.

Voici ce que renvoie la session qui a fait la modification :

```
b2=# SELECT ctid, xmin, xmax, * FROM t2;
 ctid  | xmin | xmax | c1 |   c2
-------+------+------+----+--------
 (0,1) | 1930 |    0 |  1 | un
 (0,2) | 1930 |    0 |  2 | deux
 (0,4) | 1930 |    0 |  4 | quatre
 (0,5) | 1930 |    0 |  5 | cinq
 (0,6) | 1931 |    0 |  3 | TROIS
(5 rows)
```

Et voici ce que renvoie l'autre session :

```
b2=# SELECT ctid, xmin, xmax, * FROM t2;
 ctid  | xmin | xmax | c1 |   c2
-------+------+------+----+--------
 (0,1) | 1930 |    0 |  1 | un
 (0,2) | 1930 |    0 |  2 | deux
 (0,3) | 1930 | 1931 |  3 | trois
 (0,4) | 1930 |    0 |  4 | quatre
 (0,5) | 1930 |    0 |  5 | cinq
(5 rows)
```

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, elle précise la position de l'enregistrement sur le fichier de la table.

En récupérant cette colonne, on voit bien que la première session voit la nouvelle position (enregistrement 6 du bloc 0) et que la deuxième session voit l'ancienne (enregistrement 3 du bloc 0).

Validez la transaction.

```
b2=# COMMIT;
COMMIT
```

Installez l'extension pageinspect.

```
b2=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

Décodez le bloc 0 de la table t2 à l'aide de cette extension.

```
b4=# 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 |   2169 |      0 |        0 | (0,1)  |
  2 |   8120 |        1 |     33 |   2169 |      0 |        0 | (0,2)  |
  3 |   8080 |        1 |     34 |   2169 |   2170 |        0 | (0,6)  |
  4 |   8040 |        1 |     35 |   2169 |      0 |        0 | (0,4)  |
  5 |   8000 |        1 |     33 |   2169 |      0 |        0 | (0,5)  |
  6 |   7960 |        1 |     34 |   2170 |      0 |        0 | (0,6)  |

 lp | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+-------------+------------+--------+--------+-------
  1 |           2 |       2306 |     24 |        |
  2 |           2 |       2306 |     24 |        |
  3 |       16386 |        258 |     24 |        |
  4 |           2 |       2306 |     24 |        |
  5 |           2 |       2306 |     24 |        |
```

```
  6 |       32770 |      10242 |     24 |        |
(6 rows)
```

Que peut-on remarquer ?

- les six lignes sont bien présentes ;
- le t_ctid ne contient plus (0,3) mais l'adresse de la nouvelle ligne (ie, (0,6] ;
- 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.

Traiter la fragmentation

Exécutez un VACUUM VERBOSE sur la table t2.

```
b2=# VACUUM VERBOSE t2;
INFO:  vacuuming "public.t2"
INFO:  "t2": found 1 removable, 5 nonremovable row versions in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet.
         There were 0 unused item pointers.
         0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  vacuuming "pg_toast.pg_toast_24628"
INFO:  index "pg_toast_24628_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
         0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "pg_toast_24628": found 0 removable, 0 nonremovable row versions
                        in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
         There were 0 unused item pointers.
         0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

Trouvez la ligne intéressante dans les traces de la commande, et indiquez pourquoi.

Il s'agit de la ligne suivante :

```
INFO:  "t2": found 1 removable, 5 nonremovable row versions in 1 out of 1 pages
```

Il y a en effet une ligne obsolète (et récupérable) et cinq lignes vivantes sur le seul bloc de la table.

Créez une autre table (une seule colonne de type integer est nécessaire).

```
b2=# CREATE TABLE t3(id integer);
CREATE TABLE
```

Désactivez l'autovacuum pour cette table.

```
b2=# ALTER TABLE t3 SET (autovacuum_enabled=false);
ALTER TABLE
```

Insérez un million de lignes dans cette table.

```
b2=# INSERT INTO t3 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Récupérez la taille de la table.

```
b2=# SELECT pg_size_pretty(pg_table_size('t3'));
 pg_size_pretty
----------------
 35 MB
(1 row)
```

Supprimez les 500000 premières lignes.

```
b2=# DELETE FROM t3 WHERE id<500000;
DELETE 499999
```

Récupérez la taille de la table. Qu'en déduisez-vous ?

```
b2=# SELECT pg_size_pretty(pg_table_size('t3'));
 pg_size_pretty
----------------
 35 MB
(1 row)
```

Un DELETE ne permet pas de regagner de la place sur le disque. Les lignes supprimées sont uniquement marquées comme étant mortes.

Exécutez un VACUUM.

```
b2=# VACUUM t3;
VACUUM
```

Récupérez la taille de la table. Qu'en déduisez-vous ?

```
b2=# SELECT pg_size_pretty(pg_table_size('t3'));
 pg_size_pretty
----------------
 35 MB
(1 row)
```

VACUUM ne permet pas non plus de gagner en espace disque. Principalement, il renseigne la structure FSM sur les emplacements libres dans les fichiers des tables.

Exécutez un VACUUM FULL.

```
b2=# VACUUM FULL t3;
VACUUM
```

Récupérez la taille de la table. Qu'en déduisez-vous ?

```
b2=# SELECT pg_size_pretty(pg_table_size('t3'));
 pg_size_pretty
----------------
 17 MB
(1 row)
```

Là, par contre, on gagne en place disque. Le VACUUM FULL défragmente la table et du coup, on récupère les espaces morts.

Créez encore une autre table (une seule colonne de type integer est nécessaire).

```
b2=# CREATE TABLE t4(id integer);
CREATE TABLE
```

Désactivez l'autovacuum pour cette table.

```
b2=# ALTER TABLE t4 SET (autovacuum_enabled=false);
ALTER TABLE
```

Insérez un million de lignes dans cette table.

```
b2=# INSERT INTO t4 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Récupérez la taille de la table.

```
b2=# SELECT pg_size_pretty(pg_table_size('t4'));
 pg_size_pretty
----------------
 35 MB
(1 row)
```

Supprimez les 500000 dernières lignes.

```
b2=# DELETE FROM t4 WHERE id>500000;
DELETE 500000
```

Récupérez la taille de la table. Qu'en déduisez-vous ?

```
b2=# SELECT pg_size_pretty(pg_table_size('t4'));
 pg_size_pretty
----------------
 35 MB
(1 row)
```

Là-aussi, on n'a rien perdu.

Exécutez un VACUUM.

```
b2=# VACUUM t4;
VACUUM
```

Récupérez la taille de la table. Qu'en déduisez-vous ?

```
b2=# SELECT pg_size_pretty(pg_table_size('t4'));
 pg_size_pretty
----------------
 17 MB
(1 row)
```

En fait, il existe un cas où on peut 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.

Détecter la fragmentation

Installez l'extension pg_freespacemap.

```
b2=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION
```

Créer une autre table à deux colonnes (integer et text).

```
b2=# CREATE TABLE t5 (c1 integer, c2 text);
CREATE TABLE
```

Désactivez l'autovacuum pour cette table.

```
b2=# ALTER TABLE t5 SET (autovacuum_enabled=false);
ALTER TABLE
```

Insérer un million de lignes dans cette table.

```
b2=# INSERT INTO t5 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
INSERT 0 1000000
```

Que rapporte pg_freespacemap quant à l'espace libre de la table ?

```
b2=# SELECT sum(avail) FROM pg_freespace('t5'::regclass);
 sum
-----
   0
(1 row)
```

Modifier des données (par exemple 200000 lignes).

```
b2=# UPDATE t5 SET c2=upper(c2) WHERE c1<200000;
UPDATE 199999
```

Que rapporte pg_freespacemap quant à l'espace libre de la table ?

```
b2=# SELECT sum(avail) FROM pg_freespace('t5'::regclass);
 sum
-----
```

```
  32
(1 row)
```

Exécutez un VACUUM sur la table.

```
b2=# VACUUM t5;
VACUUM
```

Que rapporte pg_freespacemap quant à l'espace libre de la table ? Qu'en déduisez-vous ?

```
b2=# SELECT sum(avail) FROM pg_freespace('t5'::regclass);
   sum
---------
 8806784
(1 row)
```

Il faut exécuter un VACUUM pour que PostgreSQL renseigne la structure FSM, ce qui nous permet de connaître le taux de fragmentation de la table.

Récupérez la taille de la table.

```
b2=# SELECT pg_size_pretty(pg_table_size('t5'));
 pg_size_pretty
----------------
 58 MB
(1 row)
```

Exécutez un VACUUM FULL sur la table.

```
b2=# VACUUM FULL t5;
VACUUM
```

Récupérez la taille de la table et l'espace libre rapporté par pg_freespacemap. Qu'en déduisez-vous ?

```
b2=# SELECT sum(avail) FROM pg_freespace('t5'::regclass);
 sum
-----
   0
(1 row)

b2=# SELECT pg_size_pretty(pg_table_size('t5'));
 pg_size_pretty
----------------
 49 MB
(1 row)
```

VACUUM FULL a supprimé 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.

Gestion de l'autovacuum

Créez une table avec une colonne de type integer.

```
b2=# CREATE TABLE t6 (id integer);
CREATE TABLE
```

Insérer un million de lignes dans cette table.

```
b2=# INSERT INTO t6 SELECT generate_series(1, 1000000);
INSERT 0 1000000
```

Que contient la vue pg_stat_user_tables pour cette table ?

```
b2=# \x
Expanded display is on.
b2=# SELECT * FROM pg_stat_user_tables WHERE relname='t6';
-[ RECORD 1 ]-------+--------
relid               | 24851
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 | 1000000
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    |
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
```

Modifiez 200000 lignes de cette table.

```
b2=# UPDATE t6 SET id=2000000 WHERE id<200001;
UPDATE 200000
```

Attendez une minute.

```
b2=# SELECT pg_sleep(60);
```

Que contient la vue pg_stat_user_tables pour cette table ?

```
b2=# SELECT * FROM pg_stat_user_tables WHERE relname='t6';
-[ RECORD 1 ]-------+------------------------------
relid               | 24851
schemaname          | public
relname             | t6
seq_scan            | 1
seq_tup_read        | 1000000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 1000000
n_tup_upd           | 200000
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
n_dead_tup          | 0
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     | 2017-09-19 09:53:22.70433-04
last_analyze        |
last_autoanalyze    | 2017-09-19 09:53:23.325561-04
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 1
```

Modifier 60 lignes supplémentaires de cette table ?

```
b2=# UPDATE t6 SET id=2000000 WHERE id<200060;
UPDATE 59
```

Attendez une minute.

```
b2=# SELECT pg_sleep(60);
```

Que contient la vue pg_stat_user_tables pour cette table ? Qu'en déduisez-vous ?

```
b2=# SELECT * FROM pg_stat_user_tables WHERE relname='t6';
-[ RECORD 1 ]-------+------------------------------
relid               | 24851
schemaname          | public
relname             | t6
seq_scan            | 2
seq_tup_read        | 2000000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 1000000
n_tup_upd           | 200059
n_tup_del           | 0
n_tup_hot_upd       | 10
```

```
n_live_tup          | 1000000
n_dead_tup          | 59
n_mod_since_analyze | 59
last_vacuum         |
last_autovacuum     | 2017-09-19 09:53:22.70433-04
last_analyze        |
last_autoanalyze    | 2017-09-19 09:53:23.325561-04
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 1
```

Un VACUUM a été automatiquement exécuté sur cette table, suite à la suppression de plus de 200050 lignes (`threshold + scale factor * #lines`). Il a fallu attendre que l'autovacuum vérifie l'état des tables, d'où l'attente de 60 secondes.

Notez aussi que `n_dead_tup` est revenu à 0 après le VACUUM. C'est le compteur qui est comparé à la limite avant exécution d'un VACUUM.

Descendez le facteur d'échelle de cette table à 10% pour le VACUUM.

```
b2=# ALTER TABLE t6 SET (autovacuum_vacuum_scale_factor=0.1);
ALTER TABLE
```

Modifiez 200000 lignes de cette table ?

```
b2=# UPDATE t6 SET id=2000000 WHERE id<=400060;
UPDATE 200000
```

Attendez une minute.

```
b2=# SELECT pg_sleep(60);
```

Que contient la vue pg_stat_user_tables pour cette table ? Qu'en déduisez-vous ?

```
b2=# SELECT * FROM pg_stat_user_tables WHERE relname='t6';
-[ RECORD 1 ]-------+------------------------------
relid               | 24851
schemaname          | public
relname             | t6
seq_scan            | 3
seq_tup_read        | 3000000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 1000000
n_tup_upd           | 400060
n_tup_del           | 0
n_tup_hot_upd       | 54
n_live_tup          | 1000000
n_dead_tup          | 0
```

```
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     | 2017-09-19 09:55:25.256378-04
last_analyze        |
last_autoanalyze    | 2017-09-19 09:55:25.878329-04
vacuum_count        | 0
autovacuum_count    | 2
analyze_count       | 0
autoanalyze_count   | 2
```

Avec un facteur d'échelle à 10%, il ne faut plus attendre que la modification de 100050 lignes.

Verrous

Ouvrez une transaction et lisez la table t1.

```
b2=# BEGIN;
BEGIN
b2=# SELECT * FROM t1;
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | TROIS
  4 | QUATRE
  5 | CINQ
(5 rows)
```

Ouvrez une autre transaction, et tentez de supprimer la table t1.

```
$ psql b2
psql (10)
Type "help" for help.

b2=# DROP TABLE t1;
```

La suppression semble bloquée.

Listez les processus du serveur PostgreSQL. Que remarquez-vous ?

```
$ ps -o pid,cmd fx
  PID CMD
 2052  \_ psql b2
 2123  \_ ps -o pid,cmd fx
 2028  \_ psql b2
 1992 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data
 1994  \_ postgres: logger process
 1996  \_ postgres: checkpointer process
 1997  \_ postgres: writer process
```

```
1998  \_ postgres: wal writer process
1999  \_ postgres: autovacuum launcher process
2000  \_ postgres: stats collector process
2001  \_ postgres: bgworker: logical replication launcher
2029  \_ postgres: postgres b2 [local] idle in transaction
2053  \_ postgres: postgres b2 [local] DROP TABLE waiting
```

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.

Récupérez la liste des sessions en attente d'un verrou avec la vue pg_stat_activity.

```
$ psql b2
psql (10)
Type "help" for help.

b2=# \x
Expanded display is on.
b2=# SELECT * FROM pg_stat_activity
  WHERE application_name='psql' AND wait_event IS NOT NULL;
-[ RECORD 1 ]----+------------------------------
datid            | 24781
datname          | b2
pid              | 2029
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-09-19 09:36:21.876533-04
xact_start       | 2017-09-19 09:55:49.204131-04
query_start      | 2017-09-19 09:55:56.803826-04
state_change     | 2017-09-19 09:55:56.804157-04
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query            | SELECT * FROM t1;
backend_type     | client backend
-[ RECORD 2 ]----+------------------------------
datid            | 24781
datname          | b2
pid              | 2053
usesysid         | 10
usename          | postgres
```

```
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-09-19 09:37:14.512091-04
xact_start       | 2017-09-19 09:56:12.79626-04
query_start      | 2017-09-19 09:56:12.79626-04
state_change     | 2017-09-19 09:56:12.796262-04
wait_event_type  | Lock
wait_event       | relation
state            | active
backend_xid      | 841
backend_xmin     | 841
query            | DROP TABLE t1;
backend_type     | client backend
```

Récupérez la liste des verrous en attente pour la requête bloquée.

```
b2=# SELECT * FROM pg_locks WHERE pid=2053 AND NOT granted;
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 24781
relation           | 24782
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 4/37
pid                | 2053
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
```

Récupérez le nom de l'objet dont on n'arrive pas à récupérer le verrou.

```
b2=# SELECT relname FROM pg_class WHERE oid=24782;
-[ RECORD 1 ]
relname | t1
```

Récupérez la liste des verrous sur cet objet. Quel processus a verrouillé la table t1 ?

```
b2=# SELECT * FROM pg_locks WHERE relation=24782;
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 24781
relation           | 24782
```

```
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 4/37
pid                | 2053
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
-[ RECORD 2 ]------+--------------------
locktype           | relation
database           | 24781
relation           | 24782
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 3/212
pid                | 2029
mode               | AccessShareLock
granted            | t
fastpath           | f
```

Le processus de PID 2029 a un verrou sur t1. Ce processus avait été listé plus haut en attente du client (`idle in transaction`).

Retrouvez les informations sur la session bloquante.

```
b2=# SELECT * FROM pg_stat_activity WHERE pid=2029;
-[ RECORD 1 ]----+------------------------------
datid            | 24781
datname          | b2
pid              | 2029
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-09-19 09:36:21.876533-04
xact_start       | 2017-09-19 09:55:49.204131-04
```

```
query_start      | 2017-09-19 09:55:56.803826-04
state_change     | 2017-09-19 09:55:56.804157-04
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query            | SELECT * FROM t1;
backend_type     | client backend
```

À partir de là, il est possible d'arrêter l'exécution de l'ordre `DROP TABLE` avec la fonction `pg_cancel_backend()` ou de déconnecter le processus en cours de transaction avec la fonction `pg_terminate_backend()`.

5 POINT IN TIME RECOVERY

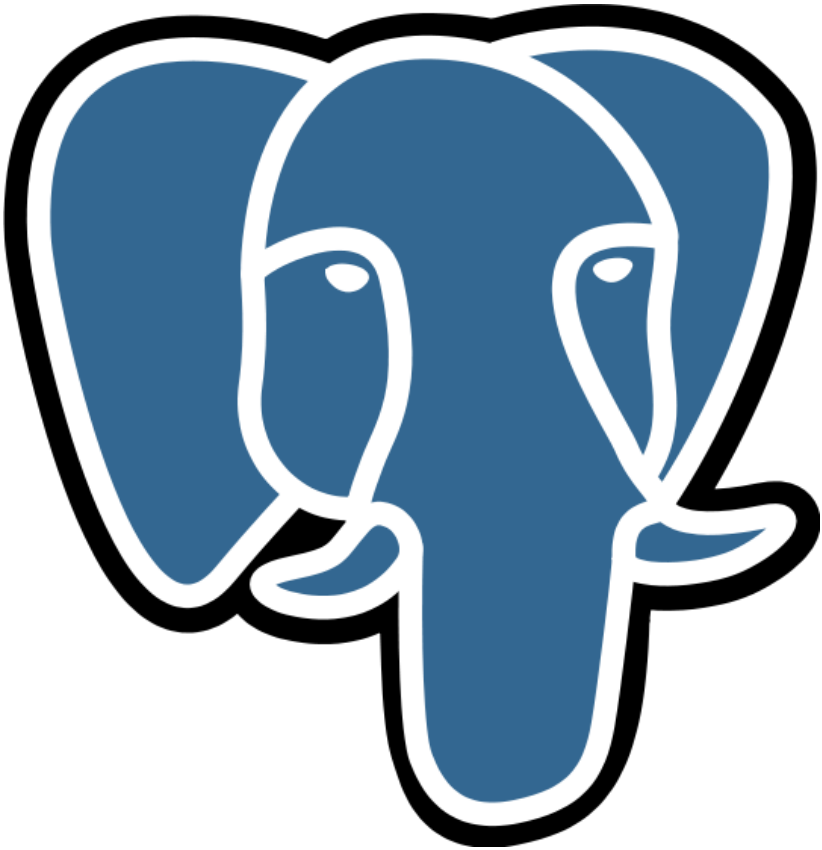


FIGURE 7: POSTGRESQL

5.1 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, 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 PITR.

5.1.1 AU MENU

- Mettre en place la sauvegarde PITR
  - archivage manuel ou avec pg_receivewal
  - sauvegarde manuelle ou avec pg_basebackup
- Restaurer une sauvegarde PITR
- Quelques outils pour aller plus loin
  - barman
  - pitrery

Ce module fait le tour de la sauvegarde PITR, de la mise en place de l'archivage (de manière manuelle ou avec l'outil `pg_receivewal`) à la sauvegarde des fichiers (là aussi, en manuel, ou avec l'outil `pg_basebackup`). Il discute aussi de la restauration d'une telle sauvegarde. Et enfin, il montre quelques outils disponibles, comme `barman` et `pitrery`.

NB : `pg_receivewal` s'appelait `pg_receivexlog` avant PostgreSQL 10.

5.2 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 continue 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.

5.2.1 PRINCIPES

- Les journaux de transactions contiennent toutes les modifications
- Il faut les archiver
- ... et avoir une image des fichiers à un instant t
- La restauration se fait en restaurant cette image
- ... et en rejouant les journaux
  - entièrement
  - partiellement (*ie* jusqu'à un certain moment)

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. 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).

5.2.2 AVANTAGES

- 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 tout 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. Généralement, une sauvegarde `pg_dump` s'exécute toutes les nuits, disons à 3 h du matin. Supposons qu'un gros problème survient à midi. S'il faut restaurer la dernière sauvegarde, la perte de données sera de 9 h. Le volume maximum de données perdu correspond à l'espacement des sauvegardes. 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 : il faut changer de fichier de journal pour que le journal précédent soit archivé et les fichiers de journaux sont de taille fixe.

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 maximale correspondant à cet intervalle.

5.2.3 INCONVÉNIENTS

- Sauvegarde de l'instance complète
- Nécessite un grand espace de stockage (données + journaux)
- Risque d'accumulation des journaux en cas d'échec d'archivage
- Restauration de l'instance complète
- Impossible de changer d'architecture
- Plus complexe

Certains inconvénients viennent directement du fait qu'on copie les fichiers : sauvegarde et restauration complète (impossible de ne restaurer qu'une seule base ou que quelques tables), restauration sur la même architecture (32/64 bits, *little/big endian*), voire probablement le même système d'exploitation.

Elle nécessite en plus un plus grand espace de stockage car il faut sauvegarder les fichiers (dont les index) ainsi que les journaux de transactions sur une certaine période, ce qui peut être volumineux (en tout cas beaucoup plus que des `pg_dump`).

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 surveiller la taille du `pg_wal`.

Enfin, cette méthode est plus complexe à mettre en place qu'une sauvegarde `pg_dump`. Elle nécessite plus d'étapes, une réflexion sur l'archicture à mettre en œuvre et une meilleure compréhension des mécanismes internes à PostgreSQL pour en avoir la maîtrise.

5.3 MISE EN PLACE

- 2 étapes :
  - Archivage des journaux de transactions
  - par archiver
  - par pg_receivewal
  - Sauvegarde des fichiers
  - manuellement (outils de copie classiques)
  - pg_basebackup

Même si la mise en place est plus complexe qu'un `pg_dump`, elle 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.

5.3.1 MÉTHODES D'ARCHIVAGE

- Deux méthodes
  - processus archiver
  - pg_receivewal sur un serveur secondaire

17.12

La méthode historique est la méthode utilisant le processus archiver. Ce processus fonctionne sur le serveur à sauvegarder et est de la responsabilité du serveur PostgreSQL. Seule sa (bonne) configuration incombe au DBA.

Une méthode plus récente a vu le jour : `pg_receivewal`. Cet outil se comporte comme un serveur secondaire. Il reconstitue les journaux de transactions à partir du flux de réplication.

Chaque solution a ses avantages et inconvénients qu'on étudiera après avoir détaillé leur mise en place.

5.3.2 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 qui va copier les journaux à l'extérieur de son répertoire de travail :

- sur un disque différent du même serveur ;
- sur un disque d'un autre serveur ;
- sur des bandes, un CDROM, etc.

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 NFS.

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 Unix (ou Windows) `postgres` peut écrire dedans :

```
$ mkdir /media/nfs1/archivage
$ chown postgres:postgres /media/nfs/archivage
```

5.3.3 PROCESSUS ARCHIVER - CONFIGURATION

- configuration (postgresql.conf)
  - wal_level = replica
  - archive_mode = on ou always

  - archive_command = '... une commande ...'
  - archive_timeout = 0
- Ne pas oublier de forcer l'écriture de l'archive sur disque

Après avoir créé le répertoire d'archivage, il faut configurer PostgreSQL pour lui indiquer comment archiver.

Le premier paramètre à modifier est `wal_level`. Ce paramètre indique le niveau des informations écrites dans les journaux de transactions. Avec un niveau `minimal`, PostgreSQL peut simplement utiliser les journaux en cas de crash pour rendre les fichiers de données cohérents au redémarrage. Dans le cas d'un archivage, il faut écrire plus d'informations, d'où l'utilisation du niveau `replica`.

Avant la version 9.6, il existait deux niveaux intermédiaires pour le paramètre `wal_level` : `archive` et `hot_standby`. Le premier permettait seulement l'archivage, le second permettait en plus d'avoir un serveur secondaire en lecture seule. Ces deux valeurs ont été fusionnées en `replica` avec la version 9.6. Les anciennes valeurs sont toujours acceptées, et remplacées silencieusement par la nouvelle valeur.

Après cela, il faut activer le mode d'archivage en positionnant le paramètre `archive_mode` à `on`. Depuis la version 9.5, il est aussi possible de mettre la valeur `always` pour qu'un esclave puisse aussi archiver les journaux de transactions. Enfin, la commande d'archivage s'indique au niveau du paramètre `archive_command`. PostgreSQL laisse le soin à l'administrateur de définir la méthode d'archivage des journaux de transaction suivant son contexte. 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, alors que le joker `%f` correspond au nom du fichier correspondant au journal de transactions une fois archivé.

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 rapidement 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.

Il est aussi possible d'y placer le nom d'un script bash, perl ou autres. L'intérêt est de pouvoir faire plus qu'une simple copie. On peut y ajouter la demande de synchronisation du cache sur disque. Il peut aussi être intéressant de tracer l'action de l'archivage par exemple, ou encore de compresser le journal avant archivage. Il faut s'assurer d'une seule chose : la commande d'archivage doit retourner 0 en cas de réussite et surtout

une valeur différente de 0 en cas d'échec. Si la commande renvoie autre chose que 0, PostgreSQL va tenter périodiquement d'archiver le fichier jusqu'à ce que la commande réussisse (autrement dit, renvoie 0). Du coup, il est important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur (disque plein, changement de bande, etc.).

Surveiller que la commande fonctionne bien peut se faire simplement en vérifiant la taille du répertoire `pg_wal`. Si ce répertoire commence à grossir fortement, c'est que PostgreSQL n'arrive plus à recycler ses journaux de transactions et ce comportement est un indicateur assez fort d'une commande d'archivage n'arrivant pas à faire son travail. Autre possibilité plus sûre et plus simple: vérifier le nombre de fichiers apparaissant dans le répertoire `pg_wal/archive_status` dont le suffixe est `.ready`. Ces fichiers, de taille nulle, indiquent en permanence quels sont les journaux prêts à être archivés. Théoriquement, leur nombre doit donc rester autour de 0 ou 1. La sonde `check_pgactivity` propose d'ailleurs une action pour faire ce test automatiquement. Voir la sonde ready_archives[17] pour plus de détails.

Si l'administrateur souhaite s'assurer qu'un archivage a lieu au moins à une certaine fréquence, il peut configurer un délai maximum avec le paramètre `archive_timeout`. L'impact de ce paramètre est d'archiver des journaux de transactions partiellement remplis. Or, ces fichiers ayant une taille fixe, nous archivons toujours 16 Mo par fichier pour un ratio de données utiles beaucoup moins important. La consommation en terme d'espace disque est donc plus importante et le temps de restauration plus long. Ce comportement est désactivé par défaut.

5.3.4 PROCESSUS ARCHIVER - RECHARGEMENT DE LA CONFIGURATION

- Par redémarrage de PostgreSQL
  - si modification de wal_level et/ou archive_mode
- Par envoi d'un signal à PostgreSQL

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 l'un des paramètres `wal_level` et `archive_mode` a été modifié, il faudra relancer PostgreSQL.

---

[17] https://github.com/OPMDG/check_pgactivity

5.3.5 PG_RECEIVEWAL - EXPLICATIONS

- Utilise le protocole de réplication
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
- Va plus loin que l'archivage standard
  - pas de `archive_timeout` car toujours au plus près du maître
- Nécessité d'utiliser les slots de réplication

`pg_receivewal` est un nouvel outil permettant de se faire passer pour un esclave dans le but d'archiver des journaux de transactions au plus près du maître grâce à l'utilisation du protocole de réplication en flux.

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`, les journaux de transactions étant écrits au fil de l'eau. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le maître.

Cet outil utilise les même 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.

Voici l'aide de cet outil :

```
pg_receivewal receives PostgreSQL streaming transaction logs.

Usage:
  pg_receivewal [OPTION]...

Options:
 -D, --directory=DIR    receive transaction log files into this directory
     --if-not-exists    do not error if slot already exists when creating a slot
 -n, --no-loop          do not loop on connection lost
 -s, --status-interval=SECS
                        time between status packets sent to server (default: 10)
 -S, --slot=SLOTNAME    replication slot to use
```

```
      --synchronous      flush transaction log immediately after writing
  -v, --verbose          output verbose messages
  -V, --version          output version information, then exit
  -?, --help             show this help, then exit

Connection options:
  -d, --dbname=CONNSTR   connection string
  -h, --host=HOSTNAME    database server host or socket directory
  -p, --port=PORT        database server port number
  -U, --username=NAME    connect as specified database user
  -w, --no-password      never prompt for password
  -W, --password         force password prompt (should happen automatically)

Optional actions:
    --create-slot      create a new replication slot (for the slot's name see
                       --slot)
    --drop-slot        drop the replication slot (for the slot's name see --slot)

Report bugs to <pgsql-bugs@postgresql.org>.
```

5.3.6 PG_RECEIVEWAL - CONFIGURATION SERVEUR

- Modification du fichier `postgresql.conf`

```
max_wal_senders = 3
max_replication_slots = 1
```

- Modification du fichier `pg_hba.conf`

```
host  replication  repli_user  192.168.0.0/24  md5
```

- Création de l'utilisateur de réplication

```
CREATE ROLE repli_user LOGIN REPLICATION PASSWORD 'supersecret'
```

- Redémarrage du serveur PostgreSQL
- Création d'un slot de réplication

```
SELECT pg_create_physical_replication_slot('archivage');
```

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. 3 est une valeur moyenne intéressante. Le paramètre `max_replication_slots` indique le nombre maximum de slots de réplication. Nous n'allons en créer qu'un seul.

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 est strictement pour des raisons de sécurité.

Une fois toutes ces modifications effectuées, il est nécessaire de redémarrer le serveur PostgreSQL.

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`.

5.3.7 PG_RECEIVEWAL - LANCEMENT DE L'OUTIL

- Exemple de lancement

```
pg_receivewal -D /data/archives -S archivage
```

- Plein d'autres options
  - notamment pour la connexion (-h, -p, -U)
- 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 maître
  - pg_receivewal s'arrête

Une fois le serveur PostgreSQL redémarré, 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 000000010000000000000E*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000F*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000010.partial*
```

En cas d'incident sur le maître, il est alors possible de partir d'une sauvegarde binaire et de rejouer les journaux de transactions disponibles (sans oublier de supprimer l'extension

`.partial` du dernier journal).

Il ne faut pas oublier de mettre en place un script de démarrage pour que `pg_receivewal` soit redémarré en cas de redémarrage du serveur.

5.3.8 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 (les quelques dernières transactions)

La méthode archiver est la méthode la plus simple à mettre en place. Elle se lance au lancement du serveur 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`), 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 être écrit et installé. Ceci étant dit, un exemple existe[18] . Cette méthode nécessite une configuration plus importante du serveur PostgreSQL. Par contre, cette méthode a le gros avantage de ne perdre pratiquement aucune transaction. 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.

5.3.9 SAUVEGARDE MANUELLE - 1/3

- Appel de la procédure stockée `pg_start_backup()`
- Argument 1
  - un label, libre
- Argument 2, optionnel

[18] https://github.com/opendbteam/pg_receivexlog-RHEL-CENTOS-startup-script/

  - un booléen indiquant si le CHECKPOINT doit être forcé
- Argument 3, optionnel
  - un booléen indiquant si la sauvegarde est concurrente
- Aucun impact pour les utilisateurs

La sauvegarde a lieu en trois temps.

Tout d'abord, il faut exécuter une procédure stockée appelée `pg_start_backup()`.

Cette procédure va réaliser entre autres choses un *checkpoint*. Le deuxième argument de cette fonction permet de préciser si on veut que ce *checkpoint* s'exécute immédiatement ou si on accepte d'attendre un certain temps (induit par la rapidité d'écriture imposé par `checkpoint_completion_target`).

Ensuite, la procédure va créer un fichier appelé `backup_label` dans le répertoire des données de PostgreSQL. Dans ce fichier, elle indique le journal de transactions et l'emplacement actuel dans le journal de transactions du *checkpoint* ainsi que le label précisé en premier argument de la procédure stockée. Ce label permet d'identifier l'opération de sauvegarde. Voici un exemple de ce fichier `backup_label` :

```
$ cat $PGDATA/backup_label
START WAL LOCATION: 8/DE000020 (file 000000010000000800000000DE)
CHECKPOINT LOCATION: 8/DE000020
START TIME: 2010-01-26 10:49:05 CET
LABEL: backup_full_2010_01-26
```

Ce fichier empêche l'exécution de deux sauvegardes PITR en parallèle. La version 9.6 supprime cette limitation en proposant de faire une sauvegarde concurrente. Pour cela, il faut indiquer un troisième argument booléen à `pg_start_backup()`. À `true`, le fichier `backup_label` n'est pas créé, ce qui permet l'exécution d'une autre sauvegarde PITR en parallèle. À `false`, le fichier est crée et le comportement est identique à celui des versions antérieures à la version 9.6. Les contraintes des sauvegardes en parallèle sont plus importantes. En effet, la session qui exécute la commande `pg_start_backup()` doit être la même que celle qui exécute `pg_stop_backup()`. Si la connexion venait à être interrompue entre-temps, alors la sauvegarde doit être considérée comme invalide. De plus il n'y a plus de fichier `backup_label` et c'est la commande `pg_stop_backup()` qui renvoie les informations qui s'y trouvaient ; elle se charge dans le même temps de créer dans `pg_wal` un fichier `000000010000000010000000XX.000000XX.backup` contenant les informations de fin de sauvegarde. Si vous voulez implémenter cette nouvelle méthode, il vous faudra donc récupérer et conserver vous-même les informations renvoyées par la commande de fin de sauvegarde. La sauvegarde PITR devient donc plus complexe, et il est donc recommandé d'utiliser plutôt `pg_basebackup` ou des outils supportant ces fonctionnalités (`pitrery`, `pg_backrest`...).

L'exécution de `pg_start_backup()` peut se faire depuis n'importe quelle base de données de l'instance. Le choix de la base n'a aucune importance en soi, et le label n'a aucune importance pour PostgreSQL (il ne sert qu'à l' administrateur, à reconnaître le backup).

Après exécution de cette procédure, les utilisateurs peuvent continuer à travailler normalement, aucune opération ne leur est interdite.

5.3.10 SAUVEGARDE MANUELLE - 2/3

- Sauvegarde des fichiers à chaud
  - le répertoire principal des données
  - les tablespaces
- Ignorer
  - postmaster.pid
  - log
  - pg_wal
  - pg_replslot

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.

La sauvegarde se fait à chaud. Il est donc possible que certains fichiers changent pendant la sauvegarde, cela n'a pas d'importance en soi. Cependant, il **faut** s'assurer que l'outil de sauvegarde continue son travail malgré tout. Si vous disposez d'un outil capable de différencier les codes d'erreurs dus à « des fichiers ont bougé ou disparu lors de la sauvegarde » des autres erreurs techniques, c'est un avantage. Le tar GNU par exemple retourne 1 pour le premier cas d'erreur, et 2 quand l'erreur est critique.

Peu d'outils sont capables de copier des fichiers en cours de modification sur les plateformes Microsoft Windows. Assurez-vous d'en utiliser un possédant cette fonctionnalité. À noter l'outil tar (ainsi que d'autres issus du projet GNU) est disponible nativement à travers le projet unxutils[19] .

Sur les fichiers et répertoires à ignorer, voici la liste exhaustive (disponible aussi dans la documentation officielle[20]) :

- `postmaster.pid`
- `postmaster.opts`

[19] http://unxutils.sourceforge.net/
[20] http://docs.postgresql.fr/current/protocol-replication.html

- différents fichiers temporaires créés pendant l'opération du serveur PostgreSQL
- `pg_wal`, ainsi que les sous-répertoires
- `pg_replslot` est copiée sous la forme d'un répertoire vide
- les fichiers autres que les fichiers et les répertoires standards

---

5.3.11 SAUVEGARDE MANUELLE - 3/3

- Appel de la procédure stockée `pg_stop_backup()`

La dernière étape correspond à l'exécution de la procédure stockée `pg_stop_backup()`. PostgreSQL va :

- marquer cette fin de backup dans le journal ;
- forcer la finalisation du journal de transactions courant et donc son archivage pour que la sauvegarde soit utilisable y compris en cas de crash immédiat ;
- archiver le fichier `backup_label` sous un autre nom (en cas de sauvegarde non concurrente).

En cas de sauvegarde concurrente, cette fonction nous renvoie l'équivalent du contenu du fichier `backup_label`. Ce contenu doit être conservé pour créer le fichier `backup_label` à stocker avec la sauvegarde des fichiers.

À partir du moment où cette fonction nous rend la main, il est possible de restaurer la sauvegarde obtenue et 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, le nom des journaux ne croit pas de manière hexadécimale simple (il y a plusieurs compteurs hexadécimaux différents dans le nom du fichier journal, qui ne sont pas incrémentés de gauche à droite).

5.3.12 PG_BASEBACKUP

- Réalise les différentes étapes d'une sauvegarde
  - ... via une connexion de réplication
- Configuration de réplication à faire sur le serveur à sauvegarder
- Exécution de pg_basebackup sur le serveur de sauvegarde
- Copie intégrale, pas d'incrémental

- Possible d'indiquer un slot de réplication (9.6)

```
$ pg_basebackup -Ft -x -c fast -P \
                -h 127.0.0.1 -U sauve -D sauve_20120625
```

pg_basebackup permet de réaliser toute la sauvegarde de la base, à distance, via une connexion PostgreSQL. Il est donc simple à mettre en place et à utiliser, et permet d'éviter de nombreuses étapes vu précédemment. Par contre, il ne permet pas de réaliser une sauvegarde incrémentale, contrairement à une méthode de sauvegarde comme rsync.

pg_basebackup nécessite une connexion de réplication. Il faut donc configurer le serveur pour accepter la connexion de pg_basebackup. Cela se passe dans un premier temps au niveau du fichier postgresql.conf. Le paramètre max_wal_senders doit avoir une valeur assez élevée pour autoriser cette connexion :

```
max_wal_senders = 1
```

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 :

```
host  replication  sauve  127.0.0.1/32  trust
```

Enfin, il faut créer l'utilisateur sauve qui sera le rôle créant la connexion :

```
$ psql -c "CREATE ROLE sauve LOGIN REPLICATION;" postgres
```

Notez qu'il serait préférable de mettre en place un mot de passe pour cet utilisateur et de forcer son utilisation avec une méthode comme md5. Nous ne le ferons pas ici.

Il ne reste plus qu'à lancer pg_basebackup :

```
$ pg_basebackup -Ft -x -c fast -P -h 127.0.0.1 -U sauve -D sauve_20120625
4163766/4163766 kB (100%), 1/1 tablespace
$ ll sauve_20120625
total 4163772
-rw-rw-r--. 1 guillaume guillaume 4263697408 Jun 25 15:16 base.tar
```

À partir de la version 9.6, il est possible d'indiquer un slot de réplication à pg_basebackup.

5.3.13 FRÉQUENCE DE LA SAUVEGARDE

- 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.

5.4 RESTAURER UNE SAUVEGARDE PITR

- Une procédure relativement simple
- Mais qui doit être effectuée rigoureusement

La restauration se déroule en trois voire quatre étapes suivant qu'elle est effectuée sur le même serveur ou sur un autre serveur.

5.4.1 RESTAURER UNE SAUVEGARDE PITR (1/4)

- S'il s'agit du même serveur
  - arrêter PostgreSQL
  - supprimer le répertoire des données
  - supprimer les tablespaces

Dans le cas où la restauration a lieu sur le même serveur, quelques étapes préliminaires sont à effectuer.

Il faut arrêter PostgreSQL s'il n'est pas arrêté. Cela arrivera quand la restauration a pour but de récupérer des données qui ont été supprimées par erreur par exemple.

Ensuite, il faut supprimer (ou archiver) l'ancien répertoire des données pour pouvoir y placer l'ancienne sauvegarde des fichiers. Écraser l'ancien répertoire n'est pas suffisant. Il faut réellement supprimer le répertoire actuel. Par conséquent, il faut aussi supprimer les répertoires des tablespaces au cas où l'instance en possède.

5.4.2 RESTAURER UNE SAUVEGARDE PITR (2/4)

- Restaurer les fichiers de la sauvegarde
- Supprimer les fichiers compris dans le répertoire pg_wal restauré
  - ou mieux, ne pas les avoir inclus dans la sauvegarde initialement
- Restaurer le dernier journal de transactions connu (si disponible).

17.12

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. Par exemple, le fichier postmaster.pid peut poser un problème au démarrage. Conserver les journaux applicatifs n'est pas en soi un problème mais peut porter à confusion. Il est donc préférable de les supprimer. Quant aux journaux de transactions compris dans la sauvegarde, bien que ceux en provenances des archives seront utilisés même s'ils sont présents aux deux emplacements, il est préférable de les supprimer. La commande sera similaire à celle-ci :

```
$ rm postmaster.pid log/* pg_wal/[0-9A-F]*
```

Enfin, s'il est possible d'accéder au journal de transactions courant au moment de l'arrêt de l'ancienne instance, il est intéressant de le restaurer dans le répertoire pg_wal fraîchement nettoyé. Ce dernier sera pris en compte en toute fin de restauration des journaux depuis les archives et permettra donc de restaurer les toutes dernières transactions validées sur l' ancienne instance, mais pas encore archivées.

5.4.3 RESTAURER UNE SAUVEGARDE PITR (3/4)

- Configuration (recovery.conf)
  - restore_command = '... une commande ...'
- Si restauration jusqu'à un certain moment
  - recovery_target_name, recovery_target_time
  - recovery_target_xid, recovery_target_lsn
  - recovery_target_inclusive
- Divers
  - recovery_target_timeline
  - pause_at_recovery_target

La restauration se configure dans un fichier spécifique, appelé `recovery.conf`.

Pour le créer, inspirez-vous de celui fourni avec PostgreSQL (sur RedHat/CentOS : `/usr/pgsql-10/share/recovery.conf.sample`, et sur Debian : `/usr/share/postgresql/10/recovery.conf.sample`).

Le paramètre essentiel est `restore_command`. Il est le pendant du paramètre `archive_command` pour l'archivage. Si nous poursuivons notre exemple, ce paramètre pourrait être :

```
restore_command = 'cp /mnt/nfs1/archivage/%f %p'
```

Si le but est de restaurer tous les journaux archivés, il n'est pas nécessaire d'aller plus loin dans la configuration. Dans le cas contraire, trois paramètres permettent de préciser jusqu'à quel point il est possible d'exécuter la restauration :

- jusqu'à un certain nom, grâce au paramètre `recovery_target_name` ;
- jusqu'à une certaine heure, grâce au paramètre `recovery_target_time` ;
- jusqu'à un certain identifiant de transactions, grâce au paramètre `recovery_target_xid` ;
- jusqu'à un certain LSN, grâce au paramètre `recovery_target_lsn`.

Le nom correspond à un label enregistré précédemment dans les journaux de transactions grâce à la fonction `pg_create_restore_point()`.

Il est possible de préciser si la restauration se fait en incluant les transactions au nom, à l'heure ou à l'identifiant de transactions, ou en les excluant. Il s'agit du paramètre `recovery_target_inclusive`. À noter qu'il n' existe actuellement aucun outil pour récupérer facilement un numéro de transaction particulier.

Il est aussi possible de demander à la restauration de marquer une pause une fois arrivé à ce niveau. Cela permet à l'utilisateur de vérifier que le serveur est bien arrivé au point qu'il désirait. Si c'est le cas, un appel à la fonction `pg_wal_replay_resume()` provoquera l'arrêt de la restauration. Et si ce n'est pas le cas, l'arrêt de PostgreSQL et le changement de la cible de restauration dans le fichier `recovery.conf` permettra de lui demander de continuer la restauration.

5.4.4 RESTAURER UNE SAUVEGARDE PITR (4/4)

- Démarrer PostgreSQL

La dernière étape est particulièrement simple. Il suffit de démarrer PostgreSQL.

Ceci fait, PostgreSQL va comprendre qu'il doit rejouer les journaux de transactions et s'en acquittera jusqu'à arriver à la limite fixée, jusqu'à ce qu'il ne trouve plus de journal à rejouer, 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).

5.4.5 RESTAURATION PITR : DIFFÉRENTES TIMELINES

- En fin de *recovery*, la *timeline* change :
  - L'historique des données prend une autre voie

  - Le nom des WAL change pour éviter d'écraser des archives suivant le point d'arrêt
  - L'aiguillage est inscrit dans un fichier `.history`, archivé
- Permet de faire plusieurs restaurations PITR à partir du même *basebackup*
- `recovery_target_timeline` permet de choisir la *timeline* à suivre

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 se matérialise par un ensemble de journaux de transactions, identifiée par un numéro. Le numéro de la *timeline* est le premier nombre hexadécimal du nom des segments de journaux de transactions (le second est le numéro du journal et le troisième le numéro du segment). Lorsqu'une instance termine une restauration PITR, elle peut archiver immédiatement ces 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
```

A la sortie du mode *recovery*, l'instance doit choisir une nouvelle *timeline*. Les *timelines* connues avec leur point de départ sont suivies grâce aux fichiers *history*, nommés d'après le numéro hexadécimal sur huit caractères de la *timeline* et le suffixe `.history`, et archivés avec les fichiers WAL. En partant de la *timeline* qu'elle quitte, l'instance restaure les fichiers *history* des *timelines* suivantes pour choisir la première disponible, et archive un nouveau fichier `.history` pour la nouvelle *timeline* sélectionnée, avec l'adresse du point de départ dans la *timeline* qu'elle quitte :

```
$ cat 00000002.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
```

Après une seconde restauration, ciblant la *timeline* 2, l'instance choisit la *timeline* 3 :

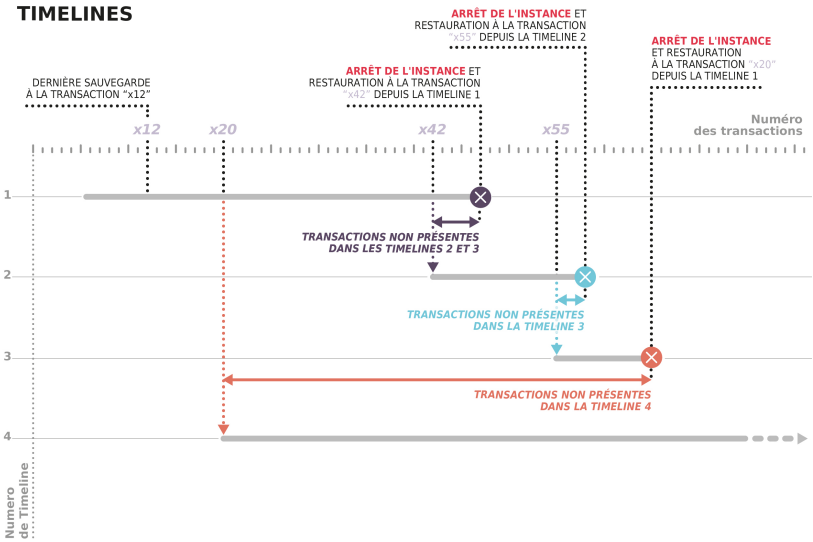
```
$ cat 00000003.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02


2   0/105AF7D0  before 2015-10-22 10:25:56.614316+02
```

On peut choisir la *timeline* cible en configurant le paramètre `recovery_target_timeline` dans le fichier `recovery.conf`. Par défaut, la restauration se fait dans la même *timeline* que le *base backup*. Pour choisir une autre *timeline*, il faut donner le numéro hexadécimal de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. On peut aussi indiquer `'latest'` pour que PostgreSQL détermine la *timeline* la plus récente en cherchant les fichiers *history*. Il prend alors le premier ID de *timeline* disponible. Attention, pour restaurer dans une *timeline* précise, il faut que le fichier *history* correspondant soit présent dans les archives, sous peine d'erreur.

En sélectionnant la *timeline* cible, on peut alors effectuer plusieurs restaurations successives à partir du même *base backup*.

5.4.6 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 configure le `recovery.conf` 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 le `recovery.conf` 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 `recovery.conf` 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.

5.5 POUR ALLER PLUS LOIN

- Gagner en place
  - ... en compressant les journaux de transactions
- Se faciliter la vie avec différents outils
  - barman
  - pitrery

5.5.1 COMPRESSER LES JOURNAUX DE TRANSACTIONS

- Objectif : éviter de consommer trop de place disque
- Méthode recommandée
  - outils de compression standards : gzip, bzip2, lzma
- Méthode déconseillée
  - outil de compression spécialisé : pglesslog

L'un des problème de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Avec un journal généré toutes les cinq minutes, cela représente 16 Mo toutes les 5 minutes, soit 192 Mo par heure. Autrement dit 5 Go par jour. Il n'est pas possible dans certains cas de conserver autant de journaux. La solution est la compression à la volée et il existe deux types de compression.

La méthode la plus simple et la plus sûre pour les données est une compression non destructive, comme celle proposée par les outils gzip, bzip2, lzma, etc. La compression peut ne pas être très intéressante en terme d' espace disque gagné. Néanmoins, un fichier de 16 Mo aura généralement une taille comprise entre 3 et 6 Mo. Attention par ailleurs au temps de compression des journaux, qui peut entraîner un retard conséquent de l'archivage par rapport à l'écriture des journaux.

L'utilisation de pglesslog[21] est déconseillée. Cet outil supprime les pages complètes inutiles et tronque le fichier dans le cas de l'utilisation du paramètre `archive_timeout`. Bien que le gain en espace disque soit certain, les utilisateurs de cet outil ont rencontré trop de problèmes pour qu'il puisse être considéré fiable : de nombreuses instances se sont avérées impossibles à restaurer après un incident. Il est préférable de rester sur des outils de compression éprouvés.

---

[21] http://pglesslog.projects.postgresql.org/

5.5.2 BARMAN

- Gère la sauvegarde et la restauration
  - mode pull
  - multi-serveurs
- Une seule commande (barman)
- Et de nombreuses actions
  - list-server, backup, list-backup, recover, ...

barman est un outil créé par 2ndQuadrant. 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).

Site web de barman[22]

5.5.3 PITRERY

- Gère la sauvegarde et la restauration
  - mode push
  - mono-serveur
- Multi-commandes
  - archive_xlog
  - pitrery
  - restore_xlog

`pitrery` a la même raison d'exister que `barman`, et a été créé par la société Dalibo. Il permet de réaliser facilement la sauvegarde et la restauration de la base. Cet outil s'appuie sur des fichiers de configuration, un par serveur de sauvegarde, qui permettent de définir la destination de l' archivage, la destination des sauvegardes ainsi que la politique de rétention à utiliser.

`pitrery` propose trois commandes :

[22] http://www.pgbarman.org/

- archive_xlog qui gère l'archivage et la compression des journaux de transactions ;
- pitrery qui gère les sauvegardes et les restaurations ;
- restore_xlog pour restaurer les journaux archivés par archive_xlog. À n' utiliser que dans le fichier recovery.conf.

L'archivage des journaux de transactions est à configurer au niveau du fichier postgresql.conf :

```
wal_level = replica
archive_mode = on
archive_command = '/usr/local/bin/archive_xlog %p'
```

Lorsque l'archivage est fonctionnel, la commande pitrery peut être utilisée pour réaliser une sauvegarde :

```
$ pitrery backup
INFO: preparing directories in 10.100.0.16:/opt/backups/prod
INFO: listing tablespaces
INFO: starting the backup process
INFO: backing up PGDATA with tar
INFO: archiving /home/postgres/postgresql-9.0.4/data
INFO: backup of PGDATA successful
INFO: backing up tablespace "ts2" with tar
INFO: archiving /home/postgres/postgresql-9.0.4/ts2
INFO: backup of tablespace "ts2" successful
INFO: stopping the backup process
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
INFO: copying the backup history file
INFO: copying the tablespaces list
INFO: backup directory is 10.100.0.16:/opt/backups/prod/2013.08.28-11.16.30
INFO: done
```

Il est possible d'obtenir la liste des sauvegardes en ligne :

```
$ pitrery list
List of backups on 10.100.0.16:

Directory:
  /usr/data/pitrery/backups/pitr13/2013.05.31_11.44.02
Minimum recovery target time:
  2013-05-31 11:44:02 CEST
Tablespaces:

Directory:
```

```
  /usr/data/pitrery/backups/pitr13/2013.05.31_11.49.37
Minimum recovery target time:
  2013-05-31 11:49:37 CEST
Tablespaces:
  ts1 /opt/postgres/ts1 (24576)
```

pitrery gère également la politique de rétention des sauvegardes. Une commande de purge permet de réaliser la purge des sauvegardes en s'appuyant sur la configuration de la rétention des sauvegardes :

```
$ pitrery purge
INFO: searching backups
INFO: purging /home/postgres/backups/prod/2011.08.17-11.16.30
INFO: purging WAL files older than 000000020000000000000060
INFO: 75 old WAL file(s) removed
INFO: done
```

Enfin, pitrery permet de restaurer une sauvegarde et de préparer la configuration de restauration (`recovery.conf`) :

```
$ pitrery -c prod restore -d '2013-06-01 13:00:00 +0200'
INFO: searching backup directory
INFO: searching for tablespaces information
INFO:
INFO: backup directory:
INFO:   /opt/postgres/pitr/prod/2013.06.01_12.15.38
INFO:
INFO: destinations directories:
INFO:   PGDATA -> /opt/postgres/data
INFO:   tablespace "ts1" -> /opt/postgres/ts1 (relocated: no)
INFO:   tablespace "ts2" -> /opt/postgres/ts2 (relocated: no)
INFO:
INFO: recovery configuration:
INFO:   target owner of the restored files: postgres
INFO:   restore_command = 'restore_xlog -L -d /opt/postgres/archives %f %p'
INFO:   recovery_target_time = '2013-06-01 13:00:00 +0200'
INFO:
INFO: checking if /opt/postgres/data is empty
INFO: checking if /opt/postgres/ts1 is empty
INFO: checking if /opt/postgres/ts2 is empty
INFO: extracting PGDATA to /opt/postgres/data
INFO: extracting tablespace "ts1" to /opt/postgres/ts1
```

17.12

```
INFO: extracting tablespace "ts2" to /opt/postgres/ts2
INFO: preparing pg_wal directory
INFO: preparing recovery.conf file
INFO: done
INFO:
INFO: please check directories and recovery.conf before starting the cluster
INFO: and do not forget to update the configuration of pitrery if needed
INFO:
```

Il ne restera plus qu'à redémarrer le serveur et surveiller les journaux applicatifs pour vérifier qu' aucune erreur ne se produit au cours de la restauration.

Site Web de pitrery[23] .

5.6 CONCLUSION

- Une sauvegarde
  - Fiable
  - Éprouvée
  - Rapide
  - Continue
- Mais
  - Plus complexe à mettre en place
  - 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).

5.7 TRAVAUX PRATIQUES

5.7.1 ÉNONCÉS

Pour simplifier les choses, merci de ne pas créer de tablespaces. La solution fournie n'est valable que dans ce cas précis. Dans le cas contraire, il vous faudra des sauvegardes séparées de chaque tablespace.

[23] http://dalibo.github.io/pitrery/

Sauvegarde

- Mettre en place l'archivage des journaux de transactions dans `/opt/pgsql/archives`.
- Créer une table suivante comprenant deux champs, le premier de type `integer` (plus exactement `serial`), le deuxième de type `text`.
- Écrire un script qui insère une grande quantité de données dans cette table. Vérifier que les journaux de transactions sont bien archivés dans le répertoire prévu pour ça. Laisser le script s'exécuter pendant la durée de la sauvegarde de l'instance.
- Sauvegarder l'instance à l'aide d'un utilitaire d'archivage (`tar` par exemple). À la fin de la sauvegarde, relever la valeur de la colonne id du dernier élément inséré dans la table créée précédemment.
- Attendre qu'un nouveau journal de transactions soit archivé dans le répertoire d'archivage, puis arrêter l'insertion des données dans la table créée précédemment (il est aussi possible de forcer la bascule avec `select pg_switch_wal()`).

Restauration

- Renommer le répertoire `/var/lib/pgsql/10/data` en `/var/lib/pgsql/10/data.old`, à froid.
- Restaurer l'instance en utilisant la sauvegarde à chaud et les journaux de transactions.
- Récupérer l'id du dernier élément inséré dans la table créée précédemment. Vérifier que cet id est supérieur à l'id relevé lors de la fin de la sauvegarde.

Utilisation de pg_receivewal et des slots de réplication

- Mettre en place pg_receivewal en lui donnant un slot de réplication

Utilisation de barman (Optionnel)

- Installer barman.
- Configurer barman pour la sauvegarde du serveur local.
- Faire une sauvegarde.
- Lister les sauvegardes.
- Afficher les informations sur une sauvegarde.
- Faire une restauration.

Utilisation de pitrery (Optionnel)

- Installer pitrery.
- Configurer pitrery pour la sauvegarde du serveur local.
- Faire une sauvegarde.
- Lister les sauvegardes.
- Faire une restauration.

5.7.2 SOLUTIONS

Toutes les opérations de sauvegarde et restauration sont exécutées avec l'utilisateur `postgres`. Lorsqu'un autre utilisateur est utilisé, ceci est précisé explicitement.

Sauvegarde

Mettre en place l'archivage des journaux de transactions dans `/opt/pgsql/archives`.

Pour mettre en place l'archivage des journaux de transactions dans `/opt/pgsql/archives`, il faut positionner la variable `archive_command` comme suit dans le fichier de configuration `postgresql.conf` :

```
archive_command = 'rsync %p /opt/pgsql/archives/%f'
```

Si c'est nécessaire pour votre version, positionner aussi les paramètres `wal_level` à `archive`, et `archive_mode` à `on`.

Créer le répertoire `/opt/pgsql/archives` (en tant que root):

```
# mkdir -p /opt/pgsql/archives
# chown -R postgres /opt/pgsql
```

Puis faire relire la configuration à PostgreSQL (avec l'utilisateur `root`) :

```
# /etc/init.d/postgresql reload
```

(Un redémarrage est nécessaire si les paramètres `wal_level` et `archive_mode` ont été modifiés.)

Créer la table.

```
$ psql
cave=> CREATE TABLE dummy (id serial, libelle text);
```

Écrire un script qui insère une grande quantité de données dans cette table.

```
#!/bin/bash

while true
do
    cat <<_QUERY | psql
INSERT INTO dummy (libelle)
  SELECT CURRENT_TIMESTAMP - (v ||' minutes')::interval
  FROM generate_series(1,5000) AS t(v)
_QUERY
    sleep 1
done
```

Vérifier que les journaux de transactions sont bien générés dans le répertoire d'archivage.

```
$ ls -l /opt/pgsql/archives
```

Sauvegarder l'instance à l'aide d'un utilitaire d'archivage.

Indiquer à PostgreSQL que vous allez faire une sauvegarde de fichiers :

```
$ psql
postgres=> SELECT pg_start_backup('backup '||current_timestamp, true);
```

Sauvegarder les fichiers d'instance :

```
$ cd /var/lib/postgresql/10/main
$ tar -cvhz . -f /opt/pgsql/backups/backup_$(date +%F).tgz
```

Indiquer la fin de la sauvegarde des fichiers.

```
$ psql postgres
postgres=> SELECT pg_stop_backup();
```

À la fin de la sauvegarde, relever l'id du dernier élément inséré dans la table créée précédemment.

```
$ psql -U caviste cave
cave=> select MAX(id) from dummy;
```

Attendre qu'un nouveau journal de transactions soit généré dans le répertoire d'archivage, puis arrêter l'insertion des données dans la table créée précédemment.

Restauration

Arrêter PostgreSQL (en tant qu'utilisateur root) :

```
# /etc/init.d/postgresql stop
```

Renommer le répertoire `/var/lib/postgresql/10/main` en `/var/lib/postgresql/10/main.old` :

```
$ mv /var/lib/postgresql/10/main /var/lib/postgresql/10/main.old
```

Restaurer l'instance en utilisant la sauvegarde à chaud et les journaux de transactions.

Restaurer l'archive de la dernière sauvegarde :

```
$ cd /var/lib/postgresql/10
$ mkdir main
$ cd main
$ tar xzvf /opt/pgsql/backups/backup_$(date +%F).tgz
```

Effacer les anciens journaux et le fichier PID :

```
$ rm -f pg_wal/00* data/postmaster.pid
```

Copier éventuellement le journal de transactions courant de l'ancien répertoire :

- Déterminer le journal de transactions (c'est le dernier de la liste)

```
$ ls -lrt /var/lib/postgresql/10/main.old/pg_wal/
```

- Puis le recopier

```
$ cp main.old/pg_wal/le_fichier_qu_on_vient_de_trouver main/pg_wal
```

Créer le fichier `recovery.conf` dans le répertoire `main` contenant les informations suivantes :

```
restore_command = 'cp /opt/pgsql/archives/%f %p'
```

Ce fichier peut aussi être copié depuis un fichier d'exemple fourni avec PostgreSQL (sur RedHat/CentOS : `/usr/pgsql-10/share/recovery.conf.sample`)

Re-démarrer le serveur PostgreSQL (avec l'utilisateur root) :

```
# /etc/init.d/postgresql start
```

Vérifier que la restauration est terminée :

```
$ ls data/recovery*
```

On devrait trouver `data/recovery.done`.

Récupérer l'id du dernier élément inséré dans la table `dummy`. Vérifier que cet id est supérieur à l'id relevé lors de la fin de la sauvegarde.

```
$ psql
cave=> select MAX(id) from dummy;
```

Utilisation de barman (Optionnel)

Installer barman

Il est préférable de passer par les paquets de la distribution. Dans le cas contraire, les développeurs de barman proposent leur propre RPM (http://sourceforge.net/projects/pgbarman/files/).

Sur Red Hat et affiliées, la commande suivante devrait suffire :

```
$ sudo yum install barman
```

alors que sur Debian et affiliées, il faudra utiliser la commande :

```
$ apt-get install barman
```

Il est possible que des dépendances soient à installer.

Configurer barman pour la sauvegarde du serveur local

barman utilise généralement un utilisateur sans droit spécifique (pas un administrateur comme root).

La première chose à faire concerne la connexion SSH. Il faut créer les clés SSH et les installer pour permettre une connexion sans mot de passe entre les deux serveurs (dans notre TP, il s'agit du même).

Ensuite, il faut s'assurer que l'utilisateur qui exécute `barman` puisse se connecter sur le serveur PostgreSQL. Ça demandera au moins une modification du fichier `pg_hba.conf` et peut-être même du `postgresql.conf`.

Reste ensuite le fichier de configuration, le voici :

```
[barman]
barman_home = /var/lib/barman
barman_user = barman
log_file = /var/log/barman/barman.log
;compression = gzip
;pre_backup_script = env | grep ^BARMAN
;post_backup_script = env | grep ^BARMAN
;pre_archive_script = env | grep ^BARMAN
;post_archive_script = env | grep ^BARMAN
configuration_files_directory = /etc/barman.d
;minimum_redundancy = 0
;retention_policy =
;bandwidth_limit = 4000
immediate_checkpoint = true
;network_compression = false

[localhost]
description =  "My own PostgreSQL Database"
ssh_command = ssh postgres@localhost
conninfo = host=localhost user=postgres
```

Ce fichier indique que l'utilisateur système est l'utilisateur barman. Les sauvegardes et journaux de transactions archivés seront placés dans `/var/lib/barman`. Le CHECKPOINT exécute par la fonction `pg_start_backup()` sera immédiat (*i.e.* on n'attend pas le CHECKPOINT planifié). Pour des raisons de facilité sur le TP, la description du seul hôte à sauvegarder se trouve dans ce fichier. Nous conseillons plutôt de faire un fichier par hôte et de les placer dans le répertoire pointé par la variable `configuration_files_directory` (`/etc/barman.d` ici). L'hôte `localhost` dispose de la commande SSH pour la connexion système et de la chaîne de connexion PostgreSQL.

Il faut ensuite configurer PostgreSQL pour qu'il archive au bon endroit. La commande suivante permet de savoir dans quel répertoire il faut archiver les journaux de transactions

17.12

:

```
$ barman show-server localhost | grep incoming_wals_directory
    incoming_wals_directory: /var/lib/barman/localhost/incoming
```

Il faut donc modifier la configuration du fichier `postgresql.conf` ainsi :

```
archive_command = 'rsync %p /var/lib/barman/localhost/incoming/%f'
```

Il est préférable de tester que la configuration est bonne. Cela se fait avec cette commande :

```
$ barman check localhost
Server localhost:
    ssh: OK
    PostgreSQL: OK
    archive_mode: OK
    archive_command: OK
    directories: OK
    retention policy settings: OK
    compression settings: OK
    minimum redundancy requirements: OK (have 0 backups, expected at least 0)
```

Si tout n'est pas à OK, c'est qu'un problème existe dans la configuration et doit être réglé avant d'aller plus loin.

Les deux problèmes habituels sont:

- ssh: FAILED. Dans ce cas, c'est que vous n'avez pas d'échange de clé entre l'utilisateur faisant fonctionner barman et l'utilisateur postgres de l'instance à sauvegarder
- PostgreSQL: FAILED. Dans ce cas, c'est que vous n'arrivez pas à vous connecter avec le protocole PostgreSQL à l'instance. Soit le `pg_hba.conf` n'est pas bon, soit vous n'avez pas renseigné le mot de passe dans un fichier `.pgpass` pour l'utilisateur barman.

Faire une sauvegarde

```
$ barman backup localhost
Starting backup for server localhost in
    /var/lib/barman/localhost/base/20140214T100017
Backup start at xlog location: 0/95000028 (000000010000000000000095, 00000028)
Copying files.
Copy done.
Asking PostgreSQL server to finalize the backup.
```

```
Backup end at xlog location: 0/950000B8 (000000010000000000000095, 000000B8)
Backup completed
```

Lister les sauvegardes.

```
$ barman list-backup localhost
localhost 20140214T100017 - Fri Feb 14 10:00:27 2014 - Size: 672.2 MiB
                          - WAL Size: 0 B
```

Afficher les informations sur une sauvegarde.

```
$ barman show-backup localhost 20140214T100017
Backup 20140214T100017:
  Server Name       : localhost
  Status            : DONE
  PostgreSQL Version: 90302
  PGDATA directory  : /var/lib/postgresql/10/main
  Tablespaces:
    ts1: /home/guillaume/ts1 (oid: 24584)

  Base backup information:
    Disk usage      : 672.2 MiB
    Timeline        : 1
    Begin WAL       : 000000010000000000000095
    End WAL         : 000000010000000000000095
    WAL number      : 0
    Begin time      : 2014-02-14 10:00:17.276202
    End time        : 2014-02-14 10:00:27.729487
    Begin Offset    : 40
    End Offset      : 184
    Begin XLOG      : 0/95000028
    End XLOG        : 0/950000B8

  WAL information:
    No of files     : 0
    Disk usage      : 0 B
    Last available  : None

  Catalog information:
    Retention Policy: not enforced
    Previous Backup : - (this is the oldest base backup)
```

```
    Next Backup     : - (this is the latest base backup)
```

Faire une restauration

```
$ barman recover localhost 20140214T100017 /tmp/test_resto
Starting local restore for server localhost using backup 20140214T100017
Destination directory: /tmp/test_resto
Copying the base backup.
Copying required wal segments.
The archive_command was set to 'false' to prevent data losses.

Your PostgreSQL server has been successfully prepared for recovery!

Please review network and archive related settings in the PostgreSQL
configuration file before starting the just recovered instance.
```

Utilisation de pitrery (Optionnel)

Installer pitrery

Les développeurs de pitrery proposent leur propre RPM (https://dl.dalibo.com/public/pitrery/rpms/). Il est également possible de l'installer en compilant les sources.

Sur Red Hat et affiliées, après avoir téléchargé localement le RPM, la commande suivante devrait suffire :

```
$ sudo yum install pitrery-2.0-1.el7.centos.noarch.rpm
```

Des paquets existent également pour Debian et affiliées.

Configurer pitrery pour la sauvegarde du serveur local

Par défaut, le fichier de configuration créé est `/etc/pitrery/pitr.conf`. Créons en une copie vide.

```
$ sudo cp /etc/pitrery/pitr.conf /etc/pitrery/pitr.conf.bck
$ sudo echo > /etc/pitrery/pitr.conf
```

Configurons le ensuite de la manière suivante :

```
####################
# Backup management
####################
PGDATA="/var/lib/pgsql/10/data"
PGUSER="postgres"
BACKUP_DIR="/var/lib/pgsql/10/backups/pitr"
```

```
####################
# WAL archiving
####################
ARCHIVE_DIR="$BACKUP_DIR/archived_wal"
```

Ce fichier indique où se trouve notre répertoire de données `PGDATA`, les informations de connexion (ici uniquement `PGUSER`) ainsi que la configuration relative au stockage des backups et des journaux de transactions archivés.

Il convient ensuite de modifier la configuration du fichier `postgresql.conf` ainsi :

```
wal_level = replica
archive_mode = on
archive_command = '/usr/bin/archive_xlog %p'
```

pitrery fournit le script `archive_xlog` pour gérer la commande d'archivage. Par défaut, ce script utilisera le `pitr.conf` que nous venons de configurer.

Il faut redémarrer le service PostgreSQL si les paramètres wal_level ou archive_mode ont été modifiés, sinon un simple rechargement de la configuration suffit.

Il est préférable de tester que la configuration est bonne. Cela se fait avec cette commande :

```
$ pitrery check
INFO: Configuration file is: /etc/pitrery/pitr.conf
INFO: loading configuration
INFO: the configuration file contains:
PGDATA="/var/lib/pgsql/10/data"
PGUSER="postgres"
BACKUP_DIR="/var/lib/pgsql/10/backups/pitr"
ARCHIVE_DIR="$BACKUP_DIR/archived_wal"

INFO: ==> checking the configuration for inconsistencies
INFO: configuration seems correct
INFO: ==> checking backup configuration
INFO: backups are local, not checking SSH
INFO: target directory '/var/lib/pgsql/10/backups' exists
INFO: target directory '/var/lib/pgsql/10/backups' is writable
INFO: ==> checking WAL files archiving configuration
INFO: WAL archiving is local, not checking SSH
INFO: checking WAL archiving directory: /var/lib/pgsql/10/backups/pitr/archived_wal
INFO: target directory '/var/lib/pgsql/10/backups/pitr/archived_wal' exists
```

```
INFO: target directory '/var/lib/pgsql/10/backups/pitr/archived_wal' is writable
INFO: ==> checking access to PostgreSQL
INFO: psql command and connection options are: psql -X -X -U postgres
INFO: connection database is: postgres
INFO: environment variables (maybe overwritten by the configuration file):
INFO:   PGDATA=/var/lib/pgsql/10/data
INFO: PostgreSQL version is: 10.0
INFO: connection role can run backup functions
INFO: current configuration:
INFO:   wal_level = replica
INFO:   archive_mode = on
INFO:   archive_command = '/usr/bin/archive_xlog %p'
INFO: ==> checking access to PGDATA
INFO: PostgreSQL and the configuration reports the same PGDATA
INFO: permissions of PGDATA ok
INFO: owner of PGDATA is the current user
INFO: access to the contents of PGDATA ok
```

Faire une sauvegarde

```
$ pitrery backup
INFO: preparing directories in /var/lib/pgsql/10/backups/pitr
INFO: listing tablespaces
INFO: starting the backup process
INFO: performing a non-exclusive backup
INFO: backing up PGDATA with tar
INFO: archiving /var/lib/pgsql/10/data
INFO: stopping the backup process
INFO: copying the backup history file
INFO: copying the tablespaces list
INFO: copying PG_VERSION
INFO: backup directory is /var/lib/pgsql/10/backups/pitr/2017.07.31_16.50.23
INFO: done
```

Lister les sauvegardes.

```
$ pitrery list
List of local backups
/var/lib/pgsql/10/backups/pitr/2017.07.31_16.50.23  9.1M      2017-07-31 16:50:23 CEST
```

Faire une restauration

```
$ pitrery restore
```

```
INFO: searching backup directory
INFO: searching for tablespaces information
INFO:
INFO: backup directory:
INFO:   /var/lib/pgsql/10/backups/pitr/2017.07.31_16.50.23
INFO:
INFO: destinations directories:
INFO:   PGDATA -> /var/lib/pgsql/10/data
INFO:
INFO: recovery configuration:
INFO:   target owner of the restored files: postgres
INFO:   restore_command = '/usr/bin/restore_xlog %f %p'
INFO:
INFO: creating /var/lib/pgsql/10/data with permission 0700
INFO: extracting PGDATA to /var/lib/pgsql/10/data
INFO: extraction of PGDATA successful
INFO: preparing pg_xlog directory
INFO: preparing recovery.conf file
INFO: done
INFO:
INFO: please check directories and recovery.conf before starting the cluster
INFO: and do not forget to update the configuration of pitrery if needed
INFO:
```

6 POSTGRESQL AVANCÉ 1

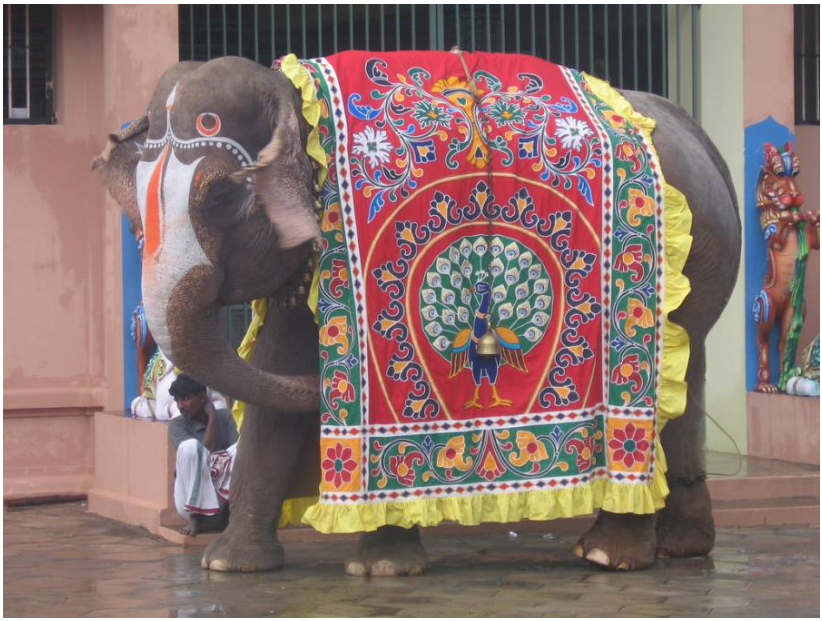


FIGURE 8: POSTGRESQL

6.1 PRÉAMBULE

Comme tous les SGBD-R, PostgreSQL fournit des fonctionnalités avancées.

Ce module présente les fonctionnalités orientées DBA.

6.2 VUES SYSTÈME

PostgreSQL propose de nombreuses vues système :

- Pour monitorer/remonter de la métrologie
- Pour diagnostiquer un incident
- Pour rapidement obtenir des informations sur le système

PostgresSQL propose, comme tout bon SGBD, de nombreuses vues, accessibles en SQL, pour obtenir des informations sur son fonctionnement interne. On peut donc avoir des informations sur le fonctionnement des bases, des processus d'arrière plan, des tables, les requêtes en cours…

6.2.1 PG_STAT_ACTIVITY

pg_stat_activity :

- Liste des processus en cours
  - sessions
  - processus en tâche de fond (10+)
- Requête en cours/dernière exécutée
- IDLE IN TRANSACTION
- Sessions en attente de verrou
- Gagne en informations au fil des versions

Cette vue donne la liste des sessions à l'instance (une ligne par session).

Au fil des versions, elle gagne de plus en plus d'informations. En version 10, elle dispose de ces colonnes :

Colonne	Type
datid	oid
datname	name
pid	integer
usesysid	oid
usename	name
application_name	text
client_addr	inet
client_hostname	text
client_port	integer
backend_start	timestamp with time zone
xact_start	timestamp with time zone
query_start	timestamp with time zone
state_change	timestamp with time zone
wait_event_type	text
wait_event	text
state	text

Colonne	Type
backend_xid	xid
backend_xmin	xid
query	text
backend_type	text

Voici la description des différents champs :

- `datid` : l'*OID* de la base à laquelle la session est connectée ;
- `datname` : le nom de la base associée à cet OID ;
- `pid` : le numéro du processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client (cette colonne avait pour nom `procpid` avant la version 9.2) ;
- `usesysid` : l'*OID* de l'utilisateur connecté ;
- `usename` : le nom de l'utilisateur associé à cet OID ;
- `application_name` : un nom facultatif renseigné par l'application cliente ;
- `client_addr` : l'adresse IP du client connecté (ou NULL si connexion sur socket Unix) ;
- `client_hostname` : le nom associé à cette IP, renseigné uniquement si `log_hostname` est à `on` (attention, ce paramètre peut fortement ralentir la connexion à cause de la résolution DNS nécessaire) ;
- `client_port` : le numéro de port à partir duquel le client est connecté, toujours s'il s'agit d'une connexion IP ;
- `backend_start` : le timestamp de l'établissement de la session ;
- `xact_start` : le timestamp de début de la dernière transaction ;
- `query_start` : le timestamp de début de la dernière requête ;
- `state_change` : le timestamp du dernier changement d'état (utile surtout dans le cas d'une session en state `idle`, pour connaître l'heure de la dernière requête exécutée, et l'heure de fin de cette dernière requête) ;
- `wait_event_type` et `wait_event` : le processus est en attente d'un verrou, ces deux colonnes permettent de savoir quel type de verrou et sur quel objet (avant la version 9.6, il n'y avait qu'une colonne, nommée `waiting` de type booléen);
- `state` : l'état du processus ;
- `backend_xid` : l'identifiant de transaction pour ce processus ;
- `backend_xmin` : l'horizon xmin du processus ;
- `query` contient la requête en cours si `state` est `active`, et la dernière requête effectuée pour les autres valeurs de `state` ;
- `backend_type` indique le type de processus (cette colonne apparaît en version 10).

Cette vue n'est renseignée que si `track_activities` est à `on` (valeur par défaut). Certains champs ne sont renseignés que pour les superutilisateurs.

6.2.2 PG_STAT_SSL

Quand le SSL est activé sur le serveur, cette vue indique pour chaque connexion cliente les informations suivantes :

- SSL activé ou non
- Version SSL
- Suite de chiffrement
- Nombre de bits pour algorithme de chiffrement
- Compression activée ou non
- Distinguished Name (DN) du certificat client

La définition de la vue est celle-ci :

Colonne	Type
pid	integer
ssl	boolean
version	text
cipher	text
bits	integer
compression	boolean
clientdn	text

- `pid` : le numéro du processus du *backend*, c'est à dire du processus PostgreSQL chargé de discuter avec le client ;
- `ssl` : ssl activé ou non ;
- `version` : version ssl utilisé, *null* si ssl n'est pas utilisé ;
- `cipher` : suite de chiffrement utilisée, *null* si ssl n'est pas utilisé ;
- `bits` : nombre de bits de la suite de chiffrement, *null* si ssl n'est pas utilisé ;
- `compression` : compression activée ou non, *null* si ssl n'est pas utilisé ;
- `clientdn` : champ *Distinguished Name (DN)* du certificat client, *null* si aucun certificat client n'est utilisé ou si ssl n'est pas utilisé ;

6.2.3 PG_STAT_DATABASE

pg_stat_database :

Des informations globales à chaque base :

- nombre de sessions
- transactions validées/annulées
- accès blocs
- accès enregistrements
- taille et nombre de fichiers temporaires
- temps d'entrées/sorties

La définition de la vue est celle-ci :

Colonne	Type
datid	oid
datname	name
numbackends	integer
xact_commit	bigint
xact_rollback	bigint
blks_read	bigint
blks_hit	bigint
tup_returned	bigint
tup_fetched	bigint
tup_inserted	bigint
tup_updated	bigint
tup_deleted	bigint
conflicts	bigint
temp_files	bigint
temp_bytes	bigint
deadlocks	bigint
blk_read_time	double precision
blk_write_time	double precision
stats_reset	timestamp with time zone

- `datid`/`datname` : l'*OID* et le nom de la base de données ;
- `numbackends` : le nombre de sessions en cours ;
- `xact_commit` : le nombre de transactions ayant terminé avec commit sur cette base ;
- `xact_rollback` : le nombre de transactions ayant terminé avec rollback sur cette

base ;

- `blks_read` : le nombre de blocs demandés au système d'exploitation ;
- `blks_hit` : le nombre de blocs trouvés dans la cache de PostgreSQL ;
- `tup_returned` : le nombre d'enregistrements réellement retournés par les accès aux tables ;
- `tup_fetched` : le nombre d'enregistrements interrogés par les accès aux tables (ces deux compteurs seront explicités dans la vue sur les index) ;
- `tup_inserted` : le nombre d'enregistrements insérés en base ;
- `tup_updated` : le nombre d'enregistrements mis à jour en base ;
- `tup_deleted` : le nombre d'enregistrements supprimés en base ;
- `conflicts` : le nombre de conflits de réplication (sur un esclave) ;
- `temp_files` : le nombre de fichiers temporaires (utilisés pour le tri) créés par cette base depuis son démarrage ;
- `temp_bytes` : le nombre d'octets correspondant à ces fichiers temporaires. Cela permet de trouver les bases effectuant beaucoup de tris sur disque ;
- `deadlocks` : le nombre de deadlocks (interblocages) ;
- `blk_read_time` et `blk_write_time` : le temps passé à faire des lectures et des écritures vers le disque (il faut que `track_io_timing` soit à `on`, ce qui n'est pas la valeur par défaut) ;
- `stats_reset` : la date de dernière remise à zéro des compteurs de cette vue.

6.2.4 PG_STAT_USER_TABLES

pg_stat_user_tables :

- statistiques niveau « ligne »
- insertions/mise à jour/suppression
- type et nombre d'accès
- opérations de maintenance
- détection des tables mal indexées ou très accédées

Voici la définition de cette vue :

Colonne	Type
relid	oid
schemaname	name
relname	name
seq_scan	bigint
seq_tup_read	bigint

Colonne	Type
idx_scan	bigint
idx_tup_fetch	bigint
n_tup_ins	bigint
n_tup_upd	bigint
n_tup_del	bigint
n_tup_hot_upd	bigint
n_live_tup	bigint
n_dead_tup	bigint
last_vacuum	timestamp with time zone
last_autovacuum	timestamp with time zone
last_analyze	timestamp with time zone
last_autoanalyze	timestamp with time zone
vacuum_count	bigint
autovacuum_count	bigint
analyze_count	bigint
autoanalyze_count	bigint

Contrairement aux vues précédentes, cette vue est locale à chaque base.

- `relid`, `relname` : *OID* et nom de la table concernée ;
- `schemaname` : le schéma contenant cette table ;
- `seq_scan` : nombre de parcours séquentiels sur cette table ;
- `seq_tup_read` : nombre d'enregistrements accédés par ces parcours séquentiels ;
- `idx_scan` : nombre de parcours d'index sur cette table ;
- `idx_tup_fetch` : nombre d'enregistrements accédés par ces parcours séquentiels ;
- `n_tup_ins`, `n_tup_upd`, `n_tup_del` : nombre d'enregistrements insérés, mis à jour, supprimés ;
- `n_tup_hot_upd` : nombre d'enregistrements mis à jour par mécanisme HOT (c'est à dire sur place) ;
- `n_live_tup` : nombre d'enregistrements « vivants » ;
- `n_dead_tup` : nombre d'enregistrements « morts » (supprimés mais non nettoyés) ;
- `last_vacuum` : timestamp de dernier *VACUUM* ;
- `last_autovacuum` : timestamp de dernier *VACUUM* automatique ;
- `last_analyze` : timestamp de dernier *ANALYZE* ;
- `last_autoanalyze` : timestamp de dernier *ANALYZE* automatique ;
- `vacuum_count` : nombre de *VACUUM* manuels ;
- `autovacuum_count` : nombre de *VACUUM* automatiques ;
- `analyze_count` : nombre d'*ANALYZE* manuels ;

- `autoanalyze_count` : nombre d'*ANALYZE* automatiques.

6.2.5 PG_STAT_USER_INDEXES

pg_stat_user_indexes :

- vue par index
- nombre d'accès et efficacite

Colonne	Type
relid	oid
indexrelid	oid
schemaname	name
relname	name
indexrelname	name
idx_scan	bigint
idx_tup_read	bigint
idx_tup_fetch	bigint

- `relid`, `relname` : *OID* et nom de la table qui possède l'index
- `indexrelid`, `indexrelname` : *OID* et nom de l'index en question
- `schemaname` : schéma contenant l'index
- `idx_scan` : nombre de parcours de cet index
- `idx_tup_read` : nombre d'enregistrements retournés par cet index
- `idx_tup_fetch` : nombre d'enregistrements accédés sur la table associée à cet index

`idx_tup_read` et `idx_tup_fetch` retournent des valeurs différentes pour plusieurs raisons :

- Un scan d'index peut très bien accéder à des enregistrements morts. Dans ce cas `idx_tup_read` > `idx_tup_fetch`.
- Un scan d'index peut très bien ne pas entraîner d'accès direct à la table :
  - Si c'est un *Index Only Scan* : dans ce cas on n'accède pas du tout à la table, puisque toutes les colonnes accédées sont dans l'index.
  - Si c'est un *Bitmap Index Scan* : dans ce cas, on va éventuellement accéder à plusieurs index, faire une fusion (*Or* ou *And*) et ensuite seulement accéder aux enregistrements (moins nombreux si c'est un *And*).

Dans tous les cas, ce qu'on surveille le plus souvent dans cette vue, c'est tout d'abord les

index ayant `idx_scan` à 0. Ils sont le signe d'un index qui ne sert probablement à rien. La seule exception éventuelle étant un index associé à une contrainte d'unicité, les parcours de l'index réalisés pour vérifier l'unicité n'étant pas comptabilisés dans cette vue.

Les autres indicateurs intéressants sont un nombre de `tup_read` très grand par rapport aux scans d'index, qui peuvent suggérer un index trop peu sélectif, et une grosse différence entre le read et le fetch. Ces indicateurs ne permettent par contre pas de conclure quoi que ce soit par eux même, ils peuvent seulement donner des pistes d'amélioration.

6.2.6 PG STATIO USER TABLES INDEXES

pg_statio_user_tables, pg_statio_user_indexes :

- opérations au niveau bloc
- demandés à l'OS ou au cache
- calculer des hit ratios

pg_statio_user_tables :

Colonne	Type
relid	oid
schemaname	name
relname	name
heap_blks_read	bigint
heap_blks_hit	bigint
idx_blks_read	bigint
idx_blks_hit	bigint
toast_blks_read	bigint
toast_blks_hit	bigint
tidx_blks_read	bigint
tidx_blks_hit	bigint

- `relid`,`relname` : *OID* et nom de la table ;
- `schemaname` : nom du schéma contenant la table ;
- `heap_blks_read` : nombre de blocs accédés de la table demandés au système d'exploitation. `Heap` signifie *tas*, et ici *données non triées*, par opposition aux index ;
- `heap_blks_hit` : nombre de blocs accédés de la table trouvés dans le cache de PostgreSQL ;
- `idx_blks_read` : nombre de blocs accédés de l'index demandés au système

d'exploitation ;

- `idx_blks_hit` : nombre de blocs accédés de l'index trouvés dans le cache de PostgreSQL ;
- `toast_blks_read`, `toast_blks_hit`, `tidx_blks_read`, `tidx_blks_hit` : idem que précédemment, mais pour TOAST et ses index (voir plus loin la présentation de TOAST).

pg_statio_user_indexes :

Colonne	Type
relid	oid
indexrelid	oid
schemaname	name
relname	name
indexrelname	name
idx_blks_read	bigint
idx_blks_hit	bigint

- `relid`, `relname` : *OID* et nom de la table associée à l'index ;
- `indexrelid`, `indexrelname` : *OID* et nom de l'index ;
- `schemaname` : nom du schéma dans lequel se trouve l'index ;
- `idx_blks_read` : nombre de blocs accédés de l'index demandés au système d'exploitation ;
- nombre de blocs accédés de l'index trouvés dans le cache de PostgreSQL.

Pour calculer un *hit ratio*, qui est un indicateur fréquemment utilisé, on utilise la formule suivante (par exemple pour les index) :

```
SELECT schemaname,
       indexrelname,
       relname,
       idx_blks_hit::float/CASE idx_blks_read+idx_blks_hit
         WHEN 0 THEN 1 ELSE idx_blks_read+idx_blks_hit END
FROM pg_statio_user_indexes;
```

Il y a deux « ruses » à signaler :

- `idx_blks_hit::float` convertit le numérateur en float, ce qui entraîne que la division est à virgule flottante. Sinon on divise des entiers par des entiers, et on obtient donc un résultat entier, 0 la plupart du temps (division euclidienne entière)
- `CASE idx_blks_read+idx_blks_hit`
  `WHEN 0 THEN 1`

`ELSE idx_blks_read+idx_blks_hit END` évite la division par zéro, en divisant par 1 quand les deux compteurs sont à 0.

6.2.7 PG_LOCKS

- pg_locks :
  - visualisation des verrous en place
  - tous types de verrous sur objets
- Complexe à interpréter :
  - verrous sur enregistrements pas directement visibles
  - Article base de connaissance Dalibo[24]

`pg_locks` :

Colonne	Type
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

C'est une vue globale à l'instance.

- `locktype` : 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) ;

[24] https://kb.dalibo.com/verrouillage

- `database` : la base dans laquelle ce verrou est pris ;
- `relation` : si locktype vaut `relation` (ou `page` ou `tuple`), l'*OID* de la relation cible ;
- `page` : le numéro de la page dans une relation (quand verrou de type `page` ou `tuple`) cible ;
- `tuple` : le numéro de l'enregistrement, (quand verrou de type `tuple`) cible;
- `virtualxid` : le numéro de la transaction virtuelle (quand verrou de type `virtualxid`) cible ;
- `transactionid` : le numéro de la transaction cible ;
- `classid` : 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` : l'*OID* de l'objet dans le catalogue système pointé par classid;
- `objsubid` : l'ID de la colonne de l'objet objid concerné par le verrou ;
- `virtualtransaction` : le numéro de transaction virtuelle possédant le verrou (ou tentant de l'acquérir si `granted` vaut `f`) ;
- `pid` : le pid de la session possédant le verrou ;
- `mode` : le niveau de verrouillage demandé ;
- `granted` : acquis ou non (donc en attente) ;
- `fastpath` : information utilisée pour le débuggage surtout. Fastpath est le mécanisme d'acquisition des verrous les plus faibles.

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 modifié 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 verrouillage manuels de la table en entier (`SHARE ROW EXCLUSIVE`).

6.2.8 PG_STAT_BGWRITER

pg_stat_bgwriter

- activité des checkpoint
- visualiser le volume d'allocations et d'écritures

Colonne	Type
checkpoints_timed	bigint
checkpoints_req	bigint
checkpoint_write_time	double precision
checkpoint_sync_time	double precision
buffers_checkpoint	bigint
buffers_clean	bigint
maxwritten_clean	bigint
buffers_backend	bigint
buffers_backend_fsync	bigint
buffers_alloc	bigint
stats_reset	timestamp with time zone

Cette vue ne comporte qu'une seule ligne.

- `checkpoints_timed` : nombre de checkpoints déclenchés quand le délai précisé par `checkpoint_timeout` est atteint ;
- `checkpoints_req` : nombre de checkpoints déclenchés quand la volumétrie précisée par `max_wal_size` (ou `checkpoint_segments` jusqu'en 9.5) est atteinte ;
- `checkpoint_write_time` : temps passé par `checkpointer` à écrire des données ;
- `checkpoint_sync_time` : temps passé à s'assurer que les écritures ont été synchronisées sur disque lors des checkpoints ;
- `buffers_checkpoint` : nombre de blocs écrits par `checkpointer` ;
- `buffers_clean` : nombre de blocs écrits par `writer` ;
- `maxwritten_clean` : nombre de fois où `writer` s'est arrêté pour avoir atteint `bgwriter_lru_maxpages` ;
- `buffers_backend` : nombre de blocs écrits par les backends avant de pouvoir allouer de la mémoire (car pas de bloc disponible) ;
- `buffers_backend_fsync` : nombre de blocs synchronisés par les backends parce que la liste des blocs à synchroniser est pleine ;
- `buffers_alloc` : nombre de blocs alloués dans le `shared_buffers` ;
- `stats_reset` : date de remise à zéro de cette vue statistique.

Les deux premières colonnes permettent de vérifier que la configuration de `max_wal_size` (ou `checkpoint_segments`) n'est pas trop basse par rapport au volume d'écriture que subit la base. Les colonnes `buffers_clean` et `maxwritten_clean`, comparées à `buffers_checkpoint` et `buffers_backend`, permettent de vérifier que la configuration du bgwriter est adéquate : si `maxwritten_clean` augmente fortement en fonctionnement normal, c'est que le paramètre `bgwriter_lru_maxpages` l'empêche de libérer

autant de buffers qu'il l'estime nécessaire (ce paramètre sert de garde-fou). Dans ce cas, `buffers_backend` va augmenter.

Il faut toutefois prendre ce compteur avec prudence : une session qui modifie énormément de blocs n'aura pas le droit de modifier tout le contenu des Shared Buffers, elle sera cantonnée à une toute petite partie. Elle sera donc obligée de vider elle-même ses buffers. C'est le cas par exemple d'une session chargeant un volume conséquent de données avec `COPY`.

Cette vue statistique peut être mise à zéro par l'appel à la fonction : `pg_stat_reset_shared('bgwriter')`.

6.2.9 PG_STAT_ARCHIVER

pg_stat_archiver (9.4+) :

- bon fonctionnement de l'archivage
- quand et combien d'erreurs d'archivages se sont produites

Colonne	Type
archived_count	bigint
last_archived_wal	text
last_archived_time	timestamp with time zone
failed_count	bigint
last_failed_wal	text
last_failed_time	timestamp with time zone
stats_reset	timestamp with time zone

Cette vue ne comporte qu'une seule ligne.

- `archived_count` : nombre de WAL archivés ;
- `last_archived_wal` : nom du dernier fichier WAL dont l'archivage a réussi ;
- `last_archived_time` : date du dernier archivage réussi ;
- `failed_count` : nombre de tentatives d'archivages échouées ;
- `last_failed_wal` : nom du dernier fichier WAL qui a rencontré des problèmes d'archivage ;
- `last_failed_time` : date de la dernière tentative d'archivage échouée ;
- `stats_reset` : date de remise à zéro de cette vue statistique.

Cette vue peut être spécifiquement remise à zéro par l'appel à la fonction

`pg_stat_reset_shared('archiver')`.

On peut facilement s'en servir pour déterminer si l'archivage fonctionne bien :

```
SELECT case WHEN (last_archived_time > last_failed_time)
    THEN 'OK' ELSE 'KO' END FROM pg_stat_archiver ;
```

6.2.10 PG_STAT_REPLICATION ET PG_STAT_DATABASE_CONFLICTS

pg_stat_replication :

- État des esclaves connectés au maître en SR
- Mesure du lag

pg_stat_database_conflicts :

- nombre de conflits de réplication
- par type

pg_stat_replication :

Colonne	Type
pid	integer
usesysid	oid
usename	name
application_name	text
client_addr	inet
client_hostname	text
client_port	integer
backend_start	timestamp with time zone
state	text
sent_lsn	text
write_lsn	text
flush_lsn	text
replay_lsn	text
sync_priority	integer
sync_state	text

- `pid` : numéro de processus du backend discutant avec l'esclave ;
- `usesysid`, `usename` : *OID* et nom de l'utilisateur utilisé pour se connecter en streaming replication ;

- `application_name` : *application_name* de la chaîne de connexion de l'esclave. Peut être paramétré dans `connection_string` (`recovery.conf`) de l'esclave, surtout utilisé dans le cas de la réplication synchrone ;
- `client_addr` : adresse IP de l'esclave (s'il n'est pas sur la même machine, ce qui est vraisemblable) ;
- `client_hostname` : nom d'hôte de l'esclave (si `log_hostname` à `on`) ;
- `client_port` : numéro de port TCP auquel est connecté l'esclave ;
- `backend_start` : timestamp de connexion de l'esclave ;
- `state` : `startup` (en cours d'initialisation), `backup` (utilisé par `pg_basebackup`), `catchup` (étape avant streaming, rattrape son retard), `streaming` (on est dans le mode streaming, les nouvelles entrées de journalisation sont envoyées au fil de l'eau) ;
- `sent_lsn` : l'adresse jusqu'à laquelle on a envoyé le contenu du WAL à cet esclave ;
- `write_lsn` l'adresse jusqu'à laquelle cet esclave a écrit le WAL sur disque ;
- `flush_lsn` : l'adresse jusqu'à laquelle cet esclave a synchronisé le WAL sur disque (l'écriture est alors garantie) ;
- `replay_lsn` : l'adresse jusqu'à laquelle l'esclave a rejoué les informations du WAL (les données sont donc visibles jusqu'à ce point, par requêtes, sur l'esclave) ;
- `sync_priority` : dans le cas d'une réplication synchrone, la priorité de ce serveur (un seul est synchrone, si celui-ci tombe, un autre est promu). Les 3 valeurs 0 (asynchrone), 1 (synchrone) et 2 (candidat) sont traduites dans `sync_state`.

pg_stat_database_conflicts :

Colonne	Type
datid	oid
datname	name
confl_tablespace	bigint
confl_lock	bigint
confl_snapshot	bigint
confl_bufferpin	bigint
confl_deadlock	bigint

- `datid`, `datname` : l'*OID* et le nom de la base ;
- `confl_tablespace` : requêtes annulées pour rejouer un `DROP TABLESPACE` ;
- `confl_lock` : requêtes annulées à cause de `lock_timeout` ;
- `confl_snapshot` : requêtes annulées à cause d'un snapshot (instantané) trop vieux. C'est dû à des données supprimées sur le maître par un VACUUM, rejoué sur l'esclave, et qui a supprimé des données encore nécessaires pour des

requêtes sur l'esclave. On peut faire disparaître totalement ce cas en activant `hot_standby_feedback` ;

- `confl_bufferpin` : requêtes annulées à cause d'un `buffer pin`, c'est à dire d'un bloc de cache mémoire en cours d'utilisation dont avait besoin la réplication. Ce cas est extrêmement rare : il faudrait un buffer pin d'une durée comparable à `max_standby_archive_delay` ou `max_standby_streaming_delay`. Or ceux-ci sont par défaut à 30 s, alors qu'un buffer pin dure quelques microsecondes ;
- `confl_deadlock` : requêtes annulées à cause d'un deadlock entre une session et le rejeu des transactions (toujours au niveau des buffers). Hautement improbable aussi.

6.3 INDEX AVANCÉS

De nombreuses fonctionnalités d'indexation sont disponibles dans PostgreSQL :

- Index multi-colonnes
- Index fonctionnels
- Index partiels
- *Covering indexes*
- Classes d'opérateurs
- GiN
- GIST
- BRIN
- Hash

PostgreSQL fournit de nombreux types d'index, afin de répondre à des problématiques de recherches complexes. L'index classique, créé comme ceci :

```
CREATE INDEX mon_index ON TABLE t1(a) ;
```

est l'index le plus fréquemment utilisé. C'est un index « BTree », c'est-à-dire un index stocké sous forme d'arbre balancé. Cette structure a de nombreux avantages :

- Performances se dégradant peu avec la taille de l'arbre : les temps de recherche sont en O(log(n)), c'est à dire qu'ils varient en fonction du logarithme du nombre d'enregistrements contenus dans l'index. Plus le nombre d'enregistrements est élevé, plus la variation est faible
- Ils permettent une excellente concurrence d'accès : on peut facilement avoir plusieurs processus en train d'insérer simultanément dans un index BTree, avec très peu de contention entre ces processus

Toutefois ils ne permettent de répondre qu'à des questions très simples : il faut qu'elles ne portent que sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre le gros des cas, mais connaître les autres possibilités du moteur vous permettra d'accélérer des requêtes plus complexes, ou d'indexer des types de données inhabituels.

6.3.1 INDEX MULTI-COLONNES

Un index peut référencer plus d'une colonne :

- CREATE INDEX idx ON ma_table (col1,col2,col3)
- Index trié sur le n-uplet (col1,col2,col3)
- Accès direct à n'importe quelle valeur de
  - (col1,col2,col3)
  - (col1,col2)
  - (col1)

Un index multi-colonnes est trié sur le n-uplet formant ses colonnes. Autrement formulé, on trie sur col1, puis sur col2 en cas d'égalité sur col1, puis col3 en cas d'égalité sur col2, dans l'exemple.

Comme les informations sont stockées dans un arbre trié, l'index peut répondre à ces clauses WHERE de façon efficace :

- `WHERE col1='val1' and col2='val2' and col3='val3'`
- `WHERE col1='val1' and col2='val2'`
- `WHERE col1='val1'`

Mais il permet aussi de répondre à des range scans (comme un index Btree mono-colonnes), mais uniquement sur la dernière colonne sur laquelle on filtre :

- `WHERE col1='val1' and col2='val2' and col3 > 'val3'` ... ou `col3 < 'val3'` ou `col3 > 'val3' and col3 < 'val4'`. Les opérateurs `<=` ou `>=` fonctionnent aussi bien sûr.
- `WHERE col1='val1' and col2>'val2'`
- `WHERE col1>'val1'`

6.3.2 INDEX FONCTIONNELS

Il s'agit d'un index sur le résultat d'une fonction :

```
WHERE upper(a)='DUPOND'
```

- l'index classique ne fonctionne pas

```
CREATE INDEX mon_idx ON ma_table ((UPPER(a))
```

- La fonction doit être IMMUTABLE

À partir du moment où une clause WHERE applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement.

C'est comme demander à un dictionnaire Anglais vers Français : « Quels sont les mots dont la traduction en français est 'fenêtre' ? ». Le tri du dictionnaire ne correspond pas à la question posée. Il nous faudrait un index non plus sur les mots anglais, mais sur leur traduction en français.

C'est exactement ce que font les index fonctionnels : ils indexent le résultat d'une fonction appliquée à l'enregistrement.

L'exemple classique est l'indexation insensible à la casse : on crée un index sur `UPPER` (ou `LOWER`) de la chaîne à indexer, et on recherche les mots convertis à la casse souhaitée.

La fonction d'indexation utilisée doit être IMMUTABLE : sa valeur de retour ne doit dépendre que d'une seule chose : ses paramètres en entrée. Si elle dépend du contenu de la base (c'est-à-dire qu'elle exécute des requêtes d'interrogation), ou qu'elle dépend d'une variable de session, elle n'est pas utilisable pour l'index : l'endroit dans lequel la donnée devrait être insérée dans l'index dépendrait de ces paramètres, et serait donc potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Si une fonction non IMMUTABLE est utilisée pour créer un index fonctionnel, PostgreSQL refuse, avec l'erreur

```
ERROR:  functions in index expression must be marked IMMUTABLE
```

6.3.3 INDEX PARTIEL

- Un index partiel n'indexe qu'une partie des données d'une table, en précisant une clause WHERE à la création de l'index :

```
CREATE INDEX idx_partiel ON trapsnmp (date_reception)
WHERE est_acquitte=false;
```

- Beaucoup plus petit que l'index complet.
- Souvent dédié à une requête précise :

```
SELECT * FROM trapsnmp WHERE est_acquitte=false
ORDER BY date_reception
```

- La clause WHERE ne porte pas forcément sur la colonne indexée, c'est même souvent plus intéressant de la faire porter sur une autre colonne.

Si cette requête est exécutée très fréquemment, il est intéressant d'avoir cet index. Par ailleurs, l'index composé (est_acquitte,date_reception) serait bien moins rentable, puisque la plupart des événements SNMP de notre table auront été acquittés.

6.3.4 COVERING INDEXES

Les *Covering Indexes* (on trouve parfois « index couvrants » dans la littérature française) :

- Répondent à la clause WHERE
- ET contiennent toutes les colonnes demandées par la requête
- `SELECT col1,col2 FROM t1 WHERE col1>12`
- `CREATE INDEX idx1 on T1 (col1,col2)`
- Pas de visite de la table (donc peu d'accès aléatoires, l'index étant à peu près trié physiquement)

Pour pouvoir bénéficier des *Covering Indexes*, il faut que toutes les colonnes retournées par la requête soient présentes dans l'index.

Un parcours d'index classique (*Index Scan*) est en fait un aller/retour entre l'index et la table : on va chercher un enregistrement dans l'index, qui nous donne son adresse dans la table, on accède à cet enregistrement dans la table, puis on passe à l'entrée d'index suivante. Le coût en entrées-sorties peut être énorme : les données de la table sont habituellement éparpillées dans tous les blocs.

Le parcours d'index permis par les *Covering Indexes* (*Index Only Scan*) n'a plus besoin de cette interrogation de la table. Les enregistrements cherchés étant contigus dans l'index (puisqu'il est trié), le nombre d'accès disque est bien plus faible, ce qui peut apporter des gains de performances énormes en sélection. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

Les *Covering Indexes* ne sont présents qu'à partir de la version 9.2.

6.3.5 CLASSES D'OPÉRATEURS

Un index utilise des opérateurs de comparaison :

- Il peut exister plusieurs façons de comparer deux données du même type
- Par exemple, pour les chaînes de caractères
  - Différentes collations
  - Tri sans collation (pour LIKE)
- CREATE INDEX idx1 ON ma_table (col_varchar varchar_pattern_ops)
- Permet SELECT ... FROM ma_table WHERE col_varchar LIKE 'chaine%'

Il est tout à fait possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison pour l'indexations, dans des cas particuliers.

Le cas d'utilisation le plus fréquent d'utilisation dans PostgreSQL est la comparaison de chaine `LIKE 'chaine%'`. L'indexation texte « classique » utilise la collation par défaut de la base (ou la collation de la colonne de la table, dans les versions récentes de PostgreSQL). Cette collation peut être complexe (par exemple, est-ce que le ß allemand est équivalent à ss, quelles sont les règles majuscules/minuscules, etc). Cette collation n'est pas compatible avec l'opérateur `LIKE`. Si on reprend l'exemple précédent, est-ce que `LIKE 'stras'` doit retourner « straße » ?

Les règles sont différentes pour chaque collation, et il serait donc très complexe de réécrire le `LIKE` en un `BETWEEN`, comme il le fait habituellement pour tous les SGBD : `col_texte LIKE 'toto%'` peut être réécrit comme `coltexte> 'toto' and coltexte< 'totop'` en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur unicode par exemple. Cela permet d'utiliser un index sur coltexte, mais uniquement si celui-ci est aussi trié en ASCII.

C'est à cela que sert la classe d'opérateurs `varchar_pattern_ops` : l'index est construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient. Il devient alors trivial pour l'optimiseur de faire la réécriture.

Il existe quelques autres cas d'utilisation d'*opclass* alternatives, mais elles sont habituellement liées à l'utilisation d'un module comme `pg_trgm`, qui fournit des types de données complexes, et sont donc clairement documentées avec ce module.

6.3.6 TOUT ENSEMBLE !

Toutes les fonctionnalités que nous venons de voir peuvent bien sûr être utilisées simultanément :

```
CREATE INDEX idx_adv ON ma_table
(f(col1), col2 varchar_pattern_ops) WHERE col3<12 ;
```

```
SELECT col2 FROM ma_table
WHERE col3<12 and f(col1)=7 and col2 LIKE 'toto%' ;
```

Toutes les fonctionnalités vues précédemment peuvent être utilisées simultanément. Il est parfois tentant de créer des index très spécialisés grâce à toutes ces fonctionnalités. Il ne faut surtout pas perdre de vue qu'un index est une structure lourde à mettre à jour, comparativement à une table. Une table avec un seul index est environ 3 fois plus lente qu'une table nue, et chaque index rajoute le même surcoût. Il est donc souvent plus judicieux d'avoir des index pouvant répondre à plusieurs requêtes différentes, et de ne pas trop les spécialiser. Il faut trouver un juste compromis entre le gain à la lecture et le surcoût à la mise à jour.

6.3.7 GIN

- Generalized Inverted iNdex
- Index inversé ?
  - Index associe une valeur à la liste de ses adresses
  - Utile pour tableaux, listes…
- Pour chaque entrée du tableau
  - Liste d'adresses (TID) où le trouver
- Option fastupdate (8.4+)
  - à désactiver pour avoir un temps de réponse stable
- Liste compressée (9.4+)
  - alternative à bitmap

Un index inversé est une structure classique, utilisée le plus souvent dans l'indexation *Full Text*. Le principe est de décomposer un document en sous-structures, qui seront indexées. Par exemple, un document sera décomposé en la liste de ses mots, et chaque mot sera une clé de l'index. Cette clé fournira la liste des documents contenant ce mot. Pour plus de détail sur la structure elle-même, cet article Wikipedia[25] est une lecture conseillée.

Les index GIN de PostgreSQL sont « généralisés » car ils sont capables d'indexer n'importe quel type de données, à partir du moment où on lui fournit les différentes fonctions d'API permettant le découpage et le stockage des différents *items* composant la donnée à indexer.

Nativement, GIN supporte dans PostgreSQL les tableaux de type natif (int, float, text, date…), les tsvector utilisés pour l'indexation *Full Text*, les jsonb (depuis PostgreSQL 9.4),

[25] https://fr.wikipedia.org/wiki/Index_inversé

ainsi que tous les types scalaires ayant un index btree, à partir du moment où on installe l'extension `btree_gin`.

L'extension `pg_trgm` utilise aussi les index GIN, pour permettre des recherches de type `SELECT * FROM ma_table WHERE ma_col_texte LIKE '%ma_chaine1%ma_chaine2%'` qui utilisent un index.

Leur structure en fait des structures lentes à la mise à jour. Par contre, elles sont extrêmement efficaces pour les interrogations multicritères, ce qui les rend très appropriées pour l'indexation *Full Text*, le jsonb…

L'option `fastupdate` permet une mise à jour bien plus rapide. Elle est activée par défaut. L'inconvénient est que le temps de réponse de l'index devient très instable : certaines recherches peuvent être très rapides et d'autres très lentes. Le seul moyen d'accélérer ces recherches revient à désactiver cette option. Cela permet en plus de diminuer drastiquement les écritures dans les journaux de transactions en cas de mises à jour massives. Mais cela a un gros impact : les mises à jour de l'index sont bien plus lentes.

Un autre cas d'utilisation, depuis PostgreSQL 9.4, est le cas d'utilisation traditionnel des index bitmap. Les index bitmap sont très compacts, mais ne permettent d'indexer que peu de valeurs différentes. Un index bitmap est une structure utilisant 1 bit par enregistrement pour chaque valeur indexable. Par exemple, on peut définir un index bitmap sur le sexe : deux (ou trois si on autorise null ou indéfini) valeurs seulement sont possibles. Indexer un enregistrement nécessitera donc un ou deux bits. Le défaut de ces index est qu'ils sont très peu performants si on rajoute des nouvelles valeurs, et se dégradent avec l'ajout de nouvelles valeurs : leur taille par enregistrement devient bien plus grosse, et l'index nécessite une réécriture complète à chaque ajout de nouvelle valeur.

Les index GIN permettent un fonctionnement sensiblement équivalent au bitmap : chaque valeur indexable contient la liste des enregistrements répondant au critère. Cette liste est compressée depuis la version 9.4. Voici un exemple :

```
CREATE TABLE demo_gin (id bigint, nom text, sexe varchar(1));
INSERT INTO demo_gin SELECT i, 'aaaaaaaa',
    CASE WHEN random()*2 <1 THEN 'H' ELSE 'F' END
    FROM generate_series(1,10000000) g(i);
CREATE INDEX idx_nom ON demo_gin(nom);
CREATE EXTENSION btree_gin;
CREATE INDEX idx_sexe ON demo_gin USING gin(sexe);
```

Voici les tailles respectives :

```
=# SELECT pg_size_pretty(pg_table_size('demo_gin'));
 pg_size_pretty
----------------
 498 MB
```

```
(1 ligne)

=# SELECT pg_size_pretty(pg_table_size('idx_nom'));
 pg_size_pretty
----------------
 301 MB
(1 ligne)

=# SELECT pg_size_pretty(pg_table_size('idx_sexe'));
 pg_size_pretty
----------------
 10 MB
(1 ligne)
```

Un index btree sur la colonne sexe aurait occupé 214 Mo. L'index est donc environ 20 fois plus compact, dans cet exemple simple.

On peut aussi utiliser un index GIN pour indexer le contenu d'une liste texte, si par exemple on a une table ne respectant pas la première forme normale. Imaginons le champ « attributs », contenant une liste d'attributs séparés par une virgule :

```
CREATE INDEX idx_attributs_array ON ma_table
USING gin (regexp_split_to_array(attributs,','));
```

La fonction `regexp_split_to_array` découpe la chaîne attribut sur le séparateur `,`, et retourne un tableau, qui peut donc être indexé avec GIN.

On peut ensuite écrire des requêtes sous la forme :

```
SELECT * FROM ma_table WHERE regexp_split_to_array(attributs,',') @>
                             '{"toit ouvrant","vitres teintées"}';
```

6.3.8 GIST

GiST : Generalized Search Tree

- Arbre de recherche généralisé
- Indexation non plus des valeurs mais de la véracité de prédicats
- Moins performants car moins sélectifs que Btree
- Mais peuvent indexer à peu près n'importe quoi
- Multi-colonnes dans n'importe quel ordre
- Sur-ensemble de Btree et Rtree

Initialement, les index GiST sont un produit de la recherche de l'université de Berkeley. L'idée fondamentale est de pouvoir indexer non plus les valeurs dans l'arbre Btree, mais

plutôt la véracité d'un prédicat : « ce prédicat est vrai sur telle sous-branche ». On dispose donc d'une API permettant au type de données d'informer le moteur GiST d'informations comme : « quel est le résultat de la fusion de tel et tel prédicat » (pour pouvoir déterminer le prédicat du nœud parent), quel est le surcout d'ajout de tel prédicat dans telle ou telle partie de l'arbre, comment réaliser un *split* (découpage) d'une page d'index, déterminer la distance entre deux prédicats, etc…

Tout ceci est très virtuel, et rarement re-développé par les utilisateurs. Ce qui est important, c'est :

- Que ces index sont moins performants que Btree ;
- Mais qu'ils permettent d'indexer des choses bien plus complexes : on peut indexer n'importe quoi avec GiST, quelle que soit la dimension, le type, tant qu'on peut utiliser des prédicats sur ce type ;
- Il est disponible pour les types natifs suivants :
  - Les types géométriques (box, circle, point, poly) ;
  - Les types range (d'int, de timestamp…) ;
  - Le *Full Text* (plus rapide à maintenir qu'un index GIN, mais moins performant à l'interrogation) ;
  - Les adresses IP/CIDR
- Il est en outre utilisé par :
  - Le projet PostGIS, pour répondre à des questions complexes telles que « quels 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 » ;
  - `pg_trgm` (moins efficace que GIN pour la recherche exacte, mais permet de rapidement trouver les N enregistrements les plus proches d'une chaîne donnée, sans tri, et est plus compact) ;
- Que les index GiST ont moins tendance à se fragmenter que les index GIN, même si c'est difficilement quantifiable, car cela dépend énormément du type de mises à jour ;
- Que les index GiST sont moins lourds à maintenir que les index GIN.

Il est utilisé pour les *Constraint Exclusions* (exclusions de contraintes), par exemple pour interdire que la même salle soit réservée simultanément sur deux intervalles en intersection.

Il est aussi intéressant si on a besoin d'indexer plusieurs colonnes sans trop savoir dans quel ordre on va les accéder. On peut faire un index GiST multi-colonnes, et voir si ses performances sont satisfaisantes.

Si on reprend l'exemple du slide précédent (la table `demo_gin` a été renommée

demo_gist):

```
=# CREATE EXTENSION btree_gist ;
CREATE EXTENSION
=# CREATE INDEX ON demo_gist USING gist(nom,sexe);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM demo_gist WHERE sexe='b';
                    QUERY PLAN
-----------------------------------------------------
 Index Scan using demo_gist_nom_sexe_idx on demo_gist
        (cost=0.42..8.44 rows=1 width=19)
        (actual time=0.054..0.054 rows=0 loops=1)
   Index Cond: ((sexe)::text = 'b'::text)
 Planning time: 0.214 ms
 Execution time: 0.159 ms
(4 lignes)

=# EXPLAIN ANALYZE SELECT * FROM demo_gist WHERE nom='b';
                     QUERY PLAN
-----------------------------------------------------
 Index Scan using demo_gist_nom_sexe_idx on demo_gist
        (cost=0.42..8.44 rows=1 width=19)
        (actual time=0.054..0.054 rows=0 loops=1)
   Index Cond: (nom = 'b'::text)
 Planning time: 0.224 ms
 Execution time: 0.178 ms
(4 lignes)
```

Ici, les clauses WHERE ne ramènent rien, on est donc dans un cas particulier où les performances sont forcément excellentes. Dans le cas d'une application réelle, il faudra tester de façon très minutieuse.

6.3.9 BRIN

BRIN : **B**lock **R**ange **IN**dex (9.5+)

- Utile pour les tables très volumineuses
  - L'index produit est petit
- Performant lorsque les valeurs sont corrélées à leur emplacement physique
- Types qui peuvent être triés linéairement (pour obtenir min/max)

Soit une table brin_demo contenant l'âge de 100 millions de personnes :

```
CREATE TABLE brin_demo (c1 int);
INSERT INTO brin_demo (SELECT trunc(random() * 90 + 1) AS i
```

```
FROM generate_series(1,100000000));

\dt+ brin_demo
                        Liste des relations
 Schéma |    Nom    | Type  | Propriétaire | Taille  | Description
--------+-----------+-------+--------------+---------+-------------
 public | brin_demo | table | postgres     | 3458 MB |
```

Un index btree va permettre d'obtenir l'emplacement physique (bloc) d'une valeur.

Un index BRIN va contenir une plage des valeurs pour chaque bloc. Dans notre exemple, l'index contiendra la valeur minimale et maximale de plusieurs blocs. La conséquence est que ce type d'index prend très peu de place, il peut facilement tenir en mémoire (réduction des IO disques) :

```
CREATE INDEX demo_btree_idx ON brin_demo USING btree (c1);
CREATE INDEX demo_brin_idx ON brin_demo USING brin (c1);

\di+
                                    Liste des relations
 Schéma |     Nom          | Type  | Propriétaire |      Table      | Taille
--------+------------------+-------+--------------+-----------------+---------
 public | demo_brin_idx    | index | postgres     | brin_demo       | 128 kB
 public | demo_btree_idx   | index | postgres     | brin_demo       | 2142 MB
```

Réduisons le nombre d'enregistrements dans la table :

```
TRUNCATE brin_demo ;
INSERT INTO brin_demo SELECT trunc(random() * 90 + 1) AS i
FROM generate_series(1,100000);
```

Créons un index BRIN avec le paramètre `pages_per_range` à 16 (le défaut est 128):

```
CREATE INDEX demo_brin_idx_16  ON brin_demo USING brin(c1)
WITH (pages_per_range = 16) ;
```

Chaque page de l'index contiendra donc la plage de valeurs de 16 blocs. On peut le voir avec cette requête (qui nécessite l'extension `pageinspect`) :

```
SELECT * FROM brin_page_items(get_raw_page('demo_brin_idx_16', 2),
'demo_brin_idx_16');

 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+-----------
          1 |      0 |      1 | f        | f        | f           | {1 .. 90}
          2 |     16 |      1 | f        | f        | f           | {1 .. 90}
          3 |     32 |      1 | f        | f        | f           | {1 .. 90}
...
```

On constate que les blocs de 0 à 16 contiennent les valeurs de 1 à 90. Ceci s'explique par le fait que les valeurs que nous avons insérées étaient aléatoires. Si nous réorganisons la table :

```
CLUSTER brin_demo USING demo_btree_idx;

SELECT * FROM brin_page_items(get_raw_page('demo_brin_idx_16', 2),
'demo_brin_idx_16');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |   value
------------+--------+--------+----------+----------+-------------+------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 4}
          2 |     16 |      1 | f        | f        | f           | {4 .. 7}
          3 |     32 |      1 | f        | f        | f           | {7 .. 10}
...
```

Les 16 premiers blocs contiennent les valeurs de 1 à 4.

Ainsi, pour une requête du type `SELECT c1 FROM brin_demo WHERE c1 BETWEEN 1 AND 9`, le moteur n'aura qu'à parcourir les 32 premiers blocs de la table.

Autre exemple avec plusieurs colonnes et un type text :

```
CREATE TABLE test (id serial primary key, val text);

INSERT INTO test (val) SELECT md5(i::text) FROM generate_series(1, 10000000) i;
```

=> colonne id qui sera corrélée (séquence), colonne md5 qui ne sera pas du tout corrélée.

On crée un index sur deux colonnes, un int (val) et un text (val)

```
CREATE INDEX brin1_idx ON test USING brin (id,val);

 \dt+ test
                    List of relations
 Schema | Name | Type  |  Owner   |  Size  | Description
--------+------+-------+----------+--------+-------------
 cave   | test | table | postgres | 651 MB |
(1 row)

\di+ brin1*
                          List of relations
 Schema |   Name    | Type  |  Owner   | Table |  Size  | Description
--------+-----------+-------+----------+-------+--------+-------------
 cave   | brin1_idx | index | postgres | test  | 104 kB |
(1 row)
```

Un aperçu du contenu de l'index :

```
SELECT itemoffset, blknum, attnum AS attn, allnulls AS alln, hasnulls AS hasn,
value
```

```
FROM brin_page_items(get_raw_page('brin1_idx', 2),'brin1_idx');

 itemoffset | blknum | attn | alln | hasn |  value
------------+--------+------+------+------+-----------------------------------
          1 |      0 |    1 | f    | f    | {1 .. 15360}
          1 |      0 |    2 | f    | f    | {00003e3b9e5336685200ae85d21b4f5e ..
            |        |      |      |      |  fffb8ef15de06d87e6ba6c830f3b6284}
          2 |    128 |    1 | f    | f    | {15361 .. 30720}
          2 |    128 |    2 | f    | f    | {00053f5e11d1fe4e49a221165b39abc9 ..
            |        |      |      |      |  fffe9f664c2ddba4a37bcd35936c7422}
          3 |    256 |    1 | f    | f    | {30721 .. 46080}
          3 |    256 |    2 | f    | f    | {0002ac0d783338cfeab0b2bdbd872cda ..
                                             fffffe98d0963d27015c198262d97221}
....
```

La colonne `blknum` indique le bloc. Par défaut, le nombre de pages est de 128. La colonne `attnum` correspond à l'attribut. On remarque bien que l'`id` est corrélé, contrairement à la colonne `val`. Ce que nous confirme bien la vue `pg_stats` :

```
SELECT tablename,attname,correlation FROM pg_stats WHERE tablename='test';
 tablename | attname | correlation
-----------+---------+-------------
 test      | id      |           1
 test      | val     |  -0.0078914
(2 rows)
```

Si l'on teste la requête suvante, on s'apercevra que PostgreSQL effectue un parcours complet (Seq Scan) et n'utilise donc pas l'index BRIN. Pour comprendre pourquoi, essayons de l'y forcer :

```
SET enable_seqscan TO off ;

EXPLAIN (buffers,analyze) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
    AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=724.60..234059.04 rows=2654947 width=37)
                           (actual time=11.046..2152.131 rows=2668675 loops=1)
   Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                       AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Rows Removed by Index Recheck: 7331325
   Heap Blocks: lossy=83334
   Buffers: shared hit=83343
   ->  Bitmap Index Scan on brin1_idx  (cost=0.00..60.86 rows=10000029 width=0)
                               (actual time=10.851..10.851 rows=834560 loops=1)
```

```
        Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                    AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
        Buffers: shared hit=9
 Planning time: 0.223 ms
 Execution time: 2252.320 ms
(10 lignes)
```

83343 blocs lus (651 Mo) soit l'intégralité de la table ! Il est donc logique que PostgreSQL préfère d'entrée un Seq Scan.

Pour pouvoir trier la table avec une commande CLUSTER il nous faut un index btree classique :

```
CREATE INDEX brin_btree_idx ON test USING btree (val);

\di+ brin_btree_idx
                              List of relations
 Schema |      Name      | Type  |  Owner   | Table |  Size  | Description
--------+----------------+-------+----------+-------+--------+-------------
 cave   | brin_btree_idx | index | postgres | test  | 563 MB |
```

Notons au passage que cet index btree est presque aussi gros que notre table !

Après la commande CLUSTER, notre table est bien corrélée avec val (mais plus avec id) :

```
CLUSTER test USING brin_btree_idx ;

ANALYZE test;

ANALYZE

SELECT tablename,attname,correlation FROM pg_stats
WHERE tablename='test';

 tablename | attname | correlation
-----------+---------+-------------
 test      | id      |  -0.0111721
 test      | val     |           1
(2 rows)
```

La requête après le cluster :

```
SET enable_seqscan TO on ;


EXPLAIN (BUFFERS,ANALYZE) SELECT * FROM test WHERE val
between 'a87ff679a2f3e71d9181a67b7542122c'
    AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                                QUERY PLAN
```

```
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=714.83..124309.35 rows=2676913 width=37)
                           (actual time=8.341..749.952 rows=2668675 loops=1)
   Recheck Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                 AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Rows Removed by Index Recheck: 19325
   Heap Blocks: lossy=22400
   Buffers: shared hit=9 read=22400
   ->  Bitmap Index Scan on brin1_idx  (cost=0.00..45.60 rows=2684035 width=0)
                                       (actual time=2.090..2.090 rows=224000 loops=1)
         Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
                      AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
         Buffers: shared hit=9
 Planning time: 0.737 ms
 Execution time: 856.974 ms
(10 lignes)
```

22409 blocs lus soit 175 Mo. Dans la table triée, l'index BRIN devient intéressant.

On supprime notre index brin et on garde l'index b-tree :

```
DROP INDEX brin1_idx;


EXPLAIN (BUFFERS,ANALYZE) SELECT * FROM test WHERE val
BETWEEN 'a87ff679a2f3e71d9181a67b7542122c'
AND 'eccbc87e4b5ce2fe28308fd9f2a7baf3';
                          QUERY PLAN
---------------------------------------------------------------------
 Index Scan using brin_btree_idx on test
        (cost=0.56..151908.16 rows=2657080 width=37)
        (actual time=0.032..449.482 rows=2668675 loops=1)
   Index Cond: ((val >= 'a87ff679a2f3e71d9181a67b7542122c'::text)
        AND (val <= 'eccbc87e4b5ce2fe28308fd9f2a7baf3'::text))
   Buffers: shared read=41306 written=10
 Planning time: 0.137 ms
 Execution time: 531.724 ms
```

Même durée d'exécution mais le nombre de blocs lus est beaucoup plus important :

41306 blocs (322 Mo), presque deux fois plus de blocs lus.

En résumé, les index BRIN sont intéressants pour les tables volumineuses et où il y a une forte corrélation entre les valeurs et leur emplacement physique.

Pour plus d'informations, voir cet article[26] .

[26] http://pythonsweetness.tumblr.com/post/119568339102/block-range-brin-indexes-in-postgresql-95

6.3.10 HASH

Index Hash :

- Non journalisés donc facilement corrompus
- Moins performants que les Btree
- Ne gèrent que les égalités, pas < et >
- Mais plus compacts
- À ne pas utiliser

Les index Hash ne sont pas journalisés. Il ne s'agit pas d'une impossibilité technique. Le code n'a pour le moment jamais été réalisé : les index Hash étant plus lents en interrogation que les index Btree à l'heure actuelle, l'effort d'écriture d'un code de journalisation n'a donc pas de sens pour le moment.

L'utilisation d'un index Hash est donc une mauvaise idée à l'heure actuelle.

6.3.11 UTILISATION D'INDEX

Index inutilisé :

- L'optimiseur pense qu'il n'est pas rentable
  - Il a le plus souvent raison
  - S'il se trompe : statistiques ? bug ?
- La requête n'est pas compatible
  - Clause WHERE avec fonction ?
  - Cast ?
- C'est souvent tout à fait normal

C'est l'optimiseur SQL qui décide si un index doit ou non être utilisé. Il est donc tout à fait possible que PostgreSQL décide ne de pas utiliser votre index.

L'optimiseur peut bien sûr avoir raison, c'est le cas le plus fréquent. S'il se trompe, vous pouvez être dans les cas suivants :

- Vos statistiques ne sont pas à jour (exécutez `ANALYZE`)
- Les statistiques ne sont pas assez fines (exécutez `ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 1000;`)
- Les statistiques sont trompeuses : par exemple, vous exécutez une clause WHERE sur deux colonnes corrélées (ville et code postal par exemple), et l'optimiseur ne le sait pas. Il n'y a pas de solution efficace à ce problème pour le moment, les développeurs de PostgreSQL y travaillent

- L'optimiseur a un bug

Il se peut aussi que votre index ne soit pas compatible avec votre clause WHERE :

- L'index n'est pas compatible avec le type de recherche que vous effectuez : vous avez créé un index btree sur un tableau, et vous exécutez une recherche sur un de ses éléments, ou vous avez un index « normal » sur une chaîne texte, et vous faites une recherche de type LIKE, etc...
- Vous appliquez une fonction sur la colonne à indexer : le classique est :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

Il n'utilisera bien sûr pas l'index sur ma_date. Il faut réécrire comme ceci :

```
SELECT * FROM ma_table WHERE ma_date >='2014-01-01' and ma_date<'2015-01-01' ;
```

par exemple. Un cast implicite sur une colonne aura le même effet.

6.3.12 CONTRAINTES D'EXCLUSION

Contrainte d'exclusion : Une extension du concept d'unicité

- Unicité : `n-uplet1 = n-uplet2` interdit dans une table
- Contrainte d'exclusion : `n-uplet1 op n-uplet2` interdit dans une table
- op est n'importe quel opérateur indexable par GiST

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

Les contraintes d'unicité sont une forme simple de contraintes d'exclusion. Si on prend l'exemple :

```
CREATE TABLE foo (
    id int,
    nom text,
    EXCLUDE (id WITH =)
);
```

cette déclaration est identique à l'utilisation d'une contrainte UNIQUE sur `foo.id`, sauf qu'ici on passe par le mécanisme des contraintes d'exclusion; on utilise aussi un index BTree pour forcer la contrainte. Les NULLs sont toujours permis, exactement comme avec une contrainte UNIQUE.

C'est la même chose si l'on veut une contrainte unique sur plusieurs colonnes :

```
CREATE TABLE foo (
    nom text,
    naissance date,
    EXCLUDE (nom WITH =, naissance WITH =)
);
```

L'intérêt avec les contraintes d'exclusion c'est qu'on peut utiliser des index d'un autre type que les BTree, comme les GiST ou les Hash, et surtout des opérateurs autres que l'égalité.

Il est par exemple impossible en utilisant une contrainte UNIQUE d'imposer que deux enregistrements contiennent des valeurs qui ne se chevauchent pas; mais il est possible de spécifier une telle contrainte avec une contrainte d'exclusion.

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
INSERT INTO circles(c) VALUES ('10, 4, 10');
INSERT INTO circles(c) VALUES ('8, 3, 8');

ERROR:  conflicting key value violates exclusion constraint "circles_c_excl"
DETAIL : Key (c)=(<(8,3),8>) conflicts with existing key (c)=(<(10,4),10>).
```

En outre, les contraintes d'exclusion supportent toutes les fonctionnalités avancées que l'on est en droit d'attendre d'un système comme PostgreSQL : mode différé, application de la contrainte à un sous-ensemble de la table (permet une clause WHERE), ou utilisation de fonctions/expressions en place de références de colonnes.

L'exemple le plus fréquemment proposé pour illustrer est le suivant :

```
=# CREATE TABLE reservation
(
  salle      TEXT,
  professeur TEXT,
  durant    tstzrange);
CREATE TABLE
=# CREATE EXTENSION btree_gist ;
CREATE EXTENSION
=# ALTER TABLE reservation ADD CONSTRAINT test_exclude EXCLUDE
USING gist (salle WITH =,durant WITH &&);
ALTER TABLE
=# INSERT INTO reservation (professeur,salle,durant) VALUES
( 'marc', 'salle techno', '[2010-06-16 09:00:00, 2010-06-16 10:00:00)');
INSERT 0 1
=# INSERT INTO reservation (professeur,salle,durant) VALUES
( 'jean', 'salle techno', '[2010-06-16 10:00:00, 2010-06-16 11:00:00)');
INSERT 0 1
=# INSERT INTO reservation (professeur,salle,durant) VALUES
```

```
( 'jean', 'salle informatique', '[2010-06-16 10:00:00, 2010-06-16 11:00:00)');
INSERT 0 1
=# INSERT INTO reservation (professeur,salle,durant) VALUES
( 'michel', 'salle techno', '[2010-06-16 10:30:00, 2010-06-16 11:00:00)');
ERROR:  conflicting key value violates exclusion constraint "test_exclude"
DETAIL : Key (salle, durant)=(salle techno,
              ["2010-06-16 10:30:00+02","2010-06-16 11:00:00+02"))
         conflicts with existing key
         (salle, durant)=(salle techno,
                          ["2010-06-16 10:00:00+02","2010-06-16 11:00:00+02")).
```

On notera l'utilisation de l'extension btree gist. Elle permet d'utiliser l'opérateur '=' , indexé normalement par un index de type BTree, avec un index GiST, ce qui nous permet d'utiliser '=' dans une contrainte d'exclusion.

Cet exemple illustre la puissance du mécanisme. Il est quasiment impossible de réaliser la même opération sans contrainte d'exclusion, à part en verrouillant intégralement la table, ou en utilisant le mode d'isolation *serializable*, qui a de nombreuses implications plus profondes sur le fonctionnement de l'application.

6.4 PARTITIONNEMENT ANCIENNE GÉNÉRATION

- Avant la version 10
- Fractionner une table en plusieurs tables plus petites
  - meilleures performances
  - maintenance plus facile
- Type de fractionnement
  - liste
  - intervalle ou échelle de valeur
  - hachage

Le partitionnement est une technique qui permet de diviser une table très volumineuse en plusieurs tables plus petites. Les petites tables sont appelées des partitions.

Ajouter du partitionnement peut avoir deux buts :

- améliorer les performances (si le partitionnement est bien fait et que les requêtes sont correctement écrites, il est possible de ne parcourir que quelques partitions, et non pas la totalité) ;
- améliorer la maintenance (le nettoyage d'une partition ou la suppression de ses données est bien plus simple et performante).

Reste à savoir comment partitionner une table. Il existe plusieurs types de partitionnements.

Dans un partitionnement de type liste, une partition porte sur une liste de valeurs. Cette liste est pré-définie et le nombre de partitions est souvent invariable dans le temps. Le volume de données de chaque partition a tendance à croître avec le temps. La maintenance est faible sur ce type de partitionnement.

Dans un partitionnement de type intervalle, une partition porte les données définies sur une échelle de valeurs bornées. Utilisé typiquement avec des échelles temporelles, où une partition contient les données d'une période de temps donnée, le nombre de partitions évolue dans le temps. Le volume de données est sensiblement le même d'une partition à une autre. Ce type de partitionnement nécessite une maintenance périodique assez lourde.

Le partitionnement de type hachage est déconseillé avec PostgreSQL car l'optimiseur ne dispose pas de moyens suffisants pour déterminer la partition adéquate. Une telle implémentation reste possible mais il est nécessaire de spécifier la fonction de hachage dans un prédicat pour que l'optimiseur sache déterminer la partition touchée.

6.4.1 PRINCIPE DU PARTITIONNEMENT

- Déterminer la clé de partitionnement
- Créer la table principale
- Créer les tables filles
- Rediriger les écritures
  - Écriture dans la dernière partition créée
  - Écriture en fonction de la clé
- Paramétrer PostgreSQL pour les lectures
- Tester les requêtes (plans d'exécution)

La mise en œuvre du partitionnement ne doit se faire qu'après avoir détecté des problèmes réels pour lesquels une solution serait le partitionnement. Il ne faut jamais mettre en place du partitionnement dès le départ en supposant que ce sera la bonne solution.

La mise en œuvre peut être relativement simple. Le plus complexe est de déterminer la bonne clé de partitionnement de la table principale. Comme l'objectif du partitionnement est d'éviter de parcourir la table entière, il faut savoir comment la table est parcourue. Sur tel type de table, ce sera sur une date, sur tel autre sur un identifiant, etc. Une fois que cette information est connue, la clé de partitionnement devient logique. Parfois, l'objectif

du partitionnement est de pouvoir supprimer un lot de données de façon efficace. Dans ce cas, la clé est encore plus évidente : c'est celle qu'on va utiliser pour la suppression.

Ceci fait, les partitions sont faciles à créer. Ce sont des tables standards qui héritent de la table mère et qui ont en plus une contrainte `CHECK` dépendant de la clé de partitionnement. Pour les lectures, l'optimiseur de PostgreSQL saura déterminer quelles partitions sont accédés à partir des prédicats de la requête. Il faudra simplement s'assurer que le paramètre `constraint_exclusion` a bien la valeur `partition` ou `on`, `partition` étant la valeur par défaut.

Le plus complexe tient dans la gestion des écritures, et notamment des insertions et mises à jour de la clé de partitionnement elle-même. Pour cela, il faudra déterminer comment sont réalisées les écritures : soit dans la dernière partition active, soit en fonction de la clé de partitionnement. Dans tous les cas, PostgreSQL ne sait pas rediriger tout seul les écritures. Il faudrait l'aider en mettant en place un trigger sur la table principale. Ce trigger s'occupera d'écriture les données dans la bonne partition.

Enfin, une fois le système de partitionnement créé (tables filles, contraintes CHECK, trigger), il faudra s'assurer que PostgreSQL sait tirer partie du partitionnement pour les requêtes à exécuter. Il faudra s'assurer que les clauses WHERE permettent bien de déterminer la partition accédée et que ces prédicats peuvent être utilisés sans difficultés par l'optimiseur (concordance des types, etc).

6.4.2 PARTITIONNEMENT ET HÉRITAGE

- Héritage de tables
- Table principale :
  - table mère définie normalement
- Partitions :
  - tables filles
  - héritent des propriétés de la table mère
  - mais pas les contraintes, index et droits

L'héritage d'une table mère transmet les propriétés suivantes à la table fille :

- les colonnes ;
- les contraintes `CHECK`.

Mais l'héritage ne transmet pas :

- les contraintes d'unicité et référentielles ;
- les index ;

- les droits. Les contraintes d'unicité ne sont pas vérifiées entre une table mère et une table fille : ce sont techniquement des tables indépendantes. Rien n'empêche d'avoir des doublons entre la table mère et la table fille. Cela empêche aussi bien sûr la mise en place de clé étrangère, puisqu'une clé étrangère s'appuie sur une contrainte d'unicité de la table référencée.

Il faut être vigilant à bien recréer les contraintes et index manquants ainsi qu'à attribuer les droits sur les objets de manière adéquate. L'une des erreurs les plus fréquentes lorsqu'on met en place le partitionnement est d'oublier de créer les contraintes, index et droits qui n'ont pas été transmis.

Le script suivant permet de créer une table `logs` qui sera partitionnée en plusieurs partitions par année, en s'appuyant sur le mécanisme d'héritage :

```
-- table mère
CREATE TABLE logs (
  id         integer,
  dreception timestamp,
  contenu    text
);

-- table fille correspondant à l'année 2014
CREATE TABLE logs_2014 (
  CHECK (dreception BETWEEN '2014-01-01' AND '2014-12-31')
) INHERITS (logs);
-- table fille correspondant à l'année 2013
CREATE TABLE logs_2013 (
  CHECK (dreception BETWEEN '2013-01-01' AND '2013-12-31')
) INHERITS (logs);
-- table fille correspondant à l'année 2012
CREATE TABLE logs_2012 (
  CHECK (dreception BETWEEN '2012-01-01' AND '2012-12-31')
) INHERITS (logs);
```

6.4.3 GESTION DES LECTURES

- Les lectures sont gérées par l'optimiseur
- `constraint_exclusion` change son comportement
  - `off`, optimisation du partitionnement désactivée
  - `partition`, optimisation activée pour les tables enfants ou requêtes avec `UNION ALL`
  - `on`, recherche d'une optimisation pour toutes les tables

- `constraint_exclusion = partition` par défaut
- L'optimisation consiste à ne parcourir que les partitions utiles

Par défaut, le planificateur ne parcourt que les partitions intéressantes si la clé de partitionnement est prise en compte dans le filtre de la requête. Par exemple :

```
partitionnement=# EXPLAIN SELECT * FROM logs WHERE dreception
BETWEEN '2013-08-03' AND '2013-09-03';
                                QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.00..26.50 rows=7 width=44)
   ->  Append  (cost=0.00..26.50 rows=7 width=44)
         ->  Seq Scan on logs  (cost=0.00..0.00 rows=1 width=44)
               Filter: ((dreception >=
                         '2013-08-03 00:00:00'::timestamp without time zone)
                       AND (dreception <=
                            '2013-09-03 00:00:00'::timestamp without time zone))
         ->  Seq Scan on logs_2013 logs  (cost=0.00..26.50 rows=6 width=44)
               Filter: ((dreception >=
                         '2013-08-03 00:00:00'::timestamp without time zone)
                       AND (dreception <=
                            '2013-09-03 00:00:00'::timestamp without time zone))
(6 rows)
```

Nous ne cherchons que des données de 2013. Les contraintes `CHECK` nous assurent que les partitions `logs_2012` et `logs_2014` ne contiennent pas de données de 2013, il n'est donc pas nécessaire de les parcourir. Par conséquent, l'optimiseur ne prévoit le parcours que de la partition `logs_2013` ainsi que de la table `logs` qui ne contient pas de contrainte CHECK sur `dreception`.

Par contre, si la requête cherche à récupérer toutes les données ou seulement certaines mais sans passer par la clé de partitionnement, le partitionnement en lui-même n'apportera aucun gain en performance, et même certainement un ralentissement (certains plans n'étant pas possibles si une table est partitionnée). Il faudra de toute façon passer par toutes les partitions :

```
partitionnement=# EXPLAIN SELECT * FROM logs WHERE id BETWEEN 10 AND 50;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.00..79.50 rows=19 width=44)
   ->  Append  (cost=0.00..79.50 rows=19 width=44)
         ->  Seq Scan on logs  (cost=0.00..0.00 rows=1 width=44)
               Filter: ((id >= 10) AND (id <= 50))
         ->  Seq Scan on logs_2014 logs  (cost=0.00..26.50 rows=6 width=44)
               Filter: ((id >= 10) AND (id <= 50))
         ->  Seq Scan on logs_2013 logs  (cost=0.00..26.50 rows=6 width=44)
               Filter: ((id >= 10) AND (id <= 50))
```

```
        ->  Seq Scan on logs_2012 logs  (cost=0.00..26.50 rows=6 width=44)
              Filter: ((id >= 10) AND (id <= 50))
(10 rows)
```

Donc attention à l'écriture des requêtes, le résultat peut être excellent ou catastrophique.

6.4.4 GESTION DES ÉCRITURES

- PostgreSQL sait gérer
  - les DELETE
  - les UPDATE (tant que la clé de partitionnement n'est pas mise à jour)
- Il faut l'aider pour :
  - rediriger les INSERT dans la bonne partition
  - aider les UPDATE sur la clé de partitionnement
- On utilisera un TRIGGER

PostgreSQL sait automatiquement gérer les suppressions. Il supprime tous les éléments dans la table mère et les tables filles associées en fonction des filtres.

Il donne le même traitement aux mises à jour. Un cas particulier concerne les mises à jour de la clé de partitionnement elle-même. Prenons un exemple. Si nous cherchons à mettre à jour la date de réception d'une trace dans notre table `logs`, il est possible que la nouvelle valeur doive placer la ligne correspondante dans une autre partition. PostgreSQL ne le fera pas automatiquement. Mais en fait, un premier problème survient : la nouvelle valeur viole la contrainte d'intégrité.

Il existe deux solutions à ce problèmes :

- l'application n'exécute plus d' `UPDATE`, mais fait un `DELETE` suivi d'un `INSERT` à la place ;
- PostgreSQL réagit à la mise à jour en faisant de lui-même un `DELETE` suivi d'un `INSERT` ;

La première solution a l'avantage d'être simple, mais demande une modification de l'application. La deuxième solution est moins simple à mettre en place, mais ne demande aucune modification de l'application. C'est d'ailleurs généralement cette solution qui est adoptée.

Restent les `INSERT`. Un `INSERT` sur la table mère est fait sur la table mère. Or, nous voulons que l'insertion se fasse réellement dans la partition adéquate. Là-aussi, PostgreSQL ne le fera pas automatiquement. Deux solutions existent :

- l'application exécute l'`INSERT` directement dans la bonne partition ;

- PostgreSQL réagit à l'insertion dans la table mère en faisant de lui-même l'insertion dans la bonne partition.

La première solution a l'avantage d'être rapide, simple, mais demande une modification de l'application. De plus, toute application qui ne suit pas ce pré-requis auront des performances en insertion réduite : la deuxième solution est plus complexe à mettre en place, ralentit les insertions, mais ne demande aucune modification de l'application. La solution adoptée par défaut est la deuxième, avec un repli sur la première si les performances ne sont pas suffisantes.

Avec PostgreSQL, il existe deux moyens de rediriger les écritures : avec des règles ou des triggers. Les règles (ou `RULES`) présentent bien plus d'inconvénients que d'avantages, leur emploi est vivement déconseillé. Un trigger est bien plus adapté à cette problématique. Nous allons donc présenter cette solution.

Dans le cas du partitionnement, avant la version 10, la clause `FOR EACH STATEMENT` ne nous intéresse pas car elle ne permet pas d'avoir accès aux lignes concernées par l'insertion ou la mise à jour. Le trigger à créer est donc `FOR EACH ROW`.

L'écriture d'une procédure trigger ne diffère pas tellement de l'écriture de n'importe quelle procédure stockée. La valeur de retour est forcément de type trigger. La valeur renvoyée est une valeur ligne indiquant la valeur de la nouvelle ligne. Si cette dernière est NULL, l'opération est annulée. Il est possible d'accéder à l'ancienne valeur de la ligne avec la variable OLD et à la nouvelle avec la variable `NEW`.

L'écriture de la procédure trigger va dépendre du partitionnement :

- écriture dans la dernière partition créée ?
- nombre de partitions ?
- SQL statique ou dynamique ?

Si le nombre de partitions est important, il est préférable d'avoir du SQL dynamique pour diminuer le nombre d'instructions dans la procédure stockée. De même si des partitions sont fréquemment ajoutées, il est préférable de passer par du SQL dynamique pour ne pas avoir à modifier la procédure à chaque ajout d'une partition. Dans les autres cas, avoir du SQL statique est intéressant en terme de performance.

6.4.5 INSERTION DE DONNÉES

```
CREATE OR REPLACE FUNCTION ins_logs() RETURNS TRIGGER
LANGUAGE plpgsql AS $FUNC$
BEGIN
  IF NEW.dreception >= '2014-01-01'
```

```
  AND NEW.dreception <= '2014-12-31' THEN
    INSERT INTO logs_2014 VALUES (NEW.*);
  ELSIF NEW.dreception >= '2013-01-01'
  AND NEW.dreception <= '2013-12-31' THEN
    INSERT INTO logs_2013 VALUES (NEW.*);
  ELSIF NEW.dreception >= '2012-01-01'
  AND NEW.dreception <= '2012-12-31' THEN
    INSERT INTO logs_2012 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$FUNC$;
```

L'écriture d'un trigger statique a l'avantage d'être très performante. Son inconvénient majeur est qu'il faut écrire une forêt d'instructions `IF` et qu'il y aura autant d'ordres `INSERT` que de partitions. Cependant, parce que les requêtes d'insertion ne sont pas dynamiques, la durée d'exécution est moindre. Par contre, elle peut varier suivant le nombre de tests effectués. Il est donc préférable que la table la plus insérée corresponde au premier test.

```
CREATE OR REPLACE FUNCTION ins_logs() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.dreception >= '2014-01-01' AND NEW.dreception <= '2014-12-31' THEN
    INSERT INTO logs_2014 VALUES (NEW.*);
  ELSIF NEW.dreception >= '2013-01-01' AND NEW.dreception <= '2013-12-31' THEN
    INSERT INTO logs_2013 VALUES (NEW.*);
  ELSIF NEW.dreception >= '2012-01-01' AND NEW.dreception <= '2012-12-31' THEN
    INSERT INTO logs_2012 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_ins_logs
   BEFORE INSERT ON logs
   FOR EACH ROW
   EXECUTE PROCEDURE ins_logs();
```

6.4.6 MISE À JOUR DE LA CLÉ DE PARTITIONNEMENT

```
CREATE OR REPLACE FUNCTION f_upd_logs() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
```

```
BEGIN
  DELETE FROM logs_2014 WHERE dreception=OLD.dreception;
  INSERT INTO logs VALUES (NEW.*);
  RETURN NULL;
END;
$$;
```

Les UPDATE sur la clé de partitionnement sont gérés par un trigger particulier qui n'est exécuté que lors d'un UPDATE sur la clé de partitionnement (voir clause `WHEN` de l'ordre de création du trigger) :

```
CREATE OR REPLACE FUNCTION f_upd_logs() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
  DELETE FROM logs_2014 WHERE dreception=OLD.dreception;
  INSERT INTO logs VALUES (NEW.*);
  RETURN NULL;
END;
$$;

CREATE TRIGGER tr_upd_logs
  BEFORE UPDATE ON logs_2014
  FOR EACH ROW
  WHEN (NEW.dreception != OLD.dreception)
  EXECUTE PROCEDURE f_upd_logs();
```

6.4.7 LIMITATIONS DU PARTITIONNEMENT

- Pas de contraintes d'unicité sur l'ensemble des tables partitionnées
  - la contrainte n'est vérifiée que dans la partition
- Performances dégradées en écriture
- Certaines requêtes ont des plans d'exécution désastreux
- L'héritage n'est pas conçu pour permettre plus de 100 partitions
- Utilisation bien plus importante des verrous

Les contraintes d'unicité ne sont vérifiées que dans une seule partition, pas pour l'ensemble de la table. Il est donc très difficile d'assurer l'unicité d'une clé primaire sur une table partitionnée. Par conséquent, il n'est pas non plus possible de positionner une clé étrangère sur les tables partitionnées.

Les performances s'en retrouvent également dégradées en insertion car ces écritures doivent être redirigées par un trigger (ou une règle) et cette exécution de code supplé-

mentaire peut fortement impacter les performances.

Il peut arriver que l'optimiseur de PostgreSQL ne génère pas des plans d'exécution optimaux. Ce dernier problème est contourné facilement en écrivant des requêtes SQL ayant un prédicat qui identifie la partition de façon claire.

Enfin, il faut avoir en tête que la modification de la définition de la table parent entraîne l'acquisition d'un verrou exclusif sur la table parent et les tables filles. Les autres transactions sont alors bloquées le temps de la modification de la définition de la table.

Enfin, PostgreSQL ne retourne plus le nombre d'enregistrements insérés sur une table partitionnée à cause du trigger d'insertion sur cette table.

L'héritage a été conçu pour avoir quelques tables filles, mais pas forcément un grand nombre. C'est plutôt de l'ordre de la dizaine à la centaine, mais pas plus.

Toute requête sur une table mère va nécessiter un grand nombre de verrous. Ce ne seront pas forcément des verrous bloquants mais ils seront vraiment nombreux.En effet, une lecture sur une table mère entraîne la prise d'un verrou sur la table mère, mais aussi sur chacune des tables filles. Ainsi, si une table est partitionnée en 100 tables, chaque requête posera 101 verrous pour protéger l'accès aux ressources. Il faudra donc très probablement augmenter `max_locks_per_transaction`. La configuration du système de supervision devra tenir compte de cet aspect.

6.4.8 OUTILS DE PARTITIONNEMENT

- Outils pour simplifier la gestion
  - pg_partman

Le logiciel pg_partman permet de simplifier la maintenance de tables partitionnées lorsque les tables sont partitionnées sur une échelle temporelle ou de valeurs (partitionnement par range). Il se présente sous forme d'extension PostgreSQL, son code source est disponible sur ce dépôt github[27] .

6.5 PARTITIONNEMENT NOUVELLE GÉNÉRATION

- À partir de la version 10
- Mise en place et administration simplifiées car intégrées au moteur

[27] https://github.com/keithf4/pg_partman

- Gestion automatique des lectures et écritures
- Partitions
  - attacher/détacher une partition
  - contrainte implicite de partitionnement
  - expression possible pour la clé de partitionnement
  - sous-partitions possibles

La version 10 apporte un nouveau système de partitionnement se basant sur une infrastructure qui existait déjà dans PostgreSQL.

Le but est de simplifier la mise en place et l'administration des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL déjà existants, comme `CREATE TABLE` et `ALTER TABLE`, pour attacher (`ATTACH PARTITION`) et détacher des partitions (`DETACH PARTITION`).

Au niveau de la simplification de la mise en place, on peut noter qu'il n'est plus nécessaire de créer une fonction trigger et d'ajouter des triggers pour gérer les insertions et mises à jour. Le routage est géré de façon automatique en fonction de la définition des partitions. Si les données insérées ne trouvent pas de partition cible, l'insertion est tout simplement en erreur. Du fait de ce routage automatique, les insertions se révèlent aussi plus rapides.

6.5.1 PARTITIONNEMENT PAR LISTE

- Liste de valeurs par partition
- Créer une table partitionnée :
  ```
  CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);
  ```
- Ajouter une partition :
  ```
  CREATE TABLE t1_a PARTITION of t1 FOR VALUES IN (1, 2, 3);
  ```
- Attacher la partition :
  ```
  ALTER TABLE t1 ATTACH PARTITION t1_a FOR VALUES IN (1, 2, 3);
  ```
- Détacher la partition :
  ```
  ALTER TABLE t1 DETACH PARTITION t1_a;
  ```

Exemple complet :

Création de la table principale et des partitions :

```
postgres=# CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);
CREATE TABLE
```

```
postgres=# CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE

postgres=# CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
CREATE TABLE
```

Insertion de données :

```
postgres=# INSERT INTO t1 VALUES (0);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

postgres=# INSERT INTO t1 VALUES (1);
INSERT 0 1

postgres=# INSERT INTO t1 VALUES (2);
INSERT 0 1

postgres=# INSERT INTO t1 VALUES (5);
INSERT 0 1

postgres=# INSERT INTO t1 VALUES (6);
ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).
```

Lors de l'insertion, les données sont correctement redirigées vers leurs partitions.

Si aucune partition correspondant à la clé insérée n'est trouvée, une erreur se produit.

6.5.2 PARTITIONNEMENT PAR INTERVALLE

- Intervalle de valeurs par partition
- Créer une table partitionnée :

  ```
  CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
  ```

- Ajouter une partition :

  ```
  CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);
  ```

- Détacher une partition :

  ```
  ALTER TABLE t2 DETACH PARTITION t2_1;
  ```

Exemple complet :

Création de la table principale et d'une partition :

```
postgres=# CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE

postgres=# CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) to (100);
CREATE TABLE
```

Insertion de données :

```
postgres=# INSERT INTO t2 VALUES (0);
ERROR:  no PARTITION OF relation "t2" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

postgres=# INSERT INTO t2 VALUES (1);
INSERT 0 1

postgres=# INSERT INTO t2 VALUES (2);
INSERT 0 1

postgres=# INSERT INTO t2 VALUES (5);
INSERT 0 1

postgres=# INSERT INTO t2 VALUES (101);
ERROR:  no PARTITION OF relation "t2" found for row
DETAIL:  Partition key of the failing row contains (c1) = (101).
```

Lors de l'insertion, les données sont correctement redirigées vers leurs partitions.

Si aucune partition correspondant à la clé insérée n'est trouvée, une erreur se produit.

6.5.3 CLÉ DE PARTITIONNEMENT MULTI-COLONNES

- Clé sur plusieurs colonnes acceptée
  - uniquement pour le partitionnement par intervalle
- Créer une table partitionnée avec une clé multi-colonnes :
  ```
  CREATE TABLE t1(c1 integer, c2 text, c3 date)
  PARTITION BY RANGE (c1, c3);
  ```
- Ajouter une partition :
  ```
  CREATE TABLE t1_a PARTITION of t1
   FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11');
  ```

Quand on utilise le partitionnement par intervalle, il est possible de créer les partitions en utilisant plusieurs colonnes.

On profitera de l'exemple ci-dessous pour montrer l'utilisation conjointe de tablespaces différents.

Commençons par créer les tablespaces :

```
postgres=# CREATE TABLESPACE ts0 LOCATION '/tablespaces/ts0';
CREATE TABLESPACE

postgres=# CREATE TABLESPACE ts1 LOCATION '/tablespaces/ts1';
CREATE TABLESPACE

postgres=# CREATE TABLESPACE ts2 LOCATION '/tablespaces/ts2';
CREATE TABLESPACE

postgres=# CREATE TABLESPACE ts3 LOCATION '/tablespaces/ts3';
CREATE TABLESPACE
```

Créons maintenant la table partitionnée et deux partitions :

```
postgres=# CREATE TABLE t2(c1 integer, c2 text, c3 date not null)
       PARTITION BY RANGE (c1, c3);
CREATE TABLE

postgres=# CREATE TABLE t2_1 PARTITION OF t2
       FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11')
       TABLESPACE ts1;
CREATE TABLE

postgres=# CREATE TABLE t2_2 PARTITION OF t2
       FOR VALUES FROM (100,'2017-08-11') TO (200, '2017-08-12')
       TABLESPACE ts2;
CREATE TABLE
```

Si les valeurs sont bien comprises dans les bornes :

```
postgres=# INSERT INTO t2 VALUES (1, 'test', '2017-08-10');
INSERT 0 1

postgres=# INSERT INTO t2 VALUES (150, 'test2', '2017-08-11');
INSERT 0 1
```

Si la valeur pour `c1` est trop petite :

```
postgres=# INSERT INTO t2 VALUES (0, 'test', '2017-08-10');
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (0, 2017-08-10).
```

Si la valeur pour `c3` (colonne de type date) est antérieure :

```
postgres=# INSERT INTO t2 VALUES (1, 'test', '2017-08-09');
ERROR:  no partition of relation "t2" found for row
```

```
DÉTAIL : Partition key of the failing row contains (c1, c3) = (1, 2017-08-09).
```

Les valeurs spéciales *MINVALUE* et *MAXVALUE* permettent de ne pas indiquer de valeur de seuil limite. Les partitions `t2_0` et `t2_3` pourront par exemple être déclarées comme suit et permettront d'insérer les lignes qui étaient ci-dessus en erreur. Attention, certains articles en ligne ont été créés avant la sortie de la version *beta3* et ils utilisent la valeur spéciale *UNBOUNDED* qui a été remplacée par *MINVALUE* et *MAXVALUE*.

```
postgres=# CREATE TABLE t2_0 PARTITION OF t2
       FOR VALUES FROM (MINVALUE, MINVALUE) TO (1,'2017-08-10')
       TABLESPACE ts0;

postgres=# CREATE TABLE t2_3 PARTITION OF t2
       FOR VALUES FROM (200,'2017-08-12') TO (MAXVALUE, MAXVALUE)
       TABLESPACE ts3;
```

Enfin, on peut consulter la table `pg_class` afin de vérifier la présence des différentes partitions :

```
postgres=# ANALYZE t2;
ANALYZE

postgres=# SELECT relname,relispartition,relkind,reltuples
           FROM pg_class WHERE relname LIKE 't2%';
 relname | relispartition | relkind | reltuples
---------+----------------+---------+-----------
 t2      | f              | p       |         0
 t2_0    | t              | r       |         2
 t2_1    | t              | r       |         1
 t2_2    | t              | r       |         1
 t2_3    | t              | r       |         0
(5 lignes)
```

6.5.4 PERFORMANCES EN INSERTION

t1 (non partitionnée) :

```
INSERT INTO t1 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 10097.098 ms (00:10.097)
```

t2 (nouveau partitionnement) :

```
INSERT INTO t2 select i, 'toto'
  FROM generate_series(0, 9999999) i;
```

```
Time: 11448.867 ms (00:11.449)
```

t3 (ancien partitionnement) :

```
INSERT INTO t3 select i, 'toto'
  FROM generate_series(0, 9999999) i;
Time: 125351.918 ms (02:05.352)
```

La table *t1* est une table non partitionnée. Elle a été créée comme suit :

```
CREATE TABLE t1 (c1 integer, c2 text);
```

La table *t2* est une table partitionnée utilisant les nouvelles fonctionnalités de la version 10 de PostgreSQL :

```
CREATE TABLE t2 (c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (      0) TO ( 1000000);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (1000000) TO ( 2000000);
CREATE TABLE t2_3 PARTITION OF t2 FOR VALUES FROM (2000000) TO ( 3000000);
CREATE TABLE t2_4 PARTITION OF t2 FOR VALUES FROM (3000000) TO ( 4000000);
CREATE TABLE t2_5 PARTITION OF t2 FOR VALUES FROM (4000000) TO ( 5000000);
CREATE TABLE t2_6 PARTITION OF t2 FOR VALUES FROM (5000000) TO ( 6000000);
CREATE TABLE t2_7 PARTITION OF t2 FOR VALUES FROM (6000000) TO ( 7000000);
CREATE TABLE t2_8 PARTITION OF t2 FOR VALUES FROM (7000000) TO ( 8000000);
CREATE TABLE t2_9 PARTITION OF t2 FOR VALUES FROM (8000000) TO ( 9000000);
CREATE TABLE t2_0 PARTITION OF t2 FOR VALUES FROM (9000000) TO (10000000);
```

Enfin, la table *t3* est une table utilisant l'ancienne méthode de partitionnement :

```
CREATE TABLE t3 (c1 integer, c2 text);
CREATE TABLE t3_1 (CHECK (c1 BETWEEN       0 AND  1000000)) INHERITS (t3);
CREATE TABLE t3_2 (CHECK (c1 BETWEEN 1000000 AND  2000000)) INHERITS (t3);
CREATE TABLE t3_3 (CHECK (c1 BETWEEN 2000000 AND  3000000)) INHERITS (t3);
CREATE TABLE t3_4 (CHECK (c1 BETWEEN 3000000 AND  4000000)) INHERITS (t3);
CREATE TABLE t3_5 (CHECK (c1 BETWEEN 4000000 AND  5000000)) INHERITS (t3);
CREATE TABLE t3_6 (CHECK (c1 BETWEEN 5000000 AND  6000000)) INHERITS (t3);
CREATE TABLE t3_7 (CHECK (c1 BETWEEN 6000000 AND  7000000)) INHERITS (t3);
CREATE TABLE t3_8 (CHECK (c1 BETWEEN 7000000 AND  8000000)) INHERITS (t3);
CREATE TABLE t3_9 (CHECK (c1 BETWEEN 8000000 AND  9000000)) INHERITS (t3);
CREATE TABLE t3_0 (CHECK (c1 BETWEEN 9000000 AND 10000000)) INHERITS (t3);

CREATE OR REPLACE FUNCTION insert_into() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.c1    BETWEEN       0 AND  1000000 THEN
    INSERT INTO t3_1 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 1000000 AND  2000000 THEN
    INSERT INTO t3_2 VALUES (NEW.*);
```

```
  ELSIF NEW.c1 BETWEEN 2000000 AND  3000000 THEN
    INSERT INTO t3_3 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 3000000 AND  4000000 THEN
    INSERT INTO t3_4 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 4000000 AND  5000000 THEN
    INSERT INTO t3_5 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 5000000 AND  6000000 THEN
    INSERT INTO t3_6 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 6000000 AND  7000000 THEN
    INSERT INTO t3_7 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 7000000 AND  8000000 THEN
    INSERT INTO t3_8 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 8000000 AND  9000000 THEN
    INSERT INTO t3_9 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 9000000 AND 10000000 THEN
    INSERT INTO t3_0 VALUES (NEW.*);
  END IF;
  RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_insert_t3 BEFORE INSERT ON t3 FOR EACH ROW EXECUTE PROCEDURE insert_into();
```

6.5.5 LIMITATIONS

- La table mère ne peut pas avoir de données
- La table mère ne peut pas avoir d'index
  - ni PK, ni UK, ni FK pointant vers elle
- Pas de colonnes additionnelles dans les partitions
- L'héritage multiple n'est pas permis
- Valeurs nulles acceptées dans les partitions uniquement si la table partitionnée le permet
- Partitions distantes pour l'instant pas supportées
- En cas d'attachement d'une partition
  - vérification du respect de la contrainte (Seq Scan de la table)
  - sauf si ajout au préalable d'une contrainte *CHECK* identique

Toute donnée doit pouvoir être placée dans une partition. Dans le cas contraire, la donnée ne sera pas placée dans la table mère (contrairement au partitionnement traditionnel). À la place, une erreur sera générée :

```
ERROR:  no partition of relation "t2" found for row
```

De même, il n'est pas possible d'ajouter un index à la table mère, sous peine de voir l'erreur suivante apparaître :

```
ERROR:  cannot create index on partitioned table "t1"
```

Ceci sous-entend qu'il n'est toujours pas possible de mettre une clé primaire, et une contrainte unique sur ce type de table. De ce fait, il n'est pas non plus possible de faire pointer une clé étrangère vers ce type de table.

6.6 TABLESPACES

Un espace de stockage :

- Un répertoire du système d'exploitation **hors de PGDATA**
- Soit pour répartition d'entrées/sorties
- Soit pour quota (taille du système de fichiers)

```
CREATE TABLESPACE tbs1 LOCATION '/fs1/';
ALTER TABLE ma_table SET TABLESPACE tbs1;
```

- `seq_page_cost` et `random_page_cost` par tablespace
- `temp_tablespaces`

Un tablespace est uniquement un endroit du système de fichiers où PostgreSQL pourra écrire des fichiers de données, impérativement hors du PGDATA. De ce fait, à partir de la version 9.5, PostgreSQL renvoie un avertissement si ce conseil n'est pas suivi :

```
postgres=# CREATE TABLESPACE ts1 LOCATION '/var/lib/postgresql/10/data/ts1';
WARNING:  tablespace location should not be inside the data directory
CREATE TABLESPACE
```

Il y a deux cas d'utilisation distincts des tablespaces dans PostgreSQL :

- La répartition des entrées-sorties. Ils sont de moins en moins fréquemment utilisés à cet effet :
  - On ne dispose souvent maintenant que d'un seul groupe RAID sur les serveurs, regroupant tous les disques
  - Ces disques sont peut-être sur un SAN, sur lequel on n'a donc ni contrôle ni visibilité
  - Le système peut être virtualisé, ce qui rajoute un niveau d'indirection vers le stockage
  - Toutefois, utilisé pour des tablespaces de performances différentes : SSD/SAS/SATA. Dans ce cas, on peut préciser des valeurs de `seq_page_cost`

et `random_page_cost` différents suivant le tablespace dans lequel se trouve l'objet. Par exemple, on abaissera `random_page_cost` sur un SSD, on l'augmentera sur un SATA.

  - On peut aussi vouloir utiliser un ou plusieurs groupes de disques spéciaux pour les tris, notamment dans un contexte d'infocentre : pour maximiser les performances, il peut être intéressant d'avoir des disques différents pour réaliser les tris, puisqu'ils ont des patterns d'accès différents. Dans ce cas, on créera un ou plusieurs tablespaces pour le tri, qui seront renseignés dans le paramètre `temp_tablespaces`. Si plusieurs sont paramétrés, ils seront utilisés de façon aléatoire à chaque création d'objet temporaire, afin de répartir la charge.

- Les quotas : PostgreSQL ne dispose pas d'un système de quotas, comme on peut en trouver dans d'autres bases de données. On ne peut donc pas limiter la taille d'un espace de stockage de façon logique dans PostgreSQL. Les tablespaces peuvent permettre de contourner cette limitation : un tablespace ne pourra pas grandir au delà de ce que permet le système de fichiers qui l'héberge. Une fois cette limite atteinte, une transaction voulant étendre un fichier sera annulée avec l'erreur `cannot extend file`.

6.7 TOAST

TOAST : The Oversized-Attribute Storage Technique

- Un enregistrement ne peut pas dépasser la taille d'un bloc
- « Contournement » : champs trop grands dans table de débordement TOAST
- Éventuellement compressé : PLAIN/MAIN/EXTERNAL/EXTENDED
- Jusqu'à 1 Go par attribut
- Transparent, seulement visible par `pg_class`

Le mécanisme TOAST est déclenché, par défaut, quand la taille d'un enregistrement dépasse 2 ko. Les enregistrements « toastables », c'est-à-dire ceux de taille variable, et déclarés comme EXTERNAL ou EXTENDED sont stockés dans le TOAST, jusqu'à ce que la taille de l'enregistrement revienne à 2 ko, ou que leur sortie n'apporte plus aucun gain.

Ce mécanisme a plusieurs avantages :

- Les « gros champs » ont moins de chance d'être accédés systématiquement dans le code applicatif
- Les « gros champs » peuvent être compressés de façon transparente
- 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 toast est juste « cloné » dans la

nouvelle version de l'enregistrement.

Les différentes politiques de stockage (clause `STORAGE` d'une colonne) sont :

- PLAIN : stocké uniquement dans la table, non compressé
- MAIN : stocké uniquement dans la table, éventuellement compressé
- EXTERNAL : stocké éventuellement dans la table TOAST, non compressé
- EXTENDED : stocké éventuellement dans la table TOAST, éventuellement compressé

Un enregistrement PLAIN ou MAIN peut tout de même être stocké dans la table TOAST, si l'enregistrement dépasse 8 ko, sinon mieux vaut « toaster » que d'empêcher l'insertion.

Un enregistrement compressible ne le sera que si le résultat compressé est plus petit que le résultat non compressé.

Chaque table est associée à une table TOAST à partir du moment où le mécanisme TOAST a eu besoin de se déclencher. Les enregistrements sont découpés en morceaux d'un peu moins de 2 ko, afin de pouvoir en placer jusqu'à 4 par page dans la table TOAST.

Pour l'utilisateur, les tables TOAST sont totalement transparentes. Elles existent dans un espace de nommage séparé nommé `pg_toast`. Leur maintenance (`autovacuum` notamment) s'effectue en même temps que la table principale. On peut constater leur présence dans `pg_class`, par exemple ainsi :

```
SELECT * FROM pg_class c
WHERE c.relname = 'longs_textes'
OR c.oid=(SELECT reltoastrelid FROM pg_class
          WHERE relname='longs_textes');

-[ RECORD 1 ]-------+----------------
relname             | longs_textes
relnamespace        | 2200
reltype             | 417234
reloftype           | 0
relowner            | 10
relam               | 0
relfilenode         | 419822
reltablespace       | 0
relpages            | 35
reltuples           | 2421
relallvisible       | 0
reltoastrelid       | 417235
...
-[ RECORD 2 ]-------+----------------
relname             | pg_toast_417232
relnamespace        | 99
```

```
reltype             | 417236
reloftype           | 0
relowner            | 10
relam               | 0
relfilenode         | 419823
reltablespace       | 0
relpages            | 0
reltuples           | 0
relallvisible       | 0
reltoastrelid       | 0
...
```

On constate que le nom de la table TOAST est lié au `relfilenode` de la table d'origine, et est donc susceptible de changer (lors d'un `VACUUM FULL` par exemple).

La volumétrie peut se calculer grâce à cette requête issue du wiki[28] :

```
SELECT *, pg_size_pretty(total_bytes) AS total
    , pg_size_pretty(index_bytes) AS INDEX
    , pg_size_pretty(toast_bytes) AS toast
    , pg_size_pretty(table_bytes) AS TABLE
  FROM (
  SELECT *, total_bytes-index_bytes-COALESCE(toast_bytes,0) AS table_bytes FROM (
      SELECT c.oid,nspname AS table_schema, relname AS TABLE_NAME
              , c.reltuples AS row_estimate
              , pg_total_relation_size(c.oid) AS total_bytes
              , pg_indexes_size(c.oid) AS index_bytes
              , pg_total_relation_size(reltoastrelid) AS toast_bytes
          FROM pg_class c
          LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
          WHERE relkind = 'r'
  ) a
) a
WHERE table_name = 'longs_textes';
```

La table TOAST reste forcément dans le même tablespace que la table principale.

6.8 OBJETS BINAIRES

Deux méthodes pour stocker des objets binaires :

- bytea : une colonne comme une autre de la table
  - Maxi 1 Go (à éviter)
  - Accès aléatoire à un morceau lent

[28] https://wiki.postgresql.org/wiki/Disk_Usage

- Large Object
  - Se manipule plutôt comme un fichier
  - Accès avec des primitives de type fichier
  - Maxi 4To (à éviter aussi…)
  - Objet séparé

On dispose de deux méthodes différentes pour gérer les données binaires :

- bytea : c'est simplement un type comme un autre.
- Large Object : ce sont des objets séparés, qu'il faut donc gérer séparément des tables

---

6.8.1 BYTEA

Type natif :

- Se manipule exactement comme les autres :
  - bytea : bytea array, tableau d'octets
  - Représentation textuelle de son contenu
  - Deux formats : `hex` et `escape` (`bytea_output`)
- Si le champ est gros, sa récupération l'alloue intégralement en mémoire
- Toute modification d'un morceau du bytea entraîne la réécriture complète du bytea
- Intéressant pour les petits volumes, jusqu'à quelques méga-octets

Voici un exemple :

```
=# CREATE TABLE demo_bytea(a bytea);
CREATE TABLE
=# INSERT INTO demo_bytea VALUES ('bonjour'::bytea);
INSERT 0 1
=# SELECT * FROM demo_bytea ;
        a
------------------
 \x626f6e6a6f7572
 (1 ligne)
```

Nous avons inséré la chaîne de caractère « bonjour » dans le champ bytea. Le bytea contenant des données binaires, nous avons en fait stocké la représentation binaire de la chaîne « bonjour », dans l'encodage courant. Si nous interrogeons la table, nous voyons la représentation textuelle `hex` du champ bytea : elle commence par `\x` pour expliquer que cette chaîne est un encodage `hex`, suivie de valeurs hexadécimales. Chaque paire de valeur hexadécimale représente donc un octet.

Un second format est disponible : `escape`. Voici comment il se présente :

```
=# SET bytea_output = escape;
=# SELECT * FROM demo_bytea ;
    a
---------
 bonjour
(1 ligne)

=# INSERT INTO demo_bytea VALUES ('journée'::bytea);
INSERT 0 1
=# SELECT * FROM demo_bytea ;
       a
----------------
 bonjour
 journ\303\251e
(2 lignes)
```

Le format `escape` ne protège que les valeurs qui ne sont pas représentables en ASCII 7 bits. Il peut être plus approprié par exemple si on cherche à manipuler une donnée textuelle sans se soucier de son encodage : le plus gros des données n'aura pas besoin d'être protégé, le format texte sera donc plus compact. Toutefois, le format `hex` est bien plus efficace à convertir, ce qui en fait le choix par défaut.

Le format `hex` est apparu avec PostgreSQL 9.0. Il est donc primordial que les librairies clientes (Java par exemple) soient d'une version au moins équivalente, afin qu'elles sachent interpréter les deux formats. Sans cela, il faut forcer `bytea_output` à `escape`, sous peine d'avoir des corruptions des champs bytea

---

6.8.2 LARGE OBJECT

Large Object :

- Totalement indépendant de la table
- Identifié par un OID (identifiant numérique unique)
- On stocke habituellement cet OID dans la table « principale »
- Suppression manuelle, par trigger, ou par batch (extensions)
- lo_create(), lo_import(), lo_seek(), lo_open(), lo_read(), lo_write()…

Un Large Object est donc un objet totalement décorrelé des tables. Le code doit donc gérer cet objet séparément :

- Créer le large object et stocker ce qu'on souhaite dedans
- Stocker la référence à ce large object dans une table (avec le type lob)

- Interroger le large object séparément de la table
- Supprimer le large object quand il n'est plus référencé

Le Large Object nécessite donc un plus gros investissement au niveau du code.

En contrepartie, il a les avantages suivant :

- Il n'est pas limité à 1 Go
- Il est très rapide de récupérer un offset dans le Large Object, par exemple les octets de 152000 à 153020, sans récupérer tout le Large Object
- On peut modifier un morceau de Large Object comme on modifierait un morceau de fichier : il est possible de modifier une séquence d'octets à l'intérieur du Large Object. Seules les pages modifiées seront réécrites. Ce cas d'utilisation reste rare
- On peut récupérer un Large Object par parties. C'est très intéressant par exemple si le contenu des Large Objects doit être transmis à un client par un serveur d'application : on peut transférer le contenu du Large Object au fur et à mesure, au lieu de devoir tout allouer puis tout envoyer comme on le ferait avec un bytea. Le driver JDBC de PostgreSQL fournit une méthode à cet usage, par exemple, dans sa classe LargeObject

Le point essentiel est que les Large Objects **doivent** être supprimés : ce n'est pas automatique, contrairement à un bytea.

Plusieurs méthodes sont possibles :

- Utiliser la fonction trigger fournie par l'extension contrib `lo` : `lo_manage`. Elle permet de supprimer automatiquement un Large Object quand l'enregistrement associé ne le référence plus. Ceci n'est évidemment possible que si les Large Objects ne sont référencés qu'une fois en base (ce qui est habituellement le cas)
- Utiliser le programme `vacuumlo` (un contrib) : celui-ci fait la liste de toutes les références à tous les Large Objects dans une base (en listant tous les OID stockés dans tous les champs de type `oid` ou `lo`), puis supprime tous les Large Objects qui ne sont pas dans cette liste. Ce traitement est bien sûr un peu lourd, et devrait être lancé en traitement de nuit, quotidien ou hebdomadaire
- Supprimer les Large Objects avec un appel à `lo_unlink` dans le code client. Il y a évidemment le risque d'oublier un cas dans le code.

6.9 UNLOGGED TABLES

Unlogged Tables :

- Les données d'une table ne nécessitent pas toujours la durabilité

  - Tables temporaires « partagées » entre plusieurs sessions
  - Intégration de données
  - Cache de données générées
  - Données « matérialisées »
- Non journalisée, donc non répliquée et remise à zéro en cas de crash

Une Unlogged Table se crée exactement comme une table classique, excepté qu'on rajoute le mot `UNLOGGED` dans la création :

```
CREATE UNLOGGED TABLE ma_table (col1 int…)
```

Cela a pour effet de désactiver la journalisation. Comme la journalisation est responsable de la durabilité, une table non journalisée n'a pas cette garantie. **Un crash système, un arrêt d'urgence entraînent la corruption de cette table.** Pour éviter ce problème, la table est remise à zéro en cas de crash.

Le fait qu'elle ne soit pas journalisée fait aussi que les données la concernant ne sont pas répliquées, puisque la réplication native de PostgreSQL utilise les journaux de transactions.

Une fois ces limitation acceptées, l'avantage de ces tables est d'être en moyenne 5 fois plus rapides à la mise à jour. Elles sont donc à réserver à des cas d'utilisation très particuliers, comme par exemple :

- Tables de spooling/staging
- Tables de cache/session applicative

6.10 UNLOGGED TABLES, SUITE

- Depuis la 9.5 on peut passer d'une table journalisée à non journalisée et vice-versa
  - `ALTER TABLE SET LOGGED`
- Attention à la production de fichiers WAL lors du passage de `UNLOGGED` à `LOGGED`.

6.11 RECHERCHE PLEIN TEXTE

Full Text Search/Recherche Plein Texte

- Recherche « à la Google » :
- On n'indexe plus une chaîne de caractère mais
  - Les mots (« lexèmes ») qui la composent

    - On peut rechercher sur chaque lexème indépendamment
- Les lexèmes sont soumis à des règles spécifiques à chaque langue
- Dictionnaires filtrants (`unaccent`)
- S'appuie sur GIN ou GiST

L'indexation FTS est un des cas les plus fréquemment d'utilisation non-relationnelle d'une base de données : les utilisateurs ont souvent besoin de pouvoir rechercher une information qu'ils ne connaissent pas parfaitement, d'une façon floue :

- Recherche d'un produit/article par rapport à sa description
- Recherche dans le contenu de livres/documents

PostgreSQL doit donc permettre de rechercher de façon efficace dans un champ texte. L'avantage de cette solution est d'être intégrée au SGBD. Le moteur de recherche est donc toujours parfaitement à jour avec le contenu de la base, puisqu'il est intégré avec le reste des transactions.

Voici un exemple succinct de mise en place de FTS :

- Création d'une configuration de dictionnaire dédiée (français+anglais sans accent)

```
CREATE TEXT SEARCH CONFIGURATION depeches (COPY= french);
CREATE EXTENSION unaccent ;
ALTER TEXT SEARCH CONFIGURATION depeches ALTER MAPPING FOR
hword, hword_part, word WITH unaccent,french_stem,english_stem;
```

- Ajout d'une colonne vectorisée à la table de depeches, afin de maximiser les performances de recherche

```
ALTER TABLE depeche ADD vect_depeche tsvector;
```

- Création du contenu de vecteur pour les données de la table depeche

```
UPDATE depeche set vect_depeche =
(setweight(to_tsvector('depeches',coalesce(titre,'')), 'A') ||
setweight(to_tsvector('depeches',coalesce(texte,'')), 'C'));
```

- Création de la fonction qui sera associée au trigger

```
CREATE FUNCTION to_vectdepeche( )
 RETURNS trigger
 LANGUAGE plpgsql
 -- common options:  IMMUTABLE  STABLE  STRICT  SECURITY DEFINER
AS $function$
BEGIN
  NEW.vect_depeche :=
    setweight(to_tsvector('depeches',coalesce(NEW.titre,'')), 'A') ||
    setweight(to_tsvector('depeches',coalesce(NEW.texte,'')), 'C');
  return NEW;
```

```
END
$function$;
```

Le rôle de cette fonction est d'automatiquement mettre à jour le champ vect_depeche par rapport à ce qui aura été modifié dans l'enregistrement. On donne aussi des poids différents aux zones titre et texte du document, pour qu'on puisse éventuellement utiliser cette information pour trier les enregistrements par pertinence lors des interrogations.

- Creation du trigger

```
CREATE TRIGGER trg_depeche before INSERT OR update ON depeche
FOR EACH ROW execute procedure to_vectdepeche();
```

Et ce trigger appelle la fonction définie précédemment à chaque insertion ou modification d'enregistrement dans la table.

- Création de l'index associé au vecteur

```
 CREATE INDEX idx_gin_texte ON depeche USING gin(vect_depeche);
```

L'index permet bien sûr une recherche plus rapide.

- Collecte des stats sur la table

```
ANALYZE depeche ;
```

- Utilisation :

```
SELECT titre,texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle');
SELECT titre,texte FROM depeche WHERE vect_depeche @@
to_tsquery('depeches','varicelle & médecin');
```

La recherche plein texte PostgreSQL consiste en la mise en relation entre un vecteur (la représentation normalisée du texte à indexer) et d'une tsquery, c'est à dire une chaîne représentant la recherche à effectuer. Ici par exemple, la première requête recherche tous les articles mentionnant « varicelle », la seconde tous ceux parlant de « varicelle » et de « médecin ». Nous obtiendrons bien sûr aussi les articles parlant de médecine, médecine ayant le même radical que médecin et étant donc automatiquement classé comme faisant partie de la même famille.

La recherche propose bien sûr d'autres opérateurs que `&` : `|` pour ou, `!` pour non. On peut effectuer des recherches de radicaux, etc... L'ensemble des opérations possibles est détaillée ici[29] .

On peut trier par pertinence :

```
SELECT titre,texte
FROM depeche
```

[29] http://docs.postgresql.fr/current/textsearch-controls.html

```
WHERE vect_depeche @@ to_tsquery('depeches','varicelle & médecin')
ORDER BY ts_rank_cd(vect_depeche, to_tsquery('depeches','varicelle & médecin'));
```

ou écrit autrement (pour éviter d'écrire deux fois `to_tsquery`) :

```
SELECT titre,ts_rank_cd(vect_depeche,query) AS rank
FROM depeche, to_tsquery('depeches','varicelle & médecin') query
WHERE query@@vect_depeche
ORDER BY rank DESC ;
```

6.12 COLLATION PAR COLONNE

Collation par colonne :

- L'ordre alphabétique pas forcément le même pour toute une base
  - Champs multi-lingues par exemple
- Possibilité de préciser la collation
  - Par colonne
  - Par index
  - Par requête
  - CREATE TABLE messages (id int,fr TEXT COLLATE "fr_FR.utf8", de TEXT COLLATE "de_DE.utf8");

L'ordre de collation peut varier suivant le contenu d'une colonne. L'ordre de tri français n'est évidemment pas le même que celui du japonais ou du chinois, mais il diffère aussi des autres langues européennes par exemple.

Dans l'exemple suivant, il peut être nécessaire de générer la locale « de_DE.UTF8 » au niveau du système d'exploitation.

```
=# CREATE COLLATION "de_DE.utf8" (LOCALE = "de_DE.utf8");
=# SELECT * from (values ('élève'),('élevé'),('élever'),('Élève')) AS tmp
ORDER BY column1 COLLATE "de_DE.utf8";
  column1
 ---------
  élevé
  élève
  Élève
  élever

=# SELECT * FROM (VALUES ('élève'),('élevé'),('élever'),('Élève')) AS tmp
ORDER BY column1 COLLATE "fr_FR.utf8";
  column1
 ---------
```

```
élève
Élève
élevé
élever
```

L'ordre des caractères accentué est le même entre l'allemand et le français. La règle de tri des mots accentués par contre est différente :

- En allemand, on trie les mots en faisant abstraction des accents. Puis, s'il y a des ex-aequo, on fait une seconde comparaison entre les mots ex-aequo, en prenant en compte les accents.
- En français, on trie les mots en faisant abstraction des accents. Puis, s'il y a des ex-aequo, on fait une seconde comparaison entre les mots ex-aequo, en prenant en compte les accents, mais en comparant les mots **de droite à gauche**.

PostgreSQL permet donc de définir la collation :

- par base
- par colonne
- par index
- par requête

Nous venons de voir un exemple de syntaxe de collation par requête et par table. On peut aussi utiliser la même syntaxe pour un index :

```
CREATE INDEX idx_french_ctype ON french_messages2(message COLLATE "fr_FR.utf8");
```

La collation effective est celle précisée dans l'ordre SQL, celui de la colonne dans la table sinon, et celui de la base en dernier recours.

La collation effective d'un index est celle précisée dans l'index, celui de la colonne de la table qu'il index sinon, et celui de la base en dernier recours.

Un index ne pourra être utilisé pour un tri que si la collation effective dans l'ordre est la même que celle de l'index.

6.13 SERIALIZABLE SNAPSHOT ISOLATION

SSI : Serializable Snapshot Isolation

- Chaque transaction est seule sur la base
- Si on ne peut maintenir l'illusion
  - Une des transactions en cours est annulée
- Sans blocage

- On doit être capable de rejouer la transaction
- Toutes les transactions impliquées doivent être `serializable`
- `default_transaction_isolation=serializable` dans la configuration

PostgreSQL fournit un mode d'isolation appelé `SERIALIZABLE`. Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base. Dès que cette garantie ne peut plus être apportée, une des transactions est annulée.

Toute transaction non déclarée comme `SERIALIZABLE` peut en théorie s'exécuter n'importe quand, ce qui rend inutile le mode `SERIALIZABLE` sur les autres. C'est donc un mode qui doit être mis en place globalement.

Voici un exemple :

Dans cet exemple, il y a des enregistrements avec une colonne couleur contenant « blanc » ou « noir ». Deux utilisateurs essayent simultanément de convertir tous les enregistrements vers une couleur unique, mais chacun dans une direction opposée. Un veut passer tous les blancs en noir, et l'autre tous les noirs en blanc.

L'exemple peut être mis en place avec ces ordres :

```
create table points
  (
    id int not null primary key,
    couleur text not null
  );
insert into points
  with x(id) as (select generate_series(1,10))
  select id, case when id % 2 = 1 then 'noir'
    else 'blanc' end from x;
```

Session 1 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'noir'
where couleur = 'blanc';
```

Session 2 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'blanc'
where couleur = 'noir';
```

À ce moment, une des deux transaction est condamnée à mourir.

Session 2 :

```
commit;
```

17.12

Le premier à valider gagne.

```
select * from points order by id;

 id | couleur
----+--------
  1 | blanc
  2 | blanc
  3 | blanc
  4 | blanc
  5 | blanc
  6 | blanc
  7 | blanc
  8 | blanc
  9 | blanc
 10 | blanc
(10 rows)
```

Session 1 : Celle-ci s'est exécutée comme si elle était seule.

```
commit;

ERROR:  could not serialize access
        due to read/write dependencies
        among transactions
DETAIL:  Cancelled on identification
         as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Une erreur de sérialisation. On annule et on réessaye.

```
rollback;
begin;
update points set couleur = 'noir'
  where couleur = 'blanc';
commit;
```

Il n'y a pas de transaction concurrente pour gêner.

```
SELECT * FROM points ORDER BY id;

 id | couleur
----+--------
  1 | noir
  2 | noir
  3 | noir
  4 | noir
  5 | noir
```

```
  6 | noir
  7 | noir
  8 | noir
  9 | noir
 10 | noir
(10 rows)
```

La transaction s'est exécutée seule, après l'autre.

Le mode `SERIALIZABLE` permet de s'affranchir des `SELECT FOR UPDATE` qu'on écrit habituellement, dans les applications en mode `READ COMMITTED`. Toutefois, il fait bien plus que ça, puisqu'il réalise du verrouillage de prédicats. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation. Il permet aussi de gérer les problèmes ci-dessus avec plus de deux sessions.

Pour des exemples plus complets, le mieux est de consulter la documentation officielle[30]
.

6.14 CONCLUSION

- Aucune information n'est cachée (*Open-Source*)
- Nombreuses tables et vues systèmes pour trouver l'information sur l'état du serveur
- PostgreSQL embarque des fonctionnalités performantes, complexes et parfois uniques dans le monde des bases de données Open Source

Cette évolution des fonctionnalités et performances justifie et encourage les mises à jours des clusters existant vers des versions récentes de PostgreSQL.

6.14.1 QUESTIONS

N'hésitez pas, c'est le moment !

6.15 TRAVAUX PRATIQUES

Ces TP traitent l'ensemble des domaines abordés dans ce module. Commencez par ceux qui vous intéressent le plus !

[30] https://wiki.postgresql.org/wiki/SSI/fr

17.12

Vous aurez besoin de la base `cave` et de la base `textes`. Le formateur doit vous les fournir.

6.15.1 ENONCÉS

Tables système - Liste des bases

a. Écrire une requête SQL qui affiche le nom et l'identifiant de toutes les bases de données.

b. Comment faire pour ne pas voir apparaître les bases du systèmes (`postgres`, `template_`) ?

c. Afficher maintenant le nom du propriétaire et l'encodage de toute les bases.

d. Comparer la requête avec celle qui est exécutée lorsque l'on tape la commande `\l` dans la console.

Tables système - Numbackends

Trouvez la relation entre la table `pg_stat_activity` et le champs `numbackends` de la table `pg_stat_database`.

Que vérifie-t-on?

Tables système - Locks

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 `locks` en cours sur la base :

- le numéro du processus
- le nom de l'utilisateur PostgreSQL
- l'âge de la transaction
- le nom de la table lockée
- le mode de lock
- les acquisitions

Afin de créer un lock de manière artificielle, utiliser dans une autre transaction :

```
CREATE TABLE foobar (a integer);
BEGIN;
LOCK TABLE foobar;
```

Puis laisser cette transaction ouverte.

Index

Nous allons commencer avec la base `cave`.

Indexation simple

Sélectionnez le nombre de bouteilles en stock de l'année 1994. Quel est son plan ?

Rajoutez un index pour que la requête soit plus rapide.

Vérifiez l'amélioration.

Indexation multi-colonnes

Créez l'index optimum pour cette requête :

```
SELECT * FROM stock WHERE vin_id=12 AND annee BETWEEN 1992 AND 1995;
```

Vérifiez cela en constatant que moins de buffers sont accédés avec votre nouveau plan.

La répartition entre *hit* et *read* pourra varier suivant l'état de votre cache.

Le second plan ne consomme qu'un seul bloc de moins ici. Il est plus efficace, mais c'est marginal, parce qu'il y a peu de `contenant_id` différents. L'index sur (`vin_id,contenant_id,annee`) est largement suffisant.

Vous noterez dans chaque cas que la dernière colonne de l'index est l'année. C'est important, car c'est la seule qui soit parcourue par inégalité. Le fait qu'elle soit la dernière de l'index permet de parcourir les feuilles sans retraverser l'arbre du BTree : toutes les feuilles sont chaînées entre elles. Leur parcours dans l'ordre retourne donc des valeurs consécutives triées, dans l'ordre de l'index.

On peut obtenir des performances encore plus fortes par un *Index Only Scan*, mais cela impose la maintenance d'un index très spécialisé. Proposez cet index, et réalisez un test avec lui.

Cet index contient dans ses premières colonnes les colonnes concernées par la clause `WHERE`, puis les colonnes supplémentaires auxquelles on accède dans la requête (ici toutes). Le plan est donc le suivant (exécutez `VACUUM stock` si vous n'obtenez pas ceci, c'est nécessaire après l'import initial pour les plans en *Index Only Scan*).

Le plan accède 4 fois moins de blocs. Par ailleurs, ils sont consécutifs dans l'index. Nous ne pourrons pas aller plus vite qu'avec cet index, mais il aura un coût important à la mise à jour.

Indexation de pattern

Pour ces exercices, il faut une base contenant une quantité de données importante. Utilisez donc la base `textes`. Il en existe deux variantes : `textes_10pct` ou `textes`.

17.12

N'utilisez la seconde que si votre machine est performante : il s'agit de l'intégralité du contenu francophone du projet Gutenberg, soit environ 12 millions de lignes de texte.

- Créez un index simple sur la colonne `contenu` de la table.
- Recherchez un enregistrement commençant par « comme disent », et vérifiez son plan.

Cet index ne fonctionne pas.

Comme expliqué dans le cours, le problème est que l'index sur contenu utilise la collation par défaut de la base. L'index est donc capable de retourner des chaînes de caractères déjà triées dans la collation de la base, mais pas de réaliser des recherches sur le début de la chaîne.

Il faut donc utiliser un autre opérateur de comparaisons entre chaînes. PostgreSQL fournit pour ça deux classes d'opérateurs: `varchar_pattern_ops` pour varchar, `text_pattern_ops` pour text.

- Créez un index utilisant la classe text_pattern_ops, et refaites le test.

On constate que comme l'ordre choisi est l'ordre ASCII, l'optimiseur sait qu'après « comme disent », c'est « comme disenu » qui apparaît dans l'index.

- Indexez pour pouvoir positionner le % au début de la chaîne plutôt qu'à la fin. Trouvez les lignes finissant par « Et vivre ».

Ceci n'est possible qu'en utilisant un index sur fonction.

Toutefois, ces méthodes ne permettent de filtrer qu'au début ou à la fin de la chaîne, et ne permettent qu'une recherche sensible ou insensible à la casse, mais pas les deux simultanément.

- Créez un index spécialisé de recherche dans les chaînes, à base de trigramme, puis recherchez toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis insensible.

Pour cela, installez l'extension `pg_trgm`, et créez un index spécialisé (GIN ici, puisque nous allons réaliser des recherches exactes).

Puis créez votre index GIN sur la colonne contenu.

Et recherchez les lignes contenant « Valjean », de façon sensible à la casse, puis insensible à la casse.

On constate que l'index est bien plus volumineux que le Btree précédent (environ 3 fois), et que la recherche est plus lente. La contrepartie est évidemment que les trigrammes

sont infiniment plus souples. On constate aussi que le LIKE a dû supprimer 4 enregistrements après le scan de l'index, ce qui est normal, puisque l'index trigramme est insensible à la casse. Il ramène donc trop d'enregistrements.

- **Si vous avez des connaissances sur les expression régulières**, nous pouvons utiliser aussi ces trigrammes pour des recherches plus avancées. Les opérateurs sont :

operateur	fonction
~	match sensible à la casse
~*	match insensible à la casse
!~	non-match sensible à la casse
!~*	non-match insensible à la casse

Recherchez toutes les lignes contenant « Fantine » OU « Valjean ».

Recherchez toutes les lignes mentionnant « Fantine » ET « Valjean ». Une formulation d'expression régulière simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

Indexation Full Text

Nous allons réaliser un autre mécanisme d'indexation pour ce texte. Cette fois-ci, il s'agit d'indexer non plus les trigrammes, mais les lexèmes (grosso modo, les radicaux des mots). Nous ne pourrons donc plus rechercher avec des LIKE ou des expressions régulières, mais seulement rechercher la présence ou nom de mots d'une famille dans le texte. En contrepartie, cette méthode utilise des index bien plus compacts, et est plus rapide.

Créons donc un index Full Text, avec un dictionnaire français, de notre table `textes`. Nous choisirons la méthode d'indexation GIN, la plus performante.

- Créez un index GIN sur le vecteur du champ `contenu` (fonction `to_tsvector`)
- Quelle est la taille de cet index ?
- Quelle performance pour trouver fantine dans la table ?

On constate donc que le Full Text Search est bien plus efficace que le trigramme, du moins pour le Full Text Search + GIN : trouver 1 mot parmi plus de cent millions, avec 300 enregistrements correspondant dure 1,5 ms. Par contre, le trigramme permet des recherches floues (orthographe approximative), et des recherches sur autre chose que des mots.

Partitionnement

Nous travaillons sur la base `cave`. Nous allons partitionner la table `stock` sur l'année.

17.12

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions) :

```
INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

Renommez `stock` en `stock_old`, puis créez une table `stock` vide. N'y mettez pas d'index, vous les créerez directement sur les partitions.

Créez les tables filles de `stock` avec la contrainte d'année, soit `stock_2001` à `stock_2005`.

Rédigez un trigger d'insertion sur `stock` (inspirez-vous de celui du cours).

Insérez tous les enregistrements venant de l'ancienne table `stock`. Cela vous permettra de valider le bon fonctionnement de ce trigger.

Passez les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

Vérifiez la présence d'enregistrements dans `stock_2001`. Vérifiez qu'il n'y en a aucun dans `stock` (syntaxe `SELECT ONLY`).

Maintenant, vérifiez qu'une requête sur `stock` qui précise l'année ne parcourt que les bonnes partitions.

Vous pouvez bien sûr remettre des index. Remettez ceux qui étaient en place dans la table `stock` originale (attention à ne pas indexer `annee`, cela ne sert à rien). Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

Les autres index ne servent à rien sur les partitions: `idx_stock_annee` est évidemment inutile, mais `idx_stock_vin_annee` aussi, puisqu'il est inclus dans l'index `stock_pkey`.

Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

Essayez de changer l'année d'un enregistrement de `stock` (la même que la précédente). Cela échoue.

Écrivez le trigger manquant permettant de déplacer l'enregistrement dans la bonne partition.

Retentez l'UPDATE. Vous avez une violation de contrainte unique, qui est une erreur normale : nous avons déjà un enregistrement de stock pour ce vin pour l'année 2004.

Tentez un DELETE.

Tout fonctionne normalement.

Large Objets

Créer une table pour stocker les informations :

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

Importer un fichier local à l'aide de psql dans un large object.

Noter le l'oid retourné.

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

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

Les sauvegarder dans des fichiers locaux :

Tables non journalisées

Restaurer le contenu de la table `stock` dans deux tables : `stock_logged` et `stock_unlogged`, l'une normale, l'autre unlogged. Surveillez la génération de WAL pour chaque opération.

6.15.2 SOLUTIONS

Tables système

Tables système - Liste des bases

a. La liste des bases de données se trouve dans le table `pg_database` :

```
SELECT db.oid, db.datname
FROM pg_database db ;
```

b. Avec l'instruction `NOT IN` on n'affiche que les bases utilisateur.

```
SELECT db.oid, db.datname
FROM pg_database db
WHERE db.datname NOT IN ('template0','template1','postgres');
```

c. On effectue une jointure avec la table `pg_roles` pour déterminer le propriétaire des bases :

```
SELECT DISTINCT db.datname, r.rolname, db.encoding
FROM pg_database db, pg_roles r
WHERE db.datdba = r.oid ;
```

d. Sortir de la console et la lancer de nouveau psql avec l'option -E :

```
$ psql -E
```

Taper la commande `\l`, la requête exécutée est affichée juste avant le résultat :

```
postgres=# \l
******** REQUETE *********
SELECT d.datname as "Nom",
     r.rolname as "Propriétaire",
     pg_catalog.pg_encoding_to_char(d.encoding) as "Encodage"
FROM pg_catalog.pg_database d
  JOIN pg_catalog.pg_roles r ON d.datdba = r.oid
ORDER BY 1;
**************************
```

Tables système - Numbackends

La requête sur `pg_stat_database` :

```
\x
Affichage étendu activé.

SELECT * FROM pg_stat_database WHERE numbackends > 0;

-[ RECORD 1 ]-+---------
datid         | 17443
datname       | docspgfr
numbackends   | 1
xact_commit   | 159564
xact_rollback | 64
blks_read     | 33452
blks_hit      | 29962750
tup_returned  | 92103870
tup_fetched   | 16491647
tup_inserted  | 9758
tup_updated   | 14
tup_deleted   | 110
-[ RECORD 2 ]-+---------
datid         | 27323
datname       | postfix
numbackends   | 1
xact_commit   | 217125
xact_rollback | 745
blks_read     | 27708
blks_hit      | 15801041
tup_returned  | 30060198
tup_fetched   | 4772744
tup_inserted  | 1932604
```

```
tup_updated    | 1126
tup_deleted    | 2468
```

La relation à trouver avec `pg_stat_activity` est que le nombre de backends se vérifie en comptant le nombre de connexions grâce à `pg_stat_activity` :

```
SELECT datname, count(*)
FROM pg_stat_activity
GROUP BY datname
HAVING count(*)>0;
```

```
 datname | count
---------+-------
 popopop |    10
 pgfr    |     5
(2 lignes)
```

Tables système - Locks

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;
```

Index

Nous allons commencer avec la base `cave`.

Indexation simple

Sélectionnez le nombre de bouteilles en stock de l'année 1994. Quel est son plan ?

```
=# EXPLAIN ANALYZE SELECT count(*) FROM stock WHERE annee=1994;
                                QUERY PLAN
```

```
------------------------------------------------------------------------
 Aggregate  (cost=16296.27..16296.28 rows=1 width=0)
            (actual time=257.555..257.556 rows=1 loops=1)
   ->  Seq Scan on stock  (cost=0.00..16285.53 rows=4298 width=0)
                          (actual time=228.758..255.792 rows=16839 loops=1)
         Filter: (annee = 1994)
         Rows Removed by Filter: 842723
 Planning time: 0.226 ms
 Execution time: 257.643 ms
```

Rajoutez un index pour que la requête soit plus rapide.

```
CREATE INDEX idx_stock_annee ON stock(annee);
```

Vérifiez l'amélioration

```
=# EXPLAIN ANALYZE SELECT count(*) FROM stock WHERE annee=1994;
                              QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=5588.14..5588.15 rows=1 width=0)
            (actual time=21.522..21.522 rows=1 loops=1)
   ->  Bitmap Heap Scan on stock (cost=81.73..5577.39 rows=4298 width=0)
                                 (actual time=10.932..17.579 rows=16839 loops=1)
         Recheck Cond: (annee = 1994)
         Heap Blocks: exact=110
         ->  Bitmap Index Scan on idx_stock_annee
                    (cost=0.00..80.66 rows=4298 width=0)
                    (actual time=10.845..10.845 rows=16839 loops=1)
               Index Cond: (annee = 1994)
 Planning time: 0.625 ms
 Execution time: 21.650 ms
(8 lignes)
```

Indexation multi-colonnes

Créez l'index optimum pour cette requête :

```
SELECT * FROM stock WHERE vin_id=12 AND annee BETWEEN 1992 AND 1995;
```

Vérifiez cela en constatant que moins de buffers sont accédés avec votre nouveau plan.

```
explain (analyze,buffers) SELECT * FROM stock WHERE vin_id=12
AND annee BETWEEN 1992 AND 1995;
                                    QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on stock  (cost=158.16..238.64 rows=21 width=16)
                            (actual time=0.139..0.190 rows=12 loops=1)
   Recheck Cond: ((vin_id = 12) AND (annee >= 1992) AND (annee <= 1995))
   Heap Blocks: exact=12
   Buffers: shared hit=16
```

```
   ->  Bitmap Index Scan on stock_pkey
               (cost=0.00..158.15 rows=21 width=0)
               (actual time=0.113..0.113 rows=12 loops=1)
         Index Cond: ((vin_id = 12) AND (annee >= 1992) AND (annee <= 1995))
         Buffers: shared hit=4
 Planning time: 0.319 ms
 Execution time: 0.280 ms
(9 lignes)
```

Cet index est un choix raisonnable :

```
CREATE INDEX idx_stock_vin_annee ON stock(vin_id,annee);
```

```
cave=# explain (analyze,buffers) SELECT * FROM stock
WHERE vin_id=12 AND annee BETWEEN 1992 AND 1995;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on stock  (cost=4.69..85.18 rows=21 width=16)
                            (actual time=0.137..0.190 rows=12 loops=1)
   Recheck Cond: ((vin_id = 12) AND (annee >= 1992) AND (annee <= 1995))
   Heap Blocks: exact=12
   Buffers: shared hit=12 read=3
   ->  Bitmap Index Scan on idx_stock_vin_annee
               (cost=0.00..4.69 rows=21 width=0)
               (actual time=0.114..0.114 rows=12 loops=1)
         Index Cond: ((vin_id = 12) AND (annee >= 1992) AND (annee <= 1995))
         Buffers: shared read=3
 Planning time: 0.698 ms
 Execution time: 0.276 ms
(9 lignes)
```

La répartition entre *hit* et *read* pourra varier suivant l'état de votre cache.

Ce second plan ne consomme qu'un seul bloc de moins ici. Il est plus efficace, mais c'est marginal, parce qu'il y a peu de `contenant_id` différents. L'index sur (`vin_id,contenant_id,annee`) est largement suffisant.

Vous noterez dans chaque cas que la dernière colonne de l'index est l'année. C'est important, car c'est la seule qui soit parcourue par inégalité. Le fait qu'elle soit la dernière de l'index permet de parcourir les feuilles sans retraverser l'arbre du BTree : toutes les feuilles sont chaînées entre elles. Leur parcours dans l'ordre retourne donc des valeurs consécutives triées, dans l'ordre de l'index.

On peut obtenir des performances encore plus fortes par un *Index Only Scan*, mais cela impose la maintenance d'un index très spécialisé:

```
CREATE INDEX idx_stock_vin_annee_contenant_id_nombre
ON stock(vin_id,annee,contenant_id,nombre);
```

Cet index contient dans ses premières colonnes les colonnes concernées par la clause WHERE, puis les colonnes supplémentaires auxquelles on accède dans la requête (ici toutes). Le plan est donc le suivant (exécutez `VACUUM stock` si vous n'obtenez pas ceci, c'est nécessaire après l'import initial pour les plans en Index Only Scan).

```
=# explain (analyze,buffers) SELECT * FROM stock
WHERE vin_id=12 AND annee BETWEEN 1992 AND 1995;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using idx_stock_vin_annee_contenant_id_nombre on stock
        (cost=0.42..4.90 rows=21 width=16)
        (actual time=0.044..0.054 rows=12 loops=1)
   Index Cond: ((vin_id = 12) AND (annee >= 1992) AND (annee <= 1995))
   Heap Fetches: 0
   Buffers: shared hit=4
 Planning time: 0.415 ms
 Execution time: 0.113 ms
```

Le plan accède 4 fois moins de blocs. Par ailleurs, ils sont consécutifs dans l'index. Nous ne pourrons pas aller plus vite qu'avec cet index, mais il aura un coût important à la mise à jour.

Indexation de pattern

Pour ces exercices, il faut une base contenant une quantité de données importante. Utilisez donc la base `textes`. Il en existe deux variantes : `textes_10pct` ou `textes`. N'utilisez la seconde que si votre machine est performante : il s'agit de l'intégralité du contenu francophone du projet Gutenberg, soit environ 12 millions de lignes de texte.

- Créez un index simple sur la colonne «contenu» de la table.

```
CREATE INDEX idx_textes_contenu ON textes(contenu);
```

- Recherchez un enregistrement commençant par «comme disent», et vérifiez son plan.

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on textes  (cost=0.00..366565.81 rows=998 width=98)
                     (actual time=0.618..3307.544 rows=28 loops=1)
   Filter: (contenu ~~ 'comme disent%'::text)
   Rows Removed by Filter: 12734159
 Planning time: 1.653 ms
 Execution time: 3307.617 ms
(5 lignes)
```

Cet index ne fonctionne pas.

Comme expliqué dans le cours, le problème est que l'index sur contenu utilise la collation par défaut de la base. L'index est donc capable de retourner des chaînes de caractères déjà triées dans la collation de la base, mais pas de réaliser des recherches sur le début de la chaîne.

Il faut donc utiliser un autre opérateur de comparaisons entre chaînes. PostgreSQL fournit pour ça deux classes d'opérateurs : `varchar_pattern_ops` pour varchar, `text_pattern_ops` pour text.

- Créez un index utilisant la classe text_pattern_ops, et refaites le test.

```
DROP INDEX idx_textes_contenu;
CREATE INDEX idx_textes_contenu ON textes(contenu text_pattern_ops);
 EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE 'comme disent%';
                     QUERY PLAN
-----------------------------------------------------
 Index Scan using idx_textes_contenu on textes
        (cost=0.56..8.58 rows=998 width=98)
        (actual time=0.048..0.162 rows=28 loops=1)
   Index Cond: ((contenu ~>=~ 'comme disent'::text)
                AND (contenu ~<~ 'comme disenu'::text))
   Filter: (contenu ~~ 'comme disent%'::text)
 Planning time: 0.747 ms
 Execution time: 0.215 ms
(5 lignes)
```

On constate que comme l'ordre choisi est l'ordre ASCII, l'optimiseur sait qu'après 'comme disent', c'est 'comme disenu' qui apparaît dans l'index.

- Indexez pour pouvoir positionner le % au début de la chaîne plutôt qu'à la fin. Trouvez les lignes finissant par 'Et vivre'.

Ceci n'est possible qu'en utilisant un index sur fonction :

```
CREATE INDEX idx_textes_revcontenu ON textes(reverse(contenu)
text_pattern_ops);
```

Il faut ensuite utiliser ce reverse systématiquement dans les requêtes :

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE reverse(contenu) LIKE reverse('%Et vivre');
                            QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=3005.19..139322.70 rows=63671 width=98)
                             (actual time=0.093..0.103 rows=2 loops=1)
   Filter: (reverse(contenu) ~~ 'erviv tE%'::text)
   Heap Blocks: exact=2
   ->  Bitmap Index Scan on idx_textes_revcontenu
            (cost=0.00..2989.28 rows=63671 width=0)
            (actual time=0.068..0.068 rows=2 loops=1)
```

```
        Index Cond: ((reverse(contenu) ~>=~ 'erviv tE'::text)
                     AND (reverse(contenu) ~<~ 'erviv tF'::text))
 Planning time: 0.345 ms
 Execution time: 0.168 ms
```

On constate que le résultat du reverse a été directement utilisé par l'optimiseur. La requête est donc très rapide. On peut utiliser une méthode similaire pour la recherche insensible à la casse, en utiliser `lower()` ou `upper()`.

Toutefois, ces méthodes ne permettent de filtrer qu'au début ou à la fin de la chaîne, et ne permettent qu'une recherche sensible ou insensible à la casse, mais pas les deux simultanément.

- Créez un index spécialisé de recherche dans les chaînes, à base de trigramme, puis recherchez toutes les lignes de texte contenant « Valjean » de façon sensible à la casse, puis insensible.

Pour cela, installez l'extension `pg_trgm`, et créez un index spécialisé (GIN ici, puisque nous allons réaliser des recherches exactes).

```
CREATE EXTENSION pg_trgm;
```

Puis créez votre index GIN sur la colonne contenu.

```
CREATE INDEX idx_textes_trgm ON textes USING gin (contenu gin_trgm_ops);
```

Et recherchez les lignes contenant « Valjean ».

De façon sensible à la casse :

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu LIKE '%Valjean%';
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=91.78..3899.69 rows=1003 width=98)
                             (actual time=22.917..32.125 rows=2085 loops=1)
   Recheck Cond: (contenu ~~ '%Valjean%'::text)
   Rows Removed by Index Recheck: 4
   Heap Blocks: exact=567
   ->  Bitmap Index Scan on idx_textes_trgm
               (cost=0.00..91.53 rows=1003 width=0=)
               (actual time=22.531..22.531 rows=2089 loops=1)
         Index Cond: (contenu ~~ '%Valjean%'::text)
 Planning time: 2.580 ms
 Execution time: 32.477 ms
(8 lignes)
```

Puis insensible à la casse :

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ILIKE '%Valjean%';
                                QUERY PLAN
```

```
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=91.78..3899.69 rows=1003 width=98)
                             (actual time=25.491..42.163 rows=2089 loops=1)
   Recheck Cond: (contenu ~~* '%Valjean%'::text)
   Heap Blocks: exact=567
   ->  Bitmap Index Scan on idx_textes_trgm
               (cost=0.00..91.53 rows=1003 width=0)
               (actual time=25.077..25.077 rows=2089 loops=1)
         Index Cond: (contenu ~~* '%Valjean%'::text)
 Planning time: 2.159 ms
 Execution time: 42.660 ms
(7 lignes)
```

On constate que l'index est bien plus volumineux que le Btree précédent (environ 3 fois), et que la recherche est plus lente. La contrepartie est évidemment que les trigrammes sont infiniment plus souples. On constate aussi que le LIKE a dû supprimer 4 enregistrements après le scan de l'index, ce qui est normal, puisque l'index trigramme est insensible à la casse. Il ramène donc trop d'enregistrements.

- **Si vous avez des connaissances sur les expression régulières**, nous pouvons utiliser aussi ces trigrammes pour des recherches plus avancées. Les opérateurs sont :

operateur	fonction
~	match sensible à la casse
~*	match insensible à la casse
!~	non-match sensible à la casse
!~*	non-match insensible à la casse

Recherchez toutes les lignes contenant « Fantine » OU « Valjean ».

```
EXPLAIN ANALYZE SELECT * FROM textes WHERE contenu ~ 'Valjean|Fantine';
                             QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=175.78..3983.69 rows=1003 width=98)
                             (actual time=480.948..498.510 rows=2312 loops=1)
   Recheck Cond: (contenu ~ 'Valjean|Fantine'::text)
   Rows Removed by Index Recheck: 741
   Heap Blocks: exact=1363
   ->  Bitmap Index Scan on idx_textes_trgm
           (cost=0.00..175.53 rows=1003 width=0)
           (actual time=480.528..480.528 rows=3053 loops=1)
         Index Cond: (contenu ~ 'Valjean|Fantine'::text)
 Planning time: 2.437 ms
 Execution time: 498.696 ms
```

```
(8 lignes)
```

Recherchez toutes les lignes mentionnant « Fantine » ET « Valjean ». Une formulation d'expression régulière simple est « Fantine puis Valjean » ou « Valjean puis Fantine ».

```
EXPLAIN ANALYZE SELECT * FROM textes
WHERE contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)';
                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=175.78..3983.69 rows=1003 width=98)
                             (actual time=379.030..379.087 rows=5 loops=1)
   Recheck Cond: (contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)'::text)
   Heap Blocks: exact=4
   ->  Bitmap Index Scan on idx_textes_trgm
           (cost=0.00..175.53 rows=1003 width=0)
           (actual time=378.994..378.994 rows=5 loops=1)
         Index Cond: (contenu ~ '(Valjean.*Fantine)|(Fantine.*Valjean)'::text)
 Planning time: 4.409 ms
 Execution time: 379.149 ms
(7 lignes)
```

Indexation Full Text

Nous allons réaliser un autre mécanisme d'indexation pour ce texte. Cette fois-ci, il s'agit d'indexer non plus les trigrammes, mais les lexèmes (grosso modo, les radicaux des mots). Nous ne pourrons donc plus rechercher avec des LIKE ou des expressions régulières, mais seulement rechercher la présence ou nom de mots d'une famille dans le texte. En contrepartie, cette méthode utilise des index bien plus compacts, et est plus rapide.

Créons donc un index Full Text, avec un dictionnaire français, de notre table textes. Nous choisirons la méthode d'indexation GIN, la plus performante.

- Créez un index GIN sur le vecteur du champ `contenu` (fonction to_tsvector)

```
textes=# create index idx_fts ON textes
USING gin (to_tsvector('french',contenu));
CREATE INDEX
```

- Quelle est la taille de cet index ?

```
textes=# select pg_size_pretty(pg_relation_size('idx_fts'));
 pg_size_pretty
----------------
 593 MB
(1 ligne)
```

- Quelle performance pour trouver « fantine » dans la table ?

```
textes=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('french','fantine');
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on textes  (cost=693.10..135759.45 rows=63109 width=97)
                             (actual time=0.278..0.699 rows=311 loops=1)
   Recheck Cond: (to_tsvector('french'::regconfig, contenu) @@
                 '''fantin'''::tsquery)
   Buffers: shared hit=153
   ->  Bitmap Index Scan on idx_fts  (cost=0.00..677.32 rows=63109 width=0)
                                     (actual time=0.222..0.222 rows=311 loops=1)
         Index Cond: (to_tsvector('french'::regconfig, contenu) @@
                      '''fantin'''::tsquery)
         Buffers: shared hit=4
 Total runtime: 0.793 ms
(7 lignes)

Temps : 1,534 ms
```

On constate donc que le Full Text Search est bien plus efficace que le trigramme, du moins pour le Full Text Search + GIN : trouver 1 mot parmi plus de cent millions, avec 300 enregistrements correspondant dure 1,5 ms. Par contre, le trigramme permet des recherches floues (orthographe approximative), et des recherches sur autre chose que des mots.

Partitionnement

Nous travaillons sur la base cave. Nous allons partitionner la table stock, sur l'année.

Pour se simplifier la vie, nous allons limiter le nombre d'années dans stock (cela nous évitera la création de 50 partitions).

```
INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

Renommez `stock` en `stock_old`, puis créez une table `stock` vide. N'y mettez pas d'index, vous les créerez directement sur les partitions.

```
ALTER TABLE stock RENAME TO stock_old;
CREATE TABLE stock (LIKE stock_old);
```

Créez les tables filles de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

```
CREATE TABLE stock_2001 (
  CHECK (annee = 2001)
) INHERITS (stock);
```

et suivants…

17.12

Rédigez un trigger d'insertion sur `stock` (inspirez-vous de celui du cours).

```
CREATE OR REPLACE FUNCTION ins_stock() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.annee = 2001 THEN
    INSERT INTO stock_2001 VALUES (NEW.*);
  ELSIF NEW.annee = 2002 THEN
    INSERT INTO stock_2002 VALUES (NEW.*);
  ELSIF NEW.annee = 2003 THEN
    INSERT INTO stock_2003 VALUES (NEW.*);
  ELSIF NEW.annee = 2004 THEN
    INSERT INTO stock_2004 VALUES (NEW.*);
  ELSIF NEW.annee = 2005 THEN
    INSERT INTO stock_2005 VALUES (NEW.*);
  ELSE
    RAISE EXCEPTION 'Partition inconnue pour l''annee %',NEW.annee;
  END IF;
  RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_ins_stock
   BEFORE INSERT ON stock
   FOR EACH ROW
   EXECUTE PROCEDURE ins_stock();
```

Insérez tous les enregistrements venant de l'ancienne table `stock`. Cela vous permettra de valider le bon fonctionnement de ce trigger.

```
INSERT INTO stock SELECT * FROM stock_old;
```

Passez les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

```
ANALYZE;
```

Vérifiez la présence d'enregistrements dans `stock_2001`. Vérifiez qu'il n'y en a aucun dans `stock`.

```
SELECT count(*) FROM stock_2001;
SELECT count(*) FROM ONLY stock;
```

Maintenant, vérifiez qu'une requête sur `stock` qui précise l'année ne parcourt que les bonnes partitions.

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE annee=2002;
```

Vous pouvez bien sûr remettre des index. Remettez ceux qui étaient en place dans la table `stock` originale (attention à ne pas indexer annee, cela ne sert à rien). Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

```
CREATE UNIQUE INDEX stock_pkey_2001 ON stock_2001 (vin_id,contenant_id);
CREATE UNIQUE INDEX stock_pkey_2002 ON stock_2002 (vin_id,contenant_id);
CREATE UNIQUE INDEX stock_pkey_2003 ON stock_2003 (vin_id,contenant_id);
CREATE UNIQUE INDEX stock_pkey_2004 ON stock_2004 (vin_id,contenant_id);
CREATE UNIQUE INDEX stock_pkey_2005 ON stock_2005 (vin_id,contenant_id);
```

Les autres index ne servent à rien sur les partitions : `idx_stock_annee` est évidemment inutile, mais `idx_stock_vin_annee` aussi, puisqu'il est inclus dans l'index `stock_pkey`.

Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE vin_id=1725 AND annee=2003;
```

Essayez de changer l'année d'un enregistrement de stock (la même que la précédente). Cela échoue.

```
UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725;
ERROR:  new row for relation "stock_2003" violates check constraint
        "stock_2003_annee_check"
```

sql: code too wide

Écrivez le trigger manquant permettant de déplacer l'enregistrement dans la bonne partition.

```
CREATE OR REPLACE FUNCTION f_upd_stock() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
  EXECUTE 'DELETE FROM ' || TG_TABLE_SCHEMA || '.' || TG_TABLE_NAME ||
          ' WHERE annee=' || OLD.ANNEE;
          -- Plus performant de faire un trigger dédié par table
  INSERT INTO stock VALUES (NEW.*);
  RETURN NULL;
END;
$$;

CREATE TRIGGER tr_upd_stock_2001
  BEFORE UPDATE ON stock_2001
  FOR EACH ROW
  WHEN (NEW.annee != OLD.annee)
  EXECUTE PROCEDURE f_upd_stock();
CREATE TRIGGER tr_upd_stock_2002
  BEFORE UPDATE ON stock_2002
```

```
  FOR EACH ROW
  WHEN (NEW.annee != OLD.annee)
  EXECUTE PROCEDURE f_upd_stock();
CREATE TRIGGER tr_upd_stock_2003
  BEFORE UPDATE ON stock_2003
  FOR EACH ROW
  WHEN (NEW.annee != OLD.annee)
  EXECUTE PROCEDURE f_upd_stock();
CREATE TRIGGER tr_upd_stock_2004
  BEFORE UPDATE ON stock_2004
  FOR EACH ROW
  WHEN (NEW.annee != OLD.annee)
  EXECUTE PROCEDURE f_upd_stock();
CREATE TRIGGER tr_upd_stock_2005
  BEFORE UPDATE ON stock_2005
  FOR EACH ROW
  WHEN (NEW.annee != OLD.annee)
  EXECUTE PROCEDURE f_upd_stock();
```

Retentez l'UPDATE. Vous avez une violation de contrainte unique, qui est une erreur normale: nous avons déjà un enregistrement de `stock` pour ce vin pour l'année 2004.

Tentez un DELETE.

```
DELETE FROM stock WHERE annee=2003 and vin_id=1725;
```

Tout fonctionne normalement.

Large Objets

Créer une table pour stocker les informations :

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

Importer un fichier local à l'aide de psql dans un large object :

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

Noter le l'oid retourné.

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

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

```
  psql -c "INSERT INTO fichiers SELECT 'postgresql.conf', \
          lo_import('/etc/postgresql/9.2/main/postgresql.conf');"
```

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

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

Les sauvegarder dans des fichiers locaux :

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

Tables non journalisées

Restaurer les deux tables et comparer les temps de chargement ainsi que la génération de fichiers WAL :

Regarder les fichiers dans le répertoire `$PGDATA/pg_wal` :

```
psql cave -c "CREATE TABLE stock_logged AS SELECT * FROM stock;"
```

Regarder les fichiers dans le répertoire `$PGDATA/pg_wal` :

```
psql cave -c "CREATE UNLOGGED TABLE stock_unlogged AS SELECT * FROM stock;"
```

Regarder les fichiers dans le répertoire `$PGDATA/pg_wal` :

7 POSTGRESQL AVANCÉ 2

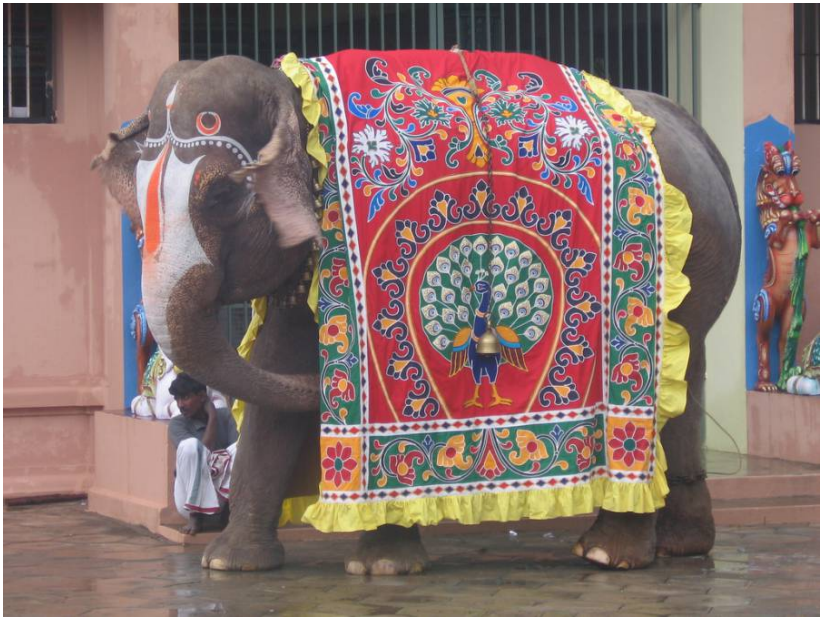


FIGURE 9: POSTGRESQL

7.1 PRÉAMBULE

Ce module présente les extensions de PostgreSQL.

Les extensions permettent de rajouter des types de données, des méthodes d'indexation, des fonctions et opérateurs, des tables, des vues…

Dans le but de rajouter des fonctionnalités.

7.2 CONTRIBS

Ce sont des fonctionnalitées :

- Livrées avec le code source de PostgreSQL

- Habituellement packagées (`postgresql-*-contrib`)
- De qualité garantie parce que maintenues par le projet
- Mais optionnelles et désactivées par défaut
- Ou fonctionnalités en cours de stabilisation
- Documentées dans les annexes de PostgreSQL !
- Chapitre F : « Additional Supplied Modules »

Les contribs sont fournis directement dans l'arborescence de PostgreSQL. Ils suivent donc strictement le rythme de révision de PostgreSQL, leur compatibilité est ainsi garantie.

Il s'agit soit de fonctionnalités qui n'intéressent pas tout le monde (`hstore`, `pg_trgm`, `pgstattuple`...), ou de fonctionnalités en cours de stabilisation (historiquement `tsearch2`, `xml2`)

La documentation des contribs est dans le chapitre F des annexes, et est donc fréquemment ratée par les nouveaux utilisateurs.

7.3 EXTENSIONS

Ce sont :

- Des « packages » pour PostgreSQL
- Un ensemble d'objets livrés ensemble
- Connus en tant que tels par le catalogue PostgreSQL
- `CREATE EXTENSION`, `ALTER EXTENSION UPDATE`, `DROP EXTENSION`
- Option `CASCADE` (à partir de 9.6)
- contrib <> extension

Les extensions sont un objet du catalogue, englobant d'autres objets. On peut les comparer à des paquetages Linux par exemple. L'installation de l'extension permet d'installer tous les objets qu'elle fournit, sa désinstallation de tous les supprimer. À partir de la version 9.6 il est possible d'utiliser l'option `CASCADE` pour installer les dépendances.

L'intérêt du mécanisme est aussi la gestion des mises à jour des extensions, et le fait qu'étant des objets de catalogue standard, elles sont exportées et importées par `pg_dump`/`pg_restore`. Une mise à jour de version majeure, par exemple, permettra donc de migrer les extensions par le même coup (changement de prototypes de fonctions, nouvelles vues, etc...)

Une contrib est habituellement sous forme d'extension, sauf quelques exceptions qui ne créent pas de nouveaux objets de catalogue (`auto_explain` par exemple). Une extension

peut provenir d'un projet séparé de PostgreSQL (PostGIS, par exemple, ou le Foreign Data Wrapper Oracle).

7.4 CONNEXIONS DISTANTES

3 approches :

- *Foreign Data Wrapper*
- dblink (historique)
- PL/Proxy (sharding)

Il existe principalement 3 méthodes pour accéder à des données externes à la base sous PostgreSQL.

Les Foreign Data Wrappers (norme SQL/MED) sont la méthode recommandée pour accéder à un objet distant. Elle permet l'accès à de nombreuses sources de données.

Historiquement, on utilisait `dblink`, qui ne fournissait cette fonctionnalité que de PostgreSQL à PostgreSQL, et de façon moins performante.

PL/Proxy est un cas d'utilisation très différent : il s'agit de distribuer des appels de fonctions PL sur plusieurs nœuds.

7.4.1 FOREIGN DATA WRAPPERS

PostgreSQL supporte SQL/MED :

- Management of External Data
- Extension de la norme SQL ISO
- Données externes présentées comme des tables
- En lecture/écriture (si supporté par le driver et à partir de PostgreSQL 9.3)
  - PostgreSQL, Oracle, MySQL (lecture/écriture)
  - fichier CSV, fichier fixe (en lecture)
  - ODBC, JDBC, Multicorn
  - CouchDB, Redis (NoSQL)

SQL/MED est un des tomes de la norme SQL, traitant de l'accès aux données externes (Management of External Data).

Elle fournit donc un certain nombre d'éléments conceptuels, et de syntaxe, permettant la déclaration d'accès à des données externes. Ces données externes sont bien sûr présentées comme des tables.

PostgreSQL suit cette norme et est ainsi capable de requêter des tables distantes à travers des connecteurs FDW (*Foreign Data Wrapper*). Les seuls connecteurs livrés par défaut sont `file_fdw` (fichier plat) et `postgres_fdw` (à partir de la 9.3) qui permet de se connecter à un autre serveur PostgreSQL.

Les deux *wrappers* les plus aboutis sont sans conteste ceux pour PostgreSQL (c'est un contrib) et Oracle (qui supporte jusqu'aux objets géométriques). Ces deux drivers supportent les écritures sur la base distante.

De nombreux drivers spécialisés existent, entre autres pour accéder à des bases NoSQL comme CouchDB ou Redis.

Il existe aussi des drivers génériques :

- ODBC : utilisation de driver ODBC
- JDBC : utilisation de driver JDBC
- Multicorn : accès aux données au travers d'une API Python, permettant donc d'écrire facilement un accès pour un nouveau type de service

La liste complète des Foreign Data Wrapper disponibles pour PostgreSQL peut être consultée à cette adresse[31] .

7.4.2 SQL/MED : UTILISATION

Installer un driver (foreign data wrapper) :

```
CREATE EXTENSION file_fdw;
```

Créer un « serveur » (ici pas d'options, vu que c'est un driver fichier) :

```
CREATE SERVER file FOREIGN DATA WRAPPER file_fdw ;
```

Créer une « foreign table »

```
CREATE FOREIGN TABLE statistical_data (f1 numeric, f2 numeric)
   SERVER file OPTIONS (filename '/tmp/statistical_data.csv',
                        format 'csv', delimiter ';') ;
```

Quel que soit le driver, la création d'un accès se fait en 3 étapes minimum :

[31] http://pgxn.org/tag/foreign%20data%20wrapper/

- Installation du driver : aucun foreign data wrapper n'est présent par défaut. Il se peut que vous ayez d'abord à l'installer sur le serveur
- Création du `SERVER` : permet de spécifier un certain nombre d'informations génériques à un serveur distant, qu'on n'aura pas à repréciser pour chaque objet de ce serveur
- Création de la `FOREIGN TABLE` : l'objet qu'on souhaite rendre visible

Éventuellement, on peut vouloir créer un `USER MAPPING`, mettant en correspondance les utilisateurs locaux et distants. Il s'agit habituellement de renseigner les paramètres de connexion d'un utilisateur (ou groupe d'utilisateurs) local à une base de donnée distante : login, mot de passe, etc...

L'étape de création des `FOREIGN TABLE` peut être remplacé par l'ordre `IMPORT FOREIGN SCHEMA`. Disponible à partir de la version 9.5, il permet l'import d'un schéma complet :

```
IMPORT FOREIGN SCHEMA remote_schema FROM SERVER server_name INTO local_schema ;
```

7.4.3 SQL/MED : HÉRITAGE

- La version 9.5 introduit la notion d'héritage
- Une table locale peut hériter d'une table distante et inversement
- Permet le partitionnement sur plusieurs serveurs
- Pour rappel, l'héritage ne permet pas de conserver :
  - les contraintes d'unicité et référentielles ;
  - les index ;
  - les droits.

Exemple d'une table locale qui hérite d'une table distante

La table parent (ici une foreign table) sera la table `fgn_stock_londre` et la table enfant sera la table `local_stock` (locale). Ainsi la lecture de la table `fgn_stock_londre` retournera les enregistrements de la table `fgn_stock_londre` et de la table `local_stock`.

Sur l'instance distante :

Créer une table `stock_londre` sur l'instance distante dans la base nommée « cave » et insérer des valeurs :

```
CREATE TABLE stock_londre (c1 int);
INSERT INTO stock_londre VALUES (1),(2),(4),(5);
```

Sur l'instance locale :

Créer le serveur et la correspondance des droits :

```
CREATE EXTENSION postgres_fdw ;

CREATE SERVER pgdistant
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '192.168.0.42', port '5432', dbname 'cave');

CREATE USER MAPPING FOR mon_utilisateur
SERVER pgdistant
OPTIONS (user 'utilisateur_distant', password 'mdp_utilisateur_distant');
```

Créer une foreign table `fgn_stock_londre` correspondant à la table `stock_londre` de l'autre instance.

```
CREATE FOREIGN TABLE fgn_stock_londre (c1 int) SERVER pgdistant
OPTIONS (schema_name 'public' , table_name 'stock_londre');
```

On peut bien lire les données :

```
SELECT tableoid::regclass,* FROM fgn_stock_londre;
     tableoid     | c1
------------------+----
 fgn_stock_londre |  1
 fgn_stock_londre |  2
 fgn_stock_londre |  4
 fgn_stock_londre |  5
(4 lignes)
```

Voici le plan d'exécution associé :

```
EXPLAIN ANALYZE SELECT * FROM fgn_stock_londre;
                                QUERY PLAN
-------------------------------------------------------------------------
 Foreign Scan on fgn_stock_londre  (cost=100.00..197.75 rows=2925 width=4)
                                   (actual time=0.388..0.389 rows=4 loops=1)
(3 lignes)
```

Créer une table `local_stock` sur l'instance locale qui va hériter de la table mère :

```
CREATE TABLE local_stock () INHERITS (fgn_stock_londre);
```

On insère des valeurs dans la table `local_stock` :

```
INSERT INTO local_stock VALUES  (10),(15);
INSERT 0 2
```

La table `local_stock` ne contient bien que 2 valeurs :

```
SELECT * FROM local_stock ;
 c1
----
 10
```

```
 15
(2 lignes)
```

En revanche, la table `fgn_stock_londre` ne contient plus 4 valeurs mais 6 valeurs :

```
SELECT tableoid::regclass,* FROM fgn_stock_londre;
     tableoid     | c1
------------------+----
 fgn_stock_londre |  1
 fgn_stock_londre |  2
 fgn_stock_londre |  4
 fgn_stock_londre |  5
 local_stock      | 10
 local_stock      | 15
(6 lignes)
```

Dans le plan d'exécution on remarque bien la lecture des deux tables :

```
EXPLAIN ANALYZE SELECT * FROM fgn_stock_londre;
                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=100.00..233.25 rows=5475 width=4)
         (actual time=0.438..0.444 rows=6 loops=1)
   ->  Foreign Scan on fgn_stock_londre
           (cost=100.00..197.75 rows=2925 width=4)
           (actual time=0.438..0.438 rows=4 loops=1)
   ->  Seq Scan on local_stock  (cost=0.00..35.50 rows=2550 width=4)
                                (actual time=0.004..0.005 rows=2 loops=1)
 Planning time: 0.066 ms
 Execution time: 0.821 ms
(5 lignes)
```

Note : Les données de la table `stock_londre` sur l'instance distante n'ont pas été modifiées.

Exemple d'une table distante qui hérite d'une table locale

La table parent sera la table `master_stock` et la table fille (ici FOREIGN TABLE) sera la table `fgn_stock_londre`. Ainsi une lecture de la table `master_ stock` retournera les valeurs de la table `master_stock` et de la table `fgn_ stock_londre`, sachant qu'une lecture de la table `fgn_stock_londre` retourne les valeurs de la table `fgn_stock_londre` et `local_stock`. Une lecture de la table `master_stock` retournera les valeurs des 3 tables : `master_stock`, `fgn_stock_londre`, `local_stock`.

Créer une table `master_stock`, insérer des valeurs dedans :

```
CREATE TABLE master_stock (LIKE fgn_stock_londre);
INSERT INTO master_stock VALUES (100),(200);
```

```
SELECT tableoid::regclass,* FROM master_stock;
   tableoid   | c1
--------------+-----
 master_stock | 100
 master_stock | 200
(2 rows)
```

Modifier la table `fgn_stock_londre` pour qu'elle hérite de la table `master_stock` :

```
ALTER TABLE fgn_stock_londre INHERIT master_stock ;
```

La lecture de la table `master_stock` nous montre bien les valeurs des 3 tables :

```
SELECT tableoid::regclass,* FROM master_stock ;
     tableoid     | c1
------------------+-----
 master_stock     | 100
 master_stock     | 200
 fgn_stock_londre |   1
 fgn_stock_londre |   2
 fgn_stock_londre |   4
 fgn_stock_londre |   5
 local_stock      |  10
 local_stock      |  15
(8 lignes)
```

Le plan d'exécution confirme bien la lecture des 3 tables :

```
EXPLAIN ANALYSE SELECT * FROM master_stock ;
                                QUERY PLAN
--------------------------------------------------------------------------
 Append  (cost=0.00..236.80 rows=5730 width=4)
         (actual time=0.004..0.440 rows=8 loops=1)
   ->  Seq Scan on master_stock  (cost=0.00..3.55 rows=255 width=4)
                                 (actual time=0.003..0.003 rows=2 loops=1)
   ->  Foreign Scan on fgn_stock_londre
           (cost=100.00..197.75 rows=2925 width=4)
           (actual time=0.430..0.430 rows=4 loops=1)
   ->  Seq Scan on local_stock  (cost=0.00..35.50 rows=2550 width=4)
                                (actual time=0.003..0.004 rows=2 loops=1)
 Planning time: 0.073 ms
 Execution time: 0.865 ms
(6 lignes)
```

Dans cet exemple, on a un héritage « imbriqué » :

- La table `master_stock` est parent de la foreign table `fgn_stock_londre`
- La foreign table `fgn_stock_londre` est enfant de la table `master_stock` et parent de la table `local_stock`

- La table `local_stock` est enfant de la foreign table `fgn_stock_londre`

```
master_stock
   fgn_stock_londre => stock_londre
       local_stock
```

Créons un index sur `master_stock` et ajoutons des données dans la table `master_stock` :

```
CREATE INDEX fgn_idx ON master_stock(c1);
INSERT INTO master_stock (SELECT generate_series(1,10000));
```

Maintenant effectuons une simple requête de sélection :

```
SELECT tableoid::regclass,* FROM master_stock WHERE c1=10;
```

```
   tableoid   | c1
--------------+----
 master_stock | 10
 local_stock  | 10
(2 lignes)
```

Étudions le plan d'exécution associé :

```
EXPLAIN ANALYZE SELECT tableoid::regclass,* FROM master_stock WHERE c1=10;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.29..192.44 rows=27 width=8)
         (actual time=0.010..0.485 rows=2 loops=1)
   ->  Append  (cost=0.29..192.44 rows=27 width=8)
               (actual time=0.009..0.483 rows=2 loops=1)
         ->  Index Scan using fgn_idx on master_stock
                     (cost=0.29..8.30 rows=1 width=8)
                     (actual time=0.009..0.010 rows=1 loops=1)
               Index Cond: (c1 = 10)
         ->  Foreign Scan on fgn_stock_londre
                     (cost=100.00..142.26 rows=13 width=8)
                     (actual time=0.466..0.466 rows=0 loops=1)
         ->  Seq Scan on local_stock  (cost=0.00..41.88 rows=13 width=8)
                                      (actual time=0.007..0.007 rows=1 loops=1)
               Filter: (c1 = 10)
               Rows Removed by Filter: 1
```

L'index ne se fait que sur master_stock.

En ajoutant l'option `ONLY` après la clause `FROM`, on demande au moteur de n'afficher que la table `master_stock` et pas les tables filles :

```
SELECT tableoid::regclass,* FROM ONLY master_stock WHERE c1=10;
```

```
   tableoid   | c1
--------------+----
 master_stock | 10
(1 ligne)
```

Attention, si on supprime les données sur la table parent, la suppression se fait aussi sur les tables filles :

```
BEGIN;
DELETE FROM master_stock;
-- [DELETE 10008]
SELECT * FROM master_stock ;

 c1
----
(0 ligne)

ROLLBACK;
```

En revanche avec l'option `ONLY`, on ne supprime que les données de la table parent :

```
BEGIN;
DELETE FROM ONLY master_stock;
-- [DELETE 10002]
ROLLBACK;
```

Enfin, si nous ajoutons une contrainte `CHECK` sur la table étrangère, l'exclusion de partition basées sur ces contraintes s'appliquent naturellement :

```
ALTER TABLE fgn_stock_londre ADD CHECK (c1 < 100);
ALTER TABLE local_stock ADD CHECK (c1 < 100);
    --local_stock hérite de fgn_stock_londre !

EXPLAIN (ANALYZE,verbose) SELECT tableoid::regclass,*
FROM master_stock WHERE c1=200;
                          QUERY PLAN
---------------------------------------------------------------
 Result  (cost=0.29..8.32 rows=2 width=8)
         (actual time=0.009..0.011 rows=2 loops=1)
   Output: (master_stock.tableoid)::regclass, master_stock.c1
   ->  Append  (cost=0.29..8.32 rows=2 width=8)
               (actual time=0.008..0.009 rows=2 loops=1)
         ->  Index Scan using fgn_idx on public.master_stock
                     (cost=0.29..8.32 rows=2 width=8)
                     (actual time=0.008..0.008 rows=2 loops=1)
               Output: master_stock.tableoid, master_stock.c1
               Index Cond: (master_stock.c1 = 200)
 Planning time: 0.157 ms
```

```
 Execution time: 0.025 ms
(8 rows)
```

Attention : La contrainte `CHECK` sur `fgn_stock_londre` est **locale** seulement. Si cette contrainte n'existe pas sur la table distants, le résultat de la requête pourra alors être faux !

Sur le serveur distant :

```
INSERT INTO stock_londre VALUES (200);
```

Sur le serveur local :

```
SELECT tableoid::regclass,* FROM master_stock WHERE c1=200;
   tableoid    | c1
---------------+-----
 master_stock | 200
 master_stock | 200
(2 rows)

ALTER TABLE fgn_stock_londre DROP CONSTRAINT fgn_stock_londre_c1_check;

SELECT tableoid::regclass,* FROM master_stock WHERE c1=200;
     tableoid     | c1
------------------+-----
 master_stock     | 200
 master_stock     | 200
 fgn_stock_londre | 200
```

7.4.4 SQL/MED : POSTGRESQL

- Ajouter le FDW
- Ajouter un serveur
- Ajouter une table distante
- Lire la table distante
- Écrire dans la table distante
- Analyser la table distante
- Plus lent qu'une table locale, surtout pour les patterns d'accès complexes

Nous créons une table sur un serveur distant. Par simplicité, nous utiliserons le même serveur mais une base différente. Créons cette base et cette table :

```
dalibo=# CREATE DATABASE distante;
CREATE DATABASE
```

```
dalibo=# \c distante
You are now connected to database "distante" as user "dalibo".

distante=# CREATE TABLE personnes (id integer, nom text);
CREATE TABLE

distante=# INSERT INTO personnes (id, nom) VALUES (1, 'alice'),
                  (2, 'bertrand'), (3, 'charlotte'), (4, 'david');
INSERT 0 4

distante=# ANALYZE personnes;
ANALYZE
```

Maintenant nous pouvons revenir à notre base d'origine et mettre en place la relation avec le « serveur distant » :

```
distante=# \c dalibo
You are now connected to database "dalibo" as user "dalibo".

dalibo=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION

dalibo=# CREATE SERVER serveur_distant FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (HOST 'localhost',PORT '5432', DBNAME 'distante');
CREATE SERVER

dalibo=# CREATE USER MAPPING FOR dalibo SERVER serveur_distant
OPTIONS (user 'dalibo', password 'mon_mdp');
CREATE USER MAPPING

dalibo=# CREATE FOREIGN TABLE personnes (id integer, nom text)
SERVER serveur_distant;
CREATE FOREIGN TABLE
```

Et c'est tout ! Nous pouvons désormais utiliser la table distante `personnes` comme si elle était une table locale de notre base.

```
dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  1 | alice
  2 | bertrand
  3 | charlotte
  4 | david
(4 rows)

dalibo=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM personnes;
                              QUERY PLAN
```

```
-------------------------------------------------------------------------------
 Foreign Scan on public.personnes  (cost=100.00..150.95 rows=1365 width=36)
                                   (actual time=0.655..0.657 rows=4 loops=1)
   Output: id, nom
   Remote SQL: SELECT id, nom FROM public.personnes
 Total runtime: 1.197 ms
(4 rows)
```

En plus, si nous filtrons notre requête, le filtre est exécuté sur le serveur distant, réduisant considérablement le trafic réseau et le traitement associé.

```
dalibo=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM personnes WHERE id = 3;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Foreign Scan on public.personnes  (cost=100.00..127.20 rows=7 width=36)
                                   (actual time=1.778..1.779 rows=1 loops=1)
   Output: id, nom
   Remote SQL: SELECT id, nom FROM public.personnes WHERE ((id = 3))
 Total runtime: 2.240 ms
(4 rows)
```

À partir de la 9.3, il est possible d'écrire vers ces tables aussi, à condition que le connecteur FDW le permette.

En utilisant l'exemple de la section précédente, on note qu'il y a un aller-retour entre la sélection des lignes à modifier et la modification de ces lignes :

```
dalibo=# EXPLAIN (ANALYZE, VERBOSE) UPDATE personnes
SET nom = 'agathe' WHERE id = 1;
                                      QUERY PLAN
-------------------------------------------------------------------------------
 Update on public.personnes  (cost=100.00..140.35 rows=12 width=10)
                          (actual time=2.086..2.086 rows=0 loops=1)
   Remote SQL: UPDATE public.personnes SET nom = $2 WHERE ctid = $1
   ->  Foreign Scan on public.personnes  (cost=100.00..140.35 rows=12 width=10)
                                      (actual time=1.040..1.042 rows=1 loops=1)
         Output: id, 'agathe'::text, ctid
         Remote SQL: SELECT id, ctid FROM public.personnes WHERE ((id = 1))
                     FOR UPDATE
 Total runtime: 2.660 ms
(6 rows)

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  2 | bertrand
  3 | charlotte
  4 | david
```

```
  1 | agathe
(4 rows)
```

On peut aussi constater que l'écriture distante respecte les transactions :

```
dalibo=# BEGIN;
BEGIN

dalibo=# DELETE FROM personnes WHERE id=2;
DELETE 1

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  3 | charlotte
  4 | david
  1 | agathe
(3 rows)

dalibo=# ROLLBACK;
ROLLBACK

dalibo=# SELECT * FROM personnes;
 id |    nom
----+-----------
  2 | bertrand
  3 | charlotte
  4 | david
  1 | agathe
(4 rows)
```

Attention à ne pas perdre de vue qu'une FOREIGN TABLE n'est pas une table locale. L'accès à ses données est plus lent, surtout quand on souhaite récupérer peu d'enregistrements : on a systématiquement une latence réseau, éventuellement un parsing de la requête envoyée au serveur distant, etc…

Les jointures ne sont pas poussées au serveur distant avant PostgreSQL 9.6 et pour des bases PostgreSQL. Un accès par `Nested Loop` (boucle imbriquée entre les deux tables) est habituellement inenvisageable entre deux FOREIGN TABLES : la boucle interne (celle qui en local serait un accès à une table par index) entraînerait une requête individuelle par itération, ce qui serait horriblement peu performant.

Les FOREIGN TABLES sont donc à réserver à des accès intermittents. Il ne faut pas les utiliser pour développer une application transactionnelle par exemple.

7.4.5 MODULE CONTRIB : DBLINK

- Permet le requêtage inter-bases PostgreSQL
- Simple et bien documenté
- L'auteur a voulu obtenir la même fonctionnalité que celle qui est disponible dans une base commerciale réputée... d'où le nom.
- En lecture seule sauf à écrire des triggers sur vue
- Ne transmet pas les prédicats au serveur distant : tout l'objet est systématiquement récupéré
- Plus d'intérêt depuis que le driver SQL/MED pour PostgreSQL est mature

Documentation officielle[32] .

Le module `dblink` de PostgreSQL n'est pas aussi riche que Microsoft SQL Linked Server ou que `dblink` d'Oracle. On peut notamment regretter l'absence de fonctionnalité d'introspection.

Le seul intérêt de `dblink`, par rapport au *Foreign Data Wrapper* pour PostgreSQL, est la possibilité d'émuler des transactions autonomes ou d'appeler des fonctions, ce qui est impossible avec `postgres_fdw`.

7.4.6 PL/PROXY : PRÉSENTATION

- Une alternative à dblink
- Possibilité de distribuer les requêtes
- Utile pour le « partionnement horizontal »
- Uniquement si votre application n'utilise que des appels de fonction à la base

Une fonction PL/Proxy peut se connecter à plusieurs hôtes distants simultanément !

PostgreSQL propose 3 modes d'exécution des fonctions PL/Proxy :

- ANY : la fonction est exécuté sur un seul noeud au hasard
- ALL : la fonction est exécuté sur tous les noeuds
- EXACT : la fonction est exécutée sur un noeud défini dans le corps de la fonction

On peut mettre en place un ensemble de fonctions PL/Proxy pour « découper » une table volumineuse et le répartir sur plusieurs instances PostgreSQL.

Le PL/Proxy offre alors la possibilité de développer une couche d'astraction transparente pour l'utilisateur final qui peut alors consulter et manipuler les données comme si elles se trouvaient dans une seule table sur une seule instance PostgreSQL.

[32] http://docs.postgresql.fr/current/contrib-dblink-function.html

7.5 HSTORE-JSON-JSONB

Stockage de données non-relationnelles :

- `hstore` : clé-valeur, stockage binaire, fonctions d'extraction, de requêtage, d'indexation avancée
- `json` : stockage texte JSON, validation syntaxique, fonctions d'extraction
- `jsonb` : stockage binaire de JSON, converti pour accès rapide, fonctions d'extraction, de requêtage, d'indexation avancée
- Alternative efficace et performante à Entité/Attribut/Valeur (très lent)

Ces types sont utilisés quand le modèle relationnel n'est pas assez souple. Les contextes où, lors de la conception, il serait nécessaire d'ajouter dynamiquement des colonnes à la table suivant les besoins du client, où le détail des attributs d'une entité ne sont pas connus (modélisation géographique par exemple), etc...

La solution traditionnelle est de créer des tables de ce format:

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

On y stocke dans entite la clé de l'enregistrement de la table principale, dans attribut la colonne supplémentaire, et dans valeur la valeur de cette colonne.

Ce modèle présente l'avantage évident de résoudre le problème. Les défauts sont par contre nombreux:

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

Toute recherche complexe est très inefficace : une recherche multi-critères sur ce schéma va être extrêmement peu performante.

Les types `hstore`, `json` et `jsonb` permettent de résoudre le problème autrement.

7.5.1 HSTORE

Stocker des données non-structurées.

```
CREATE EXTENSION hstore ;
CREATE TABLE demo_hstore(id serial, meta hstore);
INSERT INTO demo_hstore (meta) values ('river=>t');
INSERT INTO demo_hstore (meta) values ('road=>t,secondary=>t');
INSERT INTO demo_hstore (meta) values ('road=>t,primary=>t');
CREATE INDEX idxhstore ON demo_hstore USING gist (meta);
SELECT * FROM demo_hstore WHERE meta@>'river=>t';
 id |     meta
----+--------------
 15 | "river"=>"t"
```

La méthode utilisée habituellement pour pouvoir stocker des données arbitraires supplémentaires dans une table, c'est soit d'utiliser un champ « filler » comme en Cobol (où l'on concatène les données), soit utiliser un méta-modèle (entité-clé-valeur) dans une table dédiée. Ce dernier a de très mauvaises performances, et est très pénible à manipuler en SQL, sans même mentionner l'impossibilité d'avoir des contraintes d'intégrité dans un modèle entité-clé-valeur.

Les hstore sont indexables, peuvent recevoir des contraintes d'intégrité (unicité, non recouvrement…).

Il s'agit d'une extension, fournie en contrib. Elle est donc systématiquement disponible, si vous avez installé une version de PostgreSQL packagée.

Les `hstore` ne permettent par contre qu'un modèle « plat ». Il s'agit d'un pur stockage clé-valeur. Si vous avez besoin de stocker des informations davantage orientées document, vous devrez vous tourner vers le type `json`, qui n'est pas extrêmement performant car une simple représentation textuelle, ou `jsonb`, qui fournit les avantages de `hstore` et de `json`, mais n'est disponible qu'à partir de PostgreSQL 9.4.

7.5.2 JSON

- Ce n'est qu'un type texte
- Vérifie que le texte est au format JSON
- Fournit des fonctions de manipulation JSON
  - Mais ré-analyse du champ pour chaque appel de fonction
  - On peut indexer une propriété (index sur fonction)
  - Mais pas d'index avancé comme pour `hstore`

- => Peu utile (comme XML)

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

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

Le type json dispose de nombreuses fonctions de manipulation et d'extraction à partir de la 9.3 :

```
CREATE TABLE json (id serial, datas json);
INSERT INTO json (datas) VALUES ('
{
  "id": 3,
  "destinataire": {
    "nom": "Dupont",
    "ouvrages": [
      {"categorie": "Conte", "titre": "Le petit prince"},
      {"categorie": "Polar", "titre": "La chambre noire"}
    ]
  }
}
');

SELECT j.datas #> '{destinataire, ouvrages}' FROM json j;
SELECT j.datas -> 'destinataire' -> 'ouvrages'  FROM json j;
SELECT datas #>> '{destinataire, nom}'
FROM json j,
  LATERAL json_array_elements(j.datas #> '{destinataire, ouvrages}') as ouvrages
WHERE ouvrages ->>'titre' = 'Le petit prince';
```

On peut bien sûr créer des index sur certaines propriétés. Par exemple :

```
CREATE INDEX idx_test ON json(json_extract_path_text(datas,'ouvrages')) ;
```

permettra d'accélérer des requêtes utilisant une clause `WHERE` sur `json_extract_path_text(datas,'ouvrages')` uniquement. Ce type de données n'est donc pas très efficace pour une recherche rapide.

7.5.3 JSONB

- Apparu en 9.4

- Stockage de **JSON** en un format **B**inaire
- Possibilités d'indexation similaires à hstore

Apparu en 9.4, le type `jsonb` permet de stocker les données dans un format optimisé. Ainsi, il n'est plus nécessaire de désérialiser l'intégralité du document pour accèder à une propriété. Pour un exemple extrême (document JSON d'une centaine de Mo), voici le résultat :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_json;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test_json  (cost=0.00..26.38 rows=1310 width=32)
                        (actual time=893.454..912.168 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.021 ms
 Execution time: 912.194 ms
(4 lignes)

EXPLAIN (ANALYZE, BUFFERS) SELECT document->'id' FROM test_jsonb;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test_jsonb  (cost=0.00..26.38 rows=1310 width=32)
                         (actual time=77.707..84.148 rows=1 loops=1)
   Buffers: shared hit=170
 Planning time: 0.026 ms
 Execution time: 84.177 ms
(4 lignes)
```

Le principal avantage réside dans la capacité de tirer parti des fonctionnalités avancées de PostgreSQL. En effet, deux classes d'opérateurs sont proposées et mettent à profit le travail d'optimisation réalisé pour les indexes GIN :

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

Ces requêtes supportent notamment l'opérateur « contient » :

```
                              QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test_jsonb  (cost=0.00..29117.00 rows=5294 width=71)
                         (actual time=0.025..422.310 rows=1 loops=1)
   Filter: ((document -> 'id'::text) = '1'::jsonb)
   Rows Removed by Filter: 1000010
 Planning time: 0.126 ms
 Execution time: 422.353 ms
```

Le support des statistiques n'est pas encore optimal, mais devrait être amélioré dans les prochaines versions.

Il n'est en revanche pas possible de faire des recherches sur des opérateurs btree classiques (<, <=, >, >=), ou sur le contenu de tableaux. On est obligé pour cela de revenir au monde relationnel, et l'indexation devra alors utiliser des indexes fonctionnels sur les clés que l'on souhaite indexer. Il est donc préférable d'utiliser les opérateurs spécifiques, comme « contient » (@>).

7.5.4 CONVERSIONS JSONB / RELATIONNELS

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

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

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

```
{
  "id": 3,
  "sous_document": {
    "label": "mon_sous_document",
    "mon_tableau": [
      {"categorie": "categorie"},
      {"categorie": "unique"}
    ]
  }
}
```

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

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

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

7.5.5 JSQUERY

- Extension proposée
- Fournit un « langage de requête », comme tsquery
- Dépôt github[33]

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

La requête précédente peut alors s'écrire :

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

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

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

Il supporte aussi les opérateurs différents de l'égalité :

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

Cette extension est encore jeune, mais extrêmement prometteuse de part la simplification des requêtes qu'elle propose et son excellent support de l'indexation GIN.

---

[33] https://github.com/akorotkov/jsquery

7.6 PG_TRGM

```
CREATE EXTENSION pg_trgm;
SELECT similarity('bonjour','bnojour');
 similarity
------------
   0.333333

CREATE TABLE test_trgm (text_data text);

INSERT INTO test_trgm(text_data)
VALUES ('hello'), ('hello everybody'),
('helo youg man'),('hallo!'),('HELLO !');

CREATE INDEX test_trgm_idx on test_trgm
  using gist (text_data extensions.gist_trgm_ops);
SELECT text_data FROM test_trgm
 WHERE text_data like '%hello%';
```

Cette requête passe par l'index test_trgm_idx, malgré le % initial. On peut utiliser un index GIN aussi (comme pour le *Full Text Search*).

Ce module permet de décomposer en trigramme les chaînes qui lui sont proposées :

```
 SELECT show_trgm('hello');
             show_trgm
 ---------------------------------
  {"  h","  he",ell,hel,llo,"lo "}
```

Une fois les trigrammes indexés, on peut réaliser de la recherche floue, ou utiliser des clauses `LIKE` malgré la présence de jokers (%) n'importe où dans la chaîne. Les indexations simples, de type btree, ne permettent des recherches efficaces que dans un cas particulier : si le seul joker de la chaîne est à la fin de celle ci ('hello%' par exemple).

```
SELECT text_data, text_data <-> 'hello'
 FROM test_trgm
 ORDER BY text_data <-> 'hello'
 LIMIT 4;
```

nous retourne par exemple les deux enregistrements les plus proches de « hello » dans la table `test_trgm`. La recherche, sur une table de 5 millions d'enregistrements, prend 2 s avec PostgreSQL 9.0, et 20 ms avec PostgreSQL 9.1 qui apporte certaines optimisations des index. La recherche des k éléments les plus proches (on parle de recherche k-NN) n'est disponible qu'avec les index GiST. Les index GIN ont l'avantage d'être plus efficaces pour les recherches exhaustives.

7.7 CITEXT

Champ texte insensible à la casse :

- Beaucoup utilisé pour compatibilité avec SQL Server/MySQL
- Les fonctions de comparaison et tri deviennent insensibles à la casse
- Nécessite une conversion de casse à chaque comparaison
- Plus lent que le type texte

```
CREATE EXTENSION citext;
CREATE TABLE ma_table (col_insensible citext);
```

Ce type est très utile, par exemple dans le cas d'un portage d'une application de SQL Server, ou MySQL, vers PostgreSQL : ces deux moteurs sont habituellement paramétrés pour être insensibles à la casse.

Il suffit pour en profiter de créer l'extension `citext`, puis manipuler le type `citext`.

Les limitations sont les suivantes:

- Les performances sont moins bonnes sur les colonnes citext, surtout en l'absence d'index, à cause des conversions de casse
- La maintenance de l'index, s'il y en a un, est plus coûteuse
- On ne peut pas donner de limite de taille comme avec un type varchar. Cette limitation peut être contournée avec une contrainte CHECK, ou un DOMAIN.

7.8 PGCRYPTO

Le module contrib de chiffremment

- Propose de nombreuses fonctions permettant de chiffrer et de déchiffrer des données
- Gros inconvénient : oubliez les index sur les données chiffrées
- N'oubliez pas de chiffrer la connexion (SSL)
- Permet d'avoir une seule méthode de chiffrement pour tout ce qui accède à la base

Voici un exemple de code:

```
CREATE EXTENSION pgcrypto;
UPDATE utilisateurs SET mdp = crypt('mon nouveau mot de passe',gen_salt('md5'));
```

L'appel à `gen_salt` permet de rajouter une partie aléatoire à la chaîne à chiffrer, ce qui évite que la même chaîne chiffrée deux fois retourne le même résultat. Cela limite donc les attaques par dictionnaire.

7.9 POSTGIS

Pas une contrib :

- Un projet totalement indépendant
- Licence GPL (logiciel libre)
- Extension de PostgreSQL aux types géométriques/géographiques
- 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 permet donc 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')
```

Il fournit les fonctions d'indexation qui permettent d'accéder rapidement aux objets géométriques, au moyen d'index GiST. La requête ci-dessous n'a évidemment pas besoin de parcourir tous les restaurants à la recherche de ceux correspondant aux critères de recherche.

7.9.1 POSTGIS (SUITE)

- De nombreuses fonctionnalités avancées :
  - Support des coordonnées géodésiques
  - Projections, reprojections dans systèmes de coordonnées locaux (Lambert93 en France par exemple)
  - 3D, extrusions, routage, rasters
  - Opérateurs d'analyse géométrique : enveloppe convexe, simplification…
  - Intégré aux principaux serveurs de carte, ETL, outils de manipulation
- Utilisé par IGN, BRGM, AirBNB, Mappy, Openstreetmap, Agence de l'eau…

PostGIS est également respectueux des normes : Open Geospatial Consortium's "Simple Features for SQL Specification"

Voir la liste complète des fonctionnalités[34] .

[34] http://postgis.net/features

17.12

C'est donc une extension très avancée de PostgreSQL. Elle est avant tout utilisée par des spécialistes du domaine Géospatial, mais peut être utilisée aussi dans des projets moins complexes.

7.10 CONTRIBS ORIENTÉS DBA

Un certain nombre de contribs donnent accès à des informations ou des fonctions de bas niveau :

- pgstattuple : fragmentation des tables et index
- pg_buffercache : état du cache
- pg_freespacemap : liste des blocs libres
- pg_visibility : état de la *visibility map*
- pageinspect : inspection du contenu d'une page
- pgrowlocks : informations détaillées sur les enregistrements verrouillés
- pg_prewarm : sauvegarde et restauration de l'état du cache de la base

Tous ces modules permettent de manipuler une facette de PostgreSQL à laquelle on n'a normalement pas accès.

7.10.1 PGSTATTUPLE

pgstattuple fournit une mesure (par parcours complet de l'objet) sur:

- pour une table :
  - remplissage des blocs
  - enregistrements morts
  - espace libre
- pour un index :
  - profondeur de l'index
  - remplissage des feuilles
  - fragmentation (feuilles non consécutives)

Par exemple :

```
=# CREATE EXTENSION pgstattuple ;
CREATE EXTENSION
=# SELECT * FROM pgstattuple('dspam_token_data');
-[ RECORD 1]-----------------------
 table_len          | 601743360
```

```
 tuple_count        | 8587417
 tuple_len          | 412196016
 tuple_percent      | 68.5
 dead_tuple_count   | 401098
 dead_tuple_len     | 19252704
 dead_tuple_percent | 3.2
 free_space         | 93370000
 free_percent       | 15.52

=# SELECT * FROM pgstatindex('dspam_token_data_uid_key');
-[ RECORD 1]------------------------
 version            | 2
 tree_level         | 2
 index_size         | 429047808
 root_block_no      | 243
 internal_pages     | 244
 leaf_pages         | 52129
 empty_pages        | 0
 deleted_pages      | 0
 avg_leaf_density   | 51.78
 leaf_fragmentation | 43.87
```

Comme chaque interrogation nécessite une lecture complète de l'objet, ces fonctions ne sont pas à appeler en supervision.

Elles servent de façon ponctuelle pour s'assurer qu'un objet nécessite une réorganisation. Ici, l'index `dspam_token_data_uid_key` pourrait certainement être reconstruit… il deviendrait 40 % plus petit environ (remplissage à 51 % au lieu de 90 %).

`leaf_fragmentation` indique le pourcentage de pages feuilles qui ne sont pas physiquement contiguës sur le disque. Cela peut être important dans le cas d'un index utilisé pour des `Range Scans` (requête avec des inégalités), mais n'a aucune importance ici puisqu'il s'agit d'une clé primaire technique, donc d'un index qui n'est interrogé que pour récupérer des enregistrements de façon unitaire.

7.10.2 PG_BUFFERCACHE

Qu'y-a-t'il dans le cache de PostgreSQL ?

Fournit une vue :

- Pour chaque page (donc pour l'ensemble de l'instance)
  - fichier (donc objet) associé
  - OID base

- fork (0 : table, 1 : FSM, 2 : VM)
- numéro de bloc
- isdirty
- usagecount

Pour chaque entrée (bloc, par défaut de 8 ko) de la structure `Shared Buffers`, cette vue nous fournit donc les informations sur le bloc contenu dans cette entrée : le fichier (donc la table, l'index…), le bloc dans ce fichier, si ce bloc est synchonisé avec le disque (isdirty = false) ou non, et si ce bloc a été fortement utilisé récemment (de 0=pas utilisé à 5=fortement utilisé).

Cela permet donc de déterminer les *hot blocks* de la base, ou d'avoir une idée un peu plus précise de si le cache est correctement dimensionné : si rien n'atteint un usagecount de 5, le cache est manifestement trop petit : il n'est pas capable de détecter les pages devant impérativement rester en cache. Inversement, si vous avez énormément d'entrées à 0 et quelques pages avec des usagecount très élevés, toutes ces pages à 0 sont égales devant le mécanisme d'éviction du cache. Elles sont donc supprimées à peu près de la même façon que du cache du système d'exploitation. Le cache de PostgreSQL dans ce cas fait « double emploi » avec lui, et pourrait être réduit.

Attention toutefois avec les expérimentations sur les caches : il existe des effets de seuils. Un cache trop petit peut de la même façon qu'un cache trop grand avoir une grande fraction d'enregistrements avec un usagecount à 0. Par ailleurs, le cache bouge extrêmement rapidement par rapport à notre capacité d'analyse. Nous ne voyons qu'un instantané, qui peut ne pas refléter toute la réalité.

`isdirty` indique si un buffer est synchronisé avec le disque ou pas. Il est intéressant de vérifier qu'une instance dispose en permanence d'un certain nombre de buffers pour lesquels `isdirty` vaut `false` et pour lesquels `usagecount` vaut `0`. Si ce n'est pas le cas, c'est le signe :

- que `shared_buffers` est probablement trop petit (il n'arrive pas à contenir les modifications) ;
- que le `background_writer` n'est pas assez agressif.

De plus, l'utilisation de cette extension est assez coûteuse car elle a besoin d'acquérir un verrou sur chaque page de cache inspectée. Chaque verrou est acquis pour une durée très courte, mais elle peut néanmoins entraîner une contention.

À titre d'exemple, cette requête affiche les dix plus gros objets de la base en cours en mémoire cache (dont, ici, deux index) :

```
SELECT c.relname,
       c.relkind,
```

```
       count(*) AS buffers,
       pg_size_pretty(count(*)*8192) as taille_mem
FROM   pg_buffercache b
INNER JOIN pg_class c
      ON b.relfilenode = pg_relation_filenode(c.oid)
        AND b.reldatabase IN (0, (SELECT oid FROM pg_database
                                  WHERE datname = current_database()))
GROUP BY c.relname, c.relkind
ORDER BY 3 DESC
LIMIT 5 ;
```

```
            relname             | relkind | buffers | taille_mem
--------------------------------+---------+---------+------------
 test_val_idx                   | i       |  162031 | 1266 MB
 test_pkey                      | i       |   63258 | 494 MB
 test                           | r       |   36477 | 285 MB
 pg_proc                        | r       |      47 | 376 kB
 pg_proc_proname_args_nsp_index | i       |      34 | 272 kB
(5 lignes)
```

On peut suivre la quantité de blocs *dirty* et l'*usagecount* avec une requête de ce genre, ici juste après une petite mise à jour de la table `test` :

```
SELECT
    relname,
    isdirty,
    usagecount,
    pinning_backends,
    count(bufferid)
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg%'
GROUP BY
        relname,
        isdirty,
        usagecount,
        pinning_backends
ORDER BY 1, 2, 3, 4 ;
```

```
    relname     | isdirty | usagecount | pinning_backends | count
----------------+---------+------------+------------------+--------
 brin_btree_idx | f       |          0 |                0 |      1
 brin_btree_idx | f       |          1 |                0 |   7151
 brin_btree_idx | f       |          2 |                0 |   3103
 brin_btree_idx | f       |          3 |                0 |  10695
 brin_btree_idx | f       |          4 |                0 | 141078
 brin_btree_idx | f       |          5 |                0 |      2
```

```
 brin_btree_idx | t       |          1 |                0 |      9
 brin_btree_idx | t       |          2 |                0 |      1
 brin_btree_idx | t       |          5 |                0 |     60
 test           | f       |          0 |                0 |  12371
 test           | f       |          1 |                0 |   6009
 test           | f       |          2 |                0 |   8466
 test           | f       |          3 |                0 |   1682
 test           | f       |          4 |                0 |   7393
 test           | f       |          5 |                0 |    112
 test           | t       |          1 |                0 |      1
 test           | t       |          5 |                0 |    267
 test_pkey      | f       |          1 |                0 |    173
 test_pkey      | f       |          2 |                0 |  27448
 test_pkey      | f       |          3 |                0 |   6644
 test_pkey      | f       |          4 |                0 |  10324
 test_pkey      | f       |          5 |                0 |   3420
 test_pkey      | t       |          1 |                0 |     57
 test_pkey      | t       |          3 |                0 |     81
 test_pkey      | t       |          4 |                0 |    116
 test_pkey      | t       |          5 |                0 |  15067
(26 lignes)
```

7.10.3 PG_FREESPACEMAP

La `Freespacemap` :

- Est renseignée par VACUUM, par objet (table/index)
- Et consommée par les sessions modifiant des données (INSERT/UPDATE)
- Interroger la freespacemap permet de connaître l'espace libre cartographié par VACUUM
- Rarement utilisé (dans le cas de doute sur l'efficacité de VACUUM)

Voici deux exemples d'utilisation de `pg_freespacemap` :

```
dspam=# SELECT * FROM pg_freespace('dspam_token_data') LIMIT 20;
 blkno | avail
-------+-------
     0 |    32
     1 |     0
     2 |     0
     3 |    32
     4 |     0
     5 |     0
     6 |     0
```

```
     7 |     0
     8 |    32
     9 |    32
    10 |    32
    11 |     0
    12 |     0
    13 |     0
    14 |     0
    15 |     0
    16 |     0
    17 |     0
    18 |    32
    19 |    32
(20 rows)

dspam=# SELECT * FROM pg_freespace('dspam_token_data') ORDER BY avail DESC
LIMIT 20;
 blkno | avail
-------+-------
 67508 |  7520
 67513 |  7520
 67460 |  7520
 67507 |  7520
 67451 |  7520
 67512 |  7520
 67452 |  7520
 67454 |  7520
 67505 |  7520
 67447 |  7520
 67324 |  7520
 67443 |  7520
 67303 |  7520
 67509 |  7520
 67444 |  7520
 67448 |  7520
 67445 |  7520
 66888 |  7520
 67516 |  7520
 67514 |  7520
```

L'interprétation de « avail » est un peu complexe, et différente suivant qu'on inspecte une table ou un index. Il est préférable de se référer à la documentation.

7.10.4 PG_VISIBILITY

La `Visibility Map` :

- Est renseignée par `VACUUM`, par table
- Permet de savoir que l'ensemble des enregistrements de ce bloc est visible
- Indispensable pour les parcours d'index seul
- Interroger la *visibility map* permet de voir si un bloc est :
  - visible
  - gelé
- Rarement utilisé

On crée une table de test avec 451 lignes :

```
CREATE TABLE test_visibility AS SELECT generate_series(0,450) x;
SELECT 451
```

On regarde dans quel état est la *visibility map* :

```
SELECT oid FROM pg_class WHERE relname='test_visibility' ;
  oid
-------
 18370


SELECT * FROM pg_visibility(18370);

 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | f           | f          | f
     1 | f           | f          | f
```

Les deux blocs que composent la table `test_visibility` sont à `false`, ce qui est normal puisque l'opération de vacuum n'a jamais été exécutée sur cette table.

On lance donc une opération de vacuum :

```
VACUUM VERBOSE test_visibility ;

INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 0 versions de ligne supprimables,
                             451 non supprimables
parmi 2 pages sur 2
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 0 pointeurs d'éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
```

```
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

Vacuum voit bien nos 451 lignes, et met donc la *visibility map* à jour. Lorsqu'on la consulte, on voit bien que toutes les lignes sont visibles de toutes les transactions :

```
SELECT * FROM pg_visibility(33259);

 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | t           | f          | t
(2 lignes)
```

La colonne `all_frozen` passera à `t` après un `VACUUM FREEZE`.

7.10.5 PAGEINSPECT

pageinspect :

- 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 heap (table), bt (btree), entête de page, et 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 5;
 lp | lp_off | lp_flags | lp_len |   t_xmin   | t_xmax | t_field3 | t_ctid
----+--------+----------+--------+------------+--------+----------+--------
  1 |    201 |        2 |      0 |            |        |          |
  2 |   1424 |        1 |     48 | 1439252980 |      0 |        0 | (0,2)
  3 |    116 |        2 |      0 |            |        |          |
  4 |   7376 |        1 |     48 |          2 |      0 |      140 | (0,4)
  5 |   3536 |        1 |     48 | 1392499801 |      0 |        0 | (0,5)

 lp | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+-------------+------------+--------+--------+-------
  1 |             |            |        |        |
  2 |           5 |       2304 |     24 |        |
  3 |             |            |        |        |
```

```
  4 |           5 |      10496 |     24 |        |
  5 |           5 |       2304 |     24 |        |
```

Et son entête:

```
=# SELECT * FROM page_header(get_raw_page('dspam_token_data',0));
-[ RECORD 1 ]-------------------------------------------
lsn       | F1A/5A6EAC40
checksum  | 0
flags     | 1
lower     | 852
upper     | 896
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 1450780148
```

Méta-données d'un index (contenu dans la première page):

```
=# SELECT * FROM bt_metap('dspam_token_data_uid_key');
 magic  | version | root | level | fastroot | fastlevel
--------+---------+------+-------+----------+-----------
 340322 |       2 |  243 |     2 |      243 |         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
```

7.10.6 PGROWLOCKS

Les verrous mémoire de PostgreSQL ne verrouillent pas les enregistrements :

- Il est parfois compliqué de comprendre qui verrouille qui, à cause de quel enregistrement
- pgrowlocks inspecte une table pour détecter les enregistrements verrouillés, leur niveau de verrouillage, et qui les verrouille
- scan complet de la table (impossible de trouver autrement)

Par exemple:

```
=# SELECT * FROM pgrowlocks('dspam_token_data');
 locked_row |   locker   | multi |     xids     |       modes       | pids
------------+------------+-------+--------------+-------------------+-------
 (0,2)      | 1452109863 | f     | {1452109863} | {"No Key Update"} | {928}
(1 row)
```

Nous savons donc que l'enregistrement (0,2) est verrouillé par le pid 928. Nous avons le mode de verrouillage, le (ou les) numéro de transaction associés. Un enregistrement peut être verrouillé par plus d'une transaction dans le cas d'un `SELECT FOR SHARE`. Dans ce cas, PostgreSQL crée un « multixact » qui est stocké dans `locker`, `multi` vaut `true`, et `xids` contient plus d'un enregistrement. C'est un cas très rare d'utilisation.

7.10.7 PG_PREWARM

Extension à partir de PostgreSQL 9.4 :

- Charge une liste de blocs
- Dans le cache du système
  - De façon asynchrone : `prefetch` (Linux)
  - De façon synchrone : `read` (tous systèmes)
- Ou dans le cache PostgreSQL
  - De façon synchrone : `buffer` (tous systèmes)
- À coupler avec une capture du cache (`pg_buffercache` par exemple)

Par exemple, on charge la table `pomme` dans le cache de PostgreSQL ainsi, et on le vérifie avec `pg_buffercache` :

```
SELECT pg_prewarm ('pomme', 'buffer') ;

CREATE EXTENSION pg_buffercache ;

SELECT c.relname, count(*) AS buffers, pg_size_pretty(count(*)*8192) as taille_mem
FROM pg_buffercache b INNER JOIN pg_class c
                    ON b.relfilenode = pg_relation_filenode(c.oid)
GROUP BY c.relname ;

                relname                | buffers | taille_mem
---------------------------------------+---------+------------
 ...
 pomme                                 |      45 | 360 kB
 ...
```

Cet ordre sauvegarde l'état du cache (pour la base dans laquelle on l'exécute) :

```
CREATE TABLE bkp_cache AS
SELECT pg_class.oid AS objid, relforknumber, relblocknumber
FROM pg_buffercache
JOIN pg_class ON (pg_buffercache.relfilenode=pg_class.relfilenode);
```

Cet ordre recharge les blocs précédents dans le cache de du système d'exploitation :

```
SELECT pg_prewarm(objid::regclass,
                  'prefetch',
                  CASE relforknumber
                    WHEN 0 THEN 'main'
                    WHEN 1 THEN 'fsm'
                    WHEN 2 THEN 'vm'
                  END,
                  relblocknumber,relblocknumber)
FROM bkp_cache;
```

7.10.8 PG_STAT_STATEMENTS

Capture en temps réel des requêtes :

- Vue, en mémoire partagée (volumétrie contrôlée)
- Par requête
  - Nombre d'exécution, temps cumulé d'exécution, nombre d'enregistrements retournés
  - lectures/écritures en cache, demandées au système, tris
  - temps de lecture/écriture (`track_io_timing`)
  - Pas d'échantillonnage, seulement des compteurs cumulés

`pg_stat_statements` capture, à chaque exécution de requête, tous les compteurs ci-dessus. La requête étant déjà analysée, cette opération supplémentaire n'ajoute qu'un faible surcoût (de l'ordre de 5 % sur une requête extrêmement courte), fixe, pour chaque requête.

`pg_stat_statements` fournit une vue (du même nom), qui retourne un instantané des compteurs au moment de l'interrogation, ainsi qu'une fonction `pg_stat_statements_reset`. Deux méthodes sont donc possibles :

- Effectuer un reset au début d'une période, puis interroger la vue `pg_stat_statements` à la fin de cette période
- Capturer à intervalle régulier le contenu de `pg_stat_statements` et visualiser les changements dans les compteurs. Le projet powa[35] a été développé à cet effet.

7.10.9 AUTO_EXPLAIN

N'est pas une extension :

- Juste un module à charger et des variables à positionner
  - `LOAD 'auto_explain'` dans une session
  - ou `shared_preload_libraries='auto_explain'` si global
- Trace le plan de toute requête dépassant une durée d'exécution dans la log
- Éventuellement l'`EXPLAIN ANALYZE/BUFFERS` : Attention, exécute la requête une seconde fois !
- `SET auto_explain.log_min_duration = '3s'`

Cet outil est habituellement activé quand on a le sentiment qu'une requête devient subitement lente à une période de la journée, et qu'on a le sentiment que son plan a changé.

[35] http://dalibo.github.io/powa/

Elle permet donc de tracer dans la log le plan de la requête dès qu'elle dépasse une durée.

Tracer le plan d'exécution d'une requête prend de la place, et est assez coûteux. C'est donc à déconseiller pour des requêtes très courtes. Quelques secondes est un point de départ raisonnable.

Par exemple :

```
=# LOAD 'auto_explain';
=# SET auto_explain.log_min_duration = 0;
=# SET auto_explain.log_analyze = true;
=# SET client_min_messages to log;
=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
LOG:  duration: 0.311 ms  plan:
Query Text: SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
Aggregate  (cost=19.84..19.85 rows=1 width=0)
           (actual time=0.300..0.301 rows=1 loops=1)
  ->  Hash Join  (cost=5.41..19.59 rows=102 width=0)
                 (actual time=0.127..0.286 rows=105 loops=1)
        Hash Cond: (pg_class.oid = pg_index.indrelid)
        ->  Seq Scan on pg_class  (cost=0.00..10.95 rows=295 width=4)
                                  (actual time=0.005..0.080 rows=312 loops=1)
        ->  Hash  (cost=4.13..4.13 rows=102 width=4)
                  (actual time=0.091..0.091 rows=105 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 4kB
              ->  Seq Scan on pg_index
                        (cost=0.00..4.13 rows=102 width=4)
                        (actual time=0.003..0.054 rows=105 loops=1)
                    Filter: indisunique
                    Rows Removed by Filter: 15
 count
-------
   105
(1 ligne)
```

7.11 PGXN

PostgreSQL eXtension Network :

- Site WEB : pgxn.org[36]

[36] https://pgxn.org/

  - Nombreuses extensions
  - Volontariat
  - Aucune garantie de qualité
  - Tests soigneux requis
- Et optionnellement client en python pour automatisation de déploiement
- Ancêtre : pgFoundry.org[37] , toujours en service
- Beaucoup de projets sont aussi sur github

Le site PGXN fournit une vitrine à de nombreux projets gravitant autour de PostgreSQL. Ce rôle était historiquement tenu par le site pgFoundry.org, qui est sur le déclin.

PGXN a de nombreux avantages, dont celui de demander aux projets participants de respecter un certain cahier des charges permettant l'installation automatisée des modules hébergés. Ceci peut par exemple être réalisé avec le client pgxn fourni :

```
> pgxn search --dist fdw
multicdr_fdw 1.2.2
    MultiCDR *FDW* ================== Foreign Data Wrapper for representing
    CDR files stream as an external SQL table. CDR files from a directory
    can be read into a table with a specified field-to-column...

redis_fdw 1.0.0
    Redis *FDW* for PostgreSQL 9.1+ ============================== This
    PostgreSQL extension implements a Foreign Data Wrapper (*FDW*) for the
    Redis key/value database: http://redis.io/ This code is...

jdbc_fdw 1.0.0
    Also,since the JVM being used in jdbc *fdw* is created only once for the
    entire psql session,therefore,the first query issued that uses jdbc
    *fdw* shall set the value of maximum heap size of the JVM(if...

mysql_fdw 2.1.2
    ... This PostgreSQL extension implements a Foreign Data Wrapper (*FDW*)
    for [MySQL][1]. Please note that this version of mysql_fdw only works
    with PostgreSQL Version 9.3 and greater, for previous version...

www_fdw 0.1.8
    ... library contains a PostgreSQL extension, a Foreign Data Wrapper
    (*FDW*) handler of PostgreSQL which provides easy way for interacting
    with different web-services.
```

[37] http://pgfondry.org/

```
mongo_fdw 2.0.0
    MongoDB *FDW* for PostgreSQL 9.2 ============================== This
    PostgreSQL extension implements a Foreign Data Wrapper (*FDW*) for
    MongoDB.

json_fdw 1.0.0
    ... This PostgreSQL extension implements a Foreign Data Wrapper (*FDW*)
    for JSON files. The extension doesn't require any data to be loaded into
    the database, and supports analytic queries against array...

firebird_fdw 0.1.0
    ...  -
    http://www.postgresql.org/docs/current/interactive/postgres-*fdw*.html *
    Other FDWs  - https://wiki.postgresql.org/wiki/*Fdw*  -
    http://pgxn.org/tag/*fdw*/

postgres_fdw 1.0.0
    This port provides a read-only Postgres *FDW* to PostgreSQL servers in
    the 9.2 series. It is a port of the official postgres_fdw contrib module
    available in PostgreSQL version 9.3 and later.

osm_fdw 1.0.2
    ... "Openstreetmap pbf foreign data wrapper") (*FDW*) for reading
    [Openstreetmap PBF](http://wiki.openstreetmap.org/wiki/PBF_Format
    "Openstreetmap PBF") file format (*.osm.pbf) ## Requirements  *...

couchdb_fdw 0.1.0
    CouchDB *FDW* (beta) for PostgreSQL 9.1+
    ====================================== This PostgreSQL extension
    implements a Foreign Data Wrapper (*FDW*) for the CouchDB document-
    oriented database...

odbc_fdw 0.1.0
    ODBC *FDW* (beta) for PostgreSQL 9.1+
    ================================== This PostgreSQL extension implements
    a Foreign Data Wrapper (*FDW*) for remote databases using Open Database
    Connectivity(ODBC)...
```

```
treasuredata_fw 1.1.0
    ... Foreign Data Wrapper for Treasure Data ## Installation This *FDW*
    uses [td-client-rust](https://github.com/komamitsu/td-client-rust).

oracle_fdw 1.5.0
    ... here:
    http://lists.pgfoundry.org/mailman/listinfo/oracle-*fdw*-general There
    is a mail archive here:
    http://lists.pgfoundry.org/pipermail/oracle-*fdw*-general/ There is the
    option to open an issue on...

twitter_fdw 1.1.1
    Installation ------------  $ make && make install  $ psql -c "CREATE
    EXTENSION twitter_fdw" db The CREATE EXTENSION statement creates not
    only *FDW* handlers but also Data Wrapper, Foreign Server, User...

ldap_fdw 0.1.1
    ... is an initial working on a PostgreSQL's Foreign Data Wrapper (*FDW*)
    to query LDAP servers. By all means use it, but do so entirely at your
    own risk! You have been warned! Do you like to use it in...

foreign_table_exposer 1.0.0
    # foreign_table_exposer This PostgreSQL extension exposes foreign tables
    like a normal table with rewriting Query tree. Some BI tools can't
    detect foreign tables since they don't consider them when...

cstore_fdw 1.5.0
    cstore_fdw ========== [![Build Status](https://travis-
    ci.org/citusdata/cstore_fdw.svg?branch=master)][status] [![Coverage](htt
    p://img.shields.io/coveralls/citusdata/cstore_fdw/master.svg)][coverage]
    ...

multicorn 1.3.3
    [![PGXN version](https://badge.fury.io/pg/multicorn.svg)](https://badge.
    fury.io/pg/multicorn) [![Build
    Status](https://jenkins.dalibo.info/buildStatus/public/Multicorn)]()
    Multicorn =========...

tds_fdw 1.0.7
```

```
    # TDS Foreign data wrapper * **Author:** Geoff Montee * **Name:**
    tds_fdw * **File:** tds_fdw/README.md ## About This is a [PostgreSQL
    foreign data...

file_textarray_fdw 1.0.1
    ### File Text Array Foreign Data Wrapper for PostgreSQL This *FDW* is
    similar to the provided file_fdw, except that instead of the foreign
    table having named fields to match the fields in the data...

pmpp 1.2.2
    ... Having foreign server definitions and user mappings makes for
    cleaner function invocations.

pg_pathman 1.4.0
    ... event handling;  * Non-blocking concurrent table partitioning;  *
    *FDW* support (foreign partitions);  * Various GUC toggles and
    configurable settings.
```

Pour peu que le Instant Client d'Oracle soit installé, on peut par exemple lancer :

```
> pgxn install oracle_fdw
INFO: best version: oracle_fdw 1.1.0
INFO: saving /tmp/tmpihaor2is/oracle_fdw-1.1.0.zip
INFO: unpacking: /tmp/tmpihaor2is/oracle_fdw-1.1.0.zip
INFO: building extension
gcc -O3 -O0 -Wall -Wmissing-prototypes -Wpointer-arith [...]
[...]
INFO: installing extension
/usr/bin/mkdir -p '/opt/postgres/lib'
/usr/bin/mkdir -p '/opt/postgres/share/extension'
/usr/bin/mkdir -p '/opt/postgres/share/extension'
/usr/bin/mkdir -p '/opt/postgres/share/doc/extension'
/usr/bin/install -c -m 755  oracle_fdw.so '/opt/postgres/lib/oracle_fdw.so'
/usr/bin/install -c -m 644 oracle_fdw.control '/opt/postgres/share/extension/'
/usr/bin/install -c -m 644 oracle_fdw--1.1.sql\oracle_fdw--1.0--1.1.sql
                           '/opt/postgres/share/extension/'
/usr/bin/install -c -m 644 README.oracle_fdw \
                           '/opt/postgres/share/doc/extension/'
```

Attention : le fait qu'un projet soit hébergé sur PGXN n'est absolument pas une validation de la part du projet PostgreSQL. De nombreux projets hébergés sur PGXN sont encore en

phase de développement, ou même éventuellement abandonnés. Il faut avoir le même recul que pour n'importe quel autre brique libre.

Par ailleurs, il est important de noter que de nombreux projets sont encore hébergés sur pgFoundry.org, ou ont opté pour github.com, voire s'auto-hébergent.

7.12 CONCLUSION

- Un nombre toujours plus important d'extension permettant d'étendre les possibilités de PostgreSQL
- Certains modules de contribs sont inclus dans le coeur de PostgreSQL lorsqu'ils sont considérés comme matures et utiles au moteur (`tsearch`, `xml2`)
- Un site central pour les extensions PGXN.org, mais toutes n'y sont pas référencées.

Cette possibilité d'étendre les fonctionnalités de PostgreSQL est vraiment un atout majeur du projet PostgreSQL. Cela permet de tester des fonctionnalités sans avoir à toucher au moteur de PostgreSQL et risquer des états instables. Une fois l'extension mature, elle peut être intégrée directement dans le code de PostgreSQL si elle est considérée utile au moteur.

7.12.1 QUESTIONS

N'hésitez pas, c'est le moment !

7.13 TRAVAUX PRATIQUES

7.13.1 ÉNONCÉS

SQL/MED, Foreign Data Wrappers

- Créez une foreign table qui présente les champs du fichier `/etc/passwd` sous forme de table. Vérifiez son bon fonctionnement avec un SELECT.

Il s'agit d'utiliser le Foreign Data Wrapper file_fdw

- Accédez à la table `stock` de votre voisin

Tout d'abord, vérifiez qu'avec psql vous arrivez à vous connecter chez lui. Sinon, vérifiez `listen_addresses`, et le fichier `pg_hba.conf`.

Une fois que la connexion avec psql fonctionne, créez la foreign table `stock_remote` chez votre voisin. Attention, si vous avez fait le TP partitionnement précédemment, accédez plutôt à `stock_old`.

Installez d'abord le FOREIGN DATA WRAPPER:

Créez le FOREIGN SERVER (déclaration du serveur de votre voisin). Ajustez les options pour correspondre à votre environnement :

Créez un USER MAPPING, c'est-à-dire une correspondance entre votre utilisateur local et l'utilisateur distant :

Puis créez votre FOREIGN TABLE

Vérifiez le bon fonctionnement de la foreign table.

Vérifiez le plan.

Il faut l'option `VERBOSE` pour voir la requête envoyée au serveur distant. Vous constatez que le prédicat sur vin_id a été transmis, ce qui est le principal avantage de cette implémentation sur les DBLinks.

Modules contrib

- Installer le module `auto_explain`.

Modifiez le fichier `postgresql.conf` puis redémarrez PostgreSQL.

Exécutez des requêtes sur la base cave, et inspectez la log.

Vous pouvez aussi recevoir les messages directement dans votre session. Tous les messages de log sont marqués d'un niveau de priorité. Les messages produits par `auto_explain` sont au niveau 'log'. Il vous suffit donc de passer le paramètre `client_min_messages` au niveau `log` (ou inférieur, comme `debug`).

Positionnez le paramètre de session, ré-exécutez votre requête.

- Module `pg_stat_statements` :

Lui aussi nécessite une librairie préchargée. Positionnez-la dans le fichier `postgresql.conf`.

Créez l'extension.

Vous pouvez en profiter pour inspecter le contenu de l'extension `pg_stat_statements`.

Maintenant, inspectez la vue `pg_stat_statements`. Exécutez une requête coûteuse (la récupération du nombre de bouteilles de chaque appellation en stock par exemple).

17.12

Examinez la vue `pg_stat_statements` : récupérez les 5 requêtes les plus gourmandes en temps cumulé sur votre instance.

Nous allons activer la mesure de la durée des entrées sorties. Contrôlons déjà que le serveur en est capable avec `pg_test_timing`.

Si vous avez un temps de mesure de quelques dizaines de nanosecondes, c'est OK. Sinon, évitez de faire ce qui suit sur un serveur de production. Sur votre machine de formation, ce n'est pas un problème.

Activez la mesure des temps d'exécution des entrées-sorties, redémarrez PostgreSQL (pour vider son cache), remettez la vue `pg_stat_statements` à 0, et ré-exécutez la requête « lourde » précédente :

- Hstore :

Pour ce TP, il est fortement conseillé d'aller regarder la documentation officielle du type hstore.

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

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

Écrivez tout d'abord une requête classique affichant les informations supplémentaires, au moyen de jointures. Limitez la à quelques enregistrements pour afficher seulement quelques enregistrements représentatifs.

Une fois que votre requête est prête, servez-vous en pour créer une nouvelle table `stock_denorm` de cette définition :

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

Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de hstore sans se soucier de formatage de chaîne de caractères. Appuyez-vous sur la documentation officielle du type hstore pour trouver des possibilités d'écriture.

Créez maintenant un index pour accélérer les recherches.

Nous allons maintenant pouvoir réaliser une recherche. N'oubliez pas de passer les statistiques sur la table `stock_denorm`.

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

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

Re-faites la même requête sur le schéma initial.

Conclusion ?

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

JSONB

Comme lors de l'exercice précédent, nous allons créer une table dénormalisée mais cette fois au format jsonb. Celle ci aura la structure suivante :

```
document jsonb
```

Le document aura la structure suivante :

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

```
  }]
}
```

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

- Écrivez une requête permettant de générer la partie ”recoltant” du document
- Écrivez une requête permettant de générer la partie vin du document
- Écrivez une requête permettant de générer un élément du tableau stock
- Écrivez une requête permettant de générer la partie ”stocks” du document, pour un vin donné (ex: le 1)
- Enfin, écrivez la requête finale.

Indexez le document jsonb en utilisant un index de type GIN.

Calculez la taille de la table, et comparez-la à la taille du reste de la base. Que constatez vous ?

Écrivez des requêtes pour tirer parti de ce document, et de l'index créé dessus.

- Renvoyez l'ensemble des récoltants de la région Beaujolais
- Renvoyez l'ensemble des vins pour lesquels au moins une bouteille entre 92 et 95 existe

Pouvez-vous écrire une version de ces requêtes utilisant l'index ?

7.13.2 SOLUTIONS

SQL/MED, Foreign Data Wrappers

- Créez une foreign table qui présente les champs du fichier `/etc/passwd` sous forme de table. Vérifiez son bon fonctionnement avec un SELECT.

```
CREATE EXTENSION file_fdw;
CREATE SERVER files FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE passwd (
  login text,
  passwd text,
  uid int,
  gid int,
  username text,
  homedir text,
  shell text)
SERVER files
OPTIONS (filename '/etc/passwd', format 'csv', delimiter ':');
```

- Accédez à la table stock de votre voisin

Tout d'abord, vérifiez qu'avec psql vous arrivez à vous connecter chez lui. Sinon, vérifiez `listen_addresses`, et le fichier `pg_hba.conf`.

Une fois que la connexion avec psql fonctionne, créez la foreign table `stock_remote` chez votre voisin. Attention, si vous avez fait le TP partitionnement précédemment, accédez plutôt à `stock_old`.

Installez d'abord le FOREIGN DATA WRAPPER :

```
CREATE EXTENSION postgres_fdw ;
```

Créez le FOREIGN SERVER (déclaration du serveur de votre voisin). Ajustez les options pour correspondre à votre environnement :

```
CREATE SERVER serveur_voisin
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '192.168.0.18', port '5940', dbname 'cave');
```

Créez un USER MAPPING, c'est à dire une correspondance entre votre utilisateur local et l'utilisateur distant :

```
CREATE USER MAPPING FOR mon_utilisateur
SERVER serveur_voisin
OPTIONS (user 'utilisateur_distant', password 'mdp_utilisateur_distant');
```

Puis créez votre FOREIGN TABLE

```
CREATE FOREIGN TABLE stock_voisin (
vin_id integer, contenant_id integer, annee integer, nombre integer)
SERVER serveur_voisin
OPTIONS (schema_name 'public', table_name 'stock_old');
```

Vérifiez le bon fonctionnement de la foreign table :

```
SELECT * FROM stock_voisin WHERE vin_id=12;
```

Vérifiez le plan :

```
EXPLAIN ANALYZE VERBOSE SELECT * FROM stock_voisin WHERE vin_id=12;
```

Il faut l'option `VERBOSE` pour voir la requête envoyée au serveur distant. Vous constatez que le prédicat sur vin_id a été transmis, ce qui est le principal avantage de cette implémentation sur les DBLinks.

Modules contrib

- Installer le module auto_explain :

Fichier `postgresql.conf` :

```
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = 0
```

17.12

Redémarrer PostgreSQL.

Exécutez des requêtes sur la base cave, et inspectez la log.

Vous pouvez aussi recevoir les messages directement dans votre session. Tous les messages de log sont marqués d'un niveau de priorité. Les messages produits par auto_explain sont au niveau 'log'. Il vous suffit donc de passer le paramètre `client_min_messages` au niveau `log` (ou inférieur, comme `debug`).

Positionnez le paramètre de session, ré-exécutez votre requête.

```
SET client_min_messages TO log;
SELECT…
```

- Module `pg_stat_statements` :

Lui aussi nécessite une librairie préchargée :

```
shared_preload_libraries = 'auto_explain,pg_stat_statements'
```

Redémarrer PostgreSQL.

Créer l'extension :

```
CREATE EXTENSION pg_stat_statements;
```

Vous pouvez en profiter pour inspecter le contenu de l'extension `pg_stat_statements` :

```
\dx+ pg_stat_statements
```

Maintenant, inspectez la vue `pg_stat_statements`. Exécutez une requête coûteuse (la récupération du nombre de bouteilles de chaque appellation en stock par exemple).

Examinez la vue `pg_stat_statements` : récupérez les 5 requêtes les plus gourmandes en temps cumulé sur votre instance.

```
SELECT appellation.libelle,
       sum(stock.nombre)
FROM appellation
JOIN vin ON appellation.id=vin.appellation_id
JOIN stock ON vin.id=stock.vin_id
GROUP BY appellation.libelle;

SELECT * FROM pg_stat_statements ORDER BY total_time desc LIMIT 5;
```

Nous allons activer la mesure de la durée des entrées sorties. Contrôlons déjà que le serveur en est capable :

```
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 34.23 nsec
Histogram of timing durations:
```

```
< usec   % of total      count
     1     96.58665   84647529
     2      3.41157    2989865
     4      0.00044        387
     8      0.00080        702
    16      0.00052        455
    32      0.00002         16
    64      0.00000          1
   128      0.00000          1
   256      0.00000          0
   512      0.00000          1
```

Si vous avez un temps de mesure de quelques dizaines de nanosecondes, c'est OK. Sinon, évitez de faire ce qui suit sur un serveur de production. Sur votre machine de formation, ce n'est pas un problème.

Activez la mesure des temps d'exécution des entrées-sorties, redémarrez PostgreSQL (pour vider son cache), remettez la vue `pg_stat_statements` à 0, et ré-exécutez la requête « lourde » précédente :

Positionnez `track_io_timing=on` dans votre fichier `postgresql.conf`.

Redémarrez PostgreSQL.

Exécutez `SELECT pg_stat_statements_reset();` sur votre instance.

Ré-exécutez votre requête, et constatez dans `pg_stat_statements` que les colonnes `blk_read_time` et `blk_write_time` sont maintenant mises à jour.

- Hstore :

Pour ce TP, il est fortement conseillé d'aller regarder la documentation officielle du type hstore.

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

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

Écrivez tout d'abord une requête classique affichant les informations supplémentaires, au moyen de jointures.

```
SELECT stock.vin_id,
       stock.annee,
       stock.nombre,
```

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

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

Une fois que votre requête est prête, servez-vous en pour créer une nouvelle table «stock_denorm» de cette définition:

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

Une des écritures possibles passe par la génération d'un tableau, ce qui permet de passer tous les éléments au constructeur de hstore sans se soucier de formatage de chaîne de caractères.

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

Une remarque toutefois : les éléments du tableau doivent tous être de même type, d'où

la conversion en text des quelques éléments entiers. C'est aussi une limitation du type hstore : il ne supporte que les attributs texte.

Créons maintenant un index pour accélérer nos recherches :

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

Nous allons maintenant pouvoir réaliser une recherche. N'oubliez pas de passer les statistiques sur la table stock_denorm.

```
ANALYZE stock_denorm;
```

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

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

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

Re-faites la même requête sur le schéma initial.

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

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

```
                                (actual time=0.033..0.257 rows=53 loops=1)
                                 Hash Cond:
                                      (appellation.region_id = region.id)
                                 Buffers: shared hit=4
                                 ->  Seq Scan on appellation
                                      (cost=0.00..6.19 rows=319 width=24)
                                      (actual time=0.010..0.074 rows=319
                                       loops=1)
                                       Buffers: shared hit=3
                                 ->  Hash
                                        (cost=1.24..1.24 rows=1 width=15)
                                        (actual time=0.013..0.013 rows=1
                                         loops=1)
                                       Buckets: 1024  Batches: 1
                                                Memory Usage: 1kB
                                       Buffers: shared hit=1
                                       ->  Seq Scan on region
                                         (cost=0.00..1.24 rows=1 width=15)
                                         (actual time=0.005..0.012 rows=1
                                                 loops=1)
                                             Filter: (libelle =
                                                      'Alsace'::text)
                                             Rows Removed by Filter: 18
                                             Buffers: shared hit=1
                    ->  Hash  (cost=1.04..1.04 rows=1 width=9)
                              (actual time=0.008..0.008 rows=1 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 1kB
                          Buffers: shared hit=1
                          ->  Seq Scan on type_vin
                                    (cost=0.00..1.04 rows=1 width=9)
                                    (actual time=0.005..0.007 rows=1 loops=1)
                               Filter: (libelle = 'blanc'::text)
                               Rows Removed by Filter: 2
                               Buffers: shared hit=1
              ->  Hash  (cost=1.07..1.07 rows=7 width=23)
                        (actual time=0.017..0.017 rows=7 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 1kB
                    Buffers: shared hit=1
                    ->  Seq Scan on recoltant
                              (cost=0.00..1.07 rows=7 width=23)
                              (actual time=0.004..0.009 rows=7 loops=1)
                          Buffers: shared hit=1
        ->  Index Scan using idx_stock_vin_annee on stock
                    (cost=0.42..6.59 rows=15 width=16)
                    (actual time=0.013..0.038 rows=15 loops=336)
              Index Cond: (vin_id = vin.id)
```

```
              Buffers: shared hit=6248
 Planning time: 4.341 ms
 Execution time: 25.232 ms
(53 lignes)
```

Conclusion ?

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

JSONB

Pour écrire la requête correspondant à la partie recoltant, rien de plus simple :

```
select json_build_object('nom', nom, 'adresse', adresse) from recoltant;
```

Pour écrire la requête correspondant à la partie vin, il nous faut d'abord récupérer l'intégralité des données concernées à l'aide de jointures :

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

À partir de cette requête, on compose le document lui-même :

```
SELECT
  json_build_object(
    'recoltant',
    json_build_object('nom', recoltant.nom, 'adresse',
                      recoltant.adresse
    ),
    'appellation',
    json_build_object('libelle', appellation.libelle, 'region', region.libelle),
    'type_vin', type_vin.libelle
)
```

```
FROM vin
INNER JOIN recoltant on vin.recoltant_id = recoltant.id
INNER JOIN appellation on vin.appellation_id = appellation.id
INNER JOIN region on region.id = appellation.region_id
INNER JOIN type_vin on vin.type_vin_id = type_vin.id;
```

La partie stocks est un peu plus compliquée, et nécessite l'utilisation de fonctions d'aggrégations.

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

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

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

Enfin, pour la requête finale, on assemble ces deux parties :

```
CREATE TABLE stock_jsonb  AS (
  SELECT
    json_build_object(
    'vin',
      json_build_object(
        'recoltant',
        json_build_object('nom', recoltant.nom, 'adresse', recoltant.adresse),
        'appellation',
        json_build_object('libelle', appellation.libelle, 'region',
                          region.libelle),
        'type_vin', type_vin.libelle),
    'stocks',
      json_agg(json_build_object(
        'contenant',
        json_build_object('contenance', contenant.contenance, 'libelle',
                          contenant.libelle),
        'annee', stock.annee,
```

```
        'nombre', stock.nombre)))::jsonb as document
  FROM vin
  INNER JOIN recoltant on vin.recoltant_id = recoltant.id
  INNER JOIN appellation on vin.appellation_id = appellation.id
  INNER JOIN region on region.id = appellation.region_id
  INNER JOIN type_vin on vin.type_vin_id = type_vin.id
  INNER JOIN stock on stock.vin_id = vin.id
  INNER JOIN contenant on stock.contenant_id = contenant.id
  GROUP BY vin_id, recoltant.id, region.id, appellation.id, type_vin.id
);
```

Création de l'index :

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

La table contient toutes les mêmes informations que l'ensemble des tables normalisées de la base cave (à l'exception des id). Elle occupe en revanche une place beaucoup moins importante, puisque les documents individuels vont pouvoir être compressés en utilisant le mécanisme TOAST.

Récoltant de la région Beaujolais :

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

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

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

Liste des producteurs ayant du vin entre 1992 et 1995 :

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

Cette requête ne peut malheureusement pas être réécrite pour tirer partie d'un index avec les fonctionnalités intégrées à PostgreSQL.

Il est en revanche possible de le faire grâce à l'extension jsquery, qui n'est pas fournie par défaut avec PostgreSQL) :

```
CREATE INDEX ON stock_jsonb USING gin (document jsonb_path_value_ops);
SELECT DISTINCT document #> '{vin, recoltant, nom}'
FROM stock_jsonb
WHERE document @@ 'stocks.#.annee($ >= 1992 AND $ <= 1995)';
```