Formation DBA2

PostgreSQL Avancé

22.09

Dalibo SCOP

https://dalibo.com/formations

PostgreSQL Avancé

Formation DBA2

TITRE : PostgreSQL Avancé
SOUS-TITRE : Formation DBA2

REVISION: 22.09
DATE: 02 septembre 2022
ISBN: 978-2-38168-058-3

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

- 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. Le texte complet est disponible ici :

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

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

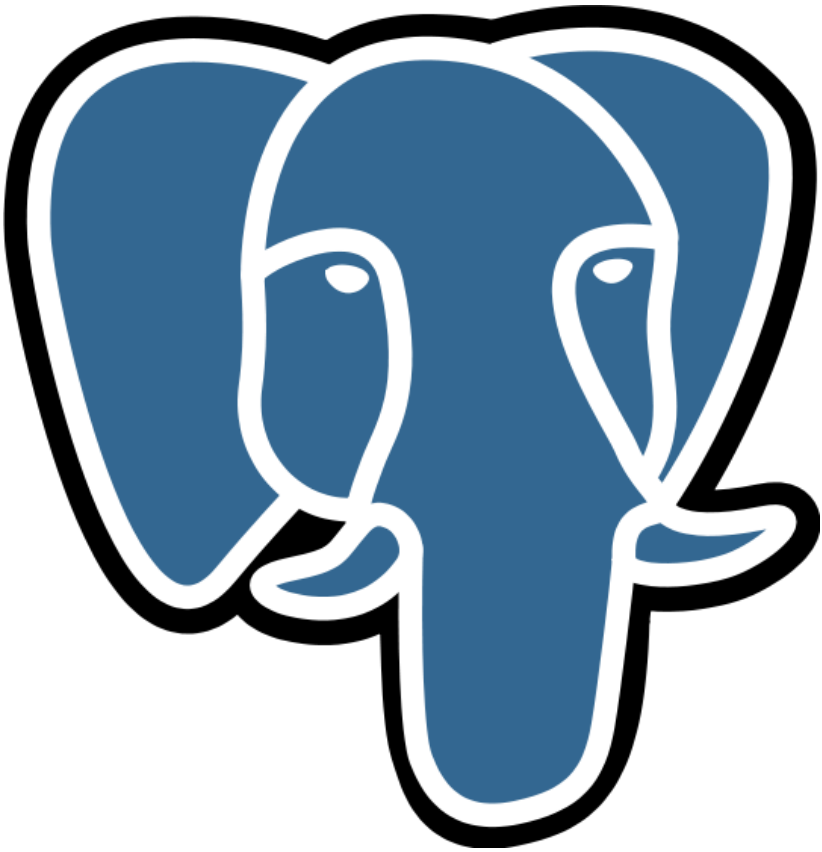
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

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

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

Table des Matières

1 ARCHITECTURE & FICHIERS DE POSTGRESQL



1.1 AU MENU

- Rappels sur l'installation
- Les processus
- Les fichiers

Le présent module vise à donner un premier aperçu global du fonctionnement interne de PostgreSQL.

Après quelques rappels sur l'installation, nous verrons essentiellement les processus et les fichiers utilisés.

1.2 RAPPELS SUR L'INSTALLATION

- Plusieurs possibilités
  - paquets Linux précompilés
  - outils externes d'installation
  - code source
- Chacun ses avantages et inconvénients
  - Dalibo recommande fortement les paquets précompilés

Nous recommandons très fortement l'utilisation des paquets Linux précompilés. Dans certains cas, il ne sera pas possible de faire autrement que de passer par des outils externes, comme l'installeur d'EntrepriseDB sous Windows.

1.2.1 PAQUETS PRÉCOMPILÉS

- Paquets Debian ou Red Hat suivant la distribution utilisée
- Préférence forte pour ceux de la communauté
- Installation du paquet
  - installation des binaires
  - création de l'utilisateur postgres
  - initialisation d'une instance (Debian seulement)
  - lancement du serveur (Debian seulement)
- (Red Hat) Script de création de l'instance

Debian et Red Hat fournissent des paquets précompilés adaptés à leur distribution. Dalibo recommande d'installer les paquets de la communauté, ces derniers étant bien plus à jour que ceux des distributions.

L'installation d'un paquet provoque la création d'un utilisateur système nommé postgres et l'installation des binaires. Suivant les distributions, l'emplacement des binaires change. Habituellement, tout est placé dans `/usr/pgsql-<version majeure>` pour les distributions Red Hat et dans `/usr/lib/postgresql/<version majeure>` pour les distributions Debian.

Dans le cas d'une distribution Debian, une instance est immédiatement créée dans `/var/lib/postgresql/<version majeure>/main`. Elle est ensuite démarrée.

Dans le cas d'une distribution Red Hat, aucune instance n'est créée automatiquement. Il faudra utiliser un script (dont le nom dépend de la version de la distribution) pour créer l'instance, puis nous pourrons utiliser le script de démarrage pour lancer le serveur.

1.2.2 INSTALLONS POSTGRESQL

- Prenons un moment pour
  - installer PostgreSQL
  - créer une instance
  - démarrer l'instance
- Pas de configuration spécifique pour l'instant

L'annexe ci-dessous décrit l'installation de PostgreSQL sans configuration particulière pour suivre le reste de la formation.

1.3 PROCESSUS DE POSTGRESQL

- 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, avec 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.

Tous les processus de PostgreSQL accèdent à une zone de « mémoire partagée ». Cette zone contient les informations devant être partagées entre les clients, comme un cache de données, ou des informations sur l'état de chaque session par exemple.

PostgreSQL utilise une architecture client-serveur. Nous ne nous connectons à PostgreSQL qu'à travers d'un protocole bien défini, nous n'accédons jamais aux fichiers de données.

1.3.1 PROCESSUS D'ARRIÈRE-PLAN

```
# ps f -e --format=pid,command | grep -E "postgres|postmaster"
  96122 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/data/
  96123  \_ postgres: logger
  96125  \_ postgres: checkpointer
  96126  \_ postgres: background writer
  96127  \_ postgres: walwriter
  96128  \_ postgres: autovacuum launcher
  96129  \_ postgres: stats collector
  96131  \_ postgres: logical replication launcher
```

(sous Rocky Linux 8)

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

NB : sur Debian, le postmaster est nommé *postgres* comme ses processus fils ; sous Windows les noms des processus sont par défaut moins verbeux.

1.3.2 PROCESSUS D'ARRIÈRE-PLAN (SUITE)

- Les processus présents au démarrage :
  - Un processus père : `postmaster`
  - `background writer`
  - `checkpointer`
  - `walwriter`
  - `autovacuum launcher`
  - `stats collector`
  - `logical replication launcher`
- et d'autres selon la configuration et le moment :
  - dont les *background workers* : parallélisation, extensions…

Le `postmaster` est responsable de la supervision des autres processus, ainsi que de la prise en compte des connexions entrantes.

Le `background writer` et le `checkpointer` s'occupent d'effectuer les écritures en arrière plan, évitant ainsi aux sessions des utilisateurs de le faire.

Le `walwriter` écrit le journal de transactions de façon anticipée, afin de limiter le travail de l'opération `COMMIT`.

L'`autovacuum launcher` pilote les opérations d'« autovacuum ».

Le `stats collector` collecte les statistiques d'exécution du serveur.

Le `logical replication launcher` est un processus dédié à la réplication logique, activé par défaut à partir de la version 10.

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

Ces différents processus seront étudiées en détail dans d'autres modules de formation.

Aucun de ces processus ne traite de requête pour le compte des utilisateurs. Ce sont des processus d'arrière-plan effectuant des tâches de maintenance.

Il existe aussi les *background workers* (processus d'arrière-plan), lancés par PostgreSQL, mais aussi par des extensions tierces. Par exemple, la parallélisation des requêtes se base sur la création temporaire de *background workers* épaulant le processus principal de la requête. La réplication logique utilise des *background workers* à plus longue durée de vie. De nombreuses extensions en utilisent pour des raisons très diverses. Le paramètre `max_worker_processes` régule le nombre de ces *workers*. Ne descendez pas en-dessous du défaut (8). Il faudra même parfois monter plus haut.

1.4 PROCESSUS PAR CLIENT (CLIENT BACKEND)

- Pour chaque client, nous avons un processus :
  - créé à la connexion
  - dédié au client…
  - …et qui dialogue avec lui
  - détruit à la déconnexion
- Un processus gère une requête
  - peut être aidé par d'autres processus (>= 9.6)
- Le nombre de processus est régi par les paramètres :
  - `max_connections` (défaut : 100)
  - `superuser_reserved_connections` (3)
    - compromis nombre requêtes actives/nombre cœurs/complexité/mé-

moire

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 (parallélisme).

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 à l'instance simultanées, actives ou non, est limité par le paramètre `max_connections`. Le défaut est 100. Afin de permettre à l'administrateur de se connecter à l'instance si cette limite était atteinte, `superuser_reserved_connections` sont réservées aux superutilisateurs 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.

La valeur 100 pour `max_connections` est généralement suffisante. Il peut être intéressant de la diminuer pour monter `work_mem` et autoriser plus de mémoire de tri. Il est possible de l'augmenter pour qu'un plus grand nombre d'utilisateurs puisse se connecter en même temps.

Il s'agit aussi d'arbitrer entre le nombre de requêtes à exécuter à un instant t, le nombre de CPU disponibles, la complexité des requêtes, et le nombre de processus que peut gérer l'OS.

Cela est encore compliqué par le parallélisme et la limitation de la bande passante des disques.

Intercaler un « pooler » entre les clients et l'instance peut se justifier dans certains cas :

- connexions/déconnexions très fréquentes (la connexion a un coût) ;
- centaines, voire milliers, de connexions généralement inactives.

Le plus réputé est PgBouncer.

1.5 GESTION DE LA MÉMOIRE

Structure de la mémoire sous PostgreSQL

- Zone de mémoire partagée :
  - *shared buffers* surtout
  - ...
- Zone de chaque processus
  - tris en mémoire (`work_mem`)
  - ...

La gestion de la mémoire dans PostgreSQL mérite un module de formation à lui tout seul.

Pour le moment, bornons-nous à la séparer en deux parties : la mémoire partagée et celle attribuée à chacun des nombreux processus.

La mémoire partagée stocke principalement le cache des données de PostgreSQL (*shared buffers*, paramètre `shared_buffers`), et d'autres zones plus petites : cache des journaux de transactions, données de sessions, les verrous, etc.

La mémoire propre à chaque processus sert notamment aux tris en mémoire (définie en premier lieu par le paramètre `work_mem`), au cache de tables temporaires, etc.

1.6 FICHIERS

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

Une instance est composée des éléments suivants :

Le répertoire de données :

Il contient les fichiers obligatoires au bon fonctionnement de l'instance : fichiers de données, journaux de transaction....

Les fichiers de configuration :

Selon la distribution, ils sont stockés dans le répertoire de données (Red Hat et dérivés comme CentOS ou Rocky Linux), ou dans `/etc/postgresql` (Debian et dérivés).

Un fichier PID :

Il permet de savoir si une instance est démarrée ou non, et donc à empêcher un second jeu de processus d'y accéder.

Le paramètre `external_pid_file` permet d'indiquer un emplacement où PostgreSQL créera un second fichier de PID, généralement à l'extérieur de son répertoire de données.

Des tablespaces :

Ils sont totalement optionnels. Ce sont des espaces de stockage supplémentaires, stockés habituellement dans d'autres systèmes de fichiers.

Le fichier de statistiques d'exécution :

Généralement dans `pg_stat_tmp/`.

Les fichiers de trace :

Typiquement, des fichiers avec une variante du nom `postgresql.log`, souvent datés. Ils sont par défaut dans le répertoire de l'instance, sous `log/`. Sur Debian, ils sont redirigés vers la sortie d'erreur du système. Ils peuvent être redirigés vers un autre mécanisme du système d'exploitation (syslog sous Unix, journal des événements sous Windows),

---

1.6.1 RÉPERTOIRE DE DONNÉES

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

Le répertoire de données est souvent appelé PGDATA, du nom de la variable d'environnement que l'on peut faire pointer vers lui pour simplifier l'utilisation de nombreux utilitaires PostgreSQL. Il est possible aussi de le connaître, une fois connecté à une base de l'instance, en interrogeant le paramètre `data_directory`.

```
SHOW data_directory;

      data_directory
--------------------------
 /var/lib/pgsql/13/data
```

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

Il est recommandé de ne jamais créer ce répertoire PGDATA à la racine d'un point de montage, quel que soit le système d'exploitation et le système de fichiers utilisé. Si un point de montage est dédié à l'utilisation de PostgreSQL, positionnez-le toujours dans un sous-répertoire, voire deux niveaux en dessous du point de montage. (par exemple `<point de montage>/<version majeure>/<nom instance>`).

Voir à ce propos le chapitre *Use of Secondary File Systems* dans la documentation officielle : https://www.postgresql.org/docs/current/creating-cluster.html.

Vous pouvez trouver une description de tous les fichiers et répertoires dans la documentation officielle[2] .

1.6.2 FICHIERS DE CONFIGURATION

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

Les fichiers de configuration sont de simples fichiers textes. Habituellement, ce sont les suivants.

`postgresql.conf` contient une liste de paramètres, sous la forme `paramètre=valeur`. Tous les paramètres sont modifiables (et présents) dans ce fichier. Selon la configuration, il peut inclure d'autres fichiers, mais ce n'est pas le cas par défaut.

`postgresql.auto.conf` stocke les paramètres de configuration fixés en utilisant la commande `ALTER SYSTEM`. Il surcharge donc `postgresql.conf`. Il est déconseillé de le modifier à la main.

`pg_hba.conf` contient les règles d'authentification à la base selon leur identité, la base, la provenance, etc.

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

`pg_ident.conf` est plus rarement utilisé. Il complète `pg_hba.conf`, par exemple pour rapprocher des utilisateurs système ou propres à PostgreSQL.

Leur configuration sera abordée plus tard.

1.6.3 AUTRES FICHIERS DANS PGDATA

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

`PG_VERSION` est un fichier. Il contient en texte lisible la version majeure devant être utilisée pour accéder au répertoire (par exemple `13`). On trouve ces fichiers `PG_VERSION` à de nombreux endroits de l'arborescence de PostgreSQL, par exemple dans chaque répertoire de base du répertoire `PGDATA/base/` ou à la racine de chaque tablespace.

Le fichier `postmaster.pid` est créé au démarrage de PostgreSQL. PostgreSQL y indique le PID du processus père sur la première ligne, l'emplacement du répertoire des données sur la deuxième ligne et la date et l'heure du lancement de postmaster sur la troisième ligne ainsi que beaucoup d'autres informations. Par exemple :

```
~$ cat /var/lib/postgresql/12/data/postmaster.pid
7771
/var/lib/postgresql/12/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/12/data
pos 7773 0 42486  4656   0 16:26 00:00  postgres: checkpointer
pos 7774 0 42486  5044   1 16:26 00:00  postgres: background writer
pos 7775 0 42486  8224   1 16:26 00:00  postgres: walwriter
pos 7776 0 42850  5640   1 16:26 00:00  postgres: autovacuum launcher
pos 7777 0  6227  2328   3 16:26 00:00  postgres: stats collector
pos 7778 0 42559  3684   0 16:26 00:00  postgres: logical replication launcher
```

```
$ ipcs -p |grep  7771
54919263   postgres   7771       10640

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

Le processus père de cette instance PostgreSQL a comme PID le 7771. Ce processus a bien réclamé une sémaphore d'identifiant 54919263. Cette sémaphore correspond à des segments de mémoire partagée pour un total de 56 octets. Le répertoire de données se trouve bien dans `/var/lib/postgresql/12/data`.

Le fichier `postmaster.pid` est supprimé lors de l'arrêt de PostgreSQL. Cependant, ce n'est pas le cas après un arrêt brutal. Dans ce genre de cas, PostgreSQL détecte le fichier et indique qu'il va malgré tout essayer de se lancer s'il ne trouve pas de processus en cours d'exécution avec ce PID. Un fichier supplémentaire peut être créé ailleurs grâce au paramètre `external_pid_file`, c'est notamment le défaut sous Debian :

```
external_pid_file = '/var/run/postgresql/12-main.pid'
```

Par contre, ce fichier ne contient que le PID du processus père.

Quant au fichier `postmaster.opts`, il contient les arguments en ligne de commande correspondant au dernier lancement de PostgreSQL. Il n'est jamais supprimé. Par exemple :

```
$ cat $PGDATA/postmaster.opts
/usr/local/pgsql/bin/postgres "-D" "/var/lib/postgresql/12/data"
```

1.6.4 FICHIERS DE DONNÉES

- `base/` : contient les fichiers de données
  - un sous-répertoire par base de données
  - `pgsql_tmp` : fichiers temporaires
- `global/` : contient les objets globaux à toute l'instance

`base/` contient les fichiers de données (tables, index, vues matérialisées, séquences). Il contient un sous-répertoire par base, le nom du répertoire étant l'OID de la base dans `pg_database`. Dans ces répertoires, nous trouvons un ou plusieurs fichiers par objet à stocker. Ils sont nommés ainsi :

- Le nom de base du fichier correspond à l'attribut `relfilenode` de l'objet stocké, dans la table `pg_class` (une table, un index...). Il peut changer dans la vie de l'objet (par exemple lors d'un `VACUUM FULL`, un `TRUNCATE`...)

- Si le nom est suffixé par un « . » suivi d'un chiffre, il s'agit d'un fichier d'extension de l'objet : un objet est découpé en fichiers de 1 Go maximum.
- Si le nom est suffixé par `_fsm`, il s'agit du fichier stockant la *Free Space Map* (liste des blocs réutilisables).
- Si le nom est suffixé par `_vm`, il s'agit du fichier stockant la *Visibility Map* (liste des blocs intégralement visibles, et donc ne nécessitant pas de traitement par `VACUUM`).

Un fichier `base/1247/14356.1` est donc le second segment de l'objet ayant comme `relfilenode` 14356 dans le catalogue `pg_class`, dans la base d'OID 1247 dans la table `pg_database`.

Savoir identifier cette correspondance ne sert que dans des cas de récupération de base très endommagée. Vous n'aurez jamais, durant une exploitation normale, besoin d'obtenir cette correspondance. Si, par exemple, vous avez besoin de connaître la taille de la table `test` dans une base, il vous suffit d'exécuter la fonction `pg_table_size()`. En voici un exemple complet :

```
CREATE TABLE test (id integer);
INSERT INTO test SELECT generate_series(1, 5000000);
SELECT pg_table_size('test');

 pg_table_size
---------------
     181305344
```

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

Le répertoire `base` peut aussi contenir un répertoire `pgsql_tmp`. Ce répertoire contient des fichiers temporaires utilisés pour stocker les résultats d'un tri ou d'un hachage. À partir de la version 12, il est possible de connaître facilement le contenu de ce répertoire en utilisant la fonction `pg_ls_tmpdir()`, ce qui peut permettre de suivre leur consommation.

Si nous demandons au sein d'une première session :

```
SELECT * FROM generate_series(1,1e9) ORDER BY random() LIMIT 1 ;
```

alors nous pourrons suivre les fichiers temporaires depuis une autre session :

```
SELECT * FROM pg_ls_tmpdir() ;

       name        |    size    |      modification
-------------------+------------+------------------------
 pgsql_tmp12851.16 | 1073741824 | 2020-09-02 15:43:27+02
 pgsql_tmp12851.11 | 1073741824 | 2020-09-02 15:42:32+02
 pgsql_tmp12851.7  | 1073741824 | 2020-09-02 15:41:49+02
 pgsql_tmp12851.5  | 1073741824 | 2020-09-02 15:41:29+02
```

```
pgsql_tmp12851.9  | 1073741824 | 2020-09-02 15:42:11+02
pgsql_tmp12851.0  | 1073741824 | 2020-09-02 15:40:36+02
pgsql_tmp12851.14 | 1073741824 | 2020-09-02 15:43:06+02
pgsql_tmp12851.4  | 1073741824 | 2020-09-02 15:41:19+02
pgsql_tmp12851.13 | 1073741824 | 2020-09-02 15:42:54+02
pgsql_tmp12851.3  | 1073741824 | 2020-09-02 15:41:09+02
pgsql_tmp12851.1  | 1073741824 | 2020-09-02 15:40:47+02
pgsql_tmp12851.15 | 1073741824 | 2020-09-02 15:43:17+02
pgsql_tmp12851.2  | 1073741824 | 2020-09-02 15:40:58+02
pgsql_tmp12851.8  | 1073741824 | 2020-09-02 15:42:00+02
pgsql_tmp12851.12 | 1073741824 | 2020-09-02 15:42:43+02
pgsql_tmp12851.10 | 1073741824 | 2020-09-02 15:42:21+02
pgsql_tmp12851.6  | 1073741824 | 2020-09-02 15:41:39+02
pgsql_tmp12851.17 |  546168976 | 2020-09-02 15:43:32+02
```

Le répertoire `global/` contient notamment les objets globaux à toute une instance, comme la table des bases de données, celle des rôles ou celle des tablespaces ainsi que leurs index.

1.6.5 FICHIERS LIÉS AUX TRANSACTIONS

- `pg_wal/` : journaux de transactions
  - `pg_xlog/` avant la v10
  - sous-répertoire `archive_status`
  - nom : *timeline*, journal, segment
  - ex : `00000002 00000142 000000FF`
- `pg_xact/` : état des transactions
  - `pg_clog/` avant la v10
- mais aussi : `pg_commit_ts/`, `pg_multixact/`, `pg_serial/` `pg_snapshots/`, `pg_subtrans/`, `pg_twophase/`
- **Ces fichiers sont vitaux !**

Le répertoire `pg_wal` contient les journaux de transactions. Ces journaux garantissent la durabilité des données dans la base, en traçant toute modification devant être effectuée **AVANT** de l'effectuer réellement en base.

Les fichiers contenus dans `pg_wal` ne doivent **jamais** être effacés manuellement. Ces fichiers sont cruciaux au bon fonctionnement de la base. PostgreSQL gère leur création et suppression. S'ils sont toujours présents, c'est que PostgreSQL en a besoin.

Par défaut, les fichiers des journaux font tous 16 Mo. Ils ont des noms sur 24 caractères, comme par exemple :

```
$ ls -l
```

```
total 2359320
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007C
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007D
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007E
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007F
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000000
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000001
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000002
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000003
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000001D
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000001E
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000001F
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001430000020
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 00000002000001430000021
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 00000002000001430000022
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 00000002000001430000023
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 00000002000001430000024
drwx------ 2 postgres postgres    16384 Mar 26 16:28 archive_status
```

La première partie d'un nom de fichier (ici `00000002`) correspond à la *timeline* (« ligne de temps »), qui ne s'incrémente que lors d'une restauration de sauvegarde ou une bascule entre serveurs primaire et secondaire. La deuxième partie (ici `00000142` puis `00000143`) correspond au numéro de journal à proprement parler, soit un ensemble de fichiers représentant 4 Go. La dernière partie correspond au numéro du segment au sein de ce journal. Selon la taille du segment fixée à l'initialisation, il peut aller de `00000000` à `000000FF` (256 segments de 16 Mo, configuration par défaut, soit 4 Go), à `00000FFF` (4096 segments de 1 Mo), ou à `0000007F` (128 segments de 32 Mo, exemple ci-dessus), etc. Une fois ce maximum atteint, le numéro de journal au centre est incrémenté et les numéros de segments reprennent à zéro.

L'ordre d'écriture des journaux est numérique (en hexadécimal), et leur archivage doit suivre cet ordre. Il ne faut pas se fier à la date des fichiers pour le tri : pour des raisons de performances, PostgreSQL recycle généralement les fichiers en les renommant. Dans l'exemple ci-dessus, le dernier journal écrit est `00000002000001430000020` et non `00000002000001430000024`. À partir de la version 12, ce mécanisme peut toutefois être désactivé en passant `wal_recycle` à `off` (ce qui a un intérêt sur certains systèmes de fichiers comme ZFS).

Dans le cadre d'un archivage PITR et/ou d'une réplication par *log shipping*, le sous-répertoire `pg_wal/archive_status` indique l'état des journaux dans le contexte de l'archivage. Les fichiers `.ready` indiquent les journaux restant à archiver (normalement peu nombreux), les `.done` ceux déjà archivés.

À partir de la version 12, il est possible de connaître facilement le contenu de ce répertoire en utilisant la fonction `pg_ls_archive_statusdir()` :

```
# SELECT * FROM pg_ls_archive_statusdir() ORDER BY 1 ;

              name               | size |      modification
---------------------------------+------+------------------------
 000000010000000000000067.done   |    0 | 2020-09-02 15:52:57+02
 000000010000000000000068.done   |    0 | 2020-09-02 15:52:57+02
 000000010000000000000069.done   |    0 | 2020-09-02 15:52:58+02
 00000001000000000000006A.ready  |    0 | 2020-09-02 15:53:53+02
 00000001000000000000006B.ready  |    0 | 2020-09-02 15:53:53+02
 00000001000000000000006C.ready  |    0 | 2020-09-02 15:53:54+02
 00000001000000000000006D.ready  |    0 | 2020-09-02 15:53:54+02
 00000001000000000000006E.ready  |    0 | 2020-09-02 15:53:54+02
 00000001000000000000006F.ready  |    0 | 2020-09-02 15:53:54+02
 000000010000000000000070.ready  |    0 | 2020-09-02 15:53:55+02
 000000010000000000000071.ready  |    0 | 2020-09-02 15:53:55+02
```

Le répertoire `pg_xact` contient l'état de toutes les transactions passées ou présentes sur la base (validées, annulées, en sous-transaction ou en cours), comme nous le détaillerons dans le module « Mécanique du moteur transactionnel ».

Les fichiers contenus dans le répertoire `pg_xact` ne doivent **jamais** être effacés. Ils sont cruciaux au bon fonctionnement de la base.

D'autres répertoires contiennent des fichiers essentiels à la gestion des transactions :

- `pg_commit_ts` contient l'horodatage de la validation de chaque transaction ;
- `pg_multixact` est utilisé dans l'implémentation des verrous partagés (`SELECT ... FOR SHARE`) ;
- `pg_serial` est utilisé dans l'implémentation de SSI (`Serializable Snapshot Isolation`) ;
- `pg_snapshots` est utilisé pour stocker les snapshots exportés de transactions ;
- `pg_subtrans` stocke l'imbrication des transactions lors de sous-transactions (les `SAVEPOINTS`) ;
- `pg_twophase` est utilisé pour l'implémentation du *Two-Phase Commit*, aussi appelé `transaction préparée`, `2PC`, ou transaction `XA` dans le monde Java par exemple.

La version 10 a été l'occasion du changement de nom de quelques répertoires pour des raisons de cohérence et pour réduire les risques de fausses manipulations. Jusqu'en 9.6, `pg_wal` s'appelait `pg_xlog`, `pg_xact` s'appelait `pg_clog`.
Les fonctions et outils ont été renommés en conséquence :

- dans les noms de fonctions et d'outils, `xlog` a été remplacé par `wal` (par exemple `pg_switch_xlog` est devenue `pg_switch_wal`) ;

- toujours dans les fonctions, `location` a été remplacé par `lsn`.

1.6.6 FICHIERS LIÉS À LA RÉPLICATION

- `pg_logical/`
- `pg_repslot/`

`pg_logical` contient des informations sur la réplication logique.

`pg_replslot` contient des informations sur les slots de réplications, qui sont un moyen de fiabiliser la réplication physique ou logique.

Sans réplication en place, ces répertoires sont quasi-vides. Là encore, il ne faut pas toucher à leur contenu.

1.6.7 RÉPERTOIRE DES TABLESPACES

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

Par défaut, `pg_tblspc` est vide. Il contient des liens symboliques vers des répertoires définis comme *tablespaces*, c'est-à-dire des espaces de stockage extérieurs. Chaque lien symbolique a comme nom l'OID du tablespace (table système `pg_tablespace`).

Ils sont totalement optionnels. Leur utilité et leur gestion seront abordés plus loin.

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.

1.6.8 FICHIERS DES STATISTIQUES D'ACTIVITÉ

- `pg_stat/`
- `pg_stat_tmp/`

`pg_stat_tmp` est le répertoire par défaut de stockage des statistiques d'activité de PostgreSQL, comme les entrées-sorties ou les opérations de modifications sur les tables. Ces fichiers pouvant générer une grande quantité d'entrées-sorties, l'emplacement du répertoire peut être modifié avec le paramètre `stats_temp_directory`. 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 `pg_stat/`.

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

```
SHOW stats_temp_directory;

          stats_temp_directory
---------------------------------------
 /var/run/postgresql/12-main.pg_stat_tmp
```

1.6.9 AUTRES RÉPERTOIRES

- `pg_dynshmem/`
- `pg_notify/`

`pg_dynshmem` est utilisé par les extensions utilisant de la mémoire partagée dynamique.

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

1.6.10 LES FICHIERS DE TRACES (JOURNAUX)

- Fichiers texte traçant l'activité
- Très paramétrables
- Gestion des fichiers soit :
  - par PostgreSQL
  - délégués au système d'exploitation (*syslog*, *eventlog*)

Le paramétrage des journaux est très fin. Leur configuration est le sujet est évoquée dans notre première formation[3] .

Si `logging_collector` est activé, c'est-à-dire que PostgreSQL collecte lui-même ses traces, l'emplacement de ces journaux se paramètre grâce aux paramètres `log_directory`, le répertoire où les stocker, et `log_filename`, le nom de fichier à utiliser, ce nom pouvant utiliser des échappements comme `%d` pour le jour de la date, par exemple. Les droits attribués au fichier sont précisés par le paramètre `log_file_mode`.

Un exemple pour `log_filename` avec date et heure serait :

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

La liste des échappements pour le paramètre `log_filename` est disponible dans la page de manuel de la fonction `strftime` sur la plupart des plateformes de type UNIX.

[3] https://dali.bo/h1_html

1.7 CONCLUSION

- PostgreSQL est complexe, avec de nombreux composants;
- Une bonne compréhension de cette architecture est la clé d'une bonne administration.

1.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.8 QUIZ

https://dali.bo/m1_quiz

1.9 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 (similaire à Red Hat 8), Red Hat/CentOS 7, et Debian/Ubuntu.

Elle ne dure que quelques minutes.

1.9.1 SUR ROCKY LINUX 8

Installation du dépôt communautaire :

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent peuvent être générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 14) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Il faut installer le dépôt et désactiver le module PostgreSQL par défaut :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 14 :

```
# dnf install -y postgresql14-server postgresql14-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Tout à fait optionnellement, une fonctionnalité avancée, le JIT (*Just In Time compilation*), nécessite un paquet séparé, qui lui-même nécessite des paquets du dépôt EPEL :

```
# dnf install postgresql14-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-14/bin/postgresql-14-setup initdb
# cat /var/lib/pgsql/14/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé.

Chemins :

Objet	**Chemin**
Binaires	`/usr/pgsql-14/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/14/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

```
# systemctl start postgresql-14
# systemctl stop postgresql-14
# systemctl reload postgresql-14
```

Test rapide de bon fonctionnement

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au démarrage du système d'exploitation :

```
# systemctl enable postgresql-14
```

Consultation de l'état de l'instance :

```
# systemctl status postgresql-14
```

Ouverture du *firewall* pour le port 5432 :

Si le *firewall* est actif (dans le doute, consulter `systemctl status firewalld`) :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faudra installer les binaires pour chacune, et l'instance par défaut de chaque version vivra dans un sous-répertoire différent de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf`.

Si plusieurs instances d'une même version majeure (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faudra les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un sous-répertoire de `/var/lib/pgsqsl/14/` (ou l'équivalent pour d'autres versions majeures).
- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-14.service \
       /etc/systemd/system/postgresql-14-secondaire.service
```

- Modification du fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/14/secondaire
```

- Option 1 : création d'une nouvelle instance vierge :

```
# /usr/pgsql-14/bin/postgresql-14-setup initdb postgresql-14-secondaire
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `postgresql.conf` (port !), `recovery.conf`...
- Commandes de maintenance :

```
# systemctl [start|stop|reload|status] postgresql-14-secondaire
# systemctl [enable|disable] postgresql-14-secondaire
```

- Ouvrir un port dans le firewall au besoin.

1.9.2 SUR RED HAT 7 / CENT OS 7

Fondamentalement, le principe reste le même qu'en version 8. Il faudra utiliser `yum` plutôt que `dnf`.

ATTENTION : Red Hat et CentOS 6 et 7 fournissent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` !
L'utilisation des dépôts du PGDG est donc obligatoire.

Il n'y a pas besoin de désactiver de module AppStream.

Le JIT (*Just In Time compilation*), nécessite un paquet séparé, qui lui-même nécessite des paquets du dépôt EPEL :

```
# yum install epel-release
# yum install postgresql14-llvmjit
```

La création de l'instance et la suite sont identiques.

1.9.3 SUR DEBIAN / UBUNTU

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Installation du dépôt communautaire :

Référence : https://apt.postgresql.org/

- Import des certificats et de la clé :

```
# apt install curl ca-certificates gnupg
# curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key add -
```

- Création du fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` (ici pour Debian 11 « bullseye » ; adapter au nom de code de la version de Debian ou Ubuntu correspondante : **stretch**, **bionic**, **focal**…) :

```
deb http://apt.postgresql.org/pub/repos/apt bullseye-pgdg main
```

Installation de PostgreSQL 14 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

```
# apt update
# apt install postgresql-common
```

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-14 postgresql-client-14
```

(Pour les versions 9.x, installer aussi le paquet `postgresql-contrib-9.x`).

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin

Binaires	`/usr/lib/postgresql/14/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/14/main`
Fichiers de configuration	dans `/etc/postgresql/14/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils :

```
# pg_ctlcluster 14 main [start|stop|reload|status]
```

Démarrage de l'instance au lancement :

C'est en place par défaut, et modifiable dans `/etc/postgresql/14/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 14 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- les paramètres de création d'instance dans `/etc/postgresql-common/createcluster.conf` peuvent être modifiés, par exemple ici pour : les *checksums*, les messages en anglais, l'authentification sécurisée, le format des traces et un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C --auth-host=scram-sha-256 --auth-local=peer'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- création de l'instance, avec possibilité là aussi de préciser certains paramètres du `postgresql.conf` voire de modifier les chemins des fichiers (déconseillé si vous pouvez l'éviter) :

```
# pg_createcluster 14 secondaire \
  --port=5433 \
  --datadir=/PGDATA/11/basedecisionnelle \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/11/basedecisionnelle/journaux
```

- démarrage :

```
# pg_ctlcluster 14 secondaire start
```

1.9.4 ACCÈS À L'INSTANCE

Par défaut, l'instance n'est accessible que par l'utilisateur système `postgres`, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (14.0)
Saisissez « help » pour l'aide.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (14.0)
Saisissez « help » pour l'aide.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, mise en place d'une authentification par mot de passe (`md5` par défaut) pour les accès à `localhost` :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
```

```
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

(une authentification `scram-sha-256` est plus conseillée mais elle impose que `password_encryption` soit à cette valeur dans `postgresql.conf` avant de définir les mots de passe).

- ajout d'un mot de passe à l'utilisateur `postgres` de l'instance ;

```
dalibo:~$ sudo -iu postgres psql
psql (14.0)
Saisissez « help » pour l'aide.
postgres=# \password
Saisissez le nouveau mot de passe :
Saisissez-le à nouveau :
postgres=# \q

dalibo:~$ psql -h localhost -U postgres
Mot de passe pour l'utilisateur postgres :
psql (14.0)
Saisissez « help » pour l'aide.
postgres=#
```

- pour se connecter sans taper le mot de passe, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

- ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/14/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` implique de recharger la configuration par une de ces trois méthodes selon le système :

```
root:~# systemctl reload postgresql-14

root:~# pg_ctlcluster 14 main reload
```

```
postgres:~$ psql -c 'SELECT pg_reload_conf();'
```

1.10 TRAVAUX PRATIQUES

1.10.1 PROCESSUS

> Si ce n'est pas déjà fait, démarrer l'instance PostgreSQL.

> Lister les processus du serveur PostgreSQL. Qu'observe-t-on ?

> Se connecter à l'instance PostgreSQL.

> Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

> Créer une nouvelle base de données nommée `b0`.

> Se connecter à la base de données `b0` et créer une table `t1` avec une colonne `id` de type integer.
> Insérer 10 millions de lignes dans la table `t1` avec
> `INSERT INTO t1 SELECT generate_series(1, 10000000);`

> Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

> Configurer la valeur du paramètre `max_connections` à `15`.

> Redémarrer l'instance PostgreSQL.

> Vérifier que la modification de la valeur du paramètre `max_connections` a été prise en compte.

> Se connecter 15 fois à l'instance PostgreSQL sans fermer les sessions, par exemple en lançant plusieurs fois `psql -c 'SELECT pg_sleep(1000)' &`.

> Se connecter une seizième fois à l'instance PostgreSQL. Qu'observe-t-on ?

> Configurer la valeur du paramètre `max_connections` à sa valeur initiale.

1.10.2 FICHIERS

> Aller dans le répertoire des données de l'instance PostgreSQL.
> Lister les fichiers.

> Aller dans le répertoire `base`.
> Lister les fichiers.

> À quelle base est lié chaque répertoire présent dans le répertoire `base` ? (*cf* `oid2name` ou `pg_database`)

> Créer une nouvelle base de données nommée `b1`. Qu'observe-t-on dans le répertoire `base` ?

> Se connecter à la base de données `b1`. Créer une table `t1` avec une colonne `id` de type integer.

> Récupérer le chemin vers le fichier correspondant à la table `t1` (il existe une fonction `pg_relation_filepath`).

> Regarder la taille du fichier correspondant à la table `t1`. Pourquoi est-il vide ?

> Insérer une ligne dans la table `t1`. Quelle taille fait le fichier de la table `t1` ?

> Insérer 500 lignes dans la table `t1` avec `generate_series`. Quelle taille fait le fichier de la table `t1` ?

> Pourquoi cette taille pour simplement 501 entiers de 4 octets chacun ?

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

1.11.1 PROCESSUS

> Si ce n'est pas déjà fait, démarrer l'instance PostgreSQL.

Sous Rocky Linux 8, CentOS ou Red Hat 7 en tant qu'utilisateur **root** :

```
# systemctl start postgresql-14
```

> Lister les processus du serveur PostgreSQL. Qu'observe-t-on ?

En tant qu'utilisateur **postgres** :

```
$ ps -o pid,cmd fx
  PID CMD
24009 ps -o pid,cmd fx
 3562 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/data/
 3624  \_ postgres: logger
 3642  \_ postgres: checkpointer
 3643  \_ postgres: background writer
 3644  \_ postgres: walwriter
 3645  \_ postgres: autovacuum launcher
 3646  \_ postgres: stats collector
 3647  \_ postgres: logical replication launcher
```

> Se connecter à l'instance PostgreSQL.

```
$ psql postgres
psql (14.1)
Type "help" for help.

postgres=#
```

> Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

```
$ ps -o pid,cmd fx
  PID CMD
 2031 -bash
 2326  \_ psql postgres
 1792 -bash
 2328  \_ ps -o pid,cmd fx
 1992 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/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
```

Il y a un nouveau processus (ici PID 2327) qui va gérer l'exécution des requêtes du client psql.

> Créer une nouvelle base de données nommée `b0`.

Depuis le shell, en tant que **postgres** :

```
$ createdb b0
```

Alternativement, depuis la session déjà ouverte dans `psql` :

```
CREATE DATABASE b0;
```

> Se connecter à la base de données `b0` et créer une table `t1` avec une colonne `id` de type integer.
> Insérer 10 millions de lignes dans la table `t1` avec
> `INSERT INTO t1 SELECT generate_series(1, 10000000);`

Pour se connecter depuis le shell :

```
psql b0
```

ou depuis la session `psql` :

```
\c b0
```

Création de la table :

```
b0=# CREATE TABLE t1 (id integer);
CREATE TABLE
b0=# INSERT INTO t1 SELECT generate_series(1, 10000000);
INSERT 0 10000000
```

> Dans un autre terminal lister de nouveau les processus du serveur PostgreSQL. Qu'observe-t-on ?

```
$ ps -o pid,cmd fx
  PID CMD
 2031 -bash
 2326  \_ psql postgres
 1792 -bash
 2363  \_ ps -o pid,cmd fx
 1992 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/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] INSERT
```

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

> Configurer la valeur du paramètre `max_connections` à `15`.

Pour cela, il faut ouvrir le fichier de configuration `postgresql.conf` et modifier la valeur du paramètre `max_connections` à 15.

Alternativement :

```
ALTER SYSTEM SET max_connections TO 15 ;
```

Ce dernier ordre écrira dans le fichier `/var/lib/pgsql/14/data/postgresql.auto.conf`.

Cependant, la prise en compte n'est pas automatique. Pour ce paramètre, il faut redémarrer l'instance PostgreSQL.

> Redémarrer l'instance PostgreSQL.

En tant qu'utilisateur **root** :

```
# systemctl restart postgresql-14
```

> Vérifier que la modification de la valeur du paramètre `max_connections` a été prise en compte.

```
postgres=# SHOW max_connections ;
```

```
 max_connections
-----------------
 15
```

> Se connecter 15 fois à l'instance PostgreSQL sans fermer les sessions, par exemple en lançant plusieurs fois `psql -c 'SELECT pg_sleep(1000)' &`.

Il est possible de le faire manuellement ou de l'automatiser avec ce petit script shell :

```
$ for i in $(seq 1 15); do psql -c "SELECT pg_sleep(1000);" postgres & done
[1] 998
[2] 999
...
[15] 1012
```

> Se connecter une seizième fois à l'instance PostgreSQL. Qu'observe-t-on ?

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

Il est impossible de se connecter une fois que le nombre de connexions a atteint sa limite configurée avec `max_connections`. Il faut donc attendre que les utilisateurs se déconnectent pour accéder de nouveau au serveur.

> Configurer la valeur du paramètre `max_connections` à sa valeur initiale.

Dans le fichier de configuration `postgresql.conf`, restaurer la valeur du paramètre `max_connections` à 100.

Si l'autre méthode `ALTER SYSTEM` a été utilisée, dans le fichier de configuration `/var/lib/pgsql/14/data/postgresql.auto.conf`, supprimer la ligne avec le paramètre `max_connections` puis redémarrer l'instance PostgreSQL.

Il est déconseillé de modifier `postgresql.auto.conf` à la main, mais pour le TP nous nous permettons quelques libertés.

Toutefois si l'instance est démarrée et qu'il est encore possible de s'y connecter, le plus propre est ceci :

```
ALTER SYSTEM RESET max_connections ;
```

Puis redémarrer PostgreSQL : toutes les connexions en cours vont être coupées.

```
# systemctl restart postgresql-14
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
[...]
FATAL:  terminating connection due to administrator command
server closed the connection unexpectedly
    This probably means the server terminated abnormally
    before or while processing the request.
connection to server was lost
```

Il est à présent possible de se reconnecter. Vérifier que cela a été pris en compte :

```
postgres=# SHOW max_connections ;

 max_connections
-----------------
 100
```

1.11.2 FICHIERS

> Aller dans le répertoire des données de l'instance PostgreSQL.
> Lister les fichiers.

En tant qu'utilisateur système **postgres** :

```
 echo $PGDATA
/var/lib/pgsql/14/data

$ cd $PGDATA

$ ls -al
total 72
drwx------ 20 postgres postgres  4096 Apr 16 15:33 .
drwx------  4 postgres postgres    48 Apr 16 15:01 ..
drwx------  7 postgres postgres    66 Apr 16 15:47 base
-rw-------  1 postgres postgres    30 Apr 16 15:33 current_logfiles
drwx------  2 postgres postgres  4096 Apr 16 15:34 global
drwx------  2 postgres postgres    31 Apr 16 15:01 log
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_commit_ts
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_dynshmem
-rw-------  1 postgres postgres  4548 Apr 16 15:01 pg_hba.conf
-rw-------  1 postgres postgres  1636 Apr 16 15:01 pg_ident.conf
drwx------  4 postgres postgres    65 Apr 16 15:48 pg_logical
```

```
drwx------  4 postgres postgres    34 Apr 16 15:01 pg_multixact
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_notify
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_replslot
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_serial
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_snapshots
drwx------  2 postgres postgres     6 Apr 16 15:33 pg_stat
drwx------  2 postgres postgres    80 Apr 16 16:08 pg_stat_tmp
drwx------  2 postgres postgres    17 Apr 16 15:01 pg_subtrans
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_tblspc
drwx------  2 postgres postgres     6 Apr 16 15:01 pg_twophase
-rw-------  1 postgres postgres     3 Apr 16 15:01 PG_VERSION
drwx------  3 postgres postgres  4096 Apr 16 15:46 pg_wal
drwx------  2 postgres postgres    17 Apr 16 15:01 pg_xact
-rw-------  1 postgres postgres    88 Apr 16 15:17 postgresql.auto.conf
-rw-------  1 postgres postgres 28007 Apr 16 15:01 postgresql.conf
-rw-------  1 postgres postgres    58 Apr 16 15:33 postmaster.opts
-rw-------  1 postgres postgres   103 Apr 16 15:33 postmaster.pid
```

> Aller dans le répertoire `base`.
> Lister les fichiers.

```
$ cd base

$ ls -al
total 44
drwx------  7 postgres postgres   66 Apr 16 15:47 .
drwx------ 20 postgres postgres 4096 Apr 16 15:33 ..
drwx------  2 postgres postgres 8192 Apr 16 15:01 1
drwx------  2 postgres postgres 8192 Apr 16 15:01 14173
drwx------  2 postgres postgres 8192 Apr 16 15:34 14174
drwx------  2 postgres postgres 8192 Apr 16 15:42 16386
drwx------  2 postgres postgres    6 Apr 16 15:58 pgsql_tmp
```

> À quelle base est lié chaque répertoire présent dans le répertoire `base` ? (*cf* `oid2name` ou `pg_database`)

Chaque répertoire correspond à une base de données. Le numéro indiqué est un identifiant système (OID). Il existe deux moyens pour récupérer cette information :

- directement dans le catalogue système `pg_database` :

```
$ psql postgres

psql (14.1)
Type "help" for help.
```

```
postgres=# SELECT oid, datname FROM pg_database;

  oid  |  datname
-------+-----------
 14174 | postgres
     1 | template1
 14173 | template0
 16386 | b0
```

- avec l'outil `oid2name` (à installer au besoin via le paquet `postgresql14-contrib`) :

```
$ /usr/pgsql-14/bin/oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  16386            b0  pg_default
  14174      postgres  pg_default
  14173     template0  pg_default
      1     template1  pg_default
```

Donc ici, le répertoire `1` correspond à la base `template1`, et le répertoire `14174` à la base `postgres` (ces nombres peuvent changer suivant l'installation).

> Créer une nouvelle base de données nommée `b1`. Qu'observe-t-on dans le répertoire `base` ?

```
$ createdb b1

$ ls -al
total 64
drwx------  8 postgres postgres   78 Apr 16 16:21 .
drwx------ 20 postgres postgres 4096 Apr 16 15:33 ..
drwx------  2 postgres postgres 8192 Apr 16 15:01 1
drwx------  2 postgres postgres 8192 Apr 16 15:01 14173
drwx------  2 postgres postgres 8192 Apr 16 15:34 14174
drwx------  2 postgres postgres 8192 Apr 16 15:42 16386
drwx------  2 postgres postgres 8192 Apr 16 16:21 16393
drwx------  2 postgres postgres    6 Apr 16 15:58 pgsql_tmp
```

Un nouveau sous-répertoire est apparu, nommé `16393`. Il correspond bien à la base `b1` d'après `oid2name`.

> Se connecter à la base de données `b1`. Créer une table `t1` avec une colonne `id` de type integer.

```
$ psql b1
```

```
psql (14.1)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);

CREATE TABLE
```

> Récupérer le chemin vers le fichier correspondant à la table `t1` (il existe une fonction `pg_relation_filepath`).

La fonction a pour définition :

```
b1=# \df pg_relation_filepath
                                 List of functions
   Schema   |         Name          | Result data type | Argument data types | Type
------------+-----------------------+------------------+---------------------+------
 pg_catalog | pg_relation_filepath | text             | regclass            | func
```

L'argument `regclass` peut être l'OID de la table, ou son nom.

L'emplacement du fichier sur le disque est donc :

```
b1=# SELECT current_setting('data_directory') || '/' || pg_relation_filepath('t1') AS chemin;

                chemin
---------------------------------------
 /var/lib/pgsql/14/data/base/16393/16398
```

> Regarder la taille du fichier correspondant à la table `t1`. Pourquoi est-il vide ?

```
$ ls -l /var/lib/pgsql/14/data/base/16393/16398
-rw------- 1 postgres postgres 0 Apr 16 16:31 /var/lib/pgsql/14/data/base/16393/16398
```

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

> Insérer une ligne dans la table `t1`. Quelle taille fait le fichier de la table `t1` ?

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

$ ls -l /var/lib/pgsql/14/data/base/16393/16398
-rw------- 1 postgres postgres 8192 Apr 16 16:32 /var/lib/pgsql/14/data/base/16393/16398
```

Il fait à présent 8 ko. En fait, PostgreSQL travaille par blocs de 8 ko. Si une ligne ne peut être placée dans un espace libre d'un bloc existant, un bloc entier est ajouté à la table.

Vous pouvez consulter le fichier avec la commande `hexdump -x <nom du fichier>` (faites un `CHECKPOINT` avant pour être sûr qu'il soit écrit sur le disque).

> Insérer 500 lignes dans la table `t1` avec `generate_series`. Quelle taille fait le fichier de la table `t1` ?

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

$ ls -l /var/lib/pgsql/14/data/base/16393/16398
-rw------- 1 postgres postgres 24576 Apr 16 16:34 /var/lib/pgsql/14/data/base/16393/16398
```

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

> Pourquoi cette taille pour simplement 501 entiers de 4 octets chacun ?

Nous avons enregisté 501 entiers dans la table. Un entier de type `int4` prend 4 octets. Donc nous avons 2004 octets de données utilisateurs. Et pourtant, nous arrivons à un fichier de 24 ko.

En fait, PostgreSQL enregistre aussi dans chaque bloc des informations systèmes en plus des données utilisateurs. Chaque bloc contient un en-tête, des pointeurs, et l'ensemble des lignes du bloc. Chaque ligne contient les colonnes utilisateurs mais aussi des colonnes système. La requête suivante permet d'en savoir plus sur les colonnes présentes dans la table :

```
b1=# SELECT CASE WHEN attnum<0 THEN 'systeme' ELSE 'utilisateur' END AS type,
     attname, attnum, typname, typlen,
     sum(typlen) OVER (PARTITION BY attnum<0) AS longueur_tot
     FROM pg_attribute a
     JOIN pg_type t ON t.oid=a.atttypid
     WHERE attrelid ='t1'::regclass
     ORDER BY attnum;

    type     | attname  | attnum | typname | typlen | sum
-------------+----------+--------+---------+--------+-----
 systeme     | tableoid |     -6 | oid     |      4 |          26
 systeme     | cmax     |     -5 | cid     |      4 |          26
 systeme     | xmax     |     -4 | xid     |      4 |          26
 systeme     | cmin     |     -3 | cid     |      4 |          26
 systeme     | xmin     |     -2 | xid     |      4 |          26
```

```
 systeme     | ctid     |     -1 | tid     |      6 |           26
 utilisateur | id       |      1 | int4    |      4 |            4
```

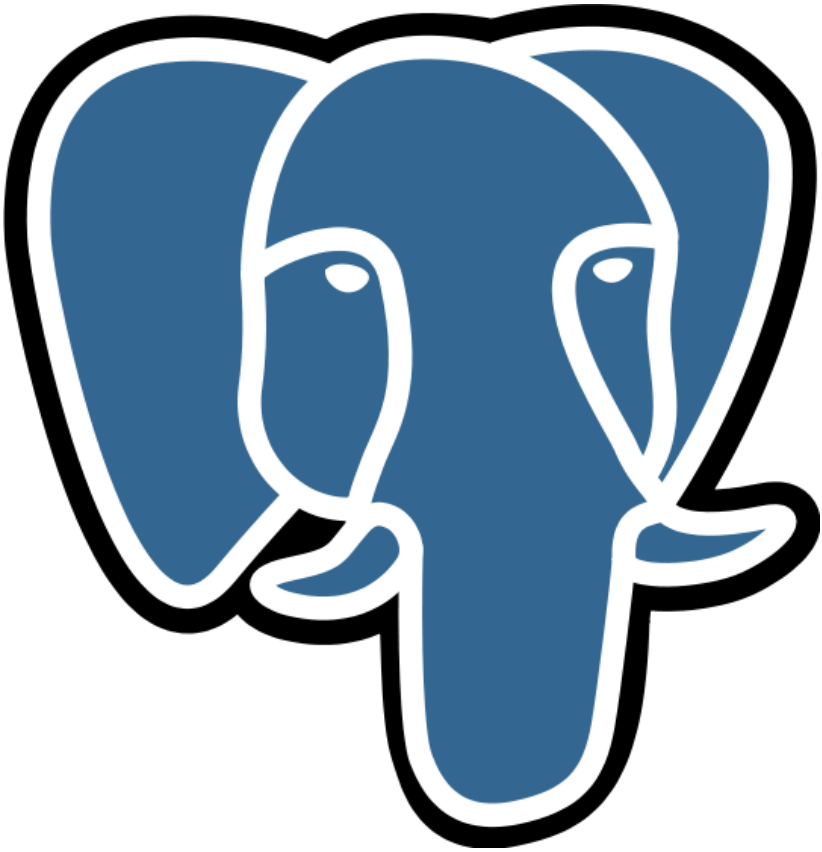
Vous pouvez voir ces colonnes système en les appelant explicitement :

```
SELECT cmin, cmax, xmin, xmax, ctid, *
FROM t1 ;
```

L'en-tête de chaque ligne pèse 26 octets dans le cas général (avant PostgreSQL 12, un éventuel champ `oid` pouvait ajouter 4 octets). Dans notre cas très particulier avec une seule petite colonne, c'est très défavorable mais ce n'est généralement pas le cas.

Avec 501 lignes de 26+4 octets, nous obtenons 15 ko. Chaque bloc possède quelques informations de maintenance : nous dépassons alors 16 ko, ce qui explique pourquoi nous en sommes à 24 ko (3 blocs).

2 CONFIGURATION DE POSTGRESQL



2.1 AU MENU

- Les paramètres en lecture seule
- Les différents fichiers de configuration
  - survol du contenu
- Quelques paramétrages importants :
  - tablespaces
  - connexions

- statistiques
- optimiseur

2.2 PARAMÈTRES EN LECTURE SEULE

- Options de compilation ou lors d'`initdb`
- Quasiment jamais modifiés
- Tailles de bloc ou de fichier
  - `block_size` : 8 ko
  - `wal_block_size` : 8 ko
  - `segment_size` : 1 Go
  - `wal_segment_size` : 16 Mo (option `--wal-segsize` d'`initdb` en v11)

Ces paramètres sont en lecture seule, mais peuvent être consultés par la commande `SHOW`, ou en interrogeant la vue `pg_settings`. Il est possible aussi d'obtenir l'information via la commande `pg_controldata`.

- `block_size` est la taille d'un bloc de données de la base, par défaut 8192 octets ;
- `wal_block_size` est la taille d'un bloc de journal, par défaut 8192 octets ;
- `segment_size` est la taille maximum d'un fichier de données, par défaut 1 Go ;
- `wal_segment_size` est la taille d'un fichier de journal de transactions (WAL), par défaut 16 Mo.

Ces paramètres sont tous fixés à la compilation, sauf `wal_segment_size` à partir de la version 11 : `initdb` accepte alors l'option `--wal-segsize` et l'on peut monter la taille des journaux de transactions à 1 Go. Cela n'a d'intérêt que pour des instances générant énormément de journaux.

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

2.3 FICHIERS DE CONFIGURATION

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

Les fichiers de configuration sont habituellement les 4 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 nous déciderons de nous reposer sur un mécanisme d'authentification extérieur à la base (identification par le système ou par un annuaire par exemple) ;
- `postgresql.auto.conf` : il stocke les paramètres de configuration fixés en utilisant la commande `ALTER SYSTEM` et surcharge donc `postgresql.conf`.

2.4 POSTGRESQL.CONF

Fichier principal de configuration :

- Emplacement :
  - défaut/Red Hat & dérivés : répertoires des données (`/var/lib/…`)
  - Debian : `/etc/postgresql/<version>/<nom>/postgresql.conf`
- Format `clé = valeur`
- Sections, commentaires (redémarrage !)

C'est le fichier le plus important. Il contient le paramétrage de l'instance. PostgreSQL le cherche au démarrage dans le PGDATA. Par défaut, dans les versions compilées, ou depuis les paquets sur Red Hat, CentOS ou Rocky Linux, il sera dans le répertoire principal avec les données (`/var/lib/pgsql/14/data/postgresql.conf` par exemple). Debian le place dans `/etc` (`/etc/postgresql/14/main/postgresql.conf` pour l'instance par défaut).

Dans le doute, il est possible de consulter la valeur du paramètre `config_file`, ici dans la configuration par défaut sur Rocky Linux :

```
# SHOW config_file;

               config_file
------------------------------------------
 /var/lib/postgresql/14/data/postgresql.conf
```

Ce fichier contient un paramètre par ligne, sous le format :

```
clé = valeur
```

Les commentaires commencent par « # » (croisillon) et les chaînes de caractères doivent être encadrées de « ' » (*single quote*). Par exemple :

```
data_directory = '/var/lib/postgresql/14/main'
listen_addresses = 'localhost'
port = 5432
shared_buffers = 128MB
```

Les valeurs de ce fichier ne seront pas forcément les valeurs actives !

Nous allons en effet voir que l'on peut les surcharger.

2.4.1 SURCHARGE DES PARAMÈTRES DE POSTGRESQL.CONF

- `pg_ctl`
- Inclusion externe : `include`, `include_if_exists`
- Surcharge :
  - `postgresql.auto.conf` (`ALTER SYSTEM SET …`)
  - paramètres de `pg_ctl`
  - `ALTER DATABASE | ROLE … SET paramètre = …` et session
  - `SET` / `SET LOCAL`
- Consulter :
  - `SHOW`
  - `pg_settings`
  - `pg_file_settings`

En effet, si des options sont passées en arguments à `pg_ctl`, elles seront prises en compte en priorité par rapport à celles du fichier de configuration.

Nous pouvons aussi inclure d'autres fichiers dans le fichier `postgresql.conf` grâce à l'une de ces directives :

```
include = 'nom_fichier'
include_if_exists = 'nom_fichier'
include_dir = 'répertoire'       # contient des fichiers .conf
```

Le ou les fichiers indiqués sont alors inclus à l'endroit où la directive est positionnée. Avec `include`, si le fichier n'existe pas, une erreur `FATAL` est levée ; au contraire la directive `include_if_exists` permet de ne pas s'arrêter si le fichier n'existe pas. Ces directives permettent notamment des ajustements de configuration propres à plusieurs machines

d'un ensemble primaire/secondaires dont le `postgresql.conf` de base est identique, ou de gérer la configuration hors de `postgresql.conf`.

Si des paramètres sont répétés dans `postgresql.conf`, éventuellement suite à des inclusions, la dernière occurrence écrase les précédentes. Si un paramètre est absent, la valeur par défaut s'applique.

Le fichier `postgresql.auto.conf` contient le résultat des commandes de ce type :

```
ALTER SYSTEM SET paramètre = valeur
```

qui sont principalement utilisés par les administrateurs et les outils n'ayant pas accès au système de fichiers.

Il est possible de surcharger les options modifiables à chaud par utilisateur, par base, et par combinaison « utilisateur+base », avec par exemple :

```
ALTER ROLE nagios SET log_min_duration_statement TO '1min';
ALTER DATABASE dwh SET work_mem TO '1GB';
ALTER ROLE patron IN DATABASE dwh SET work_mem TO '2GB';
```

Ces surcharges sont visibles dans la table `pg_db_role_setting` ou via la commande `\drds` de `psql`.

Ensuite, un utilisateur peut changer à volonté les valeurs de beaucoup de paramètres dans sa session :

```
SET parametre = valeur ;
```

ou une transaction :

```
SET LOCAL parametre = valeur ;
```

Au final, l'ordre des surcharges est le suivant :

```
paramètre par défaut
  -> postgresql.conf
    -> ALTER SYSTEM SET (postgresql.auto.conf)
      -> option de pg_ctl / postmaster
        -> paramètre par base
          -> paramètre par rôle
            -> paramètre base+rôle
              -> paramètre dans la chaîne de connexion
                -> paramètre de session (SET)
                  -> paramètre de transaction (SET LOCAL)
```

La meilleure source d'information sur les valeurs actives est la vue `pg_settings` :

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

```
        name         |  source  | context  | setting  | boot_val | reset_val
---------------------+----------+----------+----------+----------+-----------
 client_min_messages | default  | user     | notice   | notice   | notice
 log_checkpoints     | default  | sighup   | off      | off      | off
 wal_segment_size    | override | internal | 16777216 | 16777216 | 16777216
```

Nous constatons par exemple que, dans la session ayant effectué la requête, la valeur du paramètre `client_min_messages` a été modifiée à la valeur `debug`. Nous pouvons aussi voir le contexte dans lequel le paramètre est modifiable : le `client_min_messages` est modifiable par l'utilisateur dans sa session. Le `log_checkpoints` seulement par `sighup`, c'est-à-dire par un `pg_ctl reload`, et le `wal_segment_size` n'est pas modifiable après l'initialisation de l'instance.

De nombreuses autres colonnes sont disponibles dans `pg_settings`, comme une description détaillée du paramètre, l'unité de la valeur, ou le fichier et la ligne d'où provient le paramètre. Le champ `pending_restart` indique si un paramètre a été modifié mais attend encore un redémarrage pour être appliqué.

Il existe aussi une vue `pg_file_settings`, qui indique la configuration présente dans les fichiers de configuration (mais pas forcément active !). Elle peut être utile lorsque la configuration est répartie dans plusieurs fichiers. Par exemple, suite à un `ALTER SYSTEM`, les paramètres sont ajoutés dans `postgresql.auto.conf` mais un rechargement de la configuration n'est pas forcément suffisant pour qu'ils soient pris en compte :

```
ALTER SYSTEM SET work_mem TO '16MB' ;
ALTER SYSTEM SET max_connections TO 200 ;

SELECT pg_reload_conf() ;

 pg_reload_conf
----------------
 t

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

-[ RECORD 1 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/14/data/postgresql.conf
sourceline | 64
seqno      | 2
name       | max_connections
setting    | 100
```

```
applied    | f
error      |
-[ RECORD 2 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/14/data/postgresql.auto.conf
sourceline | 4
seqno      | 17
name       | max_connections
setting    | 200
applied    | f
error      | setting could not be applied
-[ RECORD 3 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/14/data/postgresql.auto.conf
sourceline | 3
seqno      | 16
name       | work_mem
setting    | 16MB
applied    | t
error      |
```

2.4.2 SURVOL DE POSTGRESQL.CONF

- Emplacement de fichiers
- Connections & authentification
- Ressources (hors journaux de transactions)
- Journaux de transactions
- Réplication
- Optimisation de requête
- Traces
- Statistiques d'activité
- Autovacuum
- Paramétrage client par défaut
- Verrous
- Compatibilité

`postgresql.conf` contient environ 300 paramètres. Il est séparé en plusieurs sections, dont les plus importantes figurent ci-dessous. Il n'est pas question de les détailler toutes.

La plupart des paramètres ne sont jamais modifiés. Les défauts sont souvent satisfaisants pour une petite installation. Les plus importants sont supposés acquis (au besoin, voir la formation DBA1[4]).

[4] https://dali.bo/dba1_html

Les principales sections sont :

Connections and authentication

S'y trouveront les classiques `listen_addresses`, `port`, `max_connections`, `password_encryption`, ainsi que les paramétrages TCP (*keepalive*) et SSL.

Resource usage (except WAL)

Cette partie fixe des limites à certaines consommations de ressources.

Sont normalement déjà bien connus `shared_buffers`, `work_mem` et `maintenance_work_mem` (qui seront couverts extensivement plus loin).

On rencontre ici aussi le paramétrage du `VACUUM` (pas directement de l'autovacuum !), du *background writer*, du parallélisme dans les requêtes.

Write-Ahead Log

Les journaux de transaction sont gérés ici. Cette partie sera également détaillée dans un autre module.

Depuis la version 10, tout est prévu pour faciliter la mise en place d'une réplication sans modification de cette partie sur le primaire (notamment `wal_level`).

Dans la partie *Archiving*, l'archivage des journaux peut être activé pour une sauvegarde PITR ou une réplication par *log shipping*.

Depuis la version 12, tous les paramètres de restauration (qui peuvent servir à la réplication) figurent aussi dans les sections *Archive Recovery* et *Recovery Target*. Auparavant, ils figuraient dans un fichier `recovery.conf` séparé.

Replication

Cette partie fournit le nécessaire pour alimenter un secondaire en réplication par *streaming*, physique ou logique.

Ici encore, depuis la version 12, l'essentiel du paramétrage nécessaire à un secondaire physique ou logique est intégré dans ce fichier.

Query tuning

Les paramètres qui peuvent influencer l'optimiseur sont à définir dans cette partie, notamment `seq_page_cost` et `random_page_cost` en fonction des disques, et éventuellement le parallélisme, le niveau de finesse des statistiques, le JIT…

Reporting and logging

Si le paramétrage par défaut des traces ne convient pas, le modifier ici. Il faudra généralement augmenter leur verbosité. Quelques paramètres `log_*` figurent dans d'autres sections.

Autovacuum

L'autovacuum fonctionne généralement convenablement, et des ajustements se font généralement table par table. Il arrive cependant que certains paramètres doivent être modifiés globalement.

Client connection defaults

Cette partie un peu fourre-tout définit le paramétrage au niveau d'un client : langue, fuseau horaire, extensions à précharger, tablespaces par défaut…

Lock management

Les paramètres de cette section sont rarement modifiés.

2.5 PG_HBA.CONF ET PG_IDENT.CONF

- Authentification multiple :
  - utilisateur / base / source de connexion
- Fichiers :
  - `pg_hba.conf` (*Host Based Authentication*)
  - `pg_ident.conf` : si mécanisme externe d'authentification
  - paramètres : `hba_file` et `ident_file`

L'authentification est paramétrée au moyen du fichier `pg_hba.conf`. Dans ce fichier, pour une tentative de connexion à une base donnée, pour un utilisateur donné, pour un transport (IP, IPV6, Socket Unix, SSL ou non), et pour une source donnée, ce fichier permet de spécifier le mécanisme d'authentification attendu.

Si le mécanisme d'authentification s'appuie sur un système externe (LDAP, Kerberos, Radius…), des tables de correspondances entre utilisateur de la base et utilisateur demandant la connexion peuvent être spécifiées dans `pg_ident.conf`.

Ces noms de fichiers ne sont que les noms par défaut. Ils peuvent tout à fait être remplacés en spécifiant de nouvelles valeurs de `hba_file` et `ident_file` dans `postgresql.conf` (les installations Red Hat et Debian utilisent là aussi des emplacements différents, comme pour `postgresql.conf`).

Leur utilisation est décrite dans notre première formation[5] .

2.6 TABLESPACES

- Espace de stockage physique d'objets
  - et non logique !
- Simple répertoire (**hors de PGDATA**) + lien symbolique
- Pour :
  - répartir I/O et volumétrie
  - quotas (par le FS, mais pas en natif)
- Utilisation selon des droits

Un *tablespace*, vu de PostgreSQL, est un espace de stockage des objets (tables et index principalement). Son rôle est purement physique, il n'a pas à être utilisé pour une séparation *logique* des tables (c'est le rôle des bases et des schémas).

Vu du système d'exploitation, il s'agit juste d'un répertoire :

```
CREATE TABLESPACE ssd LOCATION '/var/lib/postgresql/tbl_ssd';
CREATE TABLESPACE
```

Ce répertoire doit **impérativement être placé hors de PGDATA**. Certains outils poseraient problème sinon. Si ce conseil n'est pas suivi, PostgreSQL crée le tablespace mais renvoie un avertissement :

```
WARNING:  tablespace location should not be inside the data directory
CREATE TABLESPACE
```

Il est aussi déconseillé de mettre un numéro de version dans le chemin du tablespace. PostgreSQL le gère à l'intérieur du tablespace, et en tient notamment compte dans les migrations avec `pg_upgrade`.

Les tablespaces sont visibles dans la table système `pg_tablespace`, ou dans `psql` avec la méta-commande `\db+`.

Ils sont stockés au niveau du système de fichiers, sous forme de liens symboliques (ou de *reparse points* sous Windows), dans le répertoire `PGDATA/pg_tblspc`. Exemple :

```
# \db

                    Liste des tablespaces
    Nom     | Propriétaire |              Emplacement
------------+--------------+-----------------------------------------
```

[5] https://dali.bo/f_html

```
 froid      | postgres     | /HDD/tbl/froid
 chaud      | postgres     | /SSD/tbl/chaud
 pg_default | postgres     |
 pg_global  | postgres     |
```

```
# ls -l /HDD/tbl/froid
drwxr--r-- 3 postgres postgres 20 sep.  13 18:15 PG_14_202107181
```

Les cas d'utilisation des tablespaces dans PostgreSQL sont :

- la saturation de la partition du PGDATA sans possibilité de l'étendre ;
- la répartition des entrées-sorties (cela est de moins en moins courant de nos jours : les SAN ou la virtualisation ne permettent plus d'agir sur un disque à la fois) ;
- le déport des tris vers un volume dédié ;
- la séparation entre données froides et chaudes sur des disques de performances différentes ;
- les quotas : PostgreSQL ne disposant pas d'un système de quotas, les tablespaces peuvent permettre de contourner cette limitation : une transaction voulant étendre un fichier sera alors annulée avec l'erreur `cannot extend file`.

Sans un réel besoin, il n'y a pas besoin de créer des tablespaces, ce qui complexifie inutilement l'administration. Par défaut, il n'existe que les tablespaces par défaut, qui correspondent aux fichiers habituels dans `PGDATA/base/`.

2.6.1 TABLESPACES : MISE EN PLACE

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

CREATE DATABASE nom TABLESPACE 'chaud';

ALTER DATABASE nom SET default_tablespace TO 'chaud';

GRANT CREATE ON TABLESPACE chaud TO un_utilisateur ;

CREATE TABLE une_table (…) TABLESPACE chaud ;

ALTER TABLE une_table SET TABLESPACE chaud ;  -- verrou !

ALTER INDEX une_table_i_idx SET TABLESPACE chaud ; -- pas automatique
```

Les ordres ci-dessus permettent de :

- créer un tablespace simplement en indiquant son emplacement dans le système de fichiers du serveur ;

- créer une base de données dont le tablespace par défaut sera celui indiqué ;
- modifier le tablespace par défaut d'une base ;
- donner le droit de créer des tables dans un tablespace à un utilisateur (c'est nécessaire avant de l'utiliser) ;
- créer une table dans un tablespace ;
- déplacer une table dans un tablespace ;
- déplacer un index dans un tablespace.

Attention ! La table ou l'index est totalement verrouillé le temps du déplacement.
Les index existants ne « suivent » pas automatiquement une table déplacée, il faut les déplacer séparément.
Par défaut, les nouveaux index ne sont **pas** créés automatiquement dans le même tablespace que la table, mais en fonction de `default_tablespace`.

Les tablespaces sont visibles de manière simple dans la vue système `pg_indexes` :

```
# SELECT schemaname, indexname, tablespace
  FROM   pg_indexes
  WHERE  tablename = 'ma_table';

 schemaname |  indexname   | tablespace
------------+--------------+------------
 public     | matable_idx  | chaud
 public     | matable_pkey |
```

2.6.2 TABLESPACES : PARAMÈTRES LIÉS

- `default_tablespace`
- `temp_tablespaces`
- Performances :
  - `seq_page_cost`, `random_page_cost`
  - `effective_io_concurrency`, `maintenance_io_concurrency`
- Exemple :

```
ALTER TABLESPACE chaud SET ( random_page_cost = 1.1 );  --SSD
```

Le paramètre `default_tablespace` (défini sur un utilisateur, une base, voire le temps d'une session avec `SET default_tablespace TO 'chaud' ;`) permet d'utiliser un autre tablespace que celui par défaut dans PGDATA.

On peut 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 ...` pour qu'ils soient utilisables. Si plusieurs tablespaces de tri sont paramétrés, ils seront utilisés de façon aléatoire à chaque création d'objet temporaire,

afin de répartir la charge. Le premier intérêt est de dédier une partition rapide (SSD...) aux tris. Un autre est de ne plus risquer de saturer la partition du PGDATA en cas de fichiers temporaires énormes.

Reste le cas des disques de performances différentes (selon le type de disque SSD/SAS/SATA). Pour estimer les coûts d'accès aux tables et index, l'optimiseur utilise différents paramètres, décrits plus loin, et qui peuvent être adaptés différemment selon les tablespaces, si la valeur par défaut ne convient pas : `maintenance_io_concurrency`, `effective_io_concurrency`, `seq_page_cost` et `random_page_cost`.

Par exemple, pour un tablespace sur un SSD :

```
# ALTER TABLESPACE chaud SET ( random_page_cost = 1 );
# ALTER TABLESPACE chaud SET ( effective_io_concurrency = 200, maintenance_io_concurrency=300 );
```

2.7 GESTION DES CONNEXIONS

- L'accès à la base se fait par un protocole réseau clairement défini :
  - sockets TCP (IPV4 ou IPV6)
  - sockets Unix (Unix uniquement)
- Les demandes de connexion sont gérées par le *postmaster*.
- Paramètres : `port`, `listen_adresses`, `unix_socket_directories`, `unix_socket_group` et `unix_socket_permissions`

Le processus *postmaster* est en écoute sur les différentes sockets déclarées dans la configuration. Cette déclaration se fait au moyen des paramètres suivants :

- `port` : le port TCP. Il sera aussi utilisé dans le nom du fichier socket Unix (par exemple : `/tmp/.s.PGSQL.5432` ou `/var/run/postgresql/.s.PGSQL.5432` selon les distributions) ;
- `listen_adresses` : la liste des adresses IP du serveur auxquelles s'attacher ;
- `unix_socket_directories` : le répertoire où sera stocké la socket Unix ;
- `unix_socket_group` : le groupe (système) autorisé à accéder à la socket Unix ;
- `unix_socket_permissions` : les droits d'accès à la socket Unix.

Les connexions par socket Unix ne sont possibles sous Windows qu'à partir de la version 13.

2.7.1 TCP

- Paramètres de keepalive TCP
  - `tcp_keepalives_idle`
  - `tcp_keepalives_interval`
  - `tcp_keepalives_count`
- Paramètre de vérification de connexion
  - `client_connection_check_interval`

Il faut bien faire la distinction entre session TCP et session de PostgreSQL. Si une session TCP sert de support à une requête particulièrement longue, laquelle ne renvoie pas de données pendant plusieurs minutes, alors le firewall peut considérer la session inactive, même si le statut du backend dans `pg_stat_activity` est `active`.

Il est possible de préciser les propriétés *keepalive* des sockets TCP, pour peu que le système d'exploitation les gère. Le keepalive est un mécanisme de maintien et de vérification des sessions TCP, par l'envoi régulier de messages de vérification sur une session TCP inactive. `tcp_keepalives_idle` est le temps en secondes d'inactivité d'une session TCP avant l'envoi d'un message de keepalive. `tcp_keepalives_interval` est le temps entre un keepalive et le suivant, en cas de non-réponse. `tcp_keepalives_count` est le nombre maximum de paquets sans réponse accepté avant que la session ne soit déclarée comme morte.

Les valeurs par défaut (0) reviennent à utiliser les valeurs par défaut du système d'exploitation.

Le mécanisme de keepalive a deux intérêts :

- il permet de détecter les clients déconnectés même si ceux-ci ne notifient pas la déconnexion (plantage du système d'exploitation, fermeture de la session par un firewall…) ;
- il permet de maintenir une session active au travers de firewalls, qui seraient fermées sinon : la plupart des firewalls ferment une session inactive après 5 minutes, alors que la norme TCP prévoit plusieurs jours.

Un autre cas peut survenir. Parfois, un client lance une requête. Cette requête met du temps à s'exécuter et le client quitte la session avant de récupérer les résultats. Dans ce cas, le serveur continue à exécuter la requête et ne se rendra compte de l'absence du client qu'au moment de renvoyer les premiers résultats. La version 14 améliore cela en permettant une vérification de la connexion pendant l'exécution d'une requête. Il s'agit du paramètre `client_connection_check_interval`. Sa valeur par défaut est de 0, donc sans vérification. Sa valeur doit correspondre à la durée entre deux vérifications.

2.7.2 SSL

- Paramètres SSL
  - `ssl`, `ssl_ciphers`, `ssl_renegotiation_limit`

Il existe des options pour activer SSL et le paramétrer. `ssl` vaut `on` ou `off`, `ssl_ciphers` est la liste des algorithmes de chiffrement autorisés, et `ssl_renegotiation_limit` le volume maximum de données échangées sur une session avant renégociation entre le client et le serveur. Le paramétrage SSL impose aussi la présence d'un certificat. Pour plus de détails, consultez la documentation officielle[6] .

2.8 STATISTIQUES SUR L'ACTIVITÉ

- Collectées par chaque session durant son travail
- (Ne pas confondre avec statistiques sur les données !)
- 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`

PostgreSQL collecte des statistiques d'activité : elles permettent de mesurer l'activité de la base. Notamment :

- Combien de fois cette table a-t-elle été parcourue séquentiellement ?
- Combien de blocs ont été trouvés dans le cache pour ce parcours d'index, et combien ont dû être demandés au système d'exploitation ?
- Quelles sont les requêtes en cours d'exécution ?
- Combien de buffers ont été écrits par le *background writer* ? Par les processus eux-mêmes ? durant un checkpoint ?

Il ne faut pas confondre les statistiques d'activité avec celles sur les données (taille des tables, des enregistrements, fréquences des valeurs…) à destination de l'optimiseur de requête !

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

[6] https://docs.postgresql.fr/current/ssl-tcp.html

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` (`on` par défaut) précise si les processus doivent mettre à jour leur activité dans `pg_stat_activity`.

`track_counts` (`on` par défaut) indique que les processus doivent collecter des informations sur leur activité. Il est vital pour le déclenchement de l'autovacuum.

`track_activity_query_size` est la taille maximum du texte de requête pouvant être stocké dans `pg_stat_activity`. 1024 caractères est un défaut souvent insuffisant, à monter vers 10 000 si les requêtes sont longues ; cela nécessite un redémarrage.

Disponible depuis la version 14, `compute_query_id` permet d'activer le calcul de l'identifiant de la requête. Ce dernier sera visible dans le champ `query_id` de la vue `pg_stat_activity`, ainsi que dans les traces.

`track_io_timing` (`off` par défaut) précise si les processus doivent collecter des informations de chronométrage sur les lectures et écritures, pour compléter les champs `blk_read_time` et `blk_write_time` des vues `pg_stat_database` et `pg_stat_statements`, ainsi que les plans d'exécutions appelés avec `EXPLAIN (ANALYZE,BUFFERS)` et les traces de l'autovacuum (pour un `VACUUM` comme un `ANALYZE`). Avant de l'activer sur une machine peu performante, vérifiez l'impact avec l'outil `pg_test_timing`.

`track_functions` indique si les processus doivent aussi collecter des informations sur l'exécution des routines stockées. Les valeurs sont `none` par défaut, `pl` pour ne tracer que les routines en langages procéduraux, `all` pour tracer aussi les routines en C et en SQL.

`update_process_title` permet de modifier le titre du processus, visible par exemple avec `ps -ef` sous Unix. Il est à `on` par défaut sous Unix, mais il faut le laisser à `off` sous Windows pour des raisons de performance.

`stats_temp_directory` est le répertoire des statistiques temporaires, avant copie dans `pg_stat/` lors d'un arrêt propre. Ce répertoire peut devenir gros, et est réécrit fréquemment, et peut devenir source de contention. Il est conseillé de le stocker ailleurs que dans le répertoire de l'instance PostgreSQL, par exemple sur un *ramdisk* ou *tmpfs* (c'est le défaut sous Debian).

2.8.1 STATISTIQUES D'ACTIVITÉ COLLECTÉES

- Accès logiques (`INSERT`, `SELECT`…) par table et index
- Accès physiques (blocs) par table, index et séquence
- Activité du *Background Writer*
- Activité par base
- Liste des sessions et informations sur leur activité

2.8.2 VUES SYSTÈME

- Supervision / métrologie
- Diagnostiquer
- Vues système :
  - `pg_stat_user_*`
  - `pg_statio_user_*`
  - `pg_stat_activity` : requêtes
  - `pg_stat_bgwriter`
  - `pg_locks`

PostgreSQL propose de nombreuses vues, accessibles en SQL, pour obtenir des informations sur son fonctionnement interne. Il est possible d'avoir des informations sur le fonctionnement des bases, des processus d'arrière-plan, des tables, les requêtes en cours…

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 accès logiques aux objets (tables, index et routines) figurent dans les vues `pg_stat_xxx_tables`, `pg_stat_xxx_indexes` et `pg_stat_user_functions`.

Les accès physiques aux objets sont visibles dans les vues `pg_statio_xxx_tables`, `pg_statio_xxx_indexes`, et `pg_statio_xxx_sequences`.

Des statistiques globales par base sont aussi disponibles, dans `pg_stat_database` : le nombre de transactions validées et annulées, quelques statistiques sur les sessions, et quelques statistiques sur les accès physiques et en cache, ainsi que sur les opérations logiques.

`pg_stat_bgwriter` stocke les statistiques d'écriture des buffers des Background Writer, Checkpointer et des sessions elles-mêmes.

`pg_stat_activity` est une des vues les plus utilisées et est souvent le point de départ d'une recherche : elle donne des informations sur les processus en cours sur l'instance, que ce soit des processus en tâche de fond ou des processus backends associés aux clients : 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. (Noter qu'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… ; à partir de la version 14 apparaît le processus d'archivage).

```
=# SELECT datname, pid, usename, application_name, backend_start, state, backend_type, query
   FROM pg_stat_activity \gx

-[ RECORD 1 ]----+------------------------------------------------------------
datname          | ¤
pid              | 26378
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236776+02
state            | ¤
backend_type     | autovacuum launcher
query            |
-[ RECORD 2 ]----+------------------------------------------------------------
datname          | ¤
pid              | 26380
usename          | postgres
application_name |
backend_start    | 2019-10-24 18:25:28.238157+02
state            | ¤
backend_type     | logical replication launcher
query            |
-[ RECORD 3 ]----+------------------------------------------------------------
datname          | pgbench
pid              | 22324
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.167611+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + -3810 WHERE…
-[ RECORD 4 ]----+------------------------------------------------------------
datname          | postgres
pid              | 22429
usename          | postgres
```

```
application_name | psql
backend_start    | 2019-10-28 10:27:09.599426+01
state            | active
backend_type     | client backend
query            | select datname, pid, usename, application_name, backend_start…
-[ RECORD 5 ]----+-----------------------------------------------------------
datname          | pgbench
pid              | 22325
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.172585+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 4360 WHERE…
-[ RECORD 6 ]----+-----------------------------------------------------------
datname          | pgbench
pid              | 22326
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.178514+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 2865 WHERE…
-[ RECORD 7 ]----+-----------------------------------------------------------
datname          | ¤
pid              | 26376
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235574+02
state            | ¤
backend_type     | background writer
query            |
-[ RECORD 8 ]----+-----------------------------------------------------------
datname          | ¤
pid              | 26375
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235064+02
state            | ¤
backend_type     | checkpointer
query            |
-[ RECORD 9 ]----+-----------------------------------------------------------
datname          | ¤
pid              | 26377
usename          | ¤
application_name |
```

```
backend_start    | 2019-10-24 18:25:28.236239+02
state            | ¤
backend_type     | walwriter
query            |
```

Cette vue fournit aussi des informations sur ce que chaque session attend. Pour les détails sur `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente), voir le tableau des événements d'attente[7] .

```
# SELECT datname, pid, wait_event_type, wait_event, query FROM pg_stat_activity
  WHERE backend_type='client backend' AND wait_event IS NOT NULL \gx

-[ RECORD 1 ]---+------------------------------------------------------------
datname         | pgbench
pid             | 1590
state           | idle in transaction
wait_event_type | Client
wait_event      | ClientRead
query           | UPDATE pgbench_accounts SET abalance = abalance + 1438 WHERE…
-[ RECORD 2 ]---+------------------------------------------------------------
datname         | pgbench
pid             | 1591
state           | idle
wait_event_type | Client
wait_event      | ClientRead
query           | END;
-[ RECORD 3 ]---+------------------------------------------------------------
datname         | pgbench
pid             | 1593
state           | idle in transaction
wait_event_type | Client
wait_event      | ClientRead
query           | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES…
-[ RECORD 4 ]---+------------------------------------------------------------
datname         | postgres
pid             | 1018
state           | idle in transaction
wait_event_type | Client
wait_event      | ClientRead
query           | delete from t1 ;
-[ RECORD 5 ]---+------------------------------------------------------------
datname         | postgres
pid             | 1457
state           | active
wait_event_type | Lock
```

[7] https://docs.postgresql.fr/current/monitoring-stats.html#wait-event-table

```
wait_event      | transactionid
query           | delete  from t1 ;
```

Des vues plus spécialisées existent :

`pg_stat_replication` donne des informations sur les serveurs secondaires connectés. Les statistiques sur les conflits entre application de la réplication et requêtes en lecture seule sont disponibles dans `pg_stat_database_conflicts`.

`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_locks` permet de voir les verrous posés sur les objets (principalement les relations comme les tables et les index).

`pg_stat_progress_vacuum`, `pg_stat_progress_analyze`, `pg_stat_progress_create_index`, `pg_stat_progress_cluster`, `pg_stat_progress_basebackup` et `pg_stat_progress_copy` donnent respectivement des informations sur la progression des `VACUUM`, des `ANALYZE`, des créations d'index, des commandes de `VACUUM FULL` et `CLUSTER`, de la commande de réplication `BASE BACKUP` et des `COPY`.

`pg_stat_archiver` donne des informations sur l'archivage des wals et notamment sur les erreurs d'archivage.

2.9 STATISTIQUES SUR LES DONNÉES

- Statistiques sur les données : `pg_stats`
  - collectées par échantillonnage (`default_statistics_target`)
  - `ANALYZE table`
  - table par table (et pour certains index)
  - colonne par colonne
  - pour de meilleurs plans d'exécution
- `ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300;`
- Statistiques multicolonnes sur demande
  - `CREATE STATISTICS`

Afin de calculer les plans d'exécution des requêtes au mieux, le moteur a besoin de statistiques sur les données qu'il va interroger. Il est très important pour lui de pouvoir estimer la sélectivité d'une clause `WHERE`, l'augmentation ou la diminution du nombre d'enregistrements entraînée par une jointure, tout cela afin de déterminer le coût approximatif d'une requête, et donc de choisir un bon plan d'exécution.

Il ne faut pas les confondre avec les statistiques d'activité, vues précédemment !

Les statistiques sont collectées dans la table `pg_statistic`. La vue `pg_stats` affiche le contenu de cette table système de façon plus accessible.

Les statistiques sont collectées sur :

- chaque colonne de chaque table ;
- les index fonctionnels.

Le recueil des statistiques s'effectue quand on lance un ordre `ANALYZE` sur une table, ou que l'autovacuum le lance de son propre chef.

Les statistiques sont calculées sur un échantillon égal à 300 fois le paramètre `STATISTICS` de la colonne (ou, s'il n'est pas précisé, du paramètre `default_statistics_target`, 100 par défaut).

La vue `pg_stats` affiche les statistiques collectées :

```
\d pg_stats

                    View "pg_catalog.pg_stats"
         Column          |   Type   | Collation | Nullable | Default
-------------------------+----------+-----------+----------+---------
 schemaname              | name     |           |          |
 tablename               | name     |           |          |
 attname                 | name     |           |          |
 inherited               | boolean  |           |          |
 null_frac               | real     |           |          |
 avg_width               | integer  |           |          |
 n_distinct              | real     |           |          |
 most_common_vals        | anyarray |           |          |
 most_common_freqs       | real[]   |           |          |
 histogram_bounds        | anyarray |           |          |
 correlation             | real     |           |          |
 most_common_elems       | anyarray |           |          |
 most_common_elem_freqs  | real[]   |           |          |
 elem_count_histogram    | real[]   |           |          |
```

- `inherited` : la statistique concerne-t-elle un objet utilisant l'héritage (table parente, dont héritent plusieurs tables) ;
- `null_frac` : fraction d'enregistrements dont la colonne vaut NULL ;
- `avg_width` : taille moyenne de cet attribut dans l'échantillon collecté ;
- `n_distinct` : si positif, nombre de valeurs distinctes, si négatif, fraction de valeurs distinctes pour cette colonne dans la table. Il est possible de forcer le nombre de valeurs distinctes, s'il est constaté 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 celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target`. Le défaut de 100 échantillons sur 30 000 lignes peut être modifié par `ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300 ;` (avec une évolution proportionnelle du nombre de lignes consultées) sachant que le temps de planification augmente exponentiellement et qu'il vaut mieux ne pas dépasser la valeur 1000 ;
- `histogram_bounds` : les limites d'histogramme sur la colonne. Les histogrammes permettent d'évaluer la sélectivité d'un filtre par rapport à sa valeur précise. Ils permettent par exemple à l'optimiseur de déterminer que 4,3 % des enregistrements d'une colonne `noms` commencent par un A, ou 0,2 % par AL. Le principe est de regrouper les enregistrements triés dans des groupes de tailles approximativement identiques, et de stocker les limites de ces groupes (on ignore les `most_common_vals`, pour lesquelles il y a déjà une mesure plus précise). Le nombre d'`histogram_bounds` est calculé de la même façon que les `most_common_vals` ;
- `correlation` : le facteur de corrélation statistique entre l'ordre physique et l'ordre logique des enregistrements de la colonne. Il vaudra par exemple `1` si les enregistrements sont physiquement stockés dans l'ordre croissant, `-1` si ils sont dans l'ordre décroissant, ou `0` si ils sont totalement aléatoirement répartis. Ceci sert à affiner le coût d'accès aux enregistrements ;
- `most_common_elems` et `most_common_elems_freqs` : les valeurs les plus fréquentes si la colonne est un tableau (NULL dans les autres cas), et leur fréquence. Le nombre de valeurs collecté est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target` ;
- `elem_count_histogram` : les limites d'histogramme sur la colonne si elle est de type tableau.

Parfois, il est intéressant de calculer des statistiques sur un ensemble de colonnes ou d'expressions. Dans ce cas, il faut créer un objet statistique en indiquant les colonnes et/ou expressions à traiter et le type de statistiques à calculer (voir la documentation de `CREATE STATISTICS`).

2.10 OPTIMISEUR

- SQL est un langage déclaratif :
  - décrit le résultat attendu (projection, sélection, jointure, etc.)…
  - …mais pas comment l'obtenir
  - c'est le rôle de l'optimiseur

Le langage SQL décrit le résultat souhaité. Par exemple :

```
SELECT path, filename
FROM file
JOIN path ON (file.pathid=path.pathid)
WHERE path LIKE '/usr/%'
```

Cet ordre décrit le résultat souhaité. Nous ne précisons pas au moteur comment accéder aux tables `path` et `file` (par index ou parcours complet par exemple), ni comment effectuer la jointure (PostgreSQL dispose de plusieurs méthodes). C'est à l'optimiseur de prendre la décision, en fonction des informations qu'il possède.

Les informations les plus importantes pour lui, dans le contexte de cette requête, seront :

- quelle fraction de la table `path` est ramenée par le critère `path LIKE '/usr/%'` ?
- y a-t-il un index utilisable sur cette colonne ?
- y a-t-il des index utilisables sur `file.pathid`, sur `path.pathid` ?
- quelles sont les tailles des deux tables ?

La stratégie la plus efficace ne sera donc pas la même suivant les informations retournées par toutes ces questions.

Par exemple, il pourrait être intéressant de charger les deux tables séquentiellement, supprimer les enregistrements de `path` ne correspondant pas à la clause `LIKE`, trier les deux jeux d'enregistrements et fusionner les deux jeux de données triés (cette technique est un *merge join*). Cependant, si les tables sont assez volumineuses, et que le `LIKE` est très discriminant (il ramène peu d'enregistrements de la table `path`), la stratégie d'accès sera totalement différente : nous pourrions préférer récupérer les quelques enregistrements de `path` correspondant au `LIKE` par un index, puis pour chacun de ces enregistrements, aller chercher les informations correspondantes dans la table `file` (c'est un *nested loop*).

2.10.1 OPTIMISATION PAR LES COÛTS

- L'optimiseur évalue les coûts respectifs des différents plans
- Il calcule tous les plans possibles tant que c'est possible
- Le coût de planification exhaustif est exponentiel par rapport au nombre de jointures de la requête
- Il peut falloir d'autres stratégies
- Paramètres principaux :
  - `seq_page_cost`, `random_page_cost`, `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost`
  - `parallel_setup_cost`, `parallel_tuple_cost`
  - `effective_cache_size`

Afin de choisir un bon plan, le moteur essaie des plans d'exécution. Il estime, pour chacun de ces plans, le coût associé. Afin d'évaluer correctement ces coûts, il utilise plusieurs informations :

- Les statistiques sur les données, qui lui permettent d'estimer le nombre d'enregistrements ramenés par chaque étape du plan et le nombre d'opérations de lecture à effectuer pour chaque étape de ce plan ;
- Des informations de paramétrage lui permettant d'associer un coût arbitraire à chacune des opérations à effectuer. Ces informations sont les suivantes :
  - `seq_page_cost` (1 par défaut) : coût de la lecture d'une page disque de façon séquentielle (au sein d'un parcours séquentiel de table par exemple) ;
  - `random_page_cost` (4 par défaut) : coût de la lecture d'une page disque de façon aléatoire (lors d'un accès à une page d'index par exemple) ;
  - `cpu_tuple_cost` (0,01 par défaut) : coût de traitement par le processeur d'un enregistrement de table ;
  - `cpu_index_tuple_cost` (0,005 par défaut) : coût de traitement par le processeur d'un enregistrement d'index ;
  - `cpu_operator_cost` (0,0025 par défaut) : coût de traitement par le processeur de l'exécution d'un opérateur.

Ce sont les coûts relatifs de ces différentes opérations qui sont importants : l'accès à une page de façon aléatoire est par défaut 4 fois plus coûteux que de façon séquentielle, du fait du déplacement des têtes de lecture sur un disque dur. Ceci prend déjà en considération un potentiel effet du cache. Sur une base fortement en cache, il est donc possible d'être tenté d'abaisser le `random_page_cost` à 3, voire 2,5, ou des valeurs encore bien moindres dans le cas de bases totalement en mémoire.

Le cas des disques SSD est particulièrement intéressant. Ces derniers n'ont pas à proprement parler de tête de lecture. De ce fait, comme les paramètres `seq_page_cost` et

`random_page_cost` sont principalement là pour différencier un accès direct et un accès après déplacement de la tête de lecture, la différence de configuration entre ces deux paramètres n'a pas lieu d'être si les index sont placés sur des disques SSD. Dans ce cas, une configuration très basse et pratiquement identique (voire identique) de ces deux paramètres est intéressante.

Quant à `effective_io_concurrency`, il a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Le défaut vaut 1. Dans le cas d'un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1 ou RAID 10, n-1 s'il s'agit d'un RAID 5). Avec du SSD, il est possible de monter encore bien au-delà de cette valeur, étant donné la rapidité de ce type de disque. La valeur maximale est de 1000. Cependant, seuls les parcours *Bitmap Scan* sont impactés par la configuration de ce paramètre. (Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[8]).

Toujours à partir de la version 13, un nouveau paramètre apparaît : `maintenance_io_concurrency`. Il a le même sens que `effective_io_concurrency`, mais pour les opérations de maintenance, non les requêtes. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Le défaut est de 10, et il faut penser à le monter aussi si nous adaptons `effective_io_concurrency`.

`seq_page_cost`, `random_page_cost`, `effective_io_concurrency` et `maintenance_io_concurrency` peuvent être paramétrés par tablespace, afin de refléter les caractéristiques de disques différents.

La mise en place du parallélisme dans une requête représente un coût : il faut mettre en place une mémoire partagée, lancer des processus... Ce coût est pris en compte par le planificateur à l'aide du paramètre `parallel_setup_cost`. Par ailleurs, le transfert d'enregistrement entre un worker et le processus principal a également un coût représenté par le paramètre `parallel_tuple_cost`.

Ainsi une lecture complète d'une grosse table peut être moins coûteuse sans parallélisation du fait que le nombre de lignes retournées par les workers est très important. En revanche, en filtrant les résultats, le nombre de lignes retournées peut être moins important, la répartition du filtrage entre différents processeurs devient « rentable » et le planificateur peut être amené à choisir un plan comprenant la parallélisation.

Certaines autres informations permettent de nuancer les valeurs précédentes. `effective_cache_size` est la taille totale du cache. Il permet à PostgreSQL de modéliser plus finement le coût réel d'une opération disque, en prenant en compte la

[8] https://docs.postgresql.fr/13/release.html

probabilité que cette information se trouve dans le cache du système d'exploitation ou dans celui de l'instance, et soit donc moins coûteuse à accéder.

Le parcours de l'espace des solutions est un parcours exhaustif. Sa complexité est principalement liée au nombre de jointures de la requête et est de type exponentiel. Par exemple, planifier de façon 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[9] issu de la base de connaissance Dalibo.

2.10.2 PARAMÈTRES SUPPLÉMENTAIRES DE L'OPTIMISEUR (1)

- Partitionnement
  - `constraint_exclusion`
  - `enable_partition_pruning`
- Réordonne les tables
  - `from_collapse_limit` / `join_collapse_limit`
- Requêtes préparées
  - `plan_cache_mode`
- Curseurs
  - `cursor_tuple_fraction`
- Mutualiser les entrées-sorties
  - `synchronize_seqscans`

Tous les paramètres suivants peuvent être modifiés par session.

Avant la version 10, PostgreSQL ne connaissait qu'un partitionnement par héritage, où l'on crée une table parente et des tables filles héritent de celle-ci, possédant des contraintes `CHECK` comme critères de partitionnement, par exemple `CHECK (date >='2011-01-01' and date < '2011-02-01')` pour une table fille d'un partitionnement par mois.

Afin que PostgreSQL ne parcourt que les partitions correspondant à la clause `WHERE` d'une requête, le paramètre `constraint_exclusion` doit valoir `partition` (la valeur par défaut) ou `on`. `partition` est moins coûteux dans un contexte d'utilisation classique car les contraintes d'exclusion ne seront examinées que dans le cas de requêtes `UNION ALL`, qui sont les requêtes générées par le partitionnement.

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

Pour le nouveau partitionnement déclaratif, `enable_partition_pruning`, activé par défaut, est le paramètre équivalent.

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 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 raisonnable. Il est fréquent de monter les valeurs à 10 ou un peu au-delà si de longues requêtes impliquent beaucoup de tables.

Pour les requêtes préparées, l'optimiseur génère des plans personnalisés pour les cinq premières exécutions d'une requête préparée, puis il bascule sur un plan générique dès que celui-ci devient plus intéressant que la moyenne des plans personnalisés précédents. Ceci décrit le mode `auto` en place depuis de nombreuses versions. En version 12, il est possible de modifier ce comportement grâce au paramètre de configuration `plan_cache_mode` :

- `force_custom_plan` force le recalcul systématique d'un plan personnalisé pour la requête (on n'économise plus le temps de planification, mais le plan est calculé pour être optimal pour les paramètres, et l'on conserve la protection contre les injections SQL permise par les requêtes préparées) ;
- `force_generic_plan` force l'utilisation d'un seul et même plan dès le départ.

Lors de l'utilisation de curseurs, le moteur n'a aucun moyen de connaître le nombre d'enregistrements que souhaite récupérer réellement l'utilisateur : peut-être seulement les premiers enregistrements. Si c'est le cas, le plan d'exécution optimal ne sera plus le même. Le paramètre `cursor_tuple_fraction`, par défaut à 0,1, permet d'indiquer à l'optimiseur la fraction du nombre d'enregistrements qu'un curseur souhaitera vraisemblablement récupérer, et lui permettra donc de choisir un plan en conséquence. Si vous utilisez des curseurs, il vaut mieux indiquer explicitement le nombre d'enregistrements dans les requêtes avec `LIMIT`, et passer `cursor_tuple_fraction` à 1,0.

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`.

2.10.3 PARAMÈTRES SUPPLÉMENTAIRES DE L'OPTIMISEUR (2)

- GEQO :
  - un optimiseur génétique
  - état initial, puis mutations aléatoires
  - rapide, mais non optimal
  - paramètres : `geqo` et `geqo_threshold` (12 tables)

PostgreSQL, pour les requêtes trop complexes, bascule vers un optimiseur appelé GEQO (*GEnetic Query Optimizer*). Comme tout algorithme génétique, il fonctionne par introduction de mutations aléatoires sur un état initial donné. Il permet de planifier rapidement une requête complexe, et de fournir un plan d'exécution acceptable.

Le code source de PostgreSQL décrit le principe[10] , résumé aussi dans ce schéma :

Ce mécanisme est configuré par des paramètres dont le nom commence par « geqo ». Exceptés ceux évoqués ci-dessous, il est déconseillé de modifier les paramètres sans une bonne connaissance des algorithmes génétiques.

- `geqo`, par défaut à `on`, permet d'activer/désactiver GEQO ;
- `geqo_threshold`, par défaut à 12, est le nombre d'éléments minimum à joindre dans un `FROM` avant d'optimiser celui-ci par GEQO au lieu du planificateur exhaustif.

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe. 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 donc possible de faire varier la valeur de `geqo_seed` pour chercher d'autres plans (voir la documentation officielle[11]).

---

[10] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f
[11] https://www.postgresql.org/docs/current/static/geqo-pg-intro.html#AEN116517

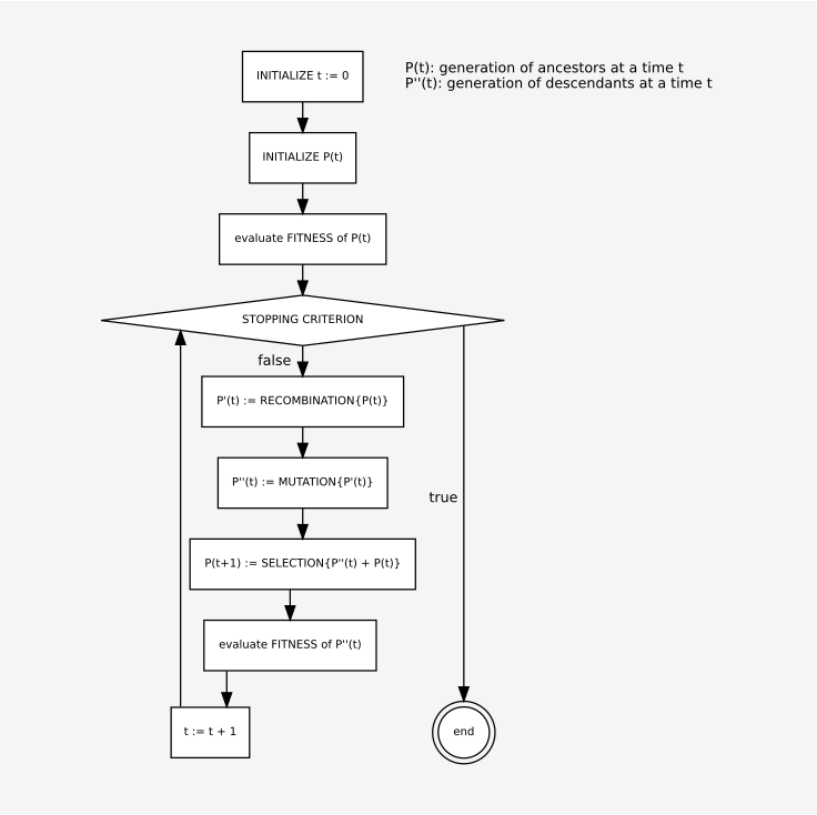


Figure 1: Principe d'un algorithme génétique (schéma de la documentation officielle, licence PostgreSQL)

2.10.4 DÉBOGAGE DE L'OPTIMISEUR

- Permet de valider qu'on est en face d'un problème d'optimiseur.
- Les paramètres sont assez grossiers :
  - défavoriser très fortement un type d'opération
  - pour du diagnostic, pas pour de la production

Ces paramètres dissuadent le moteur d'utiliser un type de nœud d'exécution (en augmentant énormément son coût). Ils permettent de vérifier ou d'invalider une erreur de l'optimiseur. Par exemple :

```
-- création de la table de test
CREATE TABLE test2(a integer, b integer);

-- insertion des données de tests
INSERT INTO test2 SELECT 1, i FROM generate_series(1, 500000) i;

-- analyse des données
ANALYZE test2;

-- désactivation de la parallélisation (pour faciliter la lecture du plan)
SET max_parallel_workers_per_gather TO 0;

-- récupération du plan d'exécution
EXPLAIN ANALYZE SELECT * FROM test2 WHERE a<3;

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on test2  (cost=0.00..8463.00 rows=500000 width=8)
             (actual time=0.031..63.194 rows=500000 loops=1)
   Filter: (a < 3)
 Planning Time: 0.411 ms
 Execution Time: 86.824 ms
```

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

```
-- désactivation des parcours SeqScan, IndexOnlyScan et BitmapScan
SET enable_seqscan TO off;
SET enable_indexonlyscan TO off;
SET enable_bitmapscan TO off;

-- création de l'index
CREATE INDEX ON test2(a);

-- récupération du plan d'exécution
EXPLAIN ANALYZE SELECT * FROM test2 WHERE a<3;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test2_a_idx on test2  (cost=0.42..16462.42 rows=500000 width=8)
                                (actual time=0.183..90.926 rows=500000 loops=1)
   Index Cond: (a < 3)
 Planning Time: 0.517 ms
 Execution Time: 111.609 ms
```

Non seulement le plan est plus coûteux, mais il est aussi (et surtout) plus lent.

Attention aux effets du cache : le parcours par index est ici relativement performant à la deuxième exécution parce que les données ont été trouvées dans le cache disque. La requête, sinon, aurait été bien plus lente. La requête initiale est donc non seulement plus rapide, mais aussi plus **sûre** : son temps d'exécution restera prévisible même en cas d'erreur d'estimation sur le nombre d'enregistrements.

Si nous supprimons l'index, nous constatons que le *sequential scan* n'a pas été désactivé. Il a juste été rendu très coûteux par ces options de débogage :

```
-- suppression de l'index
DROP INDEX test2_a_idx;

-- récupération du plan d'exécution
EXPLAIN ANALYZE SELECT * FROM test2 WHERE a<3;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on test2  (cost=10000000000.00..10000008463.00 rows=500000 width=8)
                           (actual time=0.044..60.126 rows=500000 loops=1)
   Filter: (a < 3)
 Planning Time: 0.313 ms
 Execution Time: 82.598 ms
```

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

Voici la liste des options de désactivation :

- `enable_bitmapscan` ;
- `enable_gathermerge` ;
- `enable_hashagg` ;
- `enable_hashjoin` ;
- `enable_incremental_sort` ;
- `enable_indexonlyscan` ;
- `enable_indexscan` ;
- `enable_material` ;
- `enable_mergejoin` ;

- `enable_nestloop` ;
- `enable_parallel_append` ;
- `enable_parallel_hash` ;
- `enable_partition_pruning` ;
- `enable_partitionwise_aggregate` ;
- `enable_partitionwise_join` ;
- `enable_seqscan` ;
- `enable_sort` ;
- `enable_tidscan`.

2.11 CONCLUSION

- Nombreuses fonctionnalités
  - donc nombreux paramètres

2.11.1 QUESTIONS

N'hésitez pas, c'est le moment !

2.12 QUIZ

https://dali.bo/m2_quiz

2.13 TRAVAUX PRATIQUES

2.13.1 TABLESPACE

> Créer un tablespace nommé `ts1` pointant vers `/opt/ts1`.

> Se connecter à la base de données `b1`. Créer une table `t_dans_ts1` avec une colonne `id` de type integer dans le tablespace `ts1`.

> Récupérer le chemin du fichier correspondant à la table `t_dans_ts1` avec la fonction `pg_relation_filepath`.

> Supprimer le tablespace `ts1`. Qu'observe-t-on ?

2.13.2 STATISTIQUES D'ACTIVITÉS, TABLES ET VUES SYSTÈME

> Créer une table `t3` avec une colonne `id` de type integer.

> Insérer 1000 lignes dans la table `t3` avec `generate_series`.

> Lire les statistiques d'activité de la table `t3` à l'aide de la vue système `pg_stat_user_tables`.

> Créer un utilisateur **pgbench** et créer une base `pgbench` lui appartenant.

> Écrire une requête SQL qui affiche le nom et l'identifiant de toutes les bases de données, avec le nom du propriétaire et l'encodage.
> (Utiliser les table `pg_database` et `pg_roles`).

Comparer la requête avec celle qui est exécutée lorsque l'on tape la commande `\l` dans la console (penser à `\set ECHO_HIDDEN`).

Dans un autre terminal, ouvrir une session `psql` sur la base `b0`, qu'on n'utilisera plus.
Se connecter à la base `b1` depuis une autre session.
La vue `pg_stat_activity` affiche les sessions connectées. Qu'y trouve-t-on ?

2.13.3 STATISTIQUES SUR LES DONNÉES

Se connecter à la base de données `b1` et créer une table `t4` avec une colonne `id` de type entier.

Empêcher l'autovacuum d'analyser automatiquement la table `t4`.

Insérer 1 million de lignes dans `t4` avec `generate_series`.

Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

Exécuter la commande `ANALYZE` sur la table `t4`.

Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

Ajouter un index sur la colonne `id` de la table `t4`.

Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

> Modifier le contenu de la table `t4` avec `UPDATE t4 SET id = 100000;`

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

> Exécuter la commande `ANALYZE` sur la table `t4`.

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

2.14 TRAVAUX PRATIQUES (SOLUTIONS)

2.14.1 TABLESPACE

> Créer un tablespace nommé `ts1` pointant vers `/opt/ts1`.

En tant qu'utilisateur **root** :

```
 # mkdir /opt/ts1
 # chown postgres:postgres /opt/ts1
```

En tant qu'utilisateur **postgres** :

```
$ psql

postgres=# CREATE TABLESPACE ts1 LOCATION '/opt/ts1';
CREATE TABLESPACE

postgres=# \db

          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
 pg_default | postgres     |
 pg_global  | postgres     |
 ts1        | postgres     | /opt/ts1
```

> Se connecter à la base de données `b1`. Créer une table `t_dans_ts1` avec une colonne `id` de type integer dans le tablespace `ts1`.

```
b1=# CREATE TABLE t_dans_ts1 (id integer) TABLESPACE ts1;
CREATE TABLE
```

> Récupérer le chemin du fichier correspondant à la table `t_dans_ts1` avec la fonction `pg_relation_filepath`.

```
b1=# SELECT current_setting('data_directory') || '/' || pg_relation_filepath('t_dans_ts1')
     AS chemin;

                              chemin
--------------------------------------------------------------------
  /var/lib/pgsql/14/data/pg_tblspc/16394/PG_14_202107181/16393/16395
```

Le fichier n'a pas été créé dans un sous-répertoire du répertoire `base`, mais dans le tablespace indiqué par la commande `CREATE TABLE`. `/opt/ts1` n'apparaît pas ici : il y a un lien symbolique dans le chemin.

```
$ ls -l $PGDATA/pg_tblspc/
total 0
lrwxrwxrwx 1 postgres postgres 8 Apr 16 16:26 16394 -> /opt/ts1

$ cd /opt/ts1/PG_14_202107181/
$ ls -lR
.:
total 0
drwx------ 2 postgres postgres 18 Apr 16 16:26 16393

./16393:
total 0
-rw------- 1 postgres postgres 0 Apr 16 16:26 16395
```

Il est à noter que ce fichier se trouve réellement dans un sous-répertoire de `/opt/ts1` mais que PostgreSQL le retrouve à partir de `pg_tblspc` grâce à un lien symbolique.

> Supprimer le tablespace `ts1`. Qu'observe-t-on ?

La suppression échoue tant que le tablespace est utilisé. Il faut déplacer la table dans le tablespace par défaut :

```
b1=# DROP TABLESPACE ts1 ;
ERROR:  tablespace "ts1" is not empty

b1=# ALTER TABLE t_dans_ts1 SET TABLESPACE pg_default ;
ALTER TABLE

b1=# DROP TABLESPACE ts1 ;
DROP TABLESPACE
```

2.14.2 STATISTIQUES D'ACTIVITÉS, TABLES ET VUES SYSTÈME

> Créer une table `t3` avec une colonne `id` de type integer.

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

> Insérer 1000 lignes dans la table `t3` avec `generate_series`.

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

> Lire les statistiques d'activité de la table `t3` à l'aide de la vue système `pg_stat_user_tables`.

```
b1=# \x
Expanded display is on.
b1=# SELECT * FROM pg_stat_user_tables WHERE relname = 't3';

-[ RECORD 1 ]-----+-------
relid             | 24594
schemaname        | public
relname           | t3
seq_scan          | 0
seq_tup_read      | 0
idx_scan          |
idx_tup_fetch     |
n_tup_ins         | 1000
n_tup_upd         | 0
n_tup_del         | 0
n_tup_hot_upd     | 0
n_live_tup        | 1000
n_dead_tup        | 0
last_vacuum       |
last_autovacuum   |
last_analyze      |
last_autoanalyze  |
vacuum_count      | 0
autovacuum_count  | 0
analyze_count     | 0
autoanalyze_count | 0
```

Les statistiques indiquent bien que 1000 lignes ont été insérées.

> Créer un utilisateur **pgbench** et créer une base `pgbench` lui appartenant.

```
b1=# CREATE ROLE pgbench LOGIN ;
CREATE ROLE

b1=# CREATE DATABASE pgbench OWNER pgbench ;
CREATE DATABASE
```

> Écrire une requête SQL qui affiche le nom et l'identifiant de toutes les bases de données, avec le nom du propriétaire et l'encodage.

> (Utiliser les table `pg_database` et `pg_roles`).

La liste des bases de données se trouve dans la table `pg_database` :

```
SELECT db.oid, db.datname, datdba
FROM pg_database db ;
```

Une jointure est possible avec la table `pg_roles` pour déterminer le propriétaire des bases :

```
SELECT db.datname, r.rolname, db.encoding
FROM pg_database db, pg_roles r
WHERE db.datdba = r.oid ;
```

d'où par exemple :

```
  datname  | rolname  | encoding
-----------+----------+----------
 b1        | postgres |        6
 b0        | postgres |        6
 template0 | postgres |        6
 template1 | postgres |        6
 postgres  | postgres |        6
 pgbench   | pgbench  |        6
```

L'encodage est numérique, il reste à le rendre lisible.

> Comparer la requête avec celle qui est exécutée lorsque l'on tape la commande `\l` dans la console (penser à `\set ECHO_HIDDEN`).

Il est possible de positionner le paramètre `\set ECHO_HIDDEN on`, ou sortir de la console et la lancer de nouveau psql avec l'option `-E` :

```
$ psql -E
```

Taper la commande `\l`. La requête envoyée par `psql` au serveur est affichée juste avant le résultat :

```
\l
********* QUERY **********
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges"
FROM pg_catalog.pg_database d
```

```
ORDER BY 1;
*************************
```

L'encodage se retrouve donc en appelant la fonction `pg_encoding_to_char` :

```
b1=# SELECT db.datname, r.rolname, db.encoding, pg_catalog.pg_encoding_to_char(db.encoding)
FROM pg_database db, pg_roles r
WHERE db.datdba = r.oid ;

  datname  | rolname  | encoding | pg_encoding_to_char
-----------+----------+----------+---------------------
 b1        | postgres |        6 | UTF8
 b0        | postgres |        6 | UTF8
 template0 | postgres |        6 | UTF8
 template1 | postgres |        6 | UTF8
 postgres  | postgres |        6 | UTF8
 pgbench   | pgbench  |        6 | UTF8
```

> Dans un autre terminal, ouvrir une session `psql` sur la base `b0`, qu'on n'utilisera plus.
> Se connecter à la base `b1` depuis une autre session.
> La vue `pg_stat_activity` affiche les sessions connectées. Qu'y trouve-t-on ?

```
# terminal 1
$ psql b0

# terminal 2
$ psql b1
```

La table a de nombreux champs, affichons les plus importants :

```
# SELECT datname, pid, state, usename, application_name AS appp, backend_type, query
  FROM pg_stat_activity ;

 datname | pid  | state  | usename  | appp |         backend_type          |       query
---------+------+--------+----------+------+-------------------------------+-----------------
         | 6179 |        |          |      | autovacuum launcher           |
         | 6181 |        | postgres |      | logical replication launcher  |
 b0      | 6870 | idle   | postgres | psql | client backend                |
 b1      | 6872 | active | postgres | psql | client backend                | SELECT datname, …
         | 6177 |        |          |      | background writer             |
         | 6176 |        |          |      | checkpointer                  |
         | 6178 |        |          |      | walwriter                     |
(7 rows)
```

La session dans `b1` est `idle`, c'est-à-dire en attente. La seule session active (au moment où elle tournait) est celle qui exécute la requête. Les autres lignes correspondent à des

processus système.

Remarque : Ce n'est qu'à partir de la version 10 de PostgreSQL que la vue `pg_stat_activity` liste les processus d'arrière-plan (checkpointer, background writer....). Les connexions clientes peuvent s'obtenir en filtrant sur la colonne `backend_type` le contenu *client backend*.

```
SELECT datname, count(*)
FROM pg_stat_activity
WHERE backend_type = 'client backend'
GROUP BY datname
HAVING count(*)>0;
```

Ce qui donnerait par exemple :

```
 datname | count
---------+-------
 pgbench |    10
 b0      |     5
```

2.14.3 STATISTIQUES SUR LES DONNÉES

> Se connecter à la base de données `b1` et créer une table `t4` avec une colonne `id` de type entier.

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

> Empêcher l'autovacuum d'analyser automatiquement la table `t4`.

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

NB : ceci n'est à faire qu'à titre d'exercice ! En production, c'est une très mauvaise idée.

> Insérer 1 million de lignes dans `t4` avec `generate_series`.

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

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE id = 100000;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Gather  (cost=1000.00..11866.15 rows=5642 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t4  (cost=0.00..10301.95 rows=2351 width=4)
         Filter: (id = 100000)
```

> Exécuter la commande `ANALYZE` sur la table `t4`.

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

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE id = 100000;

                              QUERY PLAN
-----------------------------------------------------------------------
 Gather  (cost=1000.00..10633.43 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on t4  (cost=0.00..9633.33 rows=1 width=4)
         Filter: (id = 100000)
```

Les statistiques sont beaucoup plus précises. PostgreSQL sait maintenant qu'il ne va récupérer qu'une seule ligne, sur le million de lignes dans la table. C'est le cas typique où un index serait intéressant.

> Ajouter un index sur la colonne `id` de la table `t4`.

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

> Rechercher la ligne ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN SELECT * FROM t4 WHERE id = 100000;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using t4_id_idx on t4  (cost=0.42..8.44 rows=1 width=4)
   Index Cond: (id = 100000)
```

Après création de l'index, nous constatons que PostgreSQL choisit un autre plan qui permet d'utiliser cet index.

> Modifier le contenu de la table `t4` avec `UPDATE t4 SET id = 100000;`

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

Toutes les lignes ont donc à présent la même valeur.

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

```
b1=# EXPLAIN ANALYZE SELECT * FROM t4 WHERE id = 100000;

                  QUERY PLAN
-------------------------------------------------------
 Index Only Scan using t4_id_idx on t4
       (cost=0.43..8.45 rows=1 width=4)
       (actual time=0.040..265.573 rows=1000000 loops=1)
   Index Cond: (id = 100000)
   Heap Fetches: 1000001
 Planning time: 0.066 ms
 Execution time: 303.026 ms
```

Là, un parcours séquentiel serait plus performant. Mais comme PostgreSQL n'a plus de statistiques à jour, il se trompe de plan et utilise toujours l'index.

> Exécuter la commande `ANALYZE` sur la table `t4`.

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

> Rechercher les lignes ayant comme valeur `100000` dans la colonne `id` et afficher le plan d'exécution.

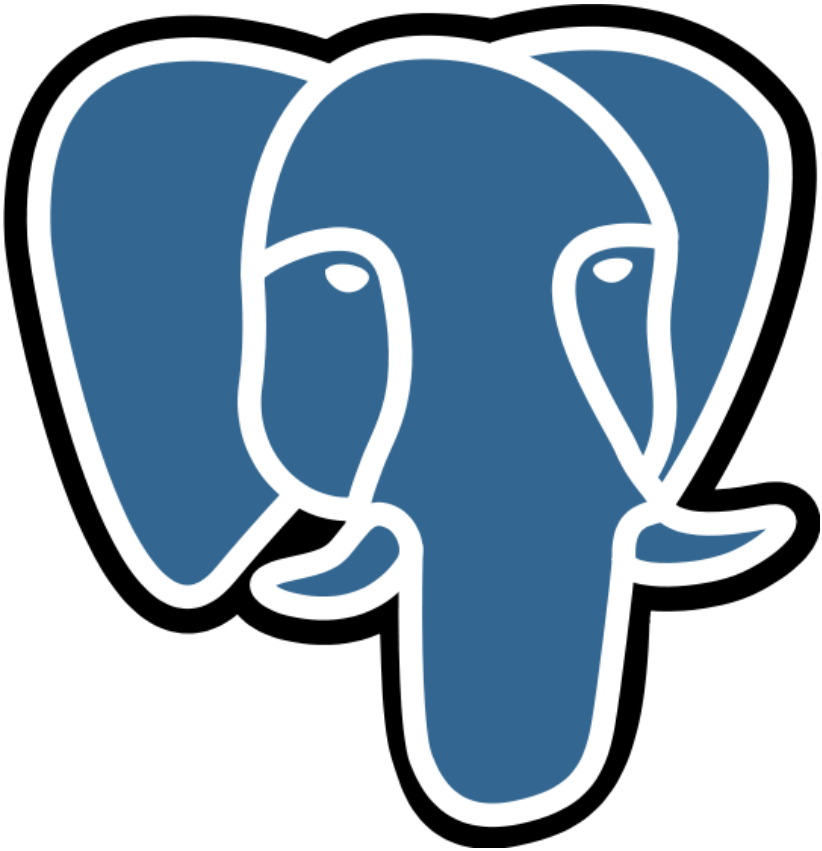
```
b1=# EXPLAIN ANALYZE SELECT * FROM t4 WHERE id = 100000;

                    QUERY PLAN
-------------------------------------------------------
 Seq Scan on t4
       (cost=0.00..21350.00 rows=1000000 width=4)
       (actual time=75.185..186.019 rows=1000000 loops=1)
   Filter: (id = 100000)
 Planning time: 0.122 ms
 Execution time: 223.357 ms
```

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

Si l'autovacuum avait été activé, les modifications massives dans la table auraient provoqué assez rapidement la mise à jour des statistiques.

3 MÉMOIRE ET JOURNALISATION DANS POSTGRESQL



3.1 AU MENU

La mémoire & PostgreSQL :

- mémoire partagée
- mémoire des processus
- les *shared buffers* & la gestion du cache
- la journalisation

3.2 MÉMOIRE PARTAGÉE

- Implémentation
  - `shared_memory_type`
- Zone de mémoire partagée :
  - cache disque des fichiers de données (`shared_buffers`)
  - cache disque des journaux de transactions (`wal_buffers`)
  - données de session (`max_connections` et `track_activity_query_size`)
  - verrous (`max_connections` et `max_locks_per_transaction`)
    - etc

La zone de mémoire partagée est allouée statiquement au démarrage de l'instance. Depuis la version 12, le type de mémoire partagée est configuré avec le paramètre `shared_memory_type`. Sous Linux, il s'agit par défaut de `mmap`, sachant qu'une très petite partie utilise toujours `sysv` (System V). Il est possible de basculer uniquement en `sysv` mais ceci n'est pas recommandé et nécessite généralement un paramétrage du noyau Linux. Sous Windows, le type est `windows`. Avant la version 12, ce paramètre n'existe pas.

Elle est calculée en fonction du dimensionnement des différentes zones :

- `shared_buffers` : le cache des fichiers de données ;
- `wal_buffers` : le cache des journaux de transaction ;
- les données de sessions : les principaux paramètres liés sont `max_connections` (défaut : 100) et `track_activity_query_size` (défaut : 1024) ;
- les verrous : les paramètres sont `max_connections` et `max_locks_per_transaction` (défaut : 64).

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

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

3.3 MÉMOIRE PAR PROCESSUS

- `work_mem`
  - × `hash_mem_multiplier` (v 13)
- `maintenance_work_mem`
  - `autovacuum_work_mem`
- `temp_buffers`
- Pas de limite stricte à la consommation mémoire d'une session !
  - ni total
- Augmenter prudemment & superviser

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, en fin de requête ou de session. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Cette mémoire est utilisée dans plusieurs contextes :

Tris, hachages, jointures

De la mémoire de tri peut être consommée lors de l'exécution de requêtes avec une clause `ORDER BY` ou certaines jointures, comme une jointure par hachage. Ce tri sera effectué en mémoire, à hauteur de la valeur de `work_mem` (seulement 4 Mo par défaut), potentiellement pour **chaque** nœud concerné. S'il estime que cette mémoire ne suffira pas, PostgreSQL triera les données sur disque, avec écriture de fichiers temporaires, ce qui peut notablement ralentir la requête.

Augmenter la valeur du `work_mem` de manière globale peut parfois mener à une consommation excessive de mémoire. PostgreSQL 13 ajoute un paramètre `hash_mem_multiplier`, par défaut à 1, qui permet de multiplier d'autant la consommation mémoire uniquement pour les hachages (jointures ou certains agrégats).

Maintenance

Les ordres `CREATE INDEX`, `REINDEX` ou `VACUUM` nécessitent aussi de la mémoire de tri. Ces besoins assez ponctuels, mais gourmands, sont gérés par le paramètre `maintenance_work_mem`, habituellement configuré à des valeurs plus hautes que `work_mem`. Sur une machine bien dotée en RAM, 1 Go est une valeur courante.

Tables temporaires

Afin de minimiser les appels systèmes dans le cas d'accès à des tables et index temporaires (locales à chaque session, et qui disparaîtront avec elle), chaque session peut allouer un cache dédié à ces tables. Sa taille dépend du paramètre `temp_buffers`. La valeur par

défaut de 8 Mo peut être insuffisant dans certains cas, ce qui peut mener à la création de fichiers sur disque dans le répertoire de la base de données.

Il n'y a pas de limite globale à la mémoire pouvant être utilisée par ces paramètres !
Il est théoriquement possible que toutes les connexions (au nombre de `max_connections`) lancent simultanément des requêtes allouant plusieurs fois `work_mem` (si la requête en cours d'exécution nécessite plusieurs tris par exemple, ou si d'autres processus sont appelés à l'aide, notamment en cas de parallélisation de la requête). Certains plans d'exécution malheureux peuvent consommer ainsi beaucoup plus que prévu.
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.

Une consommation mémoire excessive peut mener à une purge du cache de l'OS ou un *swap* excessif, tous deux désastreux pour les performances, voire un arrêt de l'instance si l'*Out-of-Memory killer* de Linux décide de tuer des processus. Au niveau système, ce phénomène peut être mitigé par le paramétrage de l'*overcommit*[12] .

3.4 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 (voir `effective_cache_size`)
  - importants pour les performances !
- Dimensionnement en première intention :
  - 1/4 RAM
  - max 8 Go

PostgreSQL dispose de son propre mécanisme de cache. Toute donnée lue l'est de ce cache. Si la donnée n'est pas dans le cache, le processus devant effectuer cette lecture l'y recopie avant d'y accéder dans le cache.

L'unité de travail du cache est le bloc (de 8 ko par défaut) de données. C'est-à-dire qu'un processus charge toujours un bloc dans son entier quand il veut lire un enregistrement. Chaque bloc du cache correspond donc exactement à un bloc d'un fichier d'un objet. Cette information est d'ailleurs, bien sûr, stockée en en-tête du bloc de cache.

Tous les processus accèdent à ce cache unique. C'est la zone la plus importante, par la taille, de la mémoire partagée. Toute modification de données est tracée dans le journal de transaction, **puis** modifiée dans ce cache. Elle n'est donc pas écrite sur le disque par

[12] https://dali.bo/j1_html#configuration-du-oom

le processus effectuant la modification, sauf en dernière extrémité (voir Synchronisation en arrière plan.

Tout accès à un bloc nécessite la prise de verrous. Un *pin lock*, qui est un simple compteur, indique qu'un processus se sert du buffer, et qu'il n'est donc pas réutilisable. C'est un verrou potentiellement de longue durée. Il existe de nombreux autres verrous, de plus courte durée, pour obtenir le droit de modifier le contenu d'un buffer, d'un enregistrement dans un buffer, le droit de recycler un buffer… mais tous ces verrous n'apparaissent pas dans la table `pg_locks`, car ils sont soit de très courte durée, soit partagés (comme le *spin lock*). Il est donc très rare qu'ils soient sources de contention, mais le diagnostic d'une contention à ce niveau est difficile.

Les lectures et écritures de PostgreSQL passent toutefois toujours par le cache du système. 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.

Dimensionner correctement ce cache est important pour de nombreuses raisons.

Un cache trop petit :

- ralentit l'accès aux données, car des données importantes risquent de ne plus s'y trouver ;
- force l'écriture de données sur le disque, ralentissant les sessions qui auraient pu effectuer uniquement des opérations en mémoire ;
- limite le regroupement d'écritures, dans le cas où un bloc viendrait à être modifié plusieurs fois.

Un cache trop grand :

- limite l'efficacité du cache système en augmentant la redondance de données entre les deux caches ;
- peut ralentir PostgreSQL, car la gestion des `shared_buffers` a un coût de traitement ;
- réduit la mémoire disponible pour d'autres opérations (tris en mémoire notamment).

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.

Pour dimensionner `shared_buffers` sur un serveur dédié à PostgreSQL, la documentation officielle[a] donne 25 % de la mémoire vive totale comme un bon point de départ et déconseille de dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

Sur une machine de 32 Go de RAM, cela donne donc :

```
shared_buffers=8GB
```

Le défaut de 128 Mo n'est pas adapté à un serveur sur une machine récente.

À cause du coût de la gestion de cette mémoire, surtout avec de nombreux processeurs ou de nombreux clients, une règle conservatrice peut être de ne pas dépasser 8 ou 10 Go, surtout sur les versions les moins récentes de PostgreSQL. Jusqu'en 9.6, sous Windows, il était même conseillé de ne pas dépasser 512 Mo.

Suivant les cas, une valeur inférieure ou supérieure à 25 % sera encore meilleure pour les performances, mais il faudra tester avec votre charge (en lecture, en écriture, et avec le bon nombre de clients).

Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres comme `max_wal_size`, et de s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…).

Un cache supplémentaire est disponible pour PostgreSQL : celui du système d'exploitation. Il est donc intéressant de préciser à PostgreSQL la taille approximative du cache, ou du moins de la part du cache qu'occupera PostgreSQL. Le paramètre `effective_cache_size` n'a pas besoin d'être très précis, mais il permet une meilleure estimation des coûts par le moteur. Il est paramétré habituellement aux alentours des 2/3 de la taille de la mémoire vive du système d'exploitation, pour un serveur dédié.

Par exemple pour une machine avec 32 Go de RAM, on peut paramétrer en première intention dans `postgresql.conf` :

```
shared_buffers = '8GB'
effective_cache_size = '21GB'
```

Cela sera à ajuster en fonction du comportement observé de l'application.

---

[a] https://www.postgresql.org/docs/current/runtime-config-resource.html

3.4.1 NOTIONS ESSENTIELLES DE GESTION DU CACHE

- Buffer pin
- Buffer dirty/clean
- Compteur d'utilisation
- Clocksweep

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

Buffer pin

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 est donc utilisé et ne peut être recyclé.

Buffer dirty/clean

Un buffer est *dirty* (« sale ») si son contenu dans le cache ne correspond pas à son contenu sur disque : il a été modifié dans le cache, ce qui a généralement été journalisé, mais le fichier de données n'est plus à jour.

Au contraire, un buffer non modifié (*clean*) peut être supprimé du cache immédiatement pour faire de la place sans être réécrit sur le disque, ce qui est le moins coûteux.

Compteur d'utilisation

Cette technique vise à garder dans le cache les blocs les plus utilisés.

À chaque fois qu'un processus a fini de se servir d'un buffer (quand il enlève son pin), ce compteur est incrémenté (à hauteur de 5 dans l'implémentation actuelle). Il est décrémenté par le *clocksweep* évoqué plus bas.

Seul un buffer dont le compteur est à zéro peut voir son contenu remplacé par un nouveau bloc.

Clocksweep (ou algorithme de balayage)

Un processus ayant besoin de charger un bloc de données dans le cache doit trouver un buffer disponible. Soit il y a encore des buffers vides (cela arrive principalement au démarrage d'une instance), soit il faut libérer un buffer.

L'algorithme *clocksweep* parcourt la liste des buffers de façon cyclique à la recherche d'un buffer *unpinned* dont le compteur d'utilisation est à zéro. Tout buffer visité voit son compteur décrémenté de 1. Le système effectue autant de passes que nécessaire sur tous les

blocs jusqu'à trouver un buffer à 0. Ce *clocksweep* est effectué par chaque processus, au moment où ce dernier a besoin d'un nouveau buffer.

3.4.2 RING BUFFER

But : ne pas purger le cache à cause :

- des grandes tables
- de certaines opérations
  - *Seq Scan* (lecture)
  - `VACUUM` (écritures)
  - `COPY`, `CREATE TABLE AS SELECT…`
  - …

Une table peut être plus grosse que les *shared buffers*. Sa lecture intégrale (lors d'un parcours complet ou d'une opération de maintenance) ne doit pas mener à l'éviction de tous les blocs du cache.

PostgreSQL utilise donc plutôt un *ring buffer* quand la taille de la relation dépasse 1/4 de `shared_buffers`. Un *ring buffer* est une zone de mémoire gérée à l'écart des autres blocs du cache. Pour un parcours complet d'une table, cette zone est de 256 ko (taille choisie pour tenir dans un cache L2). Si un bloc y est modifié (`UPDATE`…), il est traité hors du *ring buffer* comme un bloc sale normal. Pour un `VACUUM`, la même technique est utilisée, mais les écritures se font dans le *ring buffer*. Pour les écritures en masse (notamment `COPY` ou `CREATE TABLE AS SELECT`), une technique similaire utilise un *ring buffer* de 16 Mo.

Le site The Internals of PostgreSQL[13] et un README[14] dans le code de PostgreSQL entrent plus en détail sur tous ces sujets tout en restant lisibles.

3.4.3 CONTENU DU CACHE

2 extensions en « contrib » :

- `pg_buffercache`
- `pg_prewarm`

Deux extensions sont livrées dans les *contribs* de PostgreSQL qui impactent le cache.

`pg_buffercache` permet de consulter le contenu du cache (à utiliser de manière très ponctuelle). La requête suivante indique les objets non système de la base en cours,

[13] https://www.interdb.jp/pg/pgsql08.html
[14] https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README

présents dans le cache et s'ils sont *dirty* ou pas :

```
pgbench=# CREATE EXTENSION pg_buffercache ;

pgbench=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

        relname        | isdirty | blocs | taille
-----------------------+---------+-------+---------
 pgbench_accounts      | f       |  8398 | 66 MB
 pgbench_accounts      | t       |  4622 | 36 MB
 pgbench_accounts_pkey | f       |  2744 | 21 MB
 pgbench_branches      | f       |    14 | 112 kB
 pgbench_branches      | t       |     2 | 16 kB
 pgbench_branches_pkey | f       |     2 | 16 kB
 pgbench_history       | f       |   267 | 2136 kB
 pgbench_history       | t       |   102 | 816 kB
 pgbench_tellers       | f       |    13 | 104 kB
 pgbench_tellers_pkey  | f       |     2 | 16 kB
```

L'extension `pg_prewarm` permet de précharger un objet dans le cache de PostgreSQL (s'il y tient, bien sûr) :

```
=# CREATE EXTENSION pg_prewarm ;
=# SELECT pg_prewarm ('nom_table_ou_index', 'buffer') ;
```

Il permet même de recharger dès le démarrage le contenu du cache lors d'un arrêt (voir la documentation[15]).

---

[15] https://docs.postgresql.fr/current/pgprewarm.html

3.4.4 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
- Écriture directe par les *backends*
  - dernière extrémité

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 est que les temps de traitement ressentis par les utilisateurs soient les plus courts possibles, et que les écritures soient lissées sur de plus grandes plages de temps (pour ne pas saturer les disques).

Le *Background Writer* 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. Quatre paramètres régissent son comportement :

- `bgwriter_delay` (défaut : 200 ms) : la fréquence à laquelle se réveille le *Background Writer* ;
- `bgwriter_lru_maxpages` (défaut : 100) : le nombre maximum de pages pouvant être écrites sur chaque tour d'activité. Ce paramètre permet d'éviter que le *Background Writer* ne veuille synchroniser trop de pages si l'activité des sessions est trop intense : dans ce cas, autant les laisser effectuer elles-mêmes les synchronisations, étant donné que la charge est forte ;
- `bgwriter_lru_multiplier` (defaut : 2) : le coefficient multiplicateur utilisé pour calculer le nombre de buffers à libérer par rapport aux demandes d'allocation sur la période précédente ;
- `bgwriter_flush_after` (défaut : 512 ko sous Linux, 0 ou désactivé ailleurs) : à partir de quelle quantité de données écrites une synchronisation sur disque est demandée.

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

Le *checkpointer* est responsable d'un autre mécanisme : il synchronise tous les blocs modifiés lors des checkpoints. Son rôle est d'effectuer cette synchronisation, en évitant de saturer les disques en lissant la charge (voir plus loin).

Lors d'écritures intenses, il est possible que ces deux mécanismes soient débordés. Les processus *backend* peuvent alors écrire eux-mêmes dans les fichiers de données (après les journaux de transaction, bien sûr). Cette situation est évidemment à éviter, ce qui implique généralement de rendre le *bgwriter* plus agressif.

3.5 JOURNALISATION

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

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

- L'intégrité : quoi qu'il arrive, exceptée 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 qu'en mémoire, dans les *shared buffers*. Elles seront écrites de façon asynchrone, soit par un processus recherchant un buffer libre, soit par le *Background Writer*, soit par le *Checkpointer*.

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

3.5.1 JOURNAUX DE TRANSACTION (RAPPELS)

Essentiellement :

- `pg_wal/` : journaux de transactions
  - sous-répertoire `archive_status`
  - nom : *timeline*, journal, segment
  - ex : `00000002 00000142 000000FF`
- `pg_xact/` : état des transactions
- **Ces fichiers sont vitaux !**

Rappelons que les journaux de transaction sont des fichiers de 16 Mo par défaut, stockés dans `PGDATA/pg_wal` (`pg_xlog` avant la version 10), dont les noms comportent le numéro de *timeline*, un numéro de journal de 4 Go et un numéro de segment, en hexadécimal.

```
$ ls -l
total 2359320
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007C
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 00000002000001420000007D
...
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000023
-rw------- 1 postgres postgres 33554432 Mar 26 16:25 000000020000014300000024
drwx------ 2 postgres postgres    16384 Mar 26 16:28 archive_status
```

Le sous-répertoire `archive_status` est lié à l'archivage.

D'autres plus petits répertoires comme `pg_xact`, qui contient les statuts des transactions passées, ou `pg_commit_ts`, `pg_multixact`, `pg_serial`, `pg_snapshots`, `pg_subtrans` ou encore `pg_twophase` sont également impliqués.

Tous ces répertoires sont critiques, gérés par PostgreSQL, et ne doivent pas être modifiés !

3.5.2 CHECKPOINT

- Point de reprise
- À partir d'où rejouer les journaux ?
- Données écrites au moins au niveau du checkpoint
  - il peut durer
- Processus `checkpointer`

PostgreSQL trace les modifications de données dans les journaux WAL. Si le système ou l'instance sont arrêtés brutalement, il faut que PostgreSQL puisse appliquer le contenu

des journaux non traités sur les fichiers de données. Il a donc besoin de savoir à partir d'où rejouer ces données. Ce point est ce qu'on appelle un *checkpoint*, ou *point de reprise*.

Les principes sont les suivants :

Toute entrée dans les journaux est idempotente, c'est-à-dire qu'elle peut être appliquée plusieurs fois, sans que le résultat final ne soit changé. C'est nécessaire, au cas où la récupération serait interrompue, ou si un fichier sur lequel la reprise est effectuée était plus récent que l'entrée qu'on souhaite appliquer.

Tout fichier de journal antérieur à l'avant-dernier point de reprise valide (ou au dernier à partir de la version 11) **peut être supprimé** ou recyclé, car il n'est plus nécessaire à la récupération.

PostgreSQL a besoin des fichiers de données qui contiennent toutes les données jusqu'au point de reprise. Ils peuvent être plus récents et contenir des informations supplémentaires, ce n'est pas un problème.

Un checkpoint n'est pas un « instantané » cohérent de l'ensemble des fichiers. C'est simplement l'endroit à partir duquel les journaux doivent être rejoués. Il faut donc pouvoir garantir que tous les blocs modifiés dans le cache *au démarrage du checkpoint* auront été synchronisés sur le disque quand le checkpoint sera terminé, et 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.

3.5.3 DÉCLENCHEMENT & COMPORTEMENT DES CHECKPOINTS

- Déclenchement périodique (dans l'idéal)
  - `checkpoint_timeout`
- Quantité de journaux
  - `max_wal_size` (pas un plafond !)
- Dilution des écritures
  - `checkpoint_completion_target` × durée moyenne entre deux checkpoints
- Surveillance :
  - `checkpoint_warning`, `log_checkpoints`

Plusieurs paramètres influencent le comportement des checkpoints.

Dans l'idéal les checkpoints sont périodiques. Le temps maximum entre deux checkpoints

est fixé par `checkpoint_timeout` (par défaut 300 secondes). C'est parfois un peu court pour les grosses instances.

Quand le checkpoint démarre, il vise à lisser le débit en écriture, et donc le calcule à partir d'une fraction de la durée d'exécution des précédents checkpoints. Cette fraction est fixée par `checkpoint_completion_target`, et vaut 0,5 par défaut jusqu'en version 13 incluse, et 0,9 depuis la version 14. PostgreSQL prévoit donc une durée de checkpoint de 150 secondes au départ, mais cette valeur pourra évoluer ensuite suivant la durée réelle des checkpoints précédents. La valeur préconisée pour `checkpoint_completion_target` est 0,9 car elle permet de lisser davantage les écritures dues aux checkpoints dans le temps.

Le checkpoint intervient aussi quand le volume des journaux dépasse le seuil défini par `max_wal_size`, par défaut 1 Go. Un checkpoint est alors déclenché.

Attention : le terme peut porter à confusion, le volume de l'ensemble des fichiers WAL peut bien dépasser la taille fixée par le paramètre `max_wal_size` en cas de forte activité, ce n'est **pas** une valeur plafond !
De plus, les journaux peuvent encore être retenus dans `pg_wal/` pour les besoins de l'archivage ou de la réplication.

Une fois le checkpoint terminé, les journaux inutiles sont recyclés, ou effacés pour redescendre en-dessous de la quantité définie par `max_wal_size`.

Il existe un paramètre `min_wal_size` (défaut : 80 Mo) qui fixe la quantité minimale de journaux, même en cas d'activité en écriture inexistante. Ils seront prêts à être remplis en cas d'écriture imprévue. Après un gros pic d'activité suivi d'une période calme, la quantité de journaux va très progressivement redescendre de `max_wal_size` à `min_wal_size`.

Avant PostgreSQL 9.5, le rôle de `max_wal_size` était tenu par le paramètre `checkpoint_segments`, exprimé en nombre de segments de journaux maximum entre deux checkpoints, par défaut à seulement 3 (soit 48 Mo), et qu'il était conseillé d'augmenter fortement.

Le dimensionnement de ces paramètres est très dépendant du contexte et de l'activité habituelle. Le but est d'éviter des gros pics d'écriture, et donc d'avoir des checkpoints essentiellement périodiques, même si des opérations ponctuelles peuvent y échapper (gros chargements, grosse maintenance…).

Des checkpoints plus rares ont également pour effet de réduire la quantité totale de journaux écrits. Par défaut, un bloc modifié est en effet intégralement écrit dans les journaux la première fois après un checkpoint.

Par contre, un écart plus grand entre checkpoints peut entraîner une restauration plus longue après un arrêt brutal, car il y aura plus de journaux à rejouer.

Si l'on monte `max_wal_size`, par cohérence, il faudra penser à augmenter aussi `checkpoint_timeout`, et vice-versa.

Il est possible de suivre le déroulé des checkpoints dans les traces si `log_checkpoints` est à `on`. De plus, si deux checkpoints sont rapprochés d'un intervalle de temps inférieur à `checkpoint_warning` (défaut : 30 secondes), un message d'avertissement sera tracé.

Le *sync* sur disque n'a lieu qu'en fin de checkpoint. Toujours pour éviter des à-coups d'écriture, PostgreSQL demande au système d'exploitation de forcer un vidage du cache quand `checkpoint_flush_after` a déjà été écrit (par défaut 256 ko).

Avant PostgreSQL 9.6, ceci se paramétrait au niveau de Linux en abaissant les valeurs des *sysctl* `vm.dirty_*`.

3.5.4 WAL BUFFERS : JOURNALISATION EN MÉMOIRE

- Mutualiser les écritures entre transactions
- Un processus d'arrière plan : `walwriter`
- Paramètres notables :
  - `wal_buffers`
  - `wal_writer_flush_after`
- Fiabilité :
  - `fsync` = `on`
  - `full_page_writes` = `on`
  - sinon **corruption** !

La journalisation s'effectue par écriture dans les journaux de transactions. Toutefois, afin de ne pas effectuer des écritures synchrones pour chaque opération dans les fichiers de journaux, les écritures sont préparées dans des tampons (*buffers*) en mémoire. Les processus écrivent donc leur travail de journalisation dans des *buffers*, ou *WAL buffers*. Ceux-ci sont vidés quand une session demande validation de son travail (`COMMIT`), qu'il n'y a plus de *buffer* disponible, ou que le *walwriter* se réveille (`wal_writer_delay`).

Écrire un ou plusieurs blocs séquentiels de façon synchrone sur un disque a le même coût à peu de chose près. Ce mécanisme permet donc de réduire fortement les demandes d'écriture synchrone sur le journal, et augmente donc les performances.

Afin d'éviter qu'un processus n'ait tous les buffers à écrire à l'appel de `COMMIT`, et que cette opération ne dure trop longtemps, un processus d'arrière-plan appelé *walwriter* écrit à intervalle régulier tous les buffers à synchroniser.

Ce mécanisme est géré par ces paramètres, rarement modifiés :

- `wal_buffers` : taille des *WAL buffers*, soit par défaut 1/32e de `shared_buffers` avec un maximum de 16 Mo (la taille d'un segment), des valeurs supérieures pouvant être intéressantes pour les très grosses charges ;
- `wal_writer_delay` (défaut : 200 ms) : intervalle auquel le *walwriter* se réveille pour écrire les buffers non synchronisés ;
- `wal_writer_flush_after` (défaut : 1 Mo) : au-delà de cette valeur, les journaux écrits sont synchronisés sur disque pour éviter l'accumulation dans le cache de l'OS.

Pour la fiabilité, on ne touchera pas à ceux-ci :

- `wal_sync_method` : appel système à utiliser pour demander l'écriture synchrone (sauf très rare exception, PostgreSQL détecte tout seul le bon appel système à utiliser) ;
- `full_page_writes` : doit-on réécrire une image complète d'une page suite à sa première modification après un checkpoint ? Sauf cas très particulier, comme un système de fichiers *Copy On Write* comme ZFS ou btrfs, ce paramètre doit rester à `on` pour éviter des corruptions de données ;
- `fsync` : doit-on réellement effectuer les écritures synchrones ? Le défaut est `on` et **il est très fortement conseillé de le laisser ainsi en production**. Avec `off`, les performances en écritures sont certes très accélérées, mais en cas d'arrêt d'urgence de l'instance, les données seront totalement corrompues ! Ce peut être intéressant pendant le chargement initial d'une nouvelle instance par exemple, sans oublier de revenir à `on` après ce chargement initial. (D'autres paramètres et techniques existent pour accélérer les écritures et sans corrompre votre instance, si vous êtes prêt à perdre certaines données non critiques : `synchronous_commit` à `off`, les tables *unlogged*…)

3.5.5 COMPRESSION DES JOURNAUX

- `wal_compression`
  - compression
  - un peu de CPU

`wal_compression` compresse les blocs complets enregistrés dans les journaux de transactions, réduisant le volume des WAL et la charge en écriture sur les disques.

Le rejeu des WAL est aussi plus rapide, ce qui accélère la réplication et la reprise après un crash. Le prix est une augmentation de la consommation en CPU.

3.5.6 LIMITER LE COÛT DE LA JOURNALISATION

- `synchronous_commit`
  - perte potentielle de données validées
- `commit_delay` / `commit_siblings`
- Par session

Le coût d'un `fsync` est parfois rédhibitoire. Avec certains sacrifices, il est parfois possible d'améliorer les performances sur ce point.

Le paramètre `synchronous_commit` (défaut : `on`) indique si la validation de la transaction en cours doit déclencher une écriture synchrone dans le journal. Le défaut permet de garantir la pérennité des données dès la fin du `COMMIT`.

Mais ce paramètre peut être modifié dans chaque session par une commande `SET`, et passé à `off` **s'il est possible d'accepter une petite perte de données** pourtant committées. La perte peut monter à `3 × wal_writer_delay` (600 ms) ou `wal_writer_flush_after` (1 Mo) octets écrits. On accélère ainsi notablement les flux des petites transactions. Les transactions où le paramètre reste à `on` continuent de profiter de la sécurité maximale. La base restera, quoi qu'il arrive, cohérente. (Ce paramètre permet aussi de régler le niveau des transactions synchrones avec des secondaires.)

Il existe aussi `commit_delay` (défaut : `0`) et `commit_siblings` (défaut : `5`) comme mécanisme de regroupement de transactions[16] . S'il y au moins `commit_siblings` transactions en cours, PostgreSQL attendra jusqu'à `commit_delay` (en microsecondes) avant de valider une transaction pour permettre à d'autres transactions de s'y rattacher. Ce mécanisme, désactivé par défaut, accroît la latence de certaines transactions afin que plusieurs soient écrites ensembles, et n'apporte un gain de performance global qu'avec de nombreuses petites transactions en parallèle, et des disques classiques un peu lents. (En cas d'arrêt brutal, il n'y a pas à proprement parler de perte de données puisque les transactions délibérément retardées n'ont pas été signalées comme validées.)

---

[16] https://docs.postgresql.fr/current/wal-configuration.html

3.6 AU-DELÀ DE LA JOURNALISATION

- Sauvegarde PITR
- Réplication physique
  - par *log shipping*
  - par *streaming*

Le système de journalisation de PostgreSQL étant très fiable, des fonctionnalités très intéressantes ont été bâties dessus.

3.6.1 L'ARCHIVAGE DES JOURNAUX

- Repartir à partir :
  - d'une vieille sauvegarde
  - les journaux archivés
- Sauvegarde à chaud
- Sauvegarde en continu
- Paramètres : `wal_level`, `archive_mode`, `archive_command`

Les journaux permettent de rejouer, suite à un arrêt brutal de la base, toutes les modifications depuis le dernier checkpoint. Les journaux devenus obsolète depuis le dernier *checkpoint* (l'avant-dernier avant la version 11) sont à terme recyclés ou supprimés, car ils ne sont plus nécessaires à la réparation de la base.

Le but de l'archivage est de stocker ces journaux, afin de pouvoir rejouer leur contenu, non plus depuis le dernier checkpoint, mais **depuis une sauvegarde**. Le mécanisme d'archivage permet de repartir d'une sauvegarde binaire de la base (c'est-à-dire des fichiers, pas un `pg_dump`), et de réappliquer le contenu des journaux archivés.

Il suffit de rejouer tous les journaux depuis le checkpoint précédent la sauvegarde jusqu'à la fin de la sauvegarde, ou même à un point précis dans le temps. L'application de ces journaux permet de rendre à nouveau cohérents les fichiers de données, même si ils ont été sauvegardés en cours de modification.

Ce mécanisme permet aussi de fournir une sauvegarde continue de la base, alors même que celle-ci travaille.

Tout ceci est vu dans le module *Point In Time Recovery*[17] .

Même si l'archivage n'est pas en place, il faut connaître les principaux paramètres impliqués :

[17] https://dali.bo/i2_html

wal_level :

Il vaut `replica` par défaut depuis la version 10. Les journaux contiennent les informations nécessaires pour une sauvegarde PITR ou une réplication vers une instance secondaire.

Si l'on descend à `minimal` (défaut jusqu'en version 9.6 incluse), les journaux ne contiennent plus que ce qui est nécessaire à une reprise après arrêt brutal sur le serveur en cours. Ce peut être intéressant pour réduire, parfois énormément, le volume des journaux générés, si l'on a bien une sauvegarde non PITR par ailleurs.

Le niveau `logical` est destiné à la réplication logique[18] .

(Avant la version 9.6 existaient les niveaux intermédiaires `archive` et `hot_standby`, respectivement pour l'archivage et pour un serveur secondaire en lecture seule. Ils sont toujours acceptés, et assimilés à `replica`.)

archive_mode & **archive_command** :

Il faut qu'`archive_command` soit à `on` pour activer l'archivage. Les journaux sont alors copiés grâce à une commande shell à fournir dans `archive_command`. En général elle est fournie par des outils de sauvegarde dédiés (par exemple pgBackRest, pitrery ou barman).

3.6.2 RÉPLICATION

- *Log shipping* : fichier par fichier
- *Streaming* : entrée par entrée (en flux continu)
- Serveurs secondaires très proches de la production, en lecture

La restauration d'une sauvegarde peut se faire en continu sur un autre serveur, qui peut même être actif (bien que forcément en lecture seule). Les journaux peuvent être :

- envoyés régulièrement vers le secondaire, qui les rejouera : c'est le principe de la réplication par *log shipping* ;
- envoyés par fragments vers cet autre serveur : c'est la réplication par *streaming*.

Ces thèmes ne seront pas développés ici. Signalons juste que la réplication par *log shipping* implique un archivage actif sur le primaire, et l'utilisation de `restore_command` (et d'autres pour affiner) sur le secondaire. Le *streaming* permet de se passer d'archivage, même si coupler *streaming* et sauvegarde PITR est une bonne idée. Sur un PostgreSQL récent, le primaire a par défaut le nécessaire activé pour se voir doté d'un secondaire : `wal_level` est à `replica` ; `max_wal_senders` permet d'ouvrir des processus dédiés à la réplication ; et

[18] https://dali.bo/w5_html

l'on peut garder des journaux en paramétrant `wal_keep_size` (ou `wal_keep_segments` avant la version 13) pour limiter les risques de décrochage du secondaire.

Une configuration supplémentaire doit se faire sur le serveur secondaire, indiquant comment récupérer les fichiers de l'archive, et comment se connecter au primaire pour récupérer des journaux. Elle a lieu dans les fichiers `recovery.conf` (jusqu'à la version 11 comprise), ou (à partir de la version 12) `postgresql.conf` dans les sections évoquées plus haut, ou `postgresql.auto.conf`.

3.7 CONCLUSION

Mémoire et journalisation :

- complexe
- critique
- mais fiable
- et le socle de nombreuses fonctionnalités évoluées

3.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

3.8 QUIZ

https://dali.bo/m3_quiz

3.9 INTRODUCTION À PGBENCH

pgbench est un outil de test livré avec PostgreSQL. Son but est de faciliter la mise en place de benchmarks simples et rapides. Par défaut, il installe une base assez simple, génère une activité plus ou moins intense et calcule le nombre de transactions par seconde et la latence. C'est ce qui sera fait ici dans cette introduction. On peut aussi lui fournir ses propres scripts.

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[19] .

3.9.1 INSTALLATION

L'outil est installé avec les paquets habituels de PostgreSQL, client ou serveur suivant la distribution.

Dans le cas des paquets RPM du PGDG, l'outil n'est pas dans le PATH par défaut ; il faudra donc fournir le chemin complet :

```
/usr/pgsql-14/bin/pgbench
```

Il est préférable de créer un rôle non privilégié dédié, qui possédera la base de données :

```
CREATE ROLE pgbench LOGIN PASSWORD 'unmotdepassebienc0mplexe';
CREATE DATABASE pgbench OWNER pgbench ;
```

Le `pg_hba.conf` doit éventuellement être adapté.

La base par défaut s'installe ainsi (indiquer la base de données en dernier ; ajouter `-p` et `-h` au besoin) :

```
pgbench -U pgbench --initialize --scale=100 pgbench
```

`--scale` permet de faire varier proportionnellement la taille de la base. À 100, la base pèsera 1,5 Go, avec 10 millions de lignes dans la table principale `pgbench_accounts` :

```
pgbench@pgbench=# \d+
                          Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Taille  | Description
--------+------------------+-------+--------------+---------+-------------
 public | pg_buffercache   | vue   | postgres     | 0 bytes |
 public | pgbench_accounts | table | pgbench      | 1281 MB |
 public | pgbench_branches | table | pgbench      | 40 kB   |
 public | pgbench_history  | table | pgbench      | 0 bytes |
 public | pgbench_tellers  | table | pgbench      | 80 kB   |
```

[19] https://youtu.be/aTwh_CgRaE0

3.9.2 GÉNÉRER DE L'ACTIVITÉ

Pour simuler une activité de 20 clients simultanés, répartis sur 4 processeurs, pendant 100 secondes :

```
pgbench -U pgbench -c 20 -j 4 -T100 pgbench
```

NB : ne **pas** utiliser `-d` pour indiquer la base, qui signifie `--debug` pour pgbench, qui noiera alors l'affichage avec ses requêtes :

```
UPDATE pgbench_accounts SET abalance = abalance + -3455 WHERE aid = 3789437;
SELECT abalance FROM pgbench_accounts WHERE aid = 3789437;
UPDATE pgbench_tellers SET tbalance = tbalance + -3455 WHERE tid = 134;
UPDATE pgbench_branches SET bbalance = bbalance + -3455 WHERE bid = 78;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
 VALUES (134, 78, 3789437, -3455, CURRENT_TIMESTAMP);
```

À la fin, s'affichent notamment le nombre de transactions (avec et sans le temps de connexion) et la durée moyenne d'exécution du point de vue du client (`latency`) :

```
scaling factor: 100
query mode: simple
number of clients: 20
number of threads: 4
duration: 10 s
number of transactions actually processed: 20433
latency average = 9.826 ms
tps = 2035.338395 (including connections establishing)
tps = 2037.198912 (excluding connections establishing)
```

Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS='-c synchronous_commit=off -c commit_siblings=20' \
            pgbench -d pgbench -U pgbench -c 20 -j 4 -T100   2>/dev/null

latency average = 6.992 ms
tps = 2860.465176 (including connections establishing)
tps = 2862.964803 (excluding connections establishing)
```

Des tests rigoureux doivent durer bien sûr beaucoup plus longtemps que 100 s, par exemple pour tenir compte des effets de cache, des checkpoints périodiques, etc.

3.10 TRAVAUX PRATIQUES

3.10.1 MÉMOIRE PARTAGÉE

But : constater l'effet du cache sur les accès.

> Se connecter à la base de données `b0` et créer une table `t2` avec une colonne `id` de type integer.

> Insérer 500 lignes dans la table `t2` avec `generate_series`.

> Réinitialiser les statistiques pour `t2` avec la fonction `pg_stat_reset_single_table_counters` (l'OID qui lui sert de paramètre est dans la table des relations `pg_class`).

> Afin de vider le cache, redémarrer l'instance PostgreSQL.

> Se connecter à la base de données `b0`.
> Lire les données de la table `t2`.

> Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

3.10.2 MÉMOIRE DE TRI

Activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Si ce n'est pas déjà fait, laisser défiler dans une fenêtre le fichier de traces.

Activer le chronométrage dans la session (`\timing on`).
Lire les données de la table `t2` en triant par la colonne `id`
Qu'observe-t-on ?

Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

3.10.3 CACHE DISQUE DE POSTGRESQL

Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

Créer une table `t2` avec une colonne `id` de type integer.

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Nous allons vider les caches.
D'abord, redémarrer l'instance PostgreSQL.

> Vider le cache du système d'exploitation avec :
> `# sync && echo 3 > /proc/sys/vm/drop_caches`

> Se connecter à la base de données `b1`. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ?
> (Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

> Activer l'affichage de la durée des requêtes.
> Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

> Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

> Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

> Redémarrer l'instance PostgreSQL.

> Se connecter à la base de données `b1` et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

> Modifier le contenu de la table `t2` (par exemple avec `UPDATE t2 SET id = 0 WHERE id < 1000;`). Que contient le cache de PostgreSQL ?

> Exécuter un `CHECKPOINT`. Que contient le cache de PostgreSQL ?

3.10.4 JOURNAUX

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

> Exécuter un `CHECKPOINT`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

3.11 TRAVAUX PRATIQUES (SOLUTIONS)

3.11.1 MÉMOIRE PARTAGÉE

But : constater l'effet du cache sur les accès.

> Se connecter à la base de données `b0` et créer une table `t2` avec une colonne `id` de type integer.

```
$ psql b0
```

```
b0=# CREATE TABLE t2 (id integer);
CREATE TABLE
```

> Insérer 500 lignes dans la table `t2` avec `generate_series`.

```
b0=# INSERT INTO t2 SELECT generate_series(1, 500);
INSERT 0 500
```

> Réinitialiser les statistiques pour `t2` avec la fonction `pg_stat_reset_single_table_counters` (l'OID qui lui sert de paramètre est dans la table des relations `pg_class`).

Cette fonction attend un OID comme paramètre :

```
b0=# \df pg_stat_reset_single_table_counters
```

```
List of functions
-[ RECORD 1 ]-------+---------------------
Schema              | pg_catalog
Name                | pg_relation_filepath
Result data type    | text
Argument data types | regclass
Type                | func
```

L'OID est une colonne présente dans la table `pg_class` :

```
b0=# SELECT relname, pg_stat_reset_single_table_counters(oid)
     FROM pg_class WHERE relname = 't2';
```

```
 relname | pg_stat_reset_single_table_counters
---------+-------------------------------------
 t2      |
```

> Afin de vider le cache, redémarrer l'instance PostgreSQL.

```
# systemctl restart postgresql-14
```

> Se connecter à la base de données `b0`.
> Lire les données de la table `t2`.

```
b0=# SELECT * FROM t2;
[...]
```

> Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

```
b0=# \x
Expanded display is on.

b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2' ;

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 0
idx_blks_read   |
idx_blks_hit    |
toast_blks_read |
toast_blks_hit  |
tidx_blks_read  |
tidx_blks_hit   |
```

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

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2;
[...]
b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2';

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 3
…
```

Les 3 blocs sont maintenant lus à partir du cache de PostgreSQL (colonne `heap_blks_hit`).

> Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2;
[...]
b0=# SELECT * FROM pg_statio_user_tables WHERE relname = 't2';

-[ RECORD 1 ]---+-------
relid           | 24576
schemaname      | public
relname         | t2
heap_blks_read  | 3
heap_blks_hit   | 6
…
```

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

3.11.2 MÉMOIRE DE TRI

> Activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Dans la session :

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

Les paramètres `log_temp_files` et `client_min_messages` peuvent aussi être mis en place une fois pour toutes dans `postgresql.conf` (recharger la configuration). En fait, c'est généralement conseillé.

> Insérer un million de lignes dans la table `t2` avec `generate_series`.

```
b0=# INSERT INTO t2 SELECT generate_series(1, 1000000);

INSERT 0 1000000
```

> Si ce n'est pas déjà fait, laisser défiler dans une fenêtre le fichier de traces.

Le nom du fichier dépend de l'installation et du moment. Pour suivre tout ce qui se passe dans le fichier de traces, utiliser `tail -f` :

```
$ tail -f /var/lib/pgsql/14/data/log/postgresql-Tue.log
```

> Activer le chronométrage dans la session (`\timing on`).
> Lire les données de la table `t2` en triant par la colonne `id`
> Qu'observe-t-on ?

```
b0=# \timing on
b0=# SELECT * FROM t2 ORDER BY id;

LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp1197.0", size 14032896
   id
---------
       1
       1
       2
       2
       3
[...]
Time: 436.308 ms
```

Le message `LOG` apparaît aussi dans la trace, et en général il se trouvera là.

PostgreSQL a dû créer un fichier temporaire pour stocker le résultat temporaire du tri. Ce fichier s'appelle `base/pgsql_tmp/pgsql_tmp1197.0`. Il est spécifique à la session et sera détruit dès qu'il ne sera plus utile. Il fait 14 Mo.

Écrire un fichier de tri sur disque prend évidemment un certain temps, c'est généralement à éviter si le tri peut se faire en mémoire.

> Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

```
b0=# SET work_mem TO '100MB';
SET
```

> Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

```
b0=# SELECT * FROM t2 ORDER BY id;

   id
---------
```

```
       1
       1
       2
       2
[...]
Time: 240.565 ms
```

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

3.11.3 CACHE DISQUE DE POSTGRESQL

> Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

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

> Créer une table `t2` avec une colonne `id` de type integer.

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

> Insérer un million de lignes dans la table `t2` avec `generate_series`.

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

> Nous allons vider les caches.
> D'abord, redémarrer l'instance PostgreSQL.

```
 # systemctl restart postgresql-14
```

> Vider le cache du système d'exploitation avec :
> `# sync && echo 3 > /proc/sys/vm/drop_caches`

> Se connecter à la base de données `b1`. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ?
> (Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

```
b1=# SELECT relfilenode, count(*)
  FROM pg_buffercache
```

```
  GROUP BY 1
  ORDER BY 2 DESC
  LIMIT 10;

 relfilenode | count
-------------+-------
             | 16181
        1249 |    57
        1259 |    26
        2659 |    15
[...]
```

Les valeurs exactes peuvent varier. La colonne `relfilenode` correspond à l'identifiant système de la table. La deuxième colonne indique le nombre de blocs. Il y a ici 16 181 blocs non utilisés pour l'instant dans le cache (126 Mo), ce qui est logique vu que PostgreSQL vient de redémarrer. Il y a quelques blocs utilisés par des tables systèmes, mais aucune table utilisateur (on les repère par leur OID supérieur à 16384).

> Activer l'affichage de la durée des requêtes.
> Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

```
b1=# \timing on
Timing is on.

b1=# SELECT * FROM t2;

   id
---------
       1
       2
       3
       4
       5
[...]
Time: 277.800 ms

b1=# SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+-------
             | 16220
       16410 |    32
        1249 |    29
        1259 |     9
        2659 |     8
```

```
[...]
Time: 30.694 ms
```

32 blocs ont été alloués pour la lecture de la table `t2` (*filenode* 16410). Cela représente 256 ko alors que la table fait 35 Mo :

```
b1=# SELECT pg_size_pretty(pg_table_size('t2'));

 pg_size_pretty
----------------
 35 MB
(1 row)

Time: 1.913 ms
```

Un simple `SELECT *` ne suffit donc pas à maintenir la table dans le cache. Par contre, ce deuxième accès était déjà beaucoup rapide, ce qui suggère que le système d'exploitation, lui, a probablement gardé les fichiers de la table dans son propre cache.

> Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

```
b1=# SELECT * FROM t2;

   id
---------
[...]
Time: 184.529 ms

b1=#  SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+-------
             | 16039
        1249 |    85
       16410 |    64
        1259 |    39
        2659 |    22
[...]
```

Il y en en a un peu plus dans le cache (en fait, 2 fois 32 ko). Plus vous exécuterez la requête, et plus le nombre de blocs présents en cache augmentera. Sur le long terme, les 4425 blocs de la table `t2` peuvent se retrouver dans le cache.

> Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

Pour cela, il faut ouvrir le fichier de configuration `postgresql.conf` et modifier la valeur du paramètre `shared_buffers` à un quart de la mémoire. Par exemple :

```
shared_buffers = 2GB
```

> Redémarrer l'instance PostgreSQL.

```
# systemctl restart postgresql-14
```

> Se connecter à la base de données `b1` et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

```
b1=# \timing on
b1=# SELECT * FROM t2;

   id
---------
       1
[...]
Time: 340.444 ms

b1=# SELECT relfilenode, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 2 DESC LIMIT 10 ;

 relfilenode | count
-------------+--------
             | 257581
       16410 |   4425
        1249 |     29
[...]
```

PostgreSQL se retrouve avec toute la table directement dans son cache, et ce dès la première exécution.

PostgreSQL est optimisé principalement pour 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 octroyée est grosse.

> Modifier le contenu de la table `t2` (par exemple avec `UPDATE t2 SET id = 0 WHERE id < 1000;`). Que contient le cache de PostgreSQL ?

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

```
b1=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

 relname | isdirty | blocs | taille
---------+---------+-------+--------
 t2      | f       |  4419 | 35 MB
 t2      | t       |    15 | 120 kB
```

15 blocs ont été modifiés (`isdirty` est à `true`), le reste n'a pas bougé.

> Exécuter un `CHECKPOINT`. Que contient le cache de PostgreSQL ?

```
b1=# CHECKPOINT;
CHECKPOINT

b1=# SELECT
    relname,
    isdirty,
    count(bufferid) AS blocs,
    pg_size_pretty(count(bufferid) * current_setting ('block_size')::int) AS taille
FROM pg_buffercache b
INNER JOIN pg_class c ON c.relfilenode = b.relfilenode
WHERE relname NOT LIKE 'pg\_%'
GROUP BY
        relname,
        isdirty
ORDER BY 1, 2 ;

 relname | isdirty | blocs | taille
---------+---------+-------+--------
 t2      | f       |  4434 | 35 MB
```

Les blocs *dirty* ont tous été écrits sur le disque et sont devenus « propres ».

3.11.4 JOURNAUX

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

```
b1=# INSERT INTO t2 SELECT generate_series(1, 10000000);
INSERT 0 10000000

$ ls -al $PGDATA/pg_wal
total 131076
$ ls -al $PGDATA/pg_wal
total 638984
drwx------  3 postgres postgres     4096 Apr 16 17:55 .
drwx------ 20 postgres postgres     4096 Apr 16 17:48 ..
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000033
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000034
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000035
…
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000054
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000055
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000056
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000057
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000058
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000059
drwx------  2 postgres postgres        6 Apr 16 15:01 archive_status
```

Des journaux de transactions sont écrits lors des écritures dans la base. Leur nombre varie avec l'activité récente.

> Exécuter un `CHECKPOINT`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

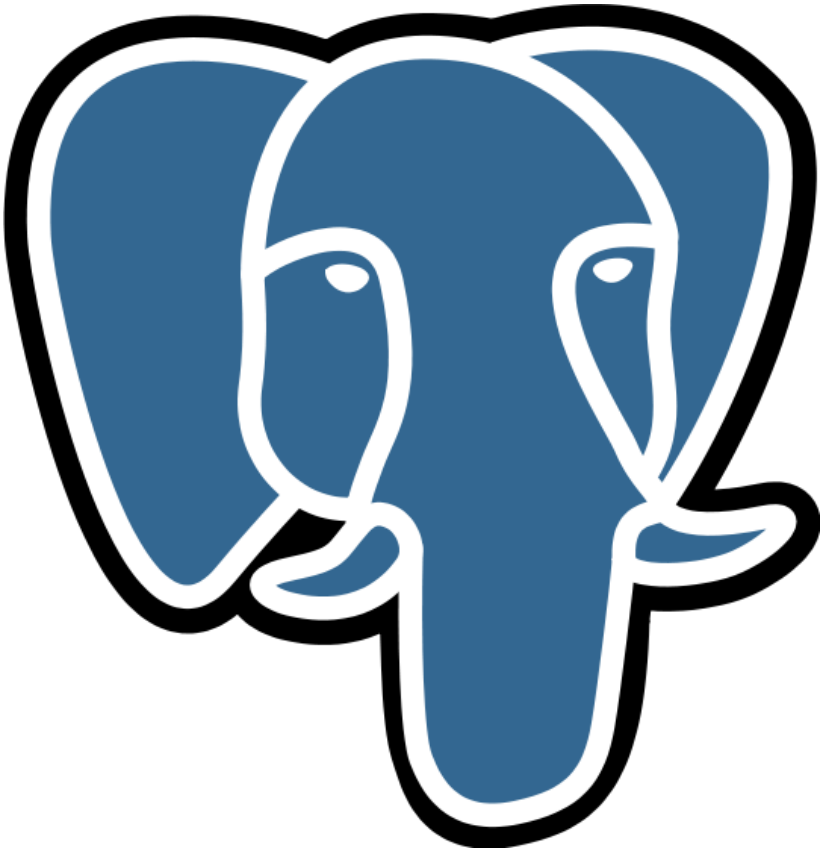
```
b1=# CHECKPOINT;
CHECKPOINT

$ ls -al $PGDATA/pg_wal
total 131076
total 638984
drwx------  3 postgres postgres     4096 Apr 16 17:56 .
drwx------ 20 postgres postgres     4096 Apr 16 17:48 ..
-rw-------  1 postgres postgres 16777216 Apr 16 17:56 000000010000000000000059
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000005A
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000005B
…
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 000000010000000000000079
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007A
```

```
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007B
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007C
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007D
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007E
-rw-------  1 postgres postgres 16777216 Apr 16 17:55 00000001000000000000007F
drwx------  2 postgres postgres        6 Apr 16 15:01 archive_status
```

Le nombre n'a pas forcément décru, mais le dernier journal d'avant le checkpoint est à présent le plus ancien (selon l'ordre des noms des journaux). Les anciens journaux devenus obsolètes sont recyclés, prêts à être remplis à nouveau. Noter que leur date de création n'a pas été mise à jour !

4 MÉCANIQUE DU MOTEUR TRANSACTIONNEL & MVCC



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.

4.2 AU MENU

- Présentation de MVCC
- Niveaux d'isolation
- Implémentation de MVCC de PostgreSQL
- Les verrous
- Le mécanisme TOAST

4.3 PRÉSENTATION DE MVCC

- *MultiVersion Concurrency Control*
- Contrôle de Concurrence Multi-Version
- Plusieurs versions du même enregistrement
- Granularité : l'enregistrement (pas le champ !)

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).

Précisons que la granularité des modifications est bien l'enregistrement (ou ligne) d'une table. Modifier un champ (colonne) revient à modifier la ligne. Deux transactions ne peuvent pas modifier deux champs différents d'un même enregistrement sans entrer en conflit, et les verrous portent toujours sur des lignes entières.

4.3.1 ALTERNATIVE À MVCC : UN SEUL ENREGISTREMENT EN BASE

- Verrouillage en lecture et exclusif en écriture
- Nombre de verrous ?
- Contention ?
- Cohérence ?
- Annulation ?

Avant d'expliquer en détail MVCC, voyons l'autre solution de gestion de la concurrence qui s'offre à nous, afin de comprendre le problème que MVCC essaye de résoudre.

Une table contient une liste d'enregistrements.

- Une transaction voulant consulter un enregistrement doit le verrouiller (pour s'assurer qu'il n'est pas modifié) de façon partagée, le consulter, puis le déverrouiller.
- Une transaction voulant modifier un enregistrement doit le verrouiller de façon exclusive (personne d'autre ne doit pouvoir le modifier ou le consulter), le modifier, puis le déverrouiller.

Cette solution a l'avantage de la simplicité : il suffit d'un gestionnaire de verrous pour gérer l'accès concurrent aux données. Elle a aussi l'avantage de la performance, dans le cas où les attentes de verrous sont peu nombreuses, la pénalité de verrouillage à payer étant peu coûteuse.

Elle a par contre des inconvénients :

- Les verrous sont en mémoire. Leur nombre est donc probablement limité. Que se passe-t-il si une transaction doit verrouiller 10 millions d'enregistrements ? Des mécanismes de promotion de verrou sont implémentés. Les verrous lignes deviennent des verrous bloc, puis des verrous table. **Le nombre de verrous est limité, et une promotion de verrou peut avoir des conséquences dramatiques** ;
- Un processus devant lire un enregistrement devra attendre la fin de la modification de celui-ci. Ceci entraîne rapidement de gros problèmes de contention. **Les écrivains bloquent les lecteurs, et les lecteurs bloquent les écrivains**. Évidemment, les écrivains se bloquent entre eux, mais cela est normal (il n'est pas possible que deux transactions modifient le même enregistrement simultanément, chacune sans conscience de ce qu'a effectué l'autre) ;
- Un ordre SQL (surtout s'il dure longtemps) n'a aucune garantie de voir des données cohérentes du début à la fin de son exécution : si, par exemple, durant un `SELECT` long, un écrivain modifie à la fois des données déjà lues par le `SELECT`, et des données qu'il va lire, le `SELECT` n'aura pas une vue cohérente de la table. Il pourrait y avoir un

total faux sur une table comptable par exemple, le `SELECT` ayant vu seulement une partie des données validées par une nouvelle transaction ;
- Comment annuler une transaction ? Il faut un moyen de défaire ce qu'une transaction a effectué, au cas où elle ne se terminerait pas par une validation mais par une annulation.

4.3.2 IMPLÉMENTATION DE MVCC PAR UNDO

- MVCC par UNDO :
  - une version de l'enregistrement dans la table
  - sauvegarde des anciennes versions
  - l'adresse physique d'un enregistrement ne change pas
  - la lecture cohérente est complexe
  - l'UNDO est complexe à dimensionner... et parfois insuffisant
  - l'annulation est lente
- Exemple : Oracle

C'est l'implémentation d'Oracle, par exemple. Un enregistrement, quand il doit être modifié, est recopié précédemment dans le tablespace d'UNDO. La nouvelle version de l'enregistrement est ensuite écrite par-dessus. Ceci implémente le MVCC (les anciennes versions de l'enregistrement sont toujours disponibles), et présente plusieurs avantages :

- Les enregistrements ne sont pas dupliqués dans la table. Celle-ci ne grandit donc pas suite à une mise à jour (si la nouvelle version n'est pas plus grande que la version précédente) ;
- Les enregistrements gardent la même adresse physique dans la table. Les index correspondant à des données non modifiées de l'enregistrement n'ont donc pas à être modifiés eux-mêmes, les index permettant justement de trouver l'adresse physique d'un enregistrement par rapport à une valeur.

Elle a aussi des défauts :

- La gestion de l'UNDO est très complexe : comment décider ce qui peut être purgé ? Il arrive que la purge soit trop agressive, et que des transactions n'aient plus accès aux vieux enregistrements (erreur `SNAPSHOT TOO OLD` sous Oracle, par exemple) ;
- La lecture cohérente est complexe à mettre en œuvre : il faut, pour tout enregistrement modifié, disposer des informations permettant de retrouver l'image avant modification de l'enregistrement (et la bonne image, il pourrait y en avoir plusieurs). Il faut ensuite pouvoir le reconstituer en mémoire ;
- Il est difficile de dimensionner correctement le fichier d'UNDO. Il arrive d'ailleurs

qu'il soit trop petit, déclenchant l'annulation d'une grosse transaction. Il est aussi potentiellement une source de contention entre les sessions ;
- L'annulation (ROLLBACK) est très lente : il faut, pour toutes les modifications d'une transaction, défaire le travail, donc restaurer les images contenues dans l'UNDO, les réappliquer aux tables (ce qui génère de nouvelles écritures). Le temps d'annulation peut être supérieur au temps de traitement initial devant être annulé.

4.3.3 L'IMPLÉMENTATION MVCC DE POSTGRESQL

- *Copy On Write* (duplication à l'écriture)
- Une version d'enregistrement n'est jamais modifiée
- Toute modification entraîne une nouvelle version
- Pas d'UNDO : pas de contention, ROLLBACK instantané

Dans une table PostgreSQL, un enregistrement peut être stocké dans plusieurs versions. Une modification d'un enregistrement entraîne l'écriture d'une nouvelle version de celui-ci. Une ancienne version ne peut être recyclée que lorsqu'aucune transaction ne peut plus en avoir besoin, c'est-à-dire qu'aucune transaction n'a un instantané de la base plus ancien que l'opération de modification de cet enregistrement, et que cette version est donc invisible pour tout le monde. Chaque version d'enregistrement contient bien sûr des informations permettant de déterminer s'il est visible ou non dans un contexte donné.

Les avantages de cette implémentation stockant plusieurs versions dans la table principale sont multiples :

- La lecture cohérente est très simple à mettre en œuvre : à chaque session de lire la version qui l'intéresse. La visibilité d'une version d'enregistrement est simple à déterminer ;
- Il n'y a pas d'UNDO. C'est un aspect de moins à gérer dans l'administration de la base ;
- Il n'y a pas de contention possible sur l'UNDO ;
- Il n'y a pas de recopie dans l'UNDO avant la mise à jour d'un enregistrement. La mise à jour est donc moins coûteuse ;
- L'annulation d'une transaction est instantanée : les anciens enregistrements sont toujours disponibles.

Cette implémentation a quelques défauts :

- Il faut supprimer régulièrement les versions obsolètes des enregistrements ;
- Il y a davantage de maintenance d'index (mais moins de contentions sur leur mise à jour) ;

- Les enregistrements embarquent des informations de visibilité, qui les rendent plus volumineux.

4.4 NIVEAUX D'ISOLATION

- Chaque transaction (et donc session) est isolée à un certain point :
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Le niveau d'isolation au démarrage d'une transaction peut être spécifié :
  - `BEGIN ISOLATION LEVEL xxx`;

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé est un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction). Quatre niveaux sont définis, ils ne sont pas tous implémentés par PostgreSQL.

4.4.1 NIVEAU READ UNCOMMITTED

- Non disponible sous PostgreSQL
  - si demandé, s'exécute en `READ COMMITTED`
- Lecture de données modifiées par d'autres transactions **non** validées
- Aussi appelé *dirty reads*
- Dangereux
- Pas de blocage entre les sessions

Ce niveau d'isolation n'est disponible que pour les SGBD non-MVCC. Il est très dangereux : il est possible de lire des données invalides, ou temporaires, puisque tous les enregistrements de la table sont lus, quels que soient leurs états. Il est utilisé dans certains cas où les performances sont cruciales, au détriment de la justesse des données.

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

4.4.2 NIVEAU READ COMMITTED

- Niveau d'isolation par défaut
- La transaction ne lit que les données validées en base
- Un ordre SQL s'exécute dans un instantané (les tables semblent figées sur la durée de l'ordre)
- L'ordre suivant s'exécute dans un instantané différent

Ce mode est le mode par défaut, et est suffisant dans de nombreux contextes. PostgreSQL étant MVCC, les écrivains et les lecteurs ne se bloquent pas mutuellement, et chaque ordre s'exécute sur un instantané de la base (ce n'est pas un prérequis de `READ COMMITTED` dans la norme SQL). Il n'y a plus de lectures d'enregistrements non valides (*dirty reads*). Il est toutefois possible d'avoir deux problèmes majeurs d'isolation dans ce mode :

- Les lectures non-répétables (*non-repeatable reads*) : une transaction peut ne pas voir les mêmes enregistrements d'une requête sur l'autre, si d'autres transactions ont validé des modifications entre temps ;
- Les lectures fantômes (*phantom reads*) : des enregistrements peuvent ne plus satisfaire une clause `WHERE` entre deux requêtes d'une même transaction.

4.4.3 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 si modification des mêmes enregistrements

Ce mode, comme son nom l'indique, permet de ne plus avoir de lectures non-répétables. Deux ordres SQL consécutifs dans la même transaction retourneront les mêmes enregistrements, dans la même version. Ceci est possible car la transaction voit une image de la base figée. L'image est figée non au démarrage de la transaction, mais à la première commande non TCL (*Transaction Control Language*) de la transaction, donc généralement au premier `SELECT` ou à la première modification.

Cette image sera utilisée pendant toute la durée de la transaction. En lecture seule, ces transactions ne peuvent pas échouer. Elles sont entre autres utilisées pour réaliser des exports des données : c'est ce que fait `pg_dump`.

Dans le standard, ce niveau d'isolation souffre toujours des lectures fantômes, c'est-à-dire de lecture d'enregistrements différents pour une même clause `WHERE` entre deux exé-

cutions de requêtes. Cependant, PostgreSQL est plus strict et ne permet pas ces lectures fantômes en `REPEATABLE READ`. Autrement dit, un même `SELECT` renverra toujours le même résultat.

En écriture, par contre (ou `SELECT FOR UPDATE`, `FOR SHARE`), si une autre transaction a modifié les enregistrements ciblés entre temps, une transaction en `REPEATABLE READ` va échouer avec l'erreur suivante :

```
ERROR: could not serialize access due to concurrent update
```

Il faut donc que l'application soit capable de la rejouer au besoin.

4.4.4 NIVEAU SERIALIZABLE

- Niveau d'isolation le plus élevé
- Chaque transaction se croit seule sur la base
  - sinon annulation d'une transaction en cours
- Avantages :
  - pas de « lectures fantômes »
  - évite des verrous, simplifie le développement
- Inconvénients :
  - pouvoir rejouer les transactions annulées
  - toutes les transactions impliquées doivent être sérialisables

PostgreSQL fournit un mode d'isolation appelé `SERIALIZABLE` :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
…
COMMIT / ROLLBACK ;
```

Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base, et comme si elles se déroulaient les unes à la suite des autres. Dès que cette garantie ne peut plus être apportée, PostgreSQL annule celle qui entraînera le moins de perte de données.

Le niveau `SERIALIZABLE` est utile quand le résultat d'une transaction peut être influencé par une transaction tournant en parallèle, par exemple quand des valeurs de lignes dépendent de valeurs d'autres lignes : mouvements de stocks, mouvements financiers… avec calculs de stocks. Autrement dit, si une transaction lit des lignes, elle a la garantie que leurs valeurs ne seront pas modifiées jusqu'à son `COMMIT`, y compris par les transactions qu'elle ne voit pas — ou bien elle tombera en erreur.

Au niveau `SERIALIZABLE` (comme en `REPEATABLE READ`), il est donc essentiel de pouvoir rejouer une transaction en cas d'échec. Par contre, nous simplifions énormément tous les autres points du développement. Il n'y a plus besoin de `SELECT FOR UPDATE`, solution courante mais très gênante pour les transactions concurrentes. Les triggers peuvent être utilisés sans soucis pour valider des opérations.

Ce mode doit être mis en place globalement, car toute transaction non sérialisable peut en théorie s'exécuter n'importe quand, ce qui rend inopérant le mode sérialisable sur les autres.

La sérialisation utilise le « verrouillage de prédicats ». Ces verrous sont visibles dans la vue `pg_locks` sous le nom `SIReadLock`, et ne gênent pas les opérations habituelles, du moins tant que la sérialisation est respectée. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation.

Le wiki PostgreSQL[20] , et la documentation officielle[21] donnent des exemples, et ajoutent quelques conseils pour l'utilisation de transactions sérialisables. Afin de tenter de réduire les verrous et le nombre d'échecs :

- faire des transactions les plus courtes possibles (si possible uniquement ce qui a trait à l'intégrité) ;
- limiter le nombre de connexions actives ;
- utiliser les transactions en mode `READ ONLY` dès que possible, voire en `SERIALIZABLE READ ONLY DEFERRABLE` (au risque d'un délai au démarrage) ;
- augmenter certains paramètres liés aux verrous, c'est-à-dire augmenter la mémoire dédiée ; car si elle manque, des verrous de niveau ligne pourraient être regroupés en verrous plus larges et plus gênants ;
- éviter les parcours de tables (*Seq Scan*), et donc privilégier les accès par index.

---

[20] https://wiki.postgresql.org/wiki/SSI/fr
[21] https://docs.postgresql.fr/current/transaction-iso.html#XACT-SERIALIZABLE

4.5 STRUCTURE D'UN BLOC

- 1 bloc = 8 ko
- `ctid` = (bloc, item dans le bloc)

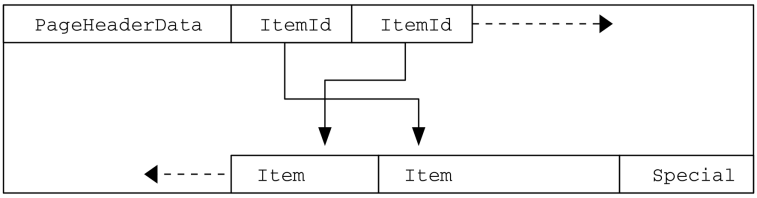


Figure 2: Répartition des lignes au sein d'un bloc (schéma de la documentation officielle, licence PostgreSQL)

Le bloc (ou page) est l'unité de base de transfert pour les I/O, le cache mémoire… Il fait généralement 8 ko (ce qui ne peut être modifié qu'en recompilant). Les lignes y sont stockées avec des informations d'administration telles que décrites dans le schéma ci-dessus. Une ligne ne fait jamais partie que d'un seul bloc (si cela ne suffit pas, un mécanisme que nous verrons plus tard, nommé TOAST, se déclenche).

Nous distinguons dans ce bloc :

- un entête de page avec diverses informations, notamment la somme de contrôle (si activée) ;
- des identificateurs de 4 octets, pointant vers les emplacements des lignes au sein du bloc ;
- les lignes, stockées à rebours depuis la fin du bloc ;
- un espace spécial, vide pour les tables ordinaires, mais utilisé par les blocs d'index.

Le `ctid` identifie une ligne, en combinant le numéro du bloc (à partir de 0) et l'identificateur dans le bloc (à partir de 1). Comme la plupart des champs administratifs liés à une ligne, il suffit de l'inclure dans un `SELECT` pour l'afficher. L'exemple suivant affiche les premiers et derniers éléments des deux blocs d'une table et vérifie qu'il n'y a pas de troisième bloc :

```
# CREATE TABLE deuxblocs AS SELECT i, i  AS j FROM generate_series(1, 452) i;
SELECT 452

# SELECT ctid, i,j FROM deuxblocs
WHERE ctid in ( '(1, 1)', '(0, 226)', '(1, 1)', '(1, 226)', '(1, 227)', '(2, 0)' );
```

```
 ctid   | i | j
--------+---+---
(0,1)   |   1 |   1
(0,226) | 226 | 226
(1,1)   | 227 | 227
(1,226) | 452 | 452
```

Un `ctid` ne doit jamais servir à désigner une ligne de manière pérenne et ne doit pas être utilisé dans des requêtes ! Il peut changer n'importe quand, notamment en cas d'`UPDATE` ou de `VACUUM FULL` !

La documentation officielle[22] contient évidemment tous les détails.

4.6 XMIN & XMAX

Table initiale :

xmin	xmax	Nom	Solde
100		M. Durand	1500
100		Mme Martin	2200

PostgreSQL stocke des informations de visibilité dans chaque version d'enregistrement.

- `xmin` : l'identifiant de la transaction créant cette version.
- `xmax` : l'identifiant de la transaction invalidant cette version.

Ici, les deux enregistrements ont été créés par la transaction 100. Il s'agit peut-être, par exemple, de la transaction ayant importé tous les soldes à l'initialisation de la base.

4.6.1 XMIN & XMAX (SUITE)

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

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

[22] https://docs.postgresql.fr/current/storage-page-layout.html

Nous décidons d'enregistrer un virement de 200 € du compte de M. Durand vers celui de Mme Martin. Ceci doit être effectué dans une seule transaction : l'opération doit être atomique, sans quoi de l'argent pourrait apparaître ou disparaître de la table.

Nous allons donc tout d'abord démarrer une transaction (ordre `SQL BEGIN`). PostgreSQL fournit donc à notre session un nouveau numéro de transaction (150 dans notre exemple). Puis nous effectuerons :

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

4.6.2 XMIN & XMAX (SUITE)

```
UPDATE soldes SET solde = solde + 200 WHERE nom = 'Mme Martin';
```

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

Puis nous effectuerons :

```
UPDATE soldes SET solde = solde + 200 WHERE nom = 'Mme Martin';
```

Nous avons maintenant deux versions de chaque enregistrement.

Notre session ne voit bien sûr plus que les nouvelles versions de ces enregistrements, sauf si elle décidait d'annuler la transaction, auquel cas elle reverrait les anciennes données.

Pour une autre session, la version visible de ces enregistrements dépend de plusieurs critères :

- La transaction 150 a-t-elle été validée ? Sinon elle est invisible ;
- La transaction 150 a-t-elle été validée après le démarrage de la transaction en cours, et sommes-nous dans un niveau d'isolation (*repeatable read* ou *serializable*) qui nous interdit de voir les modifications faites depuis le début de notre transaction ? ;
- La transaction 150 a-t-elle été validée après le démarrage de la requête en cours ? Une requête, sous PostgreSQL, voit un instantané cohérent de la base, ce qui implique que toute transaction validée après le démarrage de la requête doit être ignorée.

Dans le cas le plus simple, 150 ayant été validée, une transaction 160 ne verra pas les

premières versions : `xmax` valant 150, ces enregistrements ne sont pas visibles. Elle verra les secondes versions, puisque `xmin` = 150, et pas de `xmax`.

4.6.3 XMIN & XMAX (SUITE)

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

- Comment est effectuée la suppression d'un enregistrement ?
- Comment est effectuée l'annulation de la transaction 150 ?

- La suppression d'un enregistrement s'effectue simplement par l'écriture d'un `xmax` dans la version courante ;
- 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.

4.7 CLOG

- La CLOG (*Commit Log*) enregistre l'état des transactions.
- Chaque transaction occupe 2 bits de CLOG
- `COMMIT` ou `ROLLBACK` très rapide

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` signifie que la transaction en cours, c'est l'état initial ;
- `TRANSACTION_STATUS_COMMITTED` signifie que la la transaction a été validée ;
- `TRANSACTION_STATUS_ABORTED` signifie que la transaction a été annulée ;
- `TRANSACTION_STATUS_SUB_COMMITTED` signifie que la transaction comporte des sous-transactions, afin de valider l'ensemble des sous-transactions de façon atomique.

Nous avons donc un million d'états de transactions par fichier de 256 ko.

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, vue matérialisée), est bien sûr enregistrée tout d'abord dans les journaux de transactions (dans le répertoire `pg_wal/`).

4.8 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

Reprenons les avantages du MVCC tel qu'implémenté par PostgreSQL :

- Les lecteurs ne bloquent pas les écrivains, ni les écrivains les lecteurs ;
- Le code gérant les instantanés est simple, ce qui est excellent pour la fiabilité, la maintenabilité et les performances ;
- Les différentes sessions ne se gênent pas pour l'accès à une ressource commune (l'UNDO) ;
- Un enregistrement est facilement identifiable comme étant verrouillé en écriture : il suffit qu'il ait une version ayant un `xmax` correspondant à une transaction en cours ;
- L'annulation est instantanée : il suffit d'écrire le nouvel état de la transaction annulée dans la CLOG. Pas besoin de restaurer les valeurs précédentes, elles sont toujours là ;
- Les anciennes versions restent en ligne aussi longtemps que nécessaire. Elles ne pourront être effacées de la base qu'une fois qu'aucune transaction ne les considérera comme visibles.

(Précisons toutefois que ceci est une vision un peu simplifiée pour les cas courants. La signification du `xmax` est parfois altérée par des bits positionnés dans des champs systèmes inaccessibles par l'utilisateur. Cela arrive, par exemple, quand des transactions insèrent des lignes portant une clé étrangère, pour verrouiller la ligne pointée par cette clé, laquelle ne doit pas disparaître pendant la durée de cette transaction.)

4.9 INCONVÉNIENTS DU MVCC POSTGRESQL

- Nettoyage des enregistrements
  - VACUUM
  - automatisation : autovacuum
- Tables plus volumineuses
- Pas de visibilité dans les index
- Colonnes supprimées impliquent reconstruction

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

4.9.0.1 VACUUM

Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande VACUUM. Il a un avantage sur la technique de l'UNDO : le nettoyage n'est pas effectué par un client faisant des mises à jour (et créant donc des enregistrements morts), et le ressenti est donc meilleur.

VACUUM peut se lancer à la main, mais dans le cas général on s'en remet à l'autovacuum, un démon qui lance les VACUUM (et bien plus) en arrière-plan quand il le juge nécessaire. Tout cela sera traité en détail par la suite.

4.9.0.2 Bloat

Les tables sont forcément plus volumineuses que dans l'implémentation par UNDO, pour deux raisons :

- les informations de visibilité y sont stockées, il y a donc un surcoût d'une douzaine d'octets par enregistrement ;
- il y a toujours des enregistrements morts dans une table, une sorte de *fond de roulement*, qui se stabilise quand l'application est en régime stationnaire.

Ces enregistrements sont recyclés à chaque passage de VACUUM.

4.9.0.3 Visibilité

Les index n'ont pas d'information de visibilité. Il est donc nécessaire d'aller vérifier dans la table associée que l'enregistrement trouvé dans l'index est bien visible. Cela a un impact sur le temps d'exécution de requêtes comme `SELECT count(*)` sur une table : dans le cas le plus défavorable, il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles. La *visibility map* permet de limiter cette vérification aux données les plus récentes.

4.9.0.4 Colonnes supprimées

Un `VACUUM` ne s'occupe pas de l'espace libéré par des colonnes supprimées (fragmentation verticale). Un `VACUUM FULL` est nécessaire pour reconstruire la table.

4.9.1 LE PROBLÈME DU WRAPAROUND

Wraparound : bouclage d'un compteur

- N° de transactions dans les tables : 32 bits
  - => 4 milliards de transactions
- Et si ça boucle ?
- => `VACUUM FREEZE`
  - autovacuum
  - au pire, d'office

Le numéro de transaction stocké dans les tables de PostgreSQL est sur 32 bits, même si PostgreSQL utilise en interne 64 bits. Il y aura donc dépassement de ce compteur au bout de 4 milliards de transactions. Sur les machines actuelles, cela peut être atteint relativement rapidement.

En fait, ce 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. Si nous bouclions, 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.

La parade consiste à « geler » les lignes avec des identifiants de transaction suffisamment anciens. C'est le rôle de l'opération appelée `VACUUM FREEZE`. Ce dernier peut être déclenché manuellement, mais il fait aussi partie des tâches de maintenance habituellement gérées par le démon autovacuum, en bonne partie en même temps que les `VACUUM`

habituels. Un VACUUM FREEZE n'est pas bloquant, mais les verrous sont parfois plus gênants que lors d'un VACUUM simple.

Si cela ne suffit pas, le moteur déclenche automatiquement un VACUUM FREEZE quand les tables sont trop âgées, et ce, même si autovacuum est désactivé.

Quand le stock de transactions disponibles descend en dessous de 40 millions (10 millions avant la version 14), des messages d'avertissements apparaissent dans les traces.

Dans le pire des cas, après bien des messages d'avertissements, le moteur refuse toute nouvelle transaction dès que le stock de transactions disponibles se réduit à 3 millions (1 million avant la version 14 ; valeurs codées en dur).

Il faudra alors lancer un VACUUM FREEZE manuellement. Ceci ne peut plus arriver qu'exceptionnellement (par exemple si une transaction préparée a été oubliée depuis 2 milliards de transactions et qu'aucune supervision ne l'a détectée).

VACUUM FREEZE sera développé dans le module VACUUM et autovacuum[23] . La documentation officielle[24] contient aussi un paragraphe sur ce sujet.

4.10 OPTIMISATIONS DE MVCC

MVCC a été affiné au fil des versions :

- Mise à jour HOT (*Heap-Only Tuples*) + si place dans le bloc + si aucune colonne indexée modifiée
- *Free Space Map*
- *Visibility Map*

Les améliorations suivantes ont été ajoutées au fil des versions :

- *Heap-Only Tuples* (HOT) s'agit de pouvoir stocker, sous condition, plusieurs versions du même enregistrement dans le même bloc. Ceci permet 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 les suivantes :
  - Le bloc contient assez de place pour la nouvelle version (les enregistrements ne sont pas chaînés entre plusieurs blocs). Afin que cette première condition ait

[23] https://dali.bo/m5_html
[24] https://docs.postgresql.fr/current/maintenance.html

plus de chance d'être vérifiée, il peut être utile de baisser la valeur du paramètre `fillfactor` pour une table donnée (cf documentation officielle[25]) ;
  - Aucune colonne indexée n'a été modifiée par l'opération.

- Chaque table possède une *Free Space Map* avec une liste des espaces libres de chaque table. Elle est stockée dans les fichiers `*_fsm` associés à chaque table.

- La *Visibility Map* permet de savoir si l'ensemble des enregistrements d'un 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). À l'inverse, les parcours d'index seuls utilisent cette *Visibility Map* pour savoir s'il faut aller voir les informations de visibilité dans la table. `VACUUM` repositionne la *Visibility Map* à *vrai* après nettoyage d'un bloc, si tous les enregistrements sont visibles pour toutes les sessions. Enfin, depuis la 9.6, elle repère aussi les bloc entièrement gelés pour accélérer les `VACUUM FREEZE`.

Toutes ces optimisations visent le même but : rendre `VACUUM` le moins pénalisant possible, et simplifier la maintenance.

4.11 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

4.11.1 LE GESTIONNAIRE DE VERROUS

PostgreSQL possède un gestionnaire de verrous
- Verrous d'objet
- Niveaux de verrouillage
- Empilement des verrous
- *Deadlock*
- Vue `pg_locks`

[25] https://docs.postgresql.fr/current/sql-createtable.html#SQL-CREATETABLE-STORAGE-PARAMETERS

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

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler. Si le verrou ne peut être obtenu immédiatement, par défaut PostgreSQL attendra indéfiniment qu'il soit libéré.

Ce verrou en attente peut lui-même imposer une attente à d'autres sessions qui s'intéresseront au même objet. Si ce verrou en attente est bloquant (cas extrême : un `VACUUM FULL` sans `SKIP_LOCKED` lui-même bloqué par une session qui tarde à faire un `COMMIT`), il est possible d'assister à un phénomène d'empilement de verrous en attente.

Les noms des verrous peuvent prêter à confusion : `ROW SHARE` par exemple est un verrou de table, pas un verrou d'enregistrement. Il signifie qu'on a pris un verrou sur une table pour y faire des `SELECT FOR UPDATE` par exemple. Ce verrou est en conflit avec les verrous pris pour un `DROP TABLE`, ou pour un `LOCK TABLE`.

Le gestionnaire de verrous détecte tout verrou mortel (*deadlock*) entre deux sessions. Un *deadlock* est la suite de prise de verrous entraînant le blocage mutuel d'au moins deux sessions, chacune étant en attente d'un des verrous acquis par l'autre.

Il est possible d'accéder aux verrous actuellement utilisés sur une instance par la vue `pg_locks`.

4.11.2 VERROUS SUR ENREGISTREMENT

- Le gestionnaire de verrous possèdes des verrous sur enregistrements
  - transitoires
  - le temps de poser le `xmax`
- Utilisation de verrous sur disque
  - pas de risque de pénurie
- Les verrous entre transaction se font sur leurs ID

Le gestionnaire de verrous fournit des verrous sur enregistrement. Ceux-ci sont utilisés pour verrouiller un enregistrement le temps d'y écrire un `xmax`, puis libérés immédiatement.

Le verrouillage réel est implémenté comme suit :

- D'abord, chaque transaction verrouille son objet « identifiant de transaction » de façon exclusive.

- Une transaction voulant mettre à jour un enregistrement consulte le `xmax`. Si ce `xmax` est celui d'une transaction en cours, elle demande un verrou exclusif sur l'objet « identifiant de transaction » de cette transaction, qui ne lui est naturellement pas accordé. La transaction est donc placée en attente.
- Enfin, quand l'autre transaction possédant le verrou se termine (`COMMIT` ou `ROLLBACK`), son verrou sur l'objet « identifiant de transaction » est libéré, débloquant ainsi l'autre transaction, qui peut reprendre son travail.

Ce mécanisme ne nécessite pas un nombre de verrous mémoire proportionnel au nombre d'enregistrements à verrouiller, et simplifie le travail du gestionnaire de verrous, celui-ci ayant un nombre bien plus faible de verrous à gérer.

Le mécanisme exposé ici est évidemment simplifié.

4.11.3 LA VUE PG_LOCKS

- `pg_locks` :
  - visualisation des verrous en place
  - tous types de verrous sur objets
- Complexe à interpréter :
  - verrous sur enregistrements pas directement visibles

C'est une vue globale à l'instance.

```
# \d pg_locks

                    Vue « pg_catalog.pg_locks »
      Colonne       |   Type                   | Collationnement | NULL-able | Par défaut
--------------------+--------------------------+-----------------+-----------+------------
 locktype           | text                     |                 |           |
 database           | oid                      |                 |           |
 relation           | oid                      |                 |           |
 page               | integer                  |                 |           |
 tuple              | smallint                 |                 |           |
 virtualxid         | text                     |                 |           |
 transactionid      | xid                      |                 |           |
 classid            | oid                      |                 |           |
 objid              | oid                      |                 |           |
 objsubid           | smallint                 |                 |           |
 virtualtransaction | text                     |                 |           |
 pid                | integer                  |                 |           |
 mode               | text                     |                 |           |
 granted            | boolean                  |                 |           |
```

```
fastpath           | boolean                  |           |          |
waitstart          | timestamp with time zone |           |          |
```

- `locktype` est le type de verrou, les plus fréquents étant `relation` (table ou index), `transactionid` (transaction), `virtualxid` (transaction virtuelle, utilisée tant qu'une transaction n'a pas eu à modifier de données, donc à stocker des identifiants de transaction dans des enregistrements) ;
- `database` est la base dans laquelle ce verrou est pris ;
- `relation` est l'OID de la relation cible si locktype vaut `relation` (ou `page` ou `tuple`) ;
- `page` est le numéro de la page dans une relation (pour un verrou de type `page` ou `tuple`) cible ;
- `tuple` est le numéro de l'enregistrement cible (quand verrou de type `tuple`) ;
- `virtualxid` est le numéro de la transaction virtuelle cible (quand verrou de type `virtualxid`) ;
- `transactionid` est le numéro de la transaction cible ;
- `classid` est le numéro d'OID de la classe de l'objet verrouillé (autre que relation) dans `pg_class`. Indique le catalogue système, donc le type d'objet, concerné. Aussi utilisé pour les advisory locks ;
- `objid` est l'OID de l'objet dans le catalogue système pointé par `classid` ;
- `objsubid` correspond à l'ID de la colonne de l'objet `objid` concerné par le verrou ;
- `virtualtransaction` est le numéro de transaction virtuelle possédant le verrou (ou tentant de l'acquérir si `granted` vaut `f`) ;
- `pid` est le PID (l'identifiant de processus système) de la session possédant le verrou ;
- `mode` est le niveau de verrouillage demandé ;
- `granted` signifie si le verrou est acquis ou non (donc en attente) ;
- `fastpath` correspond à une information utilisée surtout pour le débogage (*fastpath* est le mécanisme d'acquisition des verrous les plus faibles) ;
- `waitstart` indique depuis quand le verrou est en attente.

La plupart des verrous sont de type relation, `transactionid` ou `virtualxid`. Une transaction qui démarre prend un verrou virtualxid sur son propre `virtualxid`. Elle acquiert des verrous faibles (`ACCESS SHARE`) sur tous les objets sur lesquels elle fait des `SELECT`, afin de garantir que leur structure n'est pas modifiée sur la durée de la transaction. Dès qu'une modification doit être faite, la transaction acquiert un verrou exclusif sur le numéro de transaction qui vient de lui être affecté. Tout objet modifié (table) sera verrouillé avec `ROW EXCLUSIVE`, afin d'éviter les `CREATE INDEX` non concurrents, et empêcher aussi les verrouillages manuels de la table en entier (`SHARE ROW EXCLUSIVE`).

4.11.4 VERROUS - PARAMÈTRES

- Nombre :
  - `max_locks_per_transaction` (+ paramètres pour la sérialisation)
- Durée :
  - `lock_timeout` (éviter l'empilement des verrous)
  - `deadlock_timeout` (défaut 1 s)
- Trace :
  - `log_lock_waits`

Nombre de verrous :

`max_locks_per_transaction` sert à dimensionner un espace en mémoire partagée destinée aux verrous sur des objets (notamment les tables). Le nombre de verrous est :

```
max_locks_per_transaction × max_connections
```

ou plutôt, si les transactions préparées sont activées (et `max_prepared_transactions` monté au-delà de 0) :

```
max_locks_per_transaction × (max_connections + max_prepared_transactions)
```

La valeur par défaut de 64 est largement suffisante la plupart du temps. Il peut arriver qu'il faille le monter, par exemple si l'on utilise énormément de partitions, mais le message d'erreur est explicite.

Le nombre maximum de verrous d'une session n'est pas limité à `max_locks_per_transaction`. C'est une valeur moyenne. Une session peut acquérir autant de verrous qu'elle le souhaite pourvu qu'au total la table de hachage interne soit assez grande. Les verrous de lignes sont stockés sur les lignes et donc potentiellement en nombre infini.

Pour la sérialisation, les verrous de prédicat possèdent des paramètres spécifiques. Pour économiser la mémoire, les verrous peuvent être regroupés par bloc ou relation (voir `pg_locks` pour le niveau de verrouillage). Les paramètres respectifs sont :

- `max_pred_locks_per_transaction` (64 par défaut) ;
- `max_pred_locks_per_page` (par défaut 2, donc 2 lignes verrouillées entraînent le verrouillage de tout le bloc, du moins pour la sérialisation) ;
- `max_pred_locks_per_relation` (voir la documentation[26] pour les détails).

Durées maximales de verrou :

Si une session attend un verrou depuis plus longtemps que `lock_timeout`, la requête est annulée. Il est courant de poser cela avant un ordre assez intrusif, même bref, sur une

[26] https://docs.postgresql.fr/current/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-RELATION

base utilisée. Par exemple, il faut éviter qu'un `VACUUM FULL`, s'il est bloqué par une transaction un peu longue, ne bloque lui-même toutes les transactions suivantes (phénomène d'empilement des verrous) :

```
postgres=# SET lock_timeout TO '3s' ;
SET
postgres=# VACUUM FULL t_grosse_table ;
ERROR:  canceling statement due to lock timeout
```

Il faudra bien sûr retenter le `VACUUM FULL` plus tard, mais la production n'est pas bloquée plus de 3 secondes.

PostgreSQL recherche périodiquement les *deadlocks* entre transactions en cours. La périodicité par défaut est de 1 s (paramètre `deadlock_timeout`), ce qui est largement suffisant la plupart du temps : les *deadlocks* sont assez rares, alors que la vérification est quelque chose de coûteux. L'une des transactions est alors arrêtée et annulée, pour que les autres puissent continuer :

```
postgres=*# DELETE FROM t_centmille_int WHERE i < 50000;
ERROR:  deadlock detected
DÉTAIL : Process 453259 waits for ShareLock on transaction 3010357;
blocked by process 453125.
Process 453125 waits for ShareLock on transaction 3010360;
blocked by process 453259.
ASTUCE : See server log for query details.
CONTEXTE : while deleting tuple (0,1) in relation "t_centmille_int"
```

Trace des verrous :

Pour tracer les attentes de verrous un peu longue, il est fortement conseillé de passer `log_lock_waits` à `on` (le défaut est `off`).

Le seuil est également défini par `deadlock_timeout` (1 s par défaut) Ainsi, une session toujours en attente de verrou au-delà de cette durée apparaîtra dans les traces :

```
LOG:  process 457051 still waiting for ShareLock on transaction 35373775
      after 1000.121 ms
DETAIL:  Process holding the lock: 457061. Wait queue: 457051.
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
```

S'il ne s'agit pas d'un *deadlock*, la transaction continuera, et le moment où elle obtiendra son verrou sera également tracé :

```
LOG:  process 457051 acquired ShareLock on transaction 35373775 after
      18131.402 ms
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
```

```
STATEMENT:  DELETE FROM t_centmille_int ;
LOG:  duration: 18203.059 ms  statement: DELETE FROM t_centmille_int ;
```

4.12 MÉCANISME TOAST

TOAST : *The Oversized-Attribute Storage Technique*

- Un enregistrement ne peut pas dépasser 8 ko (1 bloc)
- « Contournement » :
  - table de débordement `pg_toast_XXX` masquée
- Jusqu'à 1 Go par champ
  - texte, JSON, binaire…
- Compression optionnelle :
  - `zlib` : défaut
  - `lz4` (v14+) : généralement plus rapide
- Politiques `PLAIN`/`MAIN`/`EXTERNAL` ou `EXTENDED`

Principe du TOAST :

Une ligne ne peut déborder d'un bloc, et un bloc fait 8 ko (par défaut). Cela ne suffit pas pour certains champs beaucoup plus longs, comme certains textes, mais aussi des types composés (`json`, `jsonb`, `hstore`), ou binaires (`bytea`).

Le mécanisme TOAST consiste à déporter le contenu de certains champs d'un enregistrement vers une pseudo-table système associée à la table principale, de manière transparente pour l'utilisateur. Il permet d'éviter qu'un enregistrement ne dépasse la taille d'un bloc.

Le mécanisme TOAST a d'autres intérêts :

- la partie principale d'une table ayant des champs très longs est moins grosse, alors que les « gros champs » ont moins de chance d'être accédés systématiquement par le code applicatif ;
- ces champs peuvent être compressés de façon transparente, avec souvent de gros gains en place ;
- si un `UPDATE` ne modifie pas un de ces champs « toastés », la table TOAST n'est pas mise à jour : le pointeur vers l'enregistrement de cette table est juste « cloné » dans la nouvelle version de l'enregistrement.

Politiques de stockage :

Chaque champ possède une propriété de stockage :

```
CREATE TABLE unetable (i int, t text, b bytea, j jsonb);
# \d+ unetable

                    Table « public.unetable »
 Colonne |  Type   | Col... | NULL-able | Par défaut | Stockage | …
---------+---------+--------+-----------+------------+----------+--
 i       | integer |        |           |            | plain    |
 t       | text    |        |           |            | extended |
 b       | bytea   |        |           |            | extended |
 j2      | jsonb   |        |           |            | extended |
Méthode d'accès : heap
```

Les différentes politiques de stockage sont :

- `PLAIN` permettant de stocker uniquement dans la table, sans compression (champs numériques ou dates notamment) ;
- `MAIN` permettant de stocker dans la table tant que possible, éventuellement compressé (politique rarement utilisée) ;
- `EXTERNAL` permettant de stocker éventuellement dans la table TOAST, sans compression ;
- `EXTENDED` permettant de stocker éventuellement dans la table TOAST, éventuellement compressé (cas général des champs texte ou binaire).

Il est rare d'avoir à modifier ce paramétrage, mais cela arrive. Par exemple, certains longs champs (souvent binaires) se compressent si mal qu'il ne vaut pas la peine de gaspiller du CPU dans cette tâche. Dans le cas extrême où le champ compressé est plus grand que l'original, PostgreSQL revient à la valeur originale, mais là aussi il y a gaspillage. Il peut alors être intéressant de passer de `EXTENDED` à `EXTERNAL`, pour un gain de temps parfois non négligeable :

```
ALTER TABLE t1 ALTER COLUMN champ SET STORAGE EXTERNAL ;
```

Lors de ce changement, les données existantes ne sont pas affectées.

Les tables pg_toast_XXX :

Chaque table utilisateur 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. Tous les champs « toastés » d'une table se retrouvent dans la même table `pg_toast_XXX`, dans un espace de nommage séparé nommé `pg_toast`.

Pour l'utilisateur, les tables TOAST sont totalement transparentes. Un développeur doit juste savoir qu'il n'a pas besoin de déporter des champs texte (ou JSON, ou binaires…) imposants dans une table séparée pour des raisons de volumétrie de la table principale :

PostgreSQL le fait déjà, et de manière efficace ! Il est également souvent inutile de se donner la peine de compresser les données au niveau applicatif juste pour réduire le stockage.

La présence de ces tables n'apparaît guère que dans `pg_class`, par exemple ainsi :

```
SELECT * FROM pg_class c
WHERE c.relname = 'longs_textes'
OR c.oid = (SELECT reltoastrelid FROM pg_class
          WHERE relname = 'longs_textes');

-[ RECORD 1 ]-------+---------------
oid                 | 16614
relname             | longs_textes
relnamespace        | 2200
reltype             | 16616
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16614
reltablespace       | 0
relpages            | 35
reltuples           | 2421
relallvisible       | 35
reltoastrelid       | 16617
…
-[ RECORD 2 ]-------+---------------
oid                 | 16617
relname             | pg_toast_16614
relnamespace        | 99
reltype             | 16618
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16617
reltablespace       | 0
relpages            | 73161
reltuples           | 293188
relallvisible       | 73161
reltoastrelid       | 0
…
```

La partie TOAST est une table à part entière, avec une clé primaire. On ne peut ni ne doit y toucher !

```
\d+ pg_toast.pg_toast_16614

Table TOAST « pg_toast.pg_toast_16614 »
  Colonne   |  Type   | Stockage
```

```
------------+---------+----------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain

Table propriétaire : « public.textes_comp »
Index :
    "pg_toast_16614_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Méthode d'accès : heap
```

La volumétrie des différents éléments (partie principale, TOAST, index éventuels) peut se calculer grâce à cette requête dérivée du wiki[27] :

```
SELECT
    oid AS table_oid,
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltoastrelid,
    reltoastrelid::regclass::text AS table_toast,
    reltuples AS nb_lignes_estimees,
    pg_size_pretty(pg_table_size(c.oid)) AS " Table (dont TOAST)",
    pg_size_pretty(pg_relation_size(c.oid)) AS "  Heap",
    pg_size_pretty(pg_relation_size(reltoastrelid)) AS "  Toast",
    pg_size_pretty(pg_indexes_size(reltoastrelid)) AS  "  Toast (PK)",
    pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
    pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname = 'longs_textes'
\gx

-[ RECORD 1 ]------+------------------------
table_oid          | 16614
table              | public.longs_textes
reltoastrelid      | 16617
table_toast        | pg_toast.pg_toast_16614
nb_lignes_estimees | 2421
 Table (dont TOAST) | 578 MB
  Heap             | 280 kB
  Toast            | 572 MB
  Toast (PK)       | 6448 kB
 Index             | 560 kB
Total              | 579 MB
```

La taille des index sur les champs susceptibles d'être toastés est comptabilisée avec tous les index de la table (la clé primaire de la table TOAST est à part).

[27] https://wiki.postgresql.org/wiki/Disk_Usage

Les tables TOAST restent forcément dans le même tablespace que la table principale. Leur maintenance (notamment le nettoyage par `autovacuum`) s'effectue en même temps que la table principale, comme le montre un `VACUUM VERBOSE`.

Détails du mécanisme TOAST :

Les détails techniques du mécanisme TOAST sont dans la documentation officielle[28] . En résumé, le mécanisme TOAST est déclenché sur un enregistrement quand la taille d'un enregistrement dépasse 2 ko. Les champs « toastables » peuvent alors être compressés pour que la taille de l'enregistrement redescende en-dessous de 2 ko. Si cela ne suffit pas, des champs sont alors découpés et déportés vers la table TOAST. Dans ces champs de la table principale, l'enregistrement ne contient plus qu'un pointeur vers la table TOAST associée.

Un champ MAIN peut tout de même être stocké dans la table TOAST, si l'enregistrement dépasse 2 ko : mieux vaut « toaster » que d'empêcher l'insertion.

Cette valeur de 2 ko convient généralement. Au besoin, on peut l'augmenter (à partir de la version 11) en utilisant le paramètre de stockage `toast_tuple_target` ainsi :

```
ALTER TABLE t1 SET (toast_tuple_target = 3000);
```

mais cela est rarement utile.

Compression pgls vs lz4 :

Une nouveauté très intéressante de la version 14 permet de modifier l'algorithme de compression, défini par le nouveau paramètre `default_toast_compression`. La valeur par défaut est :

```
=# SHOW default_toast_compression ;

 default_toast_compression
---------------------------
 pglz
```

c'est-à-dire que PostgreSQL utilise la zlib, seule compression disponible jusqu'en version 13 incluse.

À partir de la version 14, il est souvent préférable d'utiliser `lz4`, un nouvel algorithme, si PostgreSQL a été compilé avec la bibliothèque du même nom (c'est le cas des paquets distribués par le PGDG).

L'activation demande soit de modifier la valeur par défaut dans `postgresql.conf` :

[28] https://doc.postgresql.fr/current/storage-toast.html

```
default_toast_compression = lz4
```

soit de déclarer la méthode de compression à la création de la table :

```
CREATE TABLE t1 (
  c1 bigint GENERATED ALWAYS AS identity,
  c2 text   COMPRESSION lz4
  ) ;
```

soit après coup :

```
ALTER TABLE t1 ALTER c2 SET COMPRESSION lz4 ;
```

De manière générale, l'algorithme `lz4` ne compresse pas mieux les données courantes, mais cela dépend des usages. Surtout, `lz4` est **beaucoup** plus rapide à compresser, et parfois à décompresser.

Par exemple, il peut accélérer une restauration logique avec beaucoup de données toastées et compressées. Si `lz4` n'a pas été activé par défaut, il peut être utilisé dès le chargement :

```
$ PGOPTIONS='-c default_toast_compression=lz4'  pg_restore …
```

> `lz4` est le choix à conseiller par défaut, même si, en toute rigueur, l'arbitrage entre consommations CPU en écriture ou lecture et place disque ne peut se faire qu'en testant soigneusement avec les données réelles.

Une table TOAST peut contenir un mélange de lignes compressées de manière différentes. En effet, l'utilisation `SET COMPRESSION` sur une colonne préexistante ne recompresse pas les données de la table TOAST. De plus, pendant une requête, des données toastées lues par une requête, puis réinsérées sans être modifiées, sont recopiées vers les champs cibles telles quelles, sans étapes de décompression/recompression, et ce même si la compression de la cible est différente. Il existe une fonction `pg_column_compression(nom_colonne)` pour consulter la compression d'un champ sur la ligne concernée.

Pour forcer la recompression de toutes les données d'une colonne, il faut modifier leur contenu, ce qui n'est pas forcément intéressant.

4.13 CONCLUSION

- PostgreSQL dispose d'une implémentation MVCC complète, permettant :
  - que les lecteurs ne bloquent pas les écrivains
  - que les écrivains ne bloquent pas les lecteurs
  - que les verrous en mémoire soient d'un nombre limité
- Cela impose par contre une mécanique un peu complexe, dont les parties visibles sont la commande `VACUUM` et le processus d'arrière-plan autovacuum.

4.13.1 QUESTIONS

N'hésitez pas, c'est le moment !

4.14 QUIZ

https://dali.bo/m4_quiz

4.15 TRAVAUX PRATIQUES

4.15.1 NIVEAUX D'ISOLATION READ COMMITTED ET REPEATABLE READ

> Créer une nouvelle base de données nommée `b2`.

> Se connecter à la base de données `b2`. Créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type text.

> Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

> Ouvrir une transaction.

> Lire les données de la table `t1`.

> Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

> Revenir à la première session et lire de nouveau toute la table `t1`.

> Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

> Lire les données de la table `t1`.

> Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

> Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

4.15.2 NIVEAU D'ISOLATION SERIALIZABLE (OPTIONNEL)

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette procédure de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement), 0)
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

> Au sein de trois transactions successives, Insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**.

> Chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`.
> La sécurité fonctionne-t-elle ?

> Dans deux sessions parallèles, pour le client **2**, procéder à deux retraits de 500 €. Appeler `CALL verifie_solde_positif (2);` dans les deux transactions, puis valider les deux. La règle du solde positif est-elle respectée ?

> Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
> Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :
>
> ```
> SELECT locktype, mode, pid, granted FROM pg_locks
> WHERE relation = (SELECT oid FROM pg_class WHERE relname =
> 'mouvements_comptes') ;
> ```

4.15.3 EFFETS DE MVCC

> Créer une nouvelle table `t2` avec les mêmes colonnes que la table `t1`.

> Insérer 5 lignes dans la table `t2` de `(1, 'un')` à `(5, 'cinq')`.

> Lire les données de la table `t2`.

> Commencer une transaction et mettre en majuscules le texte de la troisième ligne de la table `t2`.

> Lire les données de la table `t2`. Que faut-il remarquer ?

> Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

> Récupérer quelques informations systèmes (`xmin` et `xmax`) pour les deux sessions lors de la lecture des données de la table `t2`.

> Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`.

> Valider la transaction.

> Installer l'extension `pageinspect`.

> À l'aide de la documentation de l'extension sur https://docs.postgresql.fr/current/pageinspect.html, et des fonctions `get_raw_page` et `heap_page_items`, décoder le bloc 0 de la table `t2`. Que faut-il remarquer ?

4.15.4 VERROUS

> Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

> Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

> Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

> Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

> Récupérer la liste des verrous en attente pour la requête bloquée.

> Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

> Récupérer la liste des verrous sur cet objet. Quel processus a verrouillé la table `t1` ?

> Retrouver les informations sur la session bloquante.

> Retrouver cette information avec la fonction `pg_blocking_pids`.

> Détruire la session bloquant le `DROP TABLE`.

> Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

> Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

4.16 TRAVAUX PRATIQUES (SOLUTIONS)

4.16.1 NIVEAUX D'ISOLATION READ COMMITTED ET REPEATABLE READ

> Créer une nouvelle base de données nommée `b2`.

```
# createdb b2
```

> Se connecter à la base de données `b2`. Créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type text.

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

```
CREATE TABLE
```

> Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

```
INSERT INTO t1 (c1, c2) VALUES
  (1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
```

```
INSERT 0 5
```

> Ouvrir une transaction.

```
BEGIN;
```

```
BEGIN
```

> Lire les données de la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

> Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

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

```
UPDATE 1
```

> Revenir à la première session et lire de nouveau toute la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

Les modifications réalisées par la deuxième transaction sont immédiatement visibles par la première transaction. C'est le cas des transactions en niveau d'isolation READ COMMITED.

> Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

```
ROLLBACK;
```

```
ROLLBACK
```

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

> Lire les données de la table `t1`.

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

> Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

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

```
UPDATE 1
```

> Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

```
SELECT * FROM t1;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

En niveau d'isolation `REPEATABLE READ`, la transaction est certaine de ne pas voir les modifications réalisées par d'autres transactions (à partir de la première lecture de la table).

4.16.2 NIVEAU D'ISOLATION SERIALIZABLE (OPTIONNEL)

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette fonction de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement),0)
```

```
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

> Au sein de trois transactions successives, Insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**.
> Chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`.
> La sécurité fonctionne-t-elle ?

Ce client a bien 600 € :

```
SELECT * FROM mouvements_comptes WHERE client = 1 ;
```

```
 client | mouvement
--------+-----------
      1 |       100
      1 |       200
      1 |       300
```

Première transaction :

```
BEGIN ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (1, -300) ;
CALL verifie_solde_positif (1) ;
```

```
NOTICE:  Client 1 - Solde positif : 300
CALL
```

```
COMMIT ;
```

Lors d'une seconde transaction : les mêmes ordres renvoient :

```
NOTICE:  Client 1 - Solde positif : 0
```

Avec le troisième débit :

```
BEGIN ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (1, -300) ;
```

```
CALL verifie_solde_positif (1) ;

ERROR:  Client 1 - Solde négatif : -300 !
CONTEXTE : PL/pgSQL function verifie_solde_positif(integer) line 11 at RAISE
```

La transaction est annulée : il est interdit de retirer plus d'argent qu'il n'y en a.

> Dans deux sessions parallèles, pour le client **2**, procéder à deux retraits de 500 €. Appeler `CALL verifie_solde_positif (2);` dans les deux transactions, puis valider les deux. La règle du solde positif est-elle respectée ?

Chaque transaction va donc se dérouler dans une session différente.

Première transaction :

```
BEGIN ; --session 1
INSERT INTO mouvements_comptes(client, mouvement) VALUES (2, -500) ;
CALL verifie_solde_positif (2) ;

NOTICE:  Client 2 - Solde positif : 100
```

On ne commite pas encore.

Dans la deuxième session, il se passe exactement la même chose :

```
BEGIN ; --session 2
INSERT INTO mouvements_comptes(client, mouvement) VALUES (2, -500) ;
CALL verifie_solde_positif (2) ;

NOTICE:  Client 2 - Solde positif : 100
```

En effet, cette deuxième session ne voit pas encore le débit de la première session.

Les deux tests étant concluants, les deux sessions committent :

```
COMMIT ; --session 1

COMMIT

COMMIT ; --session 2

COMMIT
```

Au final, le solde est négatif, ce qui est pourtant strictement interdit !

```
CALL verifie_solde_positif (2) ;

ERROR:  Client 2 - Solde négatif : -400 !
CONTEXTE : PL/pgSQL function verifie_solde_positif(integer) line 11 at RAISE
```

Les deux sessions en parallèle sont donc un moyen de contourner la sécurité, qui porte sur le résultat d'un ensemble de lignes, et non juste sur la ligne concernée.

> Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
> Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :
> `SELECT locktype, mode, pid, granted FROM pg_locks WHERE relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;`

Première session :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (3, -500) ;
CALL verifie_solde_positif (3) ;

NOTICE:  Client 3 - Solde positif : 100
```

On ne committe pas encore.

Deuxième session :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
INSERT INTO mouvements_comptes(client, mouvement) VALUES (3, -500) ;
CALL verifie_solde_positif (3) ;

NOTICE:  Client 3 - Solde positif : 100
```

Les verrous en place sont les suivants :

```
SELECT locktype, mode, pid, granted
FROM   pg_locks
WHERE  relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;

 locktype |       mode       |   pid   | granted
----------+------------------+---------+---------
 relation | AccessShareLock  | 28304 | t
 relation | RowExclusiveLock | 28304 | t
 relation | AccessShareLock  | 28358 | t
 relation | RowExclusiveLock | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

```
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

`SIReadLock` est un verrou lié à la sérialisation : noter qu'il porte sur des lignes, portées par les deux sessions. `AccessShareLock` empêche surtout de supprimer la table. `RowExclusiveLock` est un verrou de ligne.

Validation de la première session :

```
COMMIT ;

COMMIT
```

Dans les verrous, il subsiste toujours les verrous `SIReadLock` de la session de PID 28304, qui a pourtant committé :

```
SELECT locktype, mode, pid, granted
FROM   pg_locks
WHERE  relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes') ;

 locktype |       mode       |   pid   | granted
----------+------------------+---------+---------
 relation | AccessShareLock  | 28358 | t
 relation | RowExclusiveLock | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
 tuple    | SIReadLock       | 28358 | t
 tuple    | SIReadLock       | 28304 | t
```

Tentative de validation de la seconde session :

```
COMMIT ;

ERROR:  could not serialize access due to read/write dependencies among transactions
DÉTAIL : Reason code: Canceled on identification as a pivot, during commit attempt.
ASTUCE : The transaction might succeed if retried.
```

La transaction est annulée pour erreur de sérialisation. En effet, le calcul effectué pendant la seconde transaction n'est plus valable puisque la première a modifié les lignes qu'elle a lues.

La transaction annulée doit être rejouée de zéro, et elle tombera alors bien en erreur.

4.16.3 EFFETS DE MVCC

> Créer une nouvelle table `t2` avec les mêmes colonnes que la table `t1`.

```
CREATE TABLE t2 (LIKE t1);
```

```
CREATE TABLE
```

> Insérer 5 lignes dans la table `t2` de `(1, 'un')` à `(5, 'cinq')`.

```
INSERT INTO t2(c1, c2) VALUES
  (1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre'), (5, 'cinq');
```

```
INSERT 0 5
```

> Lire les données de la table `t2`.

```
SELECT * FROM t2;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

> Commencer une transaction et mettre en majuscules le texte de la troisième ligne de la table `t2`.

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

```
UPDATE 1
```

> Lire les données de la table `t2`. Que faut-il remarquer ?

```
SELECT * FROM t2;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  4 | quatre
  5 | cinq
  3 | TROIS
```

La ligne mise à jour n'apparaît plus, ce qui est normal. Elle apparaît en fin de table. En effet, quand un `UPDATE` est exécuté, la ligne courante est considérée comme morte et une nouvelle ligne est ajoutée, avec les valeurs modifiées. Comme nous n'avons pas demandé de récupérer les résultats dans un certain ordre, les lignes sont affichées dans leur ordre de stockage dans les blocs de la table.

> Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

```
SELECT * FROM t2;
```

```
 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | trois
  4 | quatre
  5 | cinq
```

Les autres sessions voient toujours l'ancienne version de la 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.

> Récupérer quelques informations systèmes (`xmin` et `xmax`) pour les deux sessions lors de la lecture des données de la table `t2`.

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

```
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
```

Et voici ce que renvoie l'autre session :

```
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
```

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.

> Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`.

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

```
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
```

Et voici ce que renvoie l'autre session :

```
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
```

La colonne `ctid` contient une paire d'entiers. Le premier indique le numéro de bloc, le second le numéro de l'enregistrement dans le bloc. Autrement dit, elle précise la position de l'enregistrement sur le fichier de la table.

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

> Valider la transaction.

```
COMMIT;
```

```
COMMIT
```

> Installer l'extension `pageinspect`.

```
CREATE EXTENSION pageinspect;
```

```
CREATE EXTENSION
```

> À l'aide de la documentation de l'extension sur https://docs.postgresql.fr/current/pageinspect.html, et des fonctions `get_raw_page` et `heap_page_items`, décoder le bloc 0 de la table `t2`. Que faut-il remarquer ?

```
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 |  t_data
----+-------------+------------+--------+--------+-------+-------------------------
  1 |           2 |       2306 |     24 |        |       | \x0100000007756e
  2 |           2 |       2306 |     24 |        |       | \x020000000b64657578
  3 |       16386 |        258 |     24 |        |       | \x030000000d74726f6973
  4 |           2 |       2306 |     24 |        |       | \x040000000f717561747265
  5 |           2 |       2306 |     24 |        |       | \x050000000b63696e71
  6 |       32770 |      10242 |     24 |        |       | \x030000000d54524f4953
```

- Les six lignes sont bien présentes, dont les deux versions de la ligne 3 ;
- Le `t_ctid` ne contient plus (0,3) mais l'adresse de la nouvelle ligne (soit (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 (la nouvelle version de la ligne est maintenue dans le même bloc et un chaînage depuis l'ancienne est effectué) ;
- Le champ `t_data` contient les valeurs de la ligne : nous devinons `c1` au début (01 à 05), et la fin correspond aux chaînes de caractères, précédée d'un octet lié à la taille.

4.16.4 VERROUS

> Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

```
BEGIN;
SELECT * FROM t1;

 c1 |   c2
----+--------
  1 | un
  2 | deux
  3 | TROIS
  4 | QUATRE
  5 | CINQ
```

> Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

```
DROP TABLE t1;
```

La suppression semble bloquée.

> Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

En tant qu'utilisateur système `postgres` :

```
$ ps -o pid,cmd fx
  PID CMD
 2657 -bash
 2693  \_ psql
 2431 -bash
 2622  \_ psql
 2728      \_ ps -o pid,cmd fx
 2415 /usr/pgsql-11/bin/postmaster -D /var/lib/pgsql/11/data/
 2417  \_ postgres: logger
 2419  \_ postgres: checkpointer
 2420  \_ postgres: background writer
 2421  \_ postgres: walwriter
 2422  \_ postgres: autovacuum launcher
 2423  \_ postgres: stats collector
 2424  \_ postgres: logical replication launcher
 2718  \_ postgres: postgres b2 [local] DROP TABLE waiting
 2719  \_ postgres: postgres b2 [local] idle in transaction
```

La ligne intéressante est la ligne du `DROP TABLE`. Elle contient le mot clé `waiting`. Ce dernier indique que l'exécution de la requête est en attente d'un verrou sur un objet.

> Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

```
\x

Expanded display is on.

SELECT * FROM pg_stat_activity
  WHERE application_name='psql' AND wait_event IS NOT NULL;

-[ RECORD 1 ]----+------------------------------
datid            | 16387
datname          | b2
pid              | 2718
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:45.38342+00
xact_start       | 2018-11-02 15:57:32.82511+00
query_start      | 2018-11-02 15:57:32.82511+00
state_change     | 2018-11-02 15:57:32.825112+00
wait_event_type  | Lock
wait_event       | relation
state            | active
backend_xid      | 575
backend_xmin     | 575
query_id         |
query            | drop table t1 ;
backend_type     | client backend
-[ RECORD 2 ]----+------------------------------
datid            | 16387
datname          | b2
pid              | 2719
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:17.173784+00
xact_start       | 2018-11-02 15:57:25.311573+00
```

```
query_start      | 2018-11-02 15:57:25.311573+00
state_change     | 2018-11-02 15:57:25.311573+00
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT * FROM t1;
backend_type     | client backend
```

> Récupérer la liste des verrous en attente pour la requête bloquée.

```
SELECT * FROM pg_locks WHERE pid = 2718 AND NOT granted;

-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 5/7
pid                | 2718
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
waitstart          |
```

> Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

```
SELECT relname FROM pg_class WHERE oid=16394;

-[ RECORD 1 ]
relname | t1
```

Noter que l'objet n'est visible dans `pg_class` que si l'on est dans la même base de données que lui. D'autre part, la colonne `oid` des tables systèmes n'est pas visible par défaut dans les versions antérieures à la 12, il faut demander explicitement son affichage pour la voir.

> Récupérer la liste des verrous sur cet objet. Quel processus a verrouillé la table `t1` ?

```
SELECT * FROM pg_locks WHERE relation = 16394;
```

```
-[ RECORD 1 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 4/10
pid                | 2719
mode               | AccessShareLock
granted            | t
fastpath           | f
waitstart          |
-[ RECORD 2 ]------+--------------------
locktype           | relation
database           | 16387
relation           | 16394
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 5/7
pid                | 2718
mode               | AccessExclusiveLock
granted            | f
fastpath           | f
waitstart          |
```

Le processus de PID 2718 (le `DROP TABLE`) demande un verrou exclusif sur `t1`, mais ce verrou n'est pas encore accordé (`granted` est à `false`). La session `idle in transaction` a acquis un verrou `Access Share`, normalement peu gênant, qui n'entre en conflit qu'avec les verrous exclusifs.

> Retrouver les informations sur la session bloquante.

On retrouve les informations déjà affichées :

```
SELECT * FROM pg_stat_activity WHERE pid = 2719;

-[ RECORD 1 ]----+------------------------------
datid            | 16387
datname          | b2
pid              | 2719
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-11-02 15:56:17.173784+00
xact_start       | 2018-11-02 15:57:25.311573+00
query_start      | 2018-11-02 15:57:25.311573+00
state_change     | 2018-11-02 15:57:25.311573+00
wait_event_type  | Client
wait_event       | ClientRead
state            | idle in transaction
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT * FROM t1;
backend_type     | client backend
```

> Retrouver cette information avec la fonction `pg_blocking_pids`.

Il existe une fonction pratique indiquant quelles sessions bloquent une autre. En l'occurence, notre `DROP TABLE t1` est bloqué par :

```
SELECT pg_blocking_pids(2718);

-[ RECORD 1 ]----+-------
pg_blocking_pids | {2719}
```

Potentiellement, la session pourrait attendre la levée de plusieurs verrous de différentes sessions.

> Détruire la session bloquant le `DROP TABLE`.

À partir de là, il est possible d'annuler l'exécution de l'ordre bloqué, le `DROP TABLE`, avec la fonction `pg_cancel_backend()`. Si l'on veut détruire le processus bloquant, il faudra plutôt utiliser la fonction `pg_terminate_backend()` :

```
SELECT pg_terminate_backend (2719) ;
```

Dans ce dernier cas, vérifiez que la table a été supprimée, et que la session en statut `idle in transaction` affiche un message indiquant la perte de la connexion.

> Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

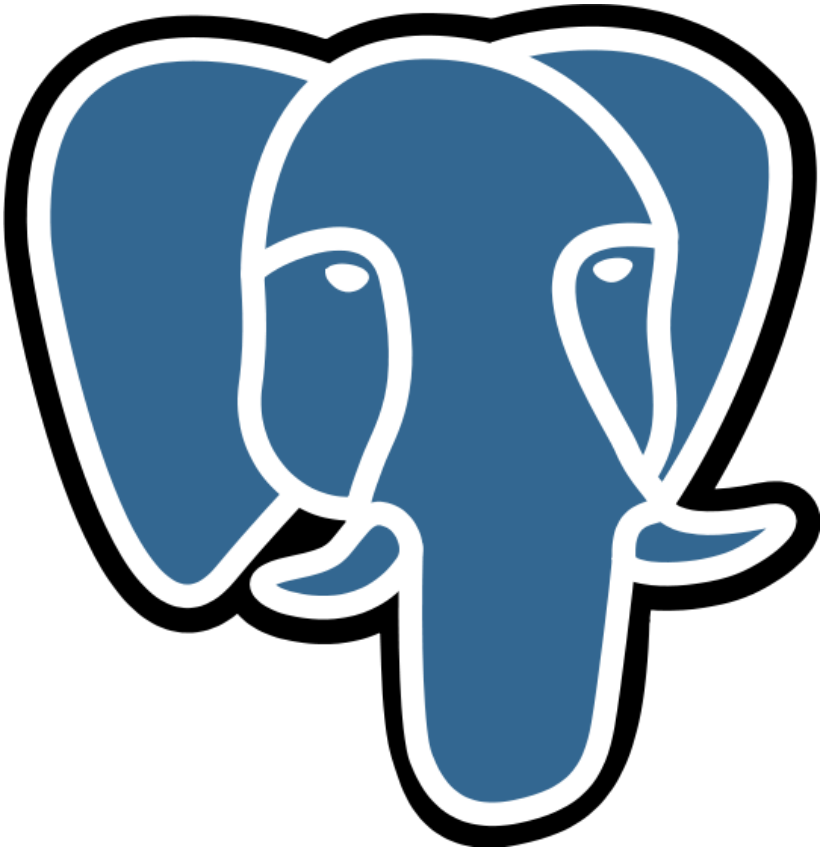
```
LOCK TABLE t1;
```

> Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

Le code source de la vue `pg_show_locks` est le suivant :

```
CREATE VIEW pg_show_locks as
SELECT
        a.pid,
        usename,
        (now() - query_start) as age,
        c.relname,
        l.mode,
        l.granted
FROM
        pg_stat_activity a
        LEFT OUTER JOIN pg_locks l
                ON (a.pid = l.pid)
        LEFT OUTER JOIN pg_class c
                ON (l.relation = c.oid)
WHERE
        c.relname !~ '^pg_'
ORDER BY
        pid;
```

5 VACUUM ET AUTOVACUUM



5.1 AU MENU

- Principe & fonctionnement du VACUUM
- Options : VACUUM seul, ANALYZE, FULL, FREEZE
  - ne pas les confondre !
- Suivi
- Autovacuum
- Paramétrages

VACUUM est la contrepartie de la flexibilité du modèle MVCC. Derrière les différentes options de VACUUM se cachent plusieurs tâches très différentes. Malheureusement, la confusion est facile. Il est capital de les connaître et de comprendre leur fonctionnement.

Autovacuum permet d'automatiser le VACUUM et allège considérablement le travail de l'administrateur.

Il fonctionne généralement bien, mais il faut savoir le surveiller et l'optimiser.

5.2 VACUUM ET AUTOVACUUM

- VACUUM : nettoie d'abord les lignes mortes
- Mais aussi d'autres opérations de maintenance
- Lancement
  - manuel
  - par le démon autovacuum (seuils)

VACUUM est né du besoin de nettoyer les lignes mortes. Au fil du temps il a été couplé à d'autres ordres (ANALYZE, VACUUM FREEZE) et s'est occupé d'autres opérations de maintenance (création de la *visibility map* par exemple).

autovacuum est un processus de l'instance PostgreSQL. Il est activé par défaut, et il fortement conseillé de le conserver ainsi. Dans le cas général, son fonctionnement convient et il ne gênera pas les utilisateurs.

L'autovacuum ne gère pas toutes les variantes de VACUUM (notamment pas le FULL).

5.3 FONCTIONNEMENT DE VACUUM

Un ordre VACUUM vise d'abord à nettoyer les lignes mortes.

Le traitement VACUUM se déroule en trois passes. Cette première passe parcourt la table à nettoyer, à la recherche d'enregistrements morts. Un enregistrement est mort s'il possède un xmax qui correspond à une transaction validée, et que cet enregistrement n'est plus visible dans l'instantané d'aucune transaction en cours sur la base. D'autres lignes mortes portent un xmin d'une transaction annulée.

L'enregistrement mort ne peut pas être supprimé immédiatement : des enregistrements d'index pointent vers lui et doivent aussi être nettoyés. La session efffectuant le vac-

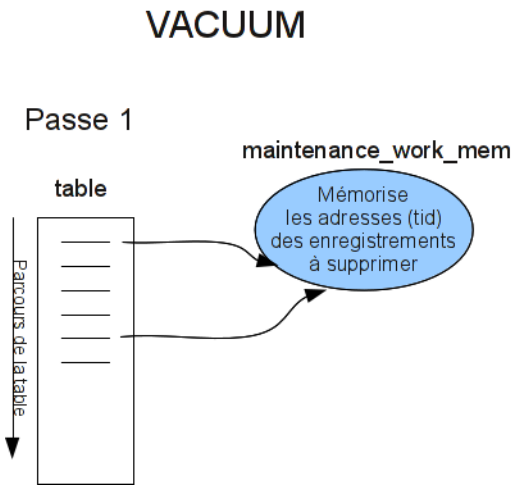


Figure 3: Phase 1/3 : recherche des enregistrements morts

uum garde en mémoire la liste des adresses des enregistrements morts, à hauteur d'une quantité indiquée par le paramètre `maintenance_work_mem`. Si cet espace est trop petit pour contenir tous les enregistrements morts, `VACUUM` effectue plusieurs séries de ces trois passes.

5.3.1 FONCTIONNEMENT DE VACUUM (SUITE)

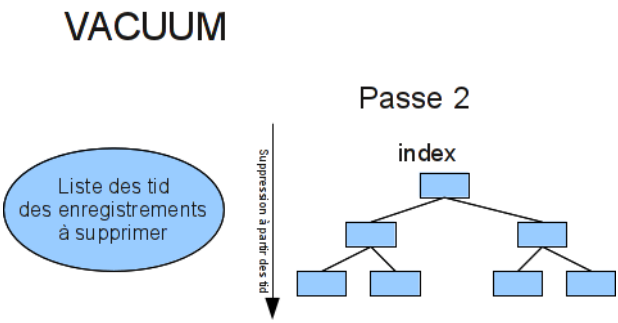


Figure 4: Phase 2/3 : nettoyage des index

La seconde passe se charge de nettoyer les entrées d'index. `VACUUM` possède une liste de `tid` (`tuple id`) à invalider. Il parcourt donc tous les index de la table à la recherche de ces `tid` et les supprime. En effet, les index sont triés afin de mettre en correspondance une valeur de clé (la colonne indexée par exemple) avec un `tid`. Il n'est par contre pas possible de trouver un `tid` directement. Les pages entièrement vides sont supprimées de l'arbre et stockées dans la liste des pages réutilisables, la *Free Space Map* (FSM).

Cette phase peut être ignorée par deux mécanismes. Le premier mécanisme apparaît en version 12 où l'option `INDEX_CLEANUP` a été ajoutée. Ce mécanisme est donc manuel et permet de gagner du temps sur l'opération de `VACUUM`. Cette option s'utilise ainsi :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) nom_table ;
```

À partir de la version 14, un autre mécanisme, automatique cette fois, a été ajouté. Le but est toujours d'exécuter rapidement le `VACUUM`, mais uniquement pour éviter le wraparound. Quand la table atteint l'âge, très élevé, de 1,6 milliard de transactions (défaut des paramètres `vacuum_failsafe_age` et `vacuum_multixact_failsafe_age`), un `VACUUM` simple va automatiquement désactiver le nettoyage des index pour nettoyer plus rapidement la table et permettre d'avancer l'identifiant le plus ancien de la table.

À partir de la version 13, cette phase peut être parallélisée (clause `PARALLEL`), chaque index pouvant être traité par un CPU.

5.3.2 FONCTIONNEMENT DE VACUUM (SUITE)

- NB : L'espace est rarement rendu à l'OS !

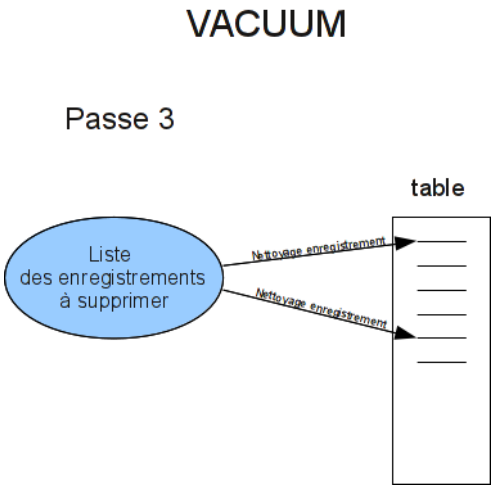


Figure 5: Phase 3/3 : suppression des enregistrements morts

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements morts identifiés, ceux-ci peuvent disparaître. C'est le rôle de cette passe. Quand un enregistrement est supprimé d'un bloc, ce bloc est réorganisé afin de consolider l'espace libre, qui est renseigné dans la *Free Space Map* (FSM).

Une fois cette passe terminée, si le parcours de la table n'a pas été terminé lors de la passe précédente, le travail reprend où il en était du parcours de la table.

Si les derniers blocs de la table sont vides, ils sont rendus au système (si le verrou nécessaire peut être obtenu, et si l'option `TRUNCATE` n'est pas `off`). C'est le seul cas où `VACUUM` réduit la taille de la table. Les espaces vides (et réutilisables) au milieu de la table constituent le *bloat* (littéralement « boursouflure » ou « gonflement », que l'on peut aussi traduire par fragmentation).

Les statistiques d'activité sont aussi mises à jour.

5.4 LES OPTIONS DE VACUUM

Différentes opérations :

- `VACUUM`
  - lignes mortes, *visibility map*, *hint bits*
- `ANALYZE`
  - statistiques
- `FREEZE`
  - gel des lignes
  - parfois gênant ou long
- `FULL`
  - bloquant !
  - non lancé par l'autovacuum

VACUUM

Par défaut, `VACUUM` procède principalement au nettoyage des lignes mortes. Pour que cela soit efficace, il met à jour la *visibility map*, et la crée au besoin. Au passage, il peut geler certaines lignes rencontrées.

L'autovacuum le déclenchera sur les tables en fonction de l'activité.

Le verrou SHARE UPDATE EXCLUSIVE posé protège la table contre les modifications simultanées du schéma, et ne gêne généralement pas les opérations, sauf les plus intrusives (il empêche par exemple un `LOCK TABLE`). L'autovacuum arrêtera spontanément un `VACUUM` qu'il aurait lancé et qui gênerait ; mais un `VACUUM` lancé manuellement continuera jusqu'à la fin.

VACUUM ANALYZE

`ANALYZE` existe en tant qu'ordre séparé, pour rafraîchir les statistiques sur un échantillon des données, à destination de l'optimiseur. L'autovacuum se charge également de lancer des `ANALYZE` en fonction de l'activité.

L'ordre `VACUUM ANALYZE` (ou `VACUUM (ANALYZE)`) force le calcul des statistiques sur les données en même temps que le `VACUUM`.

VACUUM FREEZE

`VACUUM FREEZE` procède au « gel » des lignes visibles par toutes les transactions en cours sur l'instance, afin de parer au problème du *wraparound* des identifiants de transaction Concrètement, il indique dans un *hint bit* de chaque ligne qu'elle est plus vieille que tous les numéros de transactions actuellement actives (avant la 9.4, la colonne système `xmin` était remplacée par un `FrozenXid`).

Un ordre FREEZE n'existe pas en tant que tel.

Préventivement, lors d'un VACUUM simple, l'autovacuum procède au gel de certaines des lignes rencontrées. De plus, il lancera un VACUUM FREEZE sur une table dont les plus vieilles transactions dépassent un certain âge. Ce peut être très long, et très lourd en écritures si une grosse table doit être entièrement gelée d'un coup. Autrement, l'activité n'est qu'exceptionnellement gênée (voir plus bas).

VACUUM FULL

L'ordre VACUUM FULL permet de reconstruire la table sans les espaces vides. C'est une opération très lourde, risquant de bloquer d'autres requêtes à cause du verrou exclusif qu'elle pose (on ne peut même plus lire la table !), mais il s'agit de la seule option qui permet de réduire la taille de la table au niveau du système de fichiers de façon certaine.

Il faut prévoir l'espace disque (la table est reconstruite à côté de l'ancienne, puis l'ancienne est supprimée). Les index sont reconstruits au passage.

L'autovacuum ne lancera jamais un VACUUM FULL !

Il existe aussi un ordre CLUSTER, qui permet en plus de trier la table suivant un des index.

5.4.1 AUTRES OPTIONS DE VACUUM

- VERBOSE
- Optimisations :
  - PARALLEL (v13+)
  - INDEX_CLEANUP
  - PROCESS_TOAST (v14+)
  - TRUNCATE (v12+)
- Ponctuellement :
  - SKIP_LOCKED (v12+), DISABLE_PAGE_SKIPPING (v11+)

VERBOSE :

Cette option affiche un grand nombre d'informations sur ce que fait la commande. En général c'est une bonne idée de l'activer :

```
VACUUM (VERBOSE) pgbench_accounts_5 ;

INFO:  vacuuming "public.pgbench_accounts_5"
INFO:  scanned index "pgbench_accounts_5_pkey" to remove 9999999 row versions
DÉTAIL : CPU: user: 12.16 s, system: 0.87 s, elapsed: 18.15 s
INFO:  "pgbench_accounts_5": removed 9999999 row versions in 163935 pages
```

```
DÉTAIL : CPU: user: 0.16 s, system: 0.00 s, elapsed: 0.20 s
INFO:  index "pgbench_accounts_5_pkey" now contains 100000000 row versions in 301613 pages
DÉTAIL : 9999999 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "pgbench_accounts_5": found 10000001 removable,
       10000051 nonremovable row versions in 327870 out of 1803279 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 1071186825
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 1475409 frozen pages.
0 pages are entirely empty.
CPU: user: 13.77 s, system: 0.89 s, elapsed: 19.81 s.
VACUUM
```

PARALLEL :

Apparue avec PostgreSQL 13, l'option `PARALLEL` permet le traitement parallélisé des index. Le nombre indiqué après `PARALLEL` précise le niveau de parallélisation souhaité. Par exemple :

```
VACUUM (VERBOSE, PARALLEL 4) matable ;

INFO:  vacuuming "public.matable"
INFO:  launched 3 parallel vacuum workers for index cleanup (planned: 3)
```

DISABLE_PAGE_SKIPPING :

Par défaut, PostgreSQL ne traite que les blocs modifiés depuis le dernier `VACUUM`, ce qui est un gros gain en performance (l'information est stockée dans la *Visibility Map*).

À partir de la version 11, activer l'option `DISABLE_PAGE_SKIPPING` force l'analyse de tous les blocs de la table. La table est intégralement reparcourue. Ce peut être utile en cas de problème, notamment pour reconstruire cette *Visibility Map*.

SKIP_LOCKED :

À partir de la version 12, l'option `SKIP_LOCKED` permet d'ignorer toute table pour laquelle la commande `VACUUM` ne peut pas obtenir immédiatement son verrou. Cela évite de bloquer le `VACUUM` sur une table, et peut éviter un empilement des verrous derrière celui que le `VACUUM` veut poser, surtout en cas de `VACUUM FULL`. La commande passe alors à la table suivante à traiter. Exemple :

```
# VACUUM (FULL, SKIP_LOCKED) t_un_million_int, t_cent_mille_int ;

WARNING:  skipping vacuum of "t_un_million_int" --- lock not available
VACUUM
```

Une autre technique est de paramétrer dans la session un petit délai avant abandon :

```
SET lock_timeout TO '100ms' ;
```

INDEX_CLEANUP :

L'option `INDEX_CLEANUP` (par défaut à `on` jusque PostgreSQL 13 compris) déclenche systématiquement le Quand il faut nettoyer des lignes mortes urgemment dans une grosse table, la valeur `off` fait gagner beaucoup de temps :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) unetable ;
```

Les index peuvent être nettoyés plus tard par un autre `VACUUM`, ou reconstruits.

Cette option existe aussi sous la forme d'un paramètre de stockage (`vacuum_index_cleanup`) propre à la table pour que l'autovacuum en tienne aussi compte.

En version 14, le nouveau défaut est `auto`, qui indique que PostgreSQL doit décider de faire ou non le nettoyage des index suivant la quantité d'entrées à nettoyer. Il faut au minimum 2 % d'éléments à nettoyer pour que le nettoyage ait lieu.

PROCESS_TOAST :

Cette option active ou non le traitement de la partie TOAST associée à la table. Elle est activée par défaut. Son utilité est la même que pour `INDEX_CLEANUP`.

TRUNCATE :

L'option `TRUNCATE` (à `on` par défaut) permet de tronquer les derniers blocs vides d'une table. `TRUNCATE off` évite d'avoir à poser un verrou exclusif certes court, mais parfois gênant.

Cette option existe aussi sous la forme d'un paramètre de stockage de table (`vacuum_truncate`).

Mélange des options :

Il est possible de mixer ces options presque à volonté et de préciser plusieurs tables à nettoyer :

```
VACUUM (VERBOSE, ANALYZE, INDEX_CLEANUP off, TRUNCATE off,
        DISABLE_PAGE_SKIPPING) bigtable, smalltable ;
```

5.5 SUIVI DU VACUUM

- `pg_stat_activity` ou `top`
- La table est-elle suffisamment nettoyée ?
- Vue `pg_stat_user_tables`
  - `last_vacuum` / `last_autovacuum`
  - `last_analyze` / `last_autoanalyze`
- `log_autovacuum_min_duration`

Un `VACUUM`, y compris lancé par l'autovacuum, apparaît dans `pg_stat_activity` et le processus est visible comme processus système avec `top` ou `ps` :

```
$ ps faux
…
postgres 3470724 0.0 0.0 12985308 6544  ? Ss 13:58 0:02 \_ postgres: 13/main: autovacuum launcher
postgres  795432 7.8 0.0 14034140 13424 ? Rs 16:22 0:01 \_ postgres: 13/main: autovacuum worker
                                                                                    pgbench1000p10
…
```

Il est fréquent de se demander si l'autovacuum s'occupe suffisamment d'une table qui grossit ou dont les statistiques semblent périmées. La vue `pg_stat_user_tables` contient quelques informations. Dans l'exemple ci-dessous, nous distinguons les dates des `VACUUM` et `ANALYZE` déclenchés automatiquement ou manuellement (en fait par l'application `pgbench`). Si 44 305 lignes ont été modifiées depuis le rafraîchissement des statistiques, il reste 2,3 millions de lignes mortes à nettoyer (contre 10 millions vivantes).

```
# SELECT * FROM pg_stat_user_tables WHERE relname ='pgbench_accounts' \gx

-[ RECORD 1 ]-------+------------------------------
relid               | 489050
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 1
seq_tup_read        | 10
idx_scan            | 686140
idx_tup_fetch       | 2686136
n_tup_ins           | 0
n_tup_upd           | 2343090
n_tup_del           | 452
n_tup_hot_upd       | 118551
n_live_tup          | 10044489
n_dead_tup          | 2289437
n_mod_since_analyze | 44305
n_ins_since_vacuum  | 452
last_vacuum         | 2020-01-06 18:42:50.237146+01
last_autovacuum     | 2020-01-07 14:30:30.200728+01
```

```
last_analyze        | 2020-01-06 18:42:50.504248+01
last_autoanalyze    | 2020-01-07 14:30:39.839482+01
vacuum_count        | 1
autovacuum_count    | 1
analyze_count       | 1
autoanalyze_count   | 1
```

Activer le paramètre `log_autovacuum_min_duration` avec une valeur relativement faible (dépendant des tables visées de la taille des logs générés), voire le mettre à 0, est également courant et conseillé.

5.5.1 PROGRESSION DU VACUUM

- Pour `VACUUM` simple / `VACUUM FREEZE`
  - vue `pg_stat_progress_vacuum`
  - blocs parcourus / nettoyés
  - nombre de passes dans l'index
- Partie `ANALYZE`
  - `pg_stat_progress_analyze` (v13)
- Manuel ou via autovacuum
- Pour `VACUUM FULL`
  - vue `pg_stat_progress_cluster` (v12)

La vue `pg_stat_progress_vacuum` contient une ligne par `VACUUM` (simple ou `FREEZE`) en cours d'exécution.

Voici un exemple :

```
SELECT * FROM pg_stat_progress_vacuum ;

-[ RECORD 1 ]------+--------------
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 86 665 blocs (soit 68 % de la table), et en a traité 86 664.

Dans le cas d'un `VACUUM ANALYZE`, la seconde partie de recueil des statistiques pourra être suivie dans `pg_stat_progress_analyze` (à partir de PostgreSQL 13) :

```
SELECT * FROM pg_stat_progress_analyze ;

-[ RECORD 1 ]-------------+------------------------------
pid                       | 1938258
datid                     | 748619
datname                   | grossetable
relid                     | 748698
phase                     | acquiring inherited sample rows
sample_blks_total         | 1875
sample_blks_scanned       | 1418
ext_stats_total           | 0
ext_stats_computed        | 0
child_tables_total        | 16
child_tables_done         | 6
current_child_table_relid | 748751
```

Les vues précédentes affichent aussi bien les opérations lancées manuellement que celles décidées par l'autovacuum.

Par contre, pour un `VACUUM FULL`, il faudra suivre la progression au travers de la vue `pg_stat_progress_cluster` (à partir de la version 12), qui renvoie par exemple :

```
$ psql -c 'VACUUM FULL big' &

$ psql
postgres=# \x
Affichage étendu activé.

postgres=# SELECT * FROM pg_stat_progress_cluster ;

-[ RECORD 1 ]-------+------------------
pid                 | 21157
datid               | 13444
datname             | postgres
relid               | 16384
command             | VACUUM FULL
phase               | seq scanning heap
cluster_index_relid | 0
heap_tuples_scanned | 13749388
heap_tuples_written | 13749388
heap_blks_total     | 199105
```

```
heap_blks_scanned   | 60839
index_rebuild_count | 0
```

Cette vue est utilisable aussi avec l'ordre `CLUSTER`, d'où le nom.

5.6 AUTOVACUUM

- Processus autovacuum
- But : ne plus s'occuper de `VACUUM`
- Suit l'activité
- Seuil dépassé => worker dédié
- Gère : `VACUUM`, `ANALYZE`, `FREEZE`
  - mais pas `FULL`

Le principe est le suivant :

Le démon `autovacuum launcher` s'occupe de lancer des *workers* régulièrement sur les différentes bases Ce nouveau processus inspecte les statistiques sur les tables (vue `pg_stat_all_tables`) : nombres de lignes insérées, modifiées et supprimées. Quand certains seuils sont dépassés sur un objet, le *worker* effectue un `VACUUM`, un `ANALYZE`, voire un `VACUUM FREEZE` (mais jamais, rappelons-le, un `VACUUM FULL`).

Le nombre de ces *workers* est limité, afin de ne pas engendrer de charge trop élevée.

5.6.1 PARAMÉTRAGE DU DÉCLENCHEMENT DE L'AUTOVACUUM

- `autovacuum` (on !)
- `autovacuum_naptime` (1 min)
- `autovacuum_max_workers` (3)
  - plusieurs *workers* simultanés sur une base
  - un seul par table

`autovacuum` (`on` par défaut) détermine si l'autovacuum doit être activé.

Il est fortement conseillé de laisser `autovacuum` à `on` !

S'il le faut vraiment, il est possible de désactiver l'autovacuum sur une table précise :

```
ALTER TABLE nom_table SET (autovacuum_enabled = off);
```

mais cela est très rare. La valeur `off` n'empêche pas le déclenchement d'un `VACUUM FREEZE` s'il devient nécessaire.

`autovacuum_naptime` est le temps d'attente entre deux périodes de vérification sur la même base (1 minute par défaut). Le déclenchement de l'autovacuum suite à des modifications de tables n'est donc pas instantané.

`autovacuum_max_workers` est le nombre maximum de *workers* que l'autovacuum pourra déclencher simultanément, chacun s'occupant d'une table (ou partition de table). Chaque table ne peut être traitée simultanément que par un unique *worker*. La valeur par défaut (3) est généralement suffisante. Néanmoins, s'il y a fréquemment trois *autovacuum workers* travaillant en même temps, et surtout si cela dure, il peut être nécessaire d'augmenter ce paramètre. Cela est fréquent quand il y a de nombreuses petites tables. Noter qu'il faudra peut-être être plus généreux avec les ressources allouées (paramètres `autovacuum_vacuum_cost_delay` ou `autovacuum_vacuum_cost_limit`), car les *workers* se les partagent.

5.6.2 DÉCLENCHEMENT DE L'AUTOVACUUM

Seuil de déclenchement =
threshold
+ *scale factor* × nb lignes de la table

L'autovacuum déclenche un `VACUUM` ou un `ANALYZE` à partir de seuils calculés sur le principe d'un nombre de lignes minimal (*threshold*) et d'une proportion de la table existante (*scale factor*) de lignes modifiées, insérées ou effacées. (Pour les détails précis sur ce qui suit, voir la documentation officielle[29] .)

Ces seuils pourront être adaptés table par table.

5.6.3 DÉCLENCHEMENT DE L'AUTOVACUUM (SUITE)

- Pour `VACUUM`
  - `autovacuum_vacuum_scale_factor` (20 %)
  - `autovacuum_vacuum_threshold` (50)
  - (v13) `autovacuum_vacuum_insert_threshold` (1000)
  - (v13) `autovacuum_vacuum_insert_scale_factor` (20 %)
- Pour `ANALYZE`
  - `autovacuum_analyze_scale_factor` (10 %)
  - `autovacuum_analyze_threshold` (50)
- Adapter pour une grosse table :

[29] https://docs.postgresql.fr/current/routine-vacuuming.html#AUTOVACUUM

```
ALTER TABLE table_name SET (autovacuum_vacuum_scale_factor = 0.1);
```

Pour le VACUUM, si on considère les enregistrements morts (supprimés ou anciennes versions de lignes), la condition de déclenchement est :

```
nb_enregistrements_morts (pg_stat_all_tables.n_dead_tup) >=
    autovacuum_vacuum_threshold
  + autovacuum_vacuum_scale_factor × nb_enregs (pg_class.reltuples)
```

où, par défaut :

- `autovacuum_vacuum_threshold` vaut 50 lignes ;
- `autovacuum_vacuum_scale_factor` vaut 0,2 soit 20 % de la table.

Donc, par exemple, dans une table d'un million de lignes, modifier 200 050 lignes provoquera le passage d'un VACUUM.

Pour les grosses tables avec de l'historique, modifier 20 % de la volumétrie peut être extrêmement long. Quand l'`autovacuum` lance enfin un VACUUM, celui-ci a donc beaucoup de travail et peut durer longtemps et générer beaucoup d'écritures. Il est donc fréquent de descendre la valeur de `vacuum_vacuum_scale_factor` à quelques pour cent sur les grosses tables. (Une alternative est de monter `autovacuum_vacuum_threshold` à un nombre de lignes élevé et de descendre `autovacuum_vacuum_scale_factor` à 0, mais il faut alors calculer le nombre de lignes qui déclenchera le nettoyage, et cela dépend fortement de la table et de sa fréquence de mise à jour.)

S'il faut modifier un paramètre, il est préférable de ne pas le faire au niveau global mais de cibler les tables où cela est nécessaire. Par exemple, l'ordre suivant réduit à 5 % de la table le nombre de lignes à modifier avant que l'`autovacuum` y lance un VACUUM :

```
ALTER TABLE nom_table SET (autovacuum_vacuum_scale_factor = 0.05);
```

À partir de PostgreSQL 13, le VACUUM est aussi lancé quand il n'y a que des insertions, avec deux nouveaux paramètres et un autre seuil de déclenchement :

```
nb_enregistrements_insérés (pg_stat_all_tables.n_ins_since_vacuum) >=
    autovacuum_vacuum_insert_threshold
  + autovacuum_vacuum_insert_scale_factor × nb_enregs (pg_class.reltuples)
```

Pour l'ANALYZE, le principe est le même. Il n'y a que deux paramètres, qui prennent en compte toutes les lignes modifiées *ou* insérées, pour calculer le seuil :

```
nb_insert + nb_updates + nb_delete (n_mod_since_analyze) >=
    autovacuum_analyze_threshold + nb_enregs × autovacuum_analyze_scale_factor
```

où, par défaut :

- `autovacuum_analyze_threshold` vaut 50 lignes ;

- `autovacuum_analyze_scale_factor` vaut 0,1, soit 10 %.

Dans notre exemple d'une table, modifier 100 050 lignes provoquera le passage d'un `ANALYZE`.

Là encore, il est fréquent de modifier les paramètres sur les grosses tables pour rafraîchir les statistiques plus fréquemment.

Les insertions sont prises en compte pour `ANALYZE`, puisqu'elles modifient le contenu de la table. Mais, jusque PostgreSQL 12 inclus, il semblait inutile de déclencher un `VACUUM` pour de nouvelles lignes. Cependant, cela pouvait inhiber certaines optimisations pour des tables à insertion seule. Pour cette raison, à partir de la version 13, les insertions sont aussi prises en compte pour déclencher un `VACUUM`.

5.7 PARAMÉTRAGE DE VACUUM & AUTOVACUUM

- VACUUM vs autovacuum
- Mémoire
- Gestion des coûts
- Gel des lignes

En fonction de la tâche exacte, de l'agressivité acceptable ou de l'urgence, plusieurs paramètres peuvent être mis en place.

Ces paramètres peuvent différer (par le nom ou la valeur) selon qu'ils s'appliquent à un `VACUUM` lancé manuellement ou par script, ou à un processus lancé par l'autovacuum.

5.7.1 VACUUM VS AUTOVACUUM

`VACUUM` manuel	autovacuum
Urgent	Arrière-plan
Pas de limite	Peu agressif
Paramètres	Les mêmes + paramètres de surcharge

Quand on lance un ordre `VACUUM`, il y a souvent urgence, ou l'on est dans une période de maintenance, ou dans un batch. Les paramètres que nous allons voir ne cherchent donc pas, par défaut, à économiser des ressources.

À l'inverse, un `VACUUM` lancé par l'autovacuum ne doit pas gêner une production peut-être chargée. Il existe donc des paramètres `autovacuum_*` surchargeant les précédents, et beaucoup plus conservateurs.

5.7.2 MÉMOIRE

- Quantité de mémoire allouable
  - `maintenance_work_mem` / `autovacuum_work_mem`
- Impact
  - `VACUUM`
  - construction d'index

`maintenance_work_mem` est la quantité de mémoire qu'un processus effectuant une opération de maintenance (c'est-à-dire n'exécutant pas des requêtes classiques comme `SELECT`,

INSERT, UPDATE...) est autorisé à allouer pour sa tâche de maintenance.

Cette mémoire est utilisée lors de la construction d'index ou l'ajout de clés étrangères. et, dans le contexte de VACUUM, pour stocker les adresses des enregistrements pouvant être recyclés. Cette mémoire est remplie pendant la phase 1 du processus de VACUUM, tel qu'expliqué plus haut.

Rappelons qu'une adresse d'enregistrement (tid, pour tuple id) a une taille de 6 octets et est composée du numéro dans la table, et du numéro d'enregistrement dans le bloc, par exemple (0,1), (3164,98) ou (5351510,42).

Le défaut de 64 Mo est assez faible. Si tous les enregistrements morts d'une table ne tiennent pas dans maintenance_work_mem, VACUUM est obligé de faire plusieurs passes de nettoyage, donc plusieurs parcours complets de chaque index. Une valeur assez élevée de maintenance_work_mem est donc conseillée : s'il est déjà possible de stocker plusieurs dizaines de millions d'enregistrements à effacer dans 256 Mo, 1 Go peut être utile lors de grosses purges. Attention, plusieurs VACUUM peuvent tourner simultanément.

Un maintenance_work_mem à plus de 1 Go est inutile pour le VACUUM (il ne sait pas utiliser plus), par contre il peut accélérer l'indexation de grosses tables.

autovacuum_work_mem permet de surcharger maintenance_work_mem spécifiquement pour l'autovacuum. Par défaut les deux sont identiques.

5.7.3 BRIDAGE DU VACUUM ET DE L'AUTOVACUUM

- Pauses régulières après une certaine activité
- Par bloc traité
  - `vacuum_cost_page_hit`/`_miss`/`_dirty` (1/10/20)
  - jusque total de `vacuum_cost_limit` (200)
  - pause `vacuum_cost_delay` (en manuel : 0 !)
- Surcharge pour l'autovacuum
  - `autovacuum_vacuum_cost_limit` (identique)
  - `autovacuum_vacuum_cost_delay` (20 ou 2 ms)
  - => débit en écriture max : ~ 4 ou 40 Mo/s

Les paramètres suivant permettent de provoquer une pause d'un `VACUUM` pour ne pas gêner les autres sessions en saturant le disque. Ils affectent un coût arbitraire aux trois actions suivantes :

- `vacuum_cost_page_hit` : coût d'accès à une page présente dans le cache (défaut : 1) ;
- `vacuum_cost_page_miss` : coût d'accès à une page hors du cache (défaut : 10 avant la v14, 2 à partir de la v14) ;
- `vacuum_cost_page_dirty` : coût de modification d'une page, et donc de son écriture (défaut : 20).

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

- `vacuum_cost_limit` : coût à atteindre avant de déclencher une pause (défaut : 200) ;
- `vacuum_cost_delay` : temps à attendre (défaut : 0 ms !)

En conséquence, les `VACUUM` lancés manuellement (en ligne de commande ou via `vacuumdb`) ne sont **pas** freinés par ce mécanisme et peuvent donc entraîner de fortes écritures, du moins par défaut. Mais c'est généralement dans un batch ou en urgence, et il vaut mieux alors être le plus rapide possible. Il est donc conseillé de laisser `vacuum_cost_limit` et `vacuum_cost_delay` ainsi, ou de ne les modifier que le temps d'une session ainsi :

```
SET vacuum_cost_limit = 200 ;
SET vacuum_cost_delay = '20ms' ;
VACUUM (VERBOSE) matable ;
```

(Pour les urgences, rappelons que l'option `INDEX_CLEANUP off` permet en plus d'ignorer le nettoyage des index, à partir de PostgreSQL 12.)

Les `VACUUM` d'autovacuum, eux, sont par défaut limités en débit pour ne pas gêner l'activité normale de l'instance. Deux paramètres surchargent les précédents :

- `autovacuum_cost_limit` vaut par défaut -1, donc reprend la valeur 200 de `vacuum_cost_limit` ;
- `autovacuum_vacuum_cost_delay` vaut par défaut 2 ms (mais 20 ms avant la version 12, ce qui correspond à l'exemple ci-dessus).

Un (`autovacuum_`)`vacuum_cost_limit` de 200 correspond à traiter au plus 200 blocs lus en cache (car `vacuum_cost_page_hit` = 1), soit 1,6 Mo, avant de faire une pause. Si ces blocs doivent être écrits, on descend en-dessous de 10 blocs traités avant chaque pause (`vacuum_cost_page_dirty` = 20) avant la pause de 2 ms, d'où un débit en écriture maximal de l'autovacuum de 40 Mo/s (avant la version 12 : 20 ms et seulement 4 Mo/s !), et d'au plus le double en lecture. Cela s'observe aisément par exemple avec `iotop`.

Ce débit est partagé équitablement entre les différents *workers* lancés par l'autovacuum (sauf paramétrage spécifique au niveau de la table).

Pour rendre l'`autovacuum` plus agressif, on peut augmenter la limite de coût, ou réduire le temps de pause, à condition de pouvoir assumer le débit supplémentaire pour les disques. La version 12 a justement réduit le délai pour tenir compte de l'évolution des disques et des volumétries.

5.7.4 PARAMÉTRAGE DU FREEZE

- Lors des `VACUUM`
  - `vacuum_freeze_min_age` (50 Mxact)
  - `vacuum_freeze_table_age` (150 Mxact) => vacuum agressif
- Déclenchement du gel sur toute la table
  - `autovacuum_freeze_max_age` (200 Mxact)
- Attention après des imports en masse !
  - `VACUUM FREEZE` préventif en période de maintenance
- Les blocs déjà nettoyés/gelés sont indiqués dans la *visibility map*

Afin d'éviter le *wraparound*, `VACUUM` « gèle » les vieux enregistrements, afin que ceux-ci ne se retrouvent pas brusquement dans le futur. Cela implique de réécrire le bloc. Il est inutile de geler trop tôt une ligne récente, qui sera peut-être bientôt réécrite. Plusieurs paramètres règlent ce fonctionnement.

Leurs valeurs par défaut sont satisfaisantes pour la plupart des installations et ne sont pour ainsi dire jamais modifiées. Par contre, il est important de bien connaître le fonctionnement pour ne pas être surpris.

Rappelons que le numéro de transaction (sur 32 bits) le plus ancien connu d'une table est

porté par `pgclass.relfrozenxid`. Il faut utiliser la fonction `age()` pour connaître l'écart par rapport au numéro de transaction courant (complet, sur 64 bits).

```
SELECT relname, relfrozenxid, round(age(relfrozenxid) /1e6,2) AS "age_Mtrx" FROM pg_class c
WHERE relname LIKE 'pgbench%' AND relkind='r' ORDER BY age(relfrozenxid)  ;
```

```
       relname       | relfrozenxid | age_Mtrx
---------------------+--------------+----------
 pgbench_accounts_7  |    882324041 |     0.00
 pgbench_accounts_8  |    882324041 |     0.00
 pgbench_accounts_2  |    882324041 |     0.00
 pgbench_history     |    882324040 |     0.00
 pgbench_accounts_5  |    848990708 |    33.33
 pgbench_tellers     |    832324041 |    50.00
 pgbench_accounts_3  |    719860155 |   162.46
 pgbench_accounts_9  |    719860155 |   162.46
 pgbench_accounts_4  |    719860155 |   162.46
 pgbench_accounts_6  |    719860155 |   162.46
 pgbench_accounts_1  |    719860155 |   162.46
 pgbench_branches    |    719860155 |   162.46
 pgbench_accounts_10 |    719860155 |   162.46
```

Une partie du gel se fait lors d'un `VACUUM` normal. Si ce dernier rencontre un enregistrement plus vieux que `vacuum_freeze_min_age` (par défaut 50 millions de transactions écoulées), alors le *tuple* peut et doit être gelé. Cela ne concerne que les lignes dans des blocs qui ont des lignes mortes à nettoyer : les lignes dans des blocs un peu statiques y échappent.

`VACUUM` doit donc périodiquement déclencher un nettoyage plus agressif de toute la table (et non pas uniquement des blocs modifiés depuis le dernier `VACUUM`), afin de nettoyer tous les vieux enregistrements. C'est le rôle de `vacuum_freeze_table_age` (par défaut 150 millions de transactions). Si la table a atteint cet âge, un `VACUUM` lancé dessus deviendra « agressif » :

```
VACUUM (VERBOSE) pgbench_tellers ;
INFO:  aggressively vacuuming "public.pgbench_tellers"
```

C'est équivalent à l'option `DISABLE_PAGE_SKIPPING` : les blocs ne contenant que des lignes vivantes seront tout de même parcourus. Les lignes non gelées qui s'y trouvent et plus vieilles que `vacuum_freeze_min_age` seront alors gelées. Ce peut être long, ou pas, en fonction de l'efficacité de l'étape précédente.

À côté des numéros de transaction habituels, les identifiants `multixact`, utilisés pour supporter le verrouillage de lignes par des transactions multiples évitent aussi le *wraparound* avec des paramètres spécifiques (`vacuum_multixact_freeze_min_age`, `vacuum_multixact_freeze_table_age`) qui ont les mêmes valeurs que leurs homologues.

Enfin, il faut traiter le cas de tables sur lesquelles un VACUUM complet ne s'est pas déclenché depuis très longtemps. L'autovacuum y veille : autovacuum_freeze_max_age (par défaut 200 millions de transactions) est l'âge maximum que doit avoir une table. S'il est dépassé, un VACUUM FREEZE est lancé sur cette table Il est visible dans pg_stat_activity notamment, avec la mention *to prevent wraparound* :

```
autovacuum: VACUUM public.pgbench_accounts (to prevent wraparound)
```

Ce traitement est lancé même si autovacuum est désactivé (c'est-à-dire à off).

Ce peut être très lourd s'il y a beaucoup de lignes à geler, ou très rapide si l'essentiel du travail a été fait par les nettoyages précédents. Si la table a déjà été entièrement gelée (parfois depuis des centaines de millions de transactions), il peut juste s'agir d'une mise à jour du relfrozenxid. (Avant PostgreSQL 9.6, il y avait forcément au moins un parcours complet de la table. Depuis, les blocs déjà entièrement gelés sont ignorés par le FREEZE.)

L'âge de la table peut dépasser autovacuum_freeze_max_age si le nettoyage est laborieux, ce qui explique la marge par rapport à la limite fatidique des 2 milliards de transactions.

Concrètement, on verra l'âge d'une base de données approcher peu à peu des 200 millions de transactions, ce qui correspondra à l'âge des plus vieilles tables, même si l'essentiel de leur contenu est déjà gelé, puis retomber quand un VACUUM FREEZE sera forcé sur elles, remonter, etc.

```
SELECT age(datfrozenxid) FROM pg_database WHERE datname = current_database();
SELECT relname, age (relfrozenxid) FROM pg_class WHERE relkind='r' ORDER BY  2 DESC LIMIT 1 ;

   age
---------
 2487153

 relname |   age
---------+---------
 dossier | 2487154
```

Rappelons que le FREEZE génère de fait la réécriture de tous les blocs concernés. Le déclenchement inopiné d'un VACUUM FREEZE sur l'intégralité d'une grosse table assez statique est une mauvaise surprise assez fréquente.
Une base chargée avec pg_restore et peu modifiée peut même voir le FREEZE se déclencher sur toutes les tables en même temps. Cela est moins grave depuis les optimisations de la 9.6, mais, après de très gros imports, il reste utile d'opérer un VACUUM FREEZE manuel, à un moment où cela gêne peu, pour éviter qu'ils ne se provoquent plus tard en période chargée.

5.8 AUTRES PROBLÈMES COURANTS

5.8.1 ARRÊTER UN VACUUM ?

- Lancement manuel ou script
  - risque avec certains verrous
- Autovacuum
  - interrompre s'il gêne
- Exception : *to prevent wraparound* lent **et** bloquant
  - `pg_cancel_backend` + `VACUUM FREEZE` manuel

Le cas des `VACUUM` manuels a été vu plus haut : ils peuvent gêner quelques verrous ou opérations DDL. Il faudra les arrêter manuellement au besoin.

C'est différent si l'autovacuum a lancé le processus : celui-ci sera arrêté si un utilisateur pose un verrou en conflit.

La seule exception concerne un `VACUUM FREEZE` lancé quand la table doit être gelée, donc avec la mention *to prevent wraparound* dans `pg_stat_activity` : celui-ci ne sera pas interrompu. Il ne pose qu'un verrou destinée à éviter les modifications de schéma simultanées (SHARE UPDATE EXCLUSIVE). Comme le débit en lecture et écriture est bridé par le paramétrage habituel de l'autovacuum, ce verrou peut durer assez longtemps (surtout avant PostgreSQL 9.6, où toute la table est relue à chaque `FREEZE`). Cela peut s'avérer gênant avec certaines applications. Une solution est de réduire `autovacuum_vacuum_cost_delay`, surtout avant PostgreSQL 12 (voir plus haut).

Si les opérations sont impactées, on peut vouloir lancer soi-même un `VACUUM FREEZE` manuel, non bridé. Il faudra alors repérer le PID du `VACUUM FREEZE` en cours, l'arrêter avec `pg_cancel_backend`, puis lancer manuellement l'ordre `VACUUM FREEZE` sur la table concernée, (et rapidement avant que l'autovacuum ne relance un processus).

La supervision peut se faire avec `pg_stat_progress_vacuum` et `iotop`.

5.8.2 CE QUI PEUT BLOQUER LE VACUUM FREEZE

- Causes :
  - sessions *idle in transactions* sur une longue durée
  - slot de réplication en retard/oublié
  - transactions préparées oubliées
  - erreur à l'exécution du `VACUUM`
- Conséquences :
  - processus autovacuum répétés

  - arrêt des transactions
  - mode single…
- Supervision :
  - `check_pg_activity` : `xmin`, `max_freeze_age`

Il arrive que le fonctionnement du `FREEZE` soit gêné par un problème qui lui interdit de recycler les plus anciens numéros de transactions. Les causes possibles sont :

- des sessions *idle in transactions* durent depuis des jours ou des semaines (voir le statut `idle in transaction` dans `pg_stat_activity`, et au besoin fermer la session) : au pire, elles disparaissent après redémarrage ;
- des slots de réplication pointent vers un secondaire très en retard, voire disparu (consulter `pg_replication_slots`, et supprimer le slot) ;
- des transactions préparées (pas des requêtes préparées !) n'ont jamais été validées ni annulées, (voir `pg_prepared_xacts`, et annuler la transaction) : elles ne disparaissent pas après redémarrage ;
- l'opération de `VACUUM` tombe en erreur : corruption de table ou index, fonction d'index fonctionnel buggée, etc. (voir les traces et corriger le problème, supprimer l'objet ou la fonction, etc.).

Pour effectuer le `FREEZE` en urgence le plus rapidement possible, on peut utiliser, à partir de PostgreSQL 12 :

```
VACUUM (FREEZE, VERBOSE, INDEX_CLEANUP off, TRUNCATE off) ;
```

Ne pas oublier de nettoyer toutes les bases de l'instance.

Dans le pire des cas, plus aucune transaction ne devient possible (y compris les opérations d'administration comme `DROP`, ou `VACUUM` sans `TRUNCATE off`) :

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database "db1"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
You might also need to commit or roll back old prepared transactions,
or drop stale replication slots.
```

En dernière extrémité, il reste un délai de grâce d'un million de transactions, qui ne sont accessibles que dans le très austère mode monoutilisateur[30] de PostgreSQL.

Avec la sonde Nagios check_pgactivity[31] , et les services **max_freeze_age** et **oldest_xmin**, il est possible de vérifier que l'âge des bases ne dérive pas, ou de trouver quel processus porte le `xmin` le plus ancien.

[30] https://docs.postgresql.fr/current/app-postgres.html#APP-POSTGRES-SINGLE-USER
[31] https://github.com/OPMDG/check_pgactivity

5.9 RÉSUMÉ DES CONSEILS SUR L'AUTOVACUUM (1/2)

- Laisser l'autovacuum faire son travail
- Augmenter le débit autorisé
- Surveiller `last_(auto)analyze` / `last_(auto)vacuum`
- Nombre de *workers*
- Grosses tables, par ex :

```
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.01);
ALTER TABLE table_name SET (autovacuum_vacuum_threshold = 1000000);
```

L'autovacuum fonctionne convenablement pour les charges habituelles. Il ne faut pas s'étonner qu'il fonctionne longtemps en arrière-plan : il est justement conçu pour ne pas se presser. Au besoin, ne pas hésiter à lancer manuellement l'opération, donc sans bridage en débit.

Si les disques sont bons, on peut augmenter le débit autorisé en jouant sur `autovacuum_vacuum_cost_delay`/`_cost_limit`, surtout avant la version 13.

Le déclenchement est très lié à l'activité, il faut donc vérifier que l'autovacuum passe assez souvent sur les tables sensibles en surveillant `pg_stat_all_tables.last_autovacuum` et `_autoanalyze`. Si les statistiques traînent à se rafraîchir, ne pas hésiter à activer l'autovacuum sur les grosses tables problématiques :

```
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.05);
```

De même, si la fragmentation s'envole, descendre `autovacuum_analyze_scale_factor`. (On peut préférer utiliser les variantes en `_threshold` de ces paramètres, et mettre les `_scale_factor` à 0).

Dans un modèle avec de très nombreuses tables actives, le nombre de *workers* doit parfois être augmenté.

5.10 RÉSUMÉ DES CONSEILS SUR L'AUTOVACUUM (2/2)

- Mode manuel
  - batchs / tables temporaires / tables à insertions seules (<v13)
  - si pressé !
- Danger du `FREEZE` brutal
  - prévenir
- `VACUUM FULL` : dernière extrémité

L'autovacuum n'est pas toujours assez rapide à se déclencher, par exemple entre les différentes étapes d'un batch : on intercalera des `VACUUM ANALYZE` manuels. Il faudra le faire systématiquement pour les tables temporaires (que l'autovacuum ne voit pas.) Pour les tables où il n'y a que des insertions, avant PostgreSQL 13, l'autovacuum ne lance spontanément que l'`ANALYZE`, il faudra effectuer un `VACUUM` explicite pour profiter de certaines optimisations.

Un point d'attention reste le gel brutal de grosses quantités de données chargées ou modifiées en même temps. Un `VACUUM FREEZE` préventif dans une période calme reste la meilleure solution.

Un `VACUUM FULL` sur une grande table est une opération très lourde, pour récupérer une partie significative de son espace, qui ne serait pas réutilisée plus tard.

5.11 CONCLUSION

- `VACUUM` fait de plus en plus de choses au fil des versions
- Convient généralement
- Paramétrage apparemment complexe
  - en fait relativement simple avec un peu d'habitude

5.11.1 QUESTIONS

N'hésitez pas, c'est le moment !

5.12 QUIZ

https://dali.bo/m5_quiz

5.13 TRAVAUX PRATIQUES

5.13.1 TRAITER LA FRAGMENTATION

Créer une table `t3` avec une colonne `id` de type integer.

Désactiver l'autovacuum pour la table `t3`.

Insérer un million de lignes dans la table `t3` avec `generate_series`.

Récupérer la taille de la table `t3`.

Supprimer les 500 000 premières lignes de la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Créer une table `t4` avec une colonne `id` de type integer.

Désactiver l'autovacuum pour la table `t4`.

Insérer un million de lignes dans la table `t4` avec `generate_series`.

Récupérer la taille de la table `t4`.

Supprimer les 500 000 **dernières** lignes de la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

Exécuter un `VACUUM` sur la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

5.13.2 DÉTECTER LA FRAGMENTATION

Créer une table `t5` avec deux colonnes `c1` de type integer et `c2` de type text.

Désactiver l'autovacuum pour la table `t5`.

Insérer un million de lignes dans la table `t5` avec `generate_series`.

Installer l'extension `pg_freespacemap` (documentation : https://docs.postgresql.fr/current/pgfreespacemap.html) Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?

Modifier exactement 200 000 lignes de la table `t5`. Que rapporte `pg_freespacemap` quant à l'espace libre de la table `t5` ?

Exécuter un `VACUUM` sur la table `t5`. Que rapporte `pg_freespacemap` quant à l'espace libre de la table `t5` ? Que faut-il en déduire ?

Récupérer la taille de la table `t5`.

Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

5.13.3 GESTION DE L'AUTOVACUUM

Créer une table `t6` avec une colonne `id` de type integer.

Insérer un million de lignes dans la table `t6` avec
`INSERT INTO t6(id) SELECT generate_series(1, 1000000) ;`

Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
Il faudra peut-être attendre une minute.
Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.

Vérifier le nombre de lignes dans `pg_class.reltuples`.

Modifier exactement 150 000 lignes de la table `t6` avec
`UPDATE t6 SET id = 0 WHERE id <= 150000 ;` Attendre une minute.
Que contient la vue `pg_stat_user_tables` pour la table `t6` ?

Modifier 60 000 lignes supplémentaires de la table `t6` avec :
`UPDATE t6 SET id=1 WHERE id > 940000 ;`
Attendre une minute.

> Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Que faut-il en déduire ?

> Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

> Modifier encore 200 000 autres lignes de la table `t6` avec `UPDATE t6 SET id=1 WHERE id > 740000 ;`.
> Attendre une minute.
> Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Que faut-il en déduire ?

5.14 TRAVAUX PRATIQUES (SOLUTIONS)

5.14.1 TRAITER LA FRAGMENTATION

> Créer une table `t3` avec une colonne `id` de type integer.

```
CREATE TABLE t3(id integer);
```

```
CREATE TABLE
```

> Désactiver l'autovacuum pour la table `t3`.

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

```
ALTER TABLE
```

La désactivation de l'autovacuum ici a un but uniquement pédagogique. En production, c'est une très mauvaise idée !

> Insérer un million de lignes dans la table `t3` avec `generate_series`.

```
INSERT INTO t3 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

> Récupérer la taille de la table `t3`.

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 35 MB
```

> Supprimer les 500 000 premières lignes de la table `t3`.

```
DELETE FROM t3 WHERE id <= 500000;
```

```
DELETE 500000
```

> Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 35 MB
```

`DELETE` seul ne permet pas de regagner de la place sur le disque. Les lignes supprimées sont uniquement marquées comme étant mortes. Comme l'autovacuum est ici désactivé, PostgreSQL n'a pas encore nettoyé ces lignes.

> Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

```
VACUUM VERBOSE t3;
```

```
INFO:  vacuuming "public.t3"
INFO:  "t3": removed 500000 row versions in 2213 pages
INFO:  "t3": found 500000 removable, 500000 nonremovable row versions
            in 4425 out of 4425 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 3815272
There were 0 unused item pointers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.09 s, system: 0.00 s, elapsed: 0.10 s.
VACUUM
```

L'indication :

```
removed 500000 row versions in 2213 pages
```

indique 500 000 lignes ont été nettoyées dans 2213 blocs (en gros, la moitié des blocs de la table).

Pour compléter, l'indication suivante :

```
found 500000 removable, 500000 nonremovable row versions in 4425 out of 4425 pages
```

reprend l'indication sur 500 000 lignes mortes, et précise que 500 000 autres ne le sont pas. Les 4425 pages parcourues correspondent bien à la totalité des 35 Mo de la table complète. C'est la première fois que `VACUUM` passe sur cette table, il est normal qu'elle soit intégralement parcourue.

> Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 35 MB
```

`VACUUM` ne permet pas non plus de gagner en espace disque. Principalement, il renseigne la structure FSM (*free space map*) sur les emplacements libres dans les fichiers des tables.

> Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

```
VACUUM FULL t3;
```

```
INFO:  vacuuming "public.t3"
INFO:  "t3": found 0 removable, 500000 nonremovable row versions in 4425 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.10 s, system: 0.01 s, elapsed: 0.21 s.
VACUUM
```

> Récupérer la taille de la table `t3`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t3'));
```

```
 pg_size_pretty
----------------
 17 MB
```

Là, par contre, nous gagnons en espace disque. Le `VACUUM FULL` reconstruit la table et la fragmentation disparaît.

> Créer une table `t4` avec une colonne `id` de type integer.

```
CREATE TABLE t4(id integer);
```

```
CREATE TABLE
```

> Désactiver l'autovacuum pour la table `t4`.

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

```
ALTER TABLE
```

> Insérer un million de lignes dans la table `t4` avec `generate_series`.

```
INSERT INTO t4(id) SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

> Récupérer la taille de la table `t4`.

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
 pg_size_pretty
----------------
 35 MB
```

> Supprimer les 500 000 **dernières** lignes de la table `t4`.

```
DELETE FROM t4 WHERE id > 500000;
```

```
DELETE 500000
```

> Récupérer la taille de la table `t4`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
 pg_size_pretty
----------------
 35 MB
```

Là aussi, nous n'avons rien perdu.

> Exécuter un `VACUUM` sur la table `t4`.

```
VACUUM t4;
```

```
VACUUM
```

> Récupérer la taille de la table `t4`. Que faut-il en déduire ?

```
SELECT pg_size_pretty(pg_table_size('t4'));
```

```
 pg_size_pretty
----------------
 17 MB
```

En fait, il existe un cas où il est possible de gagner de l'espace disque suite à un `VACUUM` simple : quand l'espace récupéré se trouve en fin de table et qu'il est possible de prendre rapidement un verrou exclusif sur la table pour la tronquer. C'est assez peu fréquent mais c'est une optimisation intéressante.

5.14.2 DÉTECTER LA FRAGMENTATION

> Créer une table `t5` avec deux colonnes `c1` de type integer et `c2` de type text.

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

```
CREATE TABLE
```

> Désactiver l'autovacuum pour la table `t5`.

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

```
ALTER TABLE
```

> Insérer un million de lignes dans la table `t5` avec `generate_series`.

```
INSERT INTO t5(c1, c2) SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) AS i;
```

```
INSERT 0 1000000
```

> Installer l'extension `pg_freespacemap` (documentation : https://docs.postgresql.fr/current/pgfreespacemap.html) Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?

```
CREATE EXTENSION pg_freespacemap;
```

```
CREATE EXTENSION
```

Cette extension installe une fonction nommée `pg_freespace`, dont la version la plus simple ne demande que la table en argument, et renvoie l'espace libre dans chaque bloc, en octets, *connu de la Free Space Map*.

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
 count | sum
-------+-----
  6274 |   0
```

et donc 6274 blocs (soit 51,4 Mo) sans aucun espace vide.

> Modifier exactement 200 000 lignes de la table `t5`. Que rapporte `pg_freespacemap` quant à l'espace libre de la table `t5` ?

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

```
UPDATE 200000
```

```
SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);
```

```
 count | sum
-------+-----
  7451 |  32
```

La table comporte donc 20 % de blocs en plus, où sont stockées les nouvelles versions des lignes modifiées. Le champ *avail* indique qu'il n'y a quasiment pas de place libre. (Ne

pas prendre la valeur de 32 octets au pied de la lettre, la *Free Space Map* ne cherche pas à fournir une valeur précise.)

> Exécuter un `VACUUM` sur la table `t5`. Que rapporte `pg_freespacemap` quant à l'espace libre de la table `t5` ? Que faut-il en déduire ?

```
VACUUM VERBOSE t5;

INFO:  vacuuming "public.t5"
INFO:  "t5": removed 200000 row versions in 1178 pages
INFO:  "t5": found 200000 removable, 1000000 nonremovable row versions
       in 7451 out of 7451 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 8685974
         There were 0 unused item identifiers.
         Skipped 0 pages due to buffer pins, 0 frozen pages.
         0 pages are entirely empty.
         CPU: user: 0.11 s, system: 0.03 s, elapsed: 0.33 s.
INFO:  vacuuming "pg_toast.pg_toast_4160544"
INFO:  index "pg_toast_4160544_index" now contains 0 row versions in 1 pages
DÉTAIL : 0 index row versions were removed.
         0 index pages have been deleted, 0 are currently reusable.
         CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "pg_toast_4160544": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 8685974
        There were 0 unused item identifiers.
        Skipped 0 pages due to buffer pins, 0 frozen pages.
        0 pages are entirely empty.
        CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM

SELECT count(blkno), sum(avail) FROM pg_freespace('t5'::regclass);

 count |   sum
-------+---------
  7451 | 8806816
```

Il y a toujours autant de blocs, mais environ 8,8 Mo sont à présent repérés comme libres.

Il faut donc bien exécuter un `VACUUM` pour que PostgreSQL nettoie les blocs et mette à jour la structure FSM, ce qui nous permet de déduire le taux de fragmentation de la table.

> Récupérer la taille de la table `t5`.

```
SELECT pg_size_pretty(pg_table_size('t5'));

 pg_size_pretty
----------------
```

```
 58 MB
```

> Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

```
VACUUM (FULL, VERBOSE) t5;

INFO:  vacuuming "public.t5"
INFO:  "t5": found 200000 removable, 1000000 nonremovable row versions in 7451 pages
DÉTAIL : 0 dead row versions cannot be removed yet.
CPU: user: 0.49 s, system: 0.19 s, elapsed: 1.46 s.
VACUUM
```

> Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

```
SELECT count(blkno),sum(avail)FROM pg_freespace('t5'::regclass);

 count | sum
-------+-----
  6274 |   0

SELECT pg_size_pretty(pg_table_size('t5'));

 pg_size_pretty
----------------
 49 MB
```

`VACUUM FULL` a réécrit la table sans les espaces morts, ce qui nous a fait gagner entre 8 et 9 Mo. La taille de la table maintenant correspond bien à celle de l'ancienne table, moins la place prise par les lignes mortes.

5.14.3 GESTION DE L'AUTOVACUUM

> Créer une table `t6` avec une colonne `id` de type integer.

```
CREATE TABLE t6 (id integer) ;

CREATE TABLE
```

> Insérer un million de lignes dans la table `t6` avec
> `INSERT INTO t6(id) SELECT generate_series(1, 1000000) ;`

```
INSERT INTO t6(id) SELECT generate_series(1, 1000000) ;

INSERT 0 1000000
```

> Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
> Il faudra peut-être attendre une minute.
> Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.

```
\x

Expanded display is on.

SELECT * FROM pg_stat_user_tables WHERE relname = 't6' ;

-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 0
seq_tup_read        | 0
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 0
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:42:43.612269+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:42:43.719195+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 1
```

Les deux dates `last_autovacuum` et `last_autoanalyze` sont renseignées. Il faudra peut-être attendre une minute que l'autovacuum passe sur la table (voire plus sur une instance chargée par ailleurs).

Le seuil de déclenchement de l'autoanalyze est :
`autovacuum_analyze_scale_factor` × nombre de lignes
+ `autovacuum_analyze_threshold`
soit par défaut 10 % × 0 + 50 = 50.

Quand il n'y a que des insertions, le seuil pour l'autovacuum est :

`autovacuum_vacuum_insert_scale_factor` × nombre de lignes
+ `autovacuum_vacuum_insert_threshold`
soit 20 % × 0 + 1000 = 1000.

Avec un million de nouvelles lignes, les deux seuils sont franchis.

Avec PostgreSQL 12 ou antérieur, seule la ligne `last_autoanalyze` sera remplie. S'il n'y a que des insertions, le démon autovacuum ne lance un `VACUUM` spontanément qu'à partir de PostgreSQL 13.

Jusqu'en PostgreSQL 12, il faut donc lancer manuellement :

```
ANALYZE t6 ;
```

> Vérifier le nombre de lignes dans `pg_class.reltuples`.

Vérifions que le nombre de lignes est à jour dans `pg_class` :

```
SELECT * FROM pg_class WHERE relname = 't6' ;
```

```
-[ RECORD 1 ]-------+--------
oid                 | 4160608
relname             | t6
relnamespace        | 2200
reltype             | 4160610
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 4160608
reltablespace       | 0
relpages            | 4425
reltuples           | 1e+06
...
```

L'autovacuum se base entre autres sur cette valeur pour décider s'il doit passer ou pas. Si elle n'est pas encore à jour, il faut lancer manuellement :

```
ANALYZE t6 ;
```

ce qui est d'ailleurs généralement conseillé après un gros chargement.

> Modifier exactement 150 000 lignes de la table `t6` avec `UPDATE t6 SET id = 0 WHERE id <= 150000 ;` Attendre une minute.
> Que contient la vue `pg_stat_user_tables` pour la table `t6` ?

```
UPDATE t6 SET id = 0 WHERE id <= 150000 ;

UPDATE 150000
```

Le démon *autovacuum* ne se déclenche pas instantanément après les écritures, attendons un peu :

```
SELECT pg_sleep(60) ;

SELECT * FROM pg_stat_user_tables WHERE relname = 't6' ;

-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 1
seq_tup_read        | 1000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 150000
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
n_dead_tup          | 150000
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:42:43.612269+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:43:43.561288+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 2
```

Seul `last_autoanalyze` a été modifié, et il reste entre 150 000 lignes morts (`n_dead_tup`). En effet, le démon autovacuum traite séparément l'`ANALYZE` (statistiques sur les valeurs des données) et le `VACUUM` (recherche des espaces morts).

Si l'on recalcule les seuils de déclenchement, on trouve pour l'autoanalyze :
`autovacuum_analyze_scale_factor` × nombre de lignes
+ `autovacuum_analyze_threshold`
soit par défaut 10 % × 1 000 000 + 50 = 100 050, dépassé ici.

Pour l'autovacuum, le seuil est de :
`autovacuum_vacuum_insert_scale_factor` × nombre de lignes

+ `autovacuum_vacuum_insert_threshold`

soit 20 % × 1 000 000 + 50 = 200 050, qui n'est pas atteint.

> Modifier 60 000 lignes supplémentaires de la table `t6` avec :
> `UPDATE t6 SET id=1 WHERE id > 940000 ;`
> Attendre une minute.
> Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Que faut-il en déduire ?

```
UPDATE t6 SET id=1 WHERE id > 940000 ;

UPDATE 60000
```

L'autovacuum ne passe pas tout de suite, les 210 000 lignes mortes au total sont bien visibles :

```
SELECT * FROM pg_stat_user_tables WHERE relname = 't6';

-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 3
seq_tup_read        | 3000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 210000
n_tup_del           | 0
n_tup_hot_upd       | 65
n_live_tup          | 1000000
n_dead_tup          | 210000
n_mod_since_analyze | 60000
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:42:43.612269+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:43:43.561288+01
vacuum_count        | 0
autovacuum_count    | 1
analyze_count       | 0
autoanalyze_count   | 2
```

Mais comme le seuil de 200 050 lignes modifiées à été franchi, le démon lance un VACUUM :

```
-[ RECORD 1 ]-------+------------------------------
relid               | 4160608
```

```
schemaname          | public
relname             | t6
seq_scan            | 3
seq_tup_read        | 3000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 210000
n_tup_del           | 0
n_tup_hot_upd       | 65
n_live_tup          | 896905
n_dead_tup          | 0
n_mod_since_analyze | 60000
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:47:43.740962+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:43:43.561288+01
vacuum_count        | 0
autovacuum_count    | 2
analyze_count       | 0
autoanalyze_count   | 2
```

Noter que `n_dead_tup` est revenu à 0. `last_auto_analyze` indique qu'un nouvel `ANALYZE` n'a pas été exécuté : seules 60 000 lignes ont été modifiées (voir `n_mod_since_analyze`), en-dessous du seuil de 100 050.

> Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

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

```
ALTER TABLE
```

> Modifier encore 200 000 autres lignes de la table `t6` avec `UPDATE t6 SET id=1 WHERE id > 740000 ;`.
> Attendre une minute.
> Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Que faut-il en déduire ?

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

```
UPDATE 200000
```

```
SELECT pg_sleep(60);
```

```
SELECT * FROM pg_stat_user_tables WHERE relname='t6' ;
```

```
-[ RECORD 1 ]-------+-----------------------------
relid               | 4160608
schemaname          | public
relname             | t6
seq_scan            | 4
seq_tup_read        | 4000000
idx_scan            | ¤
idx_tup_fetch       | ¤
n_tup_ins           | 1000000
n_tup_upd           | 410000
n_tup_del           | 0
n_tup_hot_upd       | 65
n_live_tup          | 1000000
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ¤
last_autovacuum     | 2021-02-22 17:53:43.563671+01
last_analyze        | ¤
last_autoanalyze    | 2021-02-22 17:53:43.681023+01
vacuum_count        | 0
autovacuum_count    | 3
analyze_count       | 0
autoanalyze_count   | 3
```

Le démon a relancé un VACUUM et un ANALYZE. Avec un facteur d'échelle à 10 %, il ne faut plus attendre que la modification de 100 050 lignes pour que le VACUUM soit déclenché par le démon. C'était déjà le seuil pour l'ANALYZE.

6 PARTITIONNEMENT DÉCLARATIF (INTRODUCTION)

- Le partitionnement déclaratif apparaît avec PostgreSQL 10
- Préférer PostgreSQL 13 ou plus récent
- Ne plus utiliser l'ancien partitionnement par héritage.

Ce module introduit le partitionnement déclaratif introduit avec PostgreSQL 10, et amélioré dans les versions suivantes. PostgreSQL 13 au minimum est conseillé pour ne pas être gêné par une des limites levées dans les versions précédentes (et non développées ici).

Le partitionnement par héritage, au fonctionnement totalement différent, reste utilisable, mais ne doit plus servir aux nouveaux développements, du moins pour les cas décrits ici.

6.1 PRINCIPE & INTÉRÊTS DU PARTITIONNEMENT

- Faciliter la maintenance de gros volumes
  - `VACUUM` (`FULL`), réindexation, déplacements, sauvegarde logique…
- Performances
  - parcours complet sur de plus petites tables
  - statistiques par partition plus précises
  - purge par partitions entières
  - `pg_dump` parallélisable
  - tablespaces différents (données froides/chaudes)
- Attention à la maintenance sur le code

Maintenir de très grosses tables peut devenir fastidieux, voire impossible : `VACUUM FULL` trop long, espace disque insuffisant, autovacuum pas assez réactif, réindexation interminable… Il est aussi aberrant de conserver beaucoup de données d'archives dans des tables lourdement sollicitées pour les données récentes.

Le partitionnement consiste à séparer une même table en plusieurs sous-tables (partitions) manipulables en tant que tables à part entière.

Maintenance

La maintenance s'effectue sur les partitions plutôt que sur l'ensemble complet des données. En particulier, un `VACUUM FULL` ou une réindexation peuvent s'effectuer partition par partition, ce qui permet de limiter les interruptions en production, et lisser la charge. `pg_dump` ne sait pas paralléliser la sauvegarde d'une table volumineuse et non partitionnée, mais parallélise celle de différentes partitions d'une même table.

C'est aussi un moyen de déplacer une partie des données dans un autre *tablespace* pour des raisons de place, ou pour déporter les parties les moins utilisées de la table vers des disques plus lents et moins chers.

Parcours complet de partitions

Certaines requêtes (notamment décisionnelles) ramènent tant de lignes, ou ont des critères si complexes, qu'un parcours complet de la table est souvent privilégié par l'optimiseur.

Un partitionnement, souvent par date, permet de ne parcourir qu'une ou quelques partitions au lieu de l'ensemble des données. C'est le rôle de l'optimiseur de choisir la partition (*partition pruning*), par exemple celle de l'année en cours, ou des mois sélectionnés.

Suppression des partitions

La suppression de données parmi un gros volume peut poser des problèmes d'accès concurrents ou de performance, par exemple dans le cas de purges.

En configurant judicieusement les partitions, on peut résoudre cette problématique en supprimant une partition (`DROP TABLE nompartition ;`), ou en la *détachant* (`ALTER TABLE table_partitionnee DETACH PARTITION nompartition ;`) pour l'archiver (et la réattacher au besoin) ou la supprimer ultérieurement.

D'autres optimisations sont décrites dans ce billet de blog d'Adrien Nayrat[32] : statistiques plus précises au niveau d'une partition, réduction plus simple de la fragmentation des index, jointure par rapprochement des partitions…

La principale difficulté d'un système de partitionnement consiste à partitionner avec un impact minimal sur la maintenance du code par rapport à une table classique.

---

[32] https://blog.anayrat.info/2021/09/01/cas-dusages-du-partitionnement-natif-dans-postgresql/

6.2 PARTITIONNEMENT DÉCLARATIF

- Table partitionnée
  - structure uniquement
  - index/contraintes répercutés sur les partitions
- Partitions :
  - 1 partition = 1 table classique, utilisable directement
  - clé de partitionnement (inclue dans PK/UK)
  - partition par défaut
  - sous-partitions possibles
  - FDW comme partitions possible
  - attacher/détacher une partition

En partitionnement déclaratif, une table partitionnée ne contient pas de données par elle-même. Elle définit la structure (champs, types) et les contraintes et index, qui sont répercutées sur ses partitions.

Une partition est une table à part entière, rattachée à une table partitionnée. Sa structure suit automatiquement celle de la table partitionnée et ses modifications. Cependant, des index ou contraintes supplémentaires propres à cette partition peuvent être ajoutées de la même manière que pour une table classique.

La partition se définit par une « clé de partitionnement », sur une ou plusieurs colonnes. Les lignes de même clé se retrouvent dans la même partition. La clé peut se définir comme :

- une liste de valeurs ;
- une plage de valeurs ;
- une valeur de hachage.

Les clés des différentes partitions ne doivent pas se recouvrir.

Une partition peut elle-même être partitionnée, sur une autre clé, ou la même.

Une table classique peut être attachée à une table partitionnée. Une partition peut être détachée et redevenir une table indépendante normale.

Même une table distante (utilisant *foreign data wrapper*) peut être définie comme partition, avec des restrictions. Pour les performances, préférer alors PostgreSQL 14 au moins.

Les clés étrangères entre tables partitionnées ne sont pas un problème dans les versions récentes de PostgreSQL.

Le routage des données insérées ou modifiées vers la bonne partition est géré de façon

automatique en fonction de la définition des partitions. La création d'une partition par défaut permet d'éviter des erreurs si aucune partition ne convient.

De même, à la lecture de la table partitionnée, les différentes partitions nécessaires sont accédées de manière transparente.

Pour le développeur, la table principale peut donc être utilisée comme une table classique. Il vaut mieux cependant qu'il connaisse le mode de partitionnement, pour utiliser la clé autant que possible. La complexité supplémentaire améliorera les performances. L'accès direct aux partitions par leur nom de table reste possible, et peut parfois améliorer les performances. Un développeur pourra aussi purger des données plus rapidement, en effectuant un simple `DROP` de la partition concernée.

6.2.1 PARTITIONNEMENT PAR LISTE

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1) ;

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3) ;
CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (4, 5) ;
…
```

Le partitionnement par liste définit les valeurs d'une colonne acceptables dans chaque partition.

Utilisations courantes : partitionnement par année, par statut, par code géographique…

6.2.2 PARTITIONNEMENT PAR INTERVALLE

```
CREATE TABLE logs ( d timestamptz, contenu text) PARTITION BY RANGE (d) ;

CREATE TABLE logs_201901 PARTITION OF logs
                        FOR VALUES FROM ('2019-01-01') TO ('2019-02-01');
CREATE TABLE logs_201902 PARTITION OF logs
                        FOR VALUES FROM ('2019-02-01') TO ('2019-03-01');
…
CREATE TABLE logs_201912 PARTITION OF logs
                        FOR VALUES FROM ('2019-12-01') TO ('2020-01-01');
…
CREATE TABLE logs_autres PARTITION OF logs
                        DEFAULT ;               -- pour ne rien perdre
```

Utilisations courantes : partitionnement par date, par plages de valeurs continues, alphabétiques…

L'exemple ci-dessus utilise le partitionnement par mois. Chaque partition est définie par des plages de date. Noter que la borne supérieure ne fait *pas* partie des données de la partition. Elle doit donc être aussi la borne inférieure de la partie suivante.

La description de la table partitionnée devient :

```
=# \d+ logs
                                                  Table partitionnée « public.logs »
 Colonne |           Type           | Collationnement | NULL-able | Par défaut | Stockage | …
---------+--------------------------+-----------------+-----------+------------+----------+ …
 d       | timestamp with time zone |                 |           |            | plain    | …
 contenu | text                     |                 |           |            | extended | …
Clé de partition : RANGE (d)
Partitions: logs_201901 FOR VALUES FROM ('2019-01-01 00:00:00+01') TO ('2019-02-01 00:00:00+01'),
            logs_201902 FOR VALUES FROM ('2019-02-01 00:00:00+01') TO ('2019-03-01 00:00:00+01'),
            …
            logs_201912 FOR VALUES FROM ('2019-12-01 00:00:00+01') TO ('2020-01-01 00:00:00+01'),
            logs_autres DEFAULT
```

La partition par défaut reçoit toutes les données qui ne vont dans aucune autre partition : cela évite des erreurs d'insertion. Il vaut mieux que la partition par défaut reste très petite.

Il est possible de définir des plages sur plusieurs champs :

```
CREATE TABLE tt_a PARTITION OF tt
FOR VALUES FROM (1,'2020-08-10') TO (100, '2020-08-11') ;
```

6.2.3 PARTITIONNEMENT PAR HACHAGE

- Hachage des valeurs
- Répartition homogène
- Indiquer un modulo et un reste

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);

CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 0) ;
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 1) ;
CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 2) ;
```

Ce type de partitionnement vise à répartir la volumétrie dans plusieurs partitions de manière homogène, quand il n'y a pas de clé évidente. En général, il y aura plus que 3 partitions.

6.3 PERFORMANCES & PARTITIONNEMENT

- Insertions via la table principale
  - quasi aucun impact
- Lecture depuis la table principale
  - attention à la clé
- Purge
  - simple `DROP` ou `DETACH`
- Trop de partitions
  - attention au temps de planification

Des `INSERT` dans la table partitionnée seront redirigés directement dans les bonnes partitions avec un impact en performances quasi négligeable.

Lors des lectures ou jointures, il est important de préciser autant que possible la clé de jointure, si elle est pertinente. Dans le cas contraire, toutes les tables de la partition seront interrogées.

Dans cet exemple, la table comprend 10 partitions :

```
=# EXPLAIN (COSTS OFF)  SELECT COUNT(*) FROM pgbench_accounts ;

                                                        QUERY PLAN
------------------------------------------------------------------------------------------------
Finalize Aggregate
-> Gather
   Workers Planned: 2
   ->  Parallel Append
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 pgbench_accounts
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_2_pkey on pgbench_accounts_2 pgbench_accounts_1
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_3_pkey on pgbench_accounts_3 pgbench_accounts_2
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_4_pkey on pgbench_accounts_4 pgbench_accounts_3
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_5_pkey on pgbench_accounts_5 pgbench_accounts_4
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_6_pkey on pgbench_accounts_6 pgbench_accounts_5
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_7_pkey on pgbench_accounts_7 pgbench_accounts_6
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_8_pkey on pgbench_accounts_8 pgbench_accounts_7
     ->  Partial Aggregate
       ->  Parallel Index Only Scan using pgbench_accounts_9_pkey on pgbench_accounts_9 pgbench_accounts_8
```

```
    ->  Partial Aggregate
      ->  Parallel Index Only Scan using pgbench_accounts_10_pkey on pgbench_accounts_10 pgbench_accounts_9
```

Avec la clé, PostgreSQL se restreint à la (ou les) bonne(s) partition(s) :

```
=# EXPLAIN (COSTS OFF)  SELECT * FROM pgbench_accounts WHERE aid = 599999 ;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 pgbench_accounts
   Index Cond: (aid = 599999)
```

Si l'on connaît la clé et que le développeur sait en déduire la table, il est aussi possible d'accéder directement à la partition :

```
=# EXPLAIN (COSTS OFF) SELECT * FROM pgbench_accounts_6 WHERE aid = 599999 ;

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using pgbench_accounts_6_pkey on pgbench_accounts_6
  Index Cond: (aid = 599999)
```

Cela allège la planification, surtout s'il y a beaucoup de partitions.

Exemples :

- dans une table partitionnée par statut de commande, beaucoup de requêtes ne s'occupent que d'un statut particulier, et peuvent donc n'appeler que la partition concernée (attention si le statut change…) ;
- dans une table partitionnée par mois de création d'une facture, la date de la facture permet de s'adresser directement à la bonne partition.

Il est courant que les ORM ne sachent pas exploiter cette fonctionnalité.

6.3.1 ATTACHER/DÉTACHER UNE PARTITION

```
ALTER TABLE logs ATTACH PARTITION logs_archives
FOR VALUES FROM (MINVALUE) TO ('2019-01-01') ;
```

- Vérification du respect de la contrainte
  - parcours complet de la table: lent + verrou !

```
ALTER TABLE logs DETACH PARTITION logs_archives ;
```

- Rapide… mais verrou

Attacher une table existante à une table partitionnée implique de définir une clé de partitionnement. PostgreSQL vérifiera que les valeurs présentes correspondent bien à cette clé. Cela peut être long, surtout que le verrou nécessaire sur la table est gênant. Pour

accélérer les choses, il est conseillé d'ajouter au préalable une contrainte CHECK correspondant à la clé, voire d'ajouter d'avance les index qui seraient ajoutés lors du rattachement.

Détacher une partition est beaucoup plus rapide qu'en attacher une. Cependant, là encore, le verrou peut être gênant.

6.3.2 SUPPRIMER UNE PARTITION

```
DROP TABLE logs_2018 ;
```

Là aussi, l'opération est simple et rapide, mais demande un verrou exclusif.

6.3.3 LIMITATIONS PRINCIPALES DU PARTITIONNEMENT DÉCLARATIF

- Temps de planification ! Max ~ 100 partitions
- Création non automatique
- Pas d'héritage multiple, schéma fixe
- Partitions distantes sans propagation d'index
- Limitations avant PostgreSQL 13/14

Certaines limitations du partitionnement sont liées à son principe. Les partitions ont forcément le même schéma de données que leur partition mère. Il n'y a pas de notion d'héritage multiple.

La création des partitions n'est pas automatique (par exemple dans un partitionnement par date). Il faudra prévoir de les créer par avance.

Une limitation sérieuse du partitionnement tient au temps de planification qui augmente très vite avec le nombre de partitions, même petites. En général, on considère qu'il ne faut pas dépasser 100 partitions.

Pour contourner cette limite, il reste possible de manipuler directement les partitions s'il est facile de trouver leur nom.

Avant PostgreSQL 13, de nombreuses limitations rendent l'utilisation moins pratique ou moins performante. Si le partitionnement vous intéresse, il est conseillé d'utiliser une version la plus récente possible.

6.4 CONCLUSION

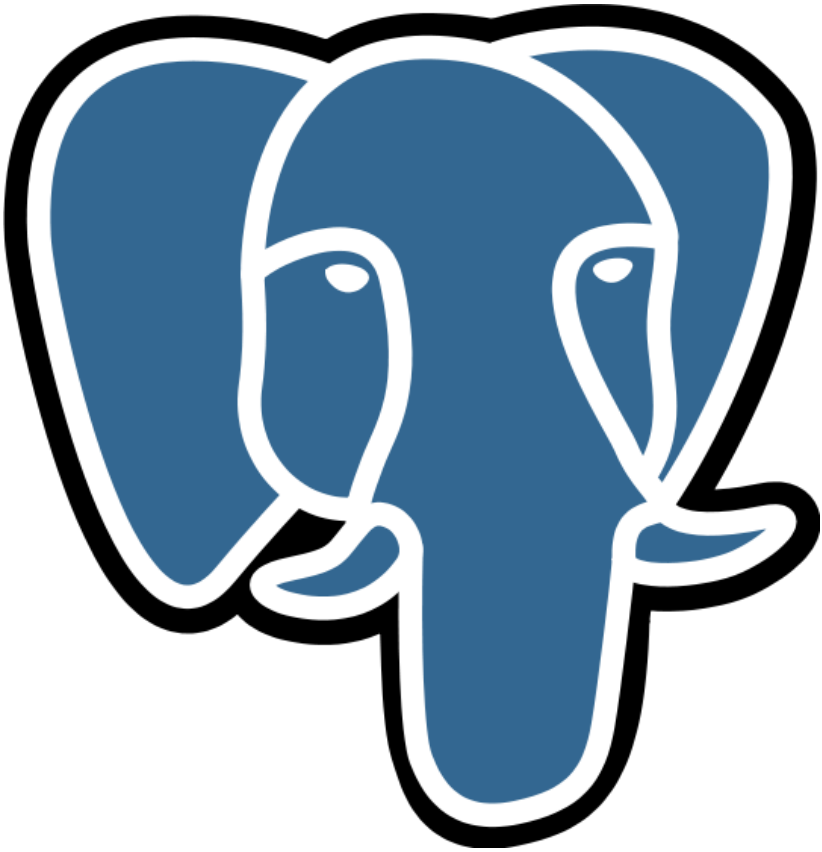
- Préférer une version récente de PostgreSQL
- Pour plus de détails sur le partitionnement
  - https://dali.bo/v1_html

Le partitionnement déclaratif apparu en version 10 est mûr dans les dernières versions. Il introduit une complexité supplémentaire, mais peut rendre de grands services quand la volumétrie augmente.

6.5 QUIZ

https://dali.bo/v0_quiz

7 SAUVEGARDE PHYSIQUE À CHAUD ET PITR



7.1 INTRODUCTION

- Sauvegarde traditionnelle
  - sauvegarde `pg_dump` à chaud
  - sauvegarde des fichiers à froid
- Insuffisant pour les grosses bases
  - long à sauvegarder
  - encore plus long à restaurer

- Perte de données potentiellement importante
  - car impossible de réaliser fréquemment une sauvegarde
- Une solution : la sauvegarde PITR

La sauvegarde traditionnelle, qu'elle soit logique ou physique à froid, répond à beaucoup de besoins. Cependant, ce type de sauvegarde montre de plus en plus ses faiblesses pour les gros volumes : la sauvegarde est longue à réaliser et encore plus longue à restaurer. Et plus une sauvegarde met du temps, moins fréquemment on l'exécute. La fenêtre de perte de données devient plus importante.

PostgreSQL propose une solution à ce problème avec la sauvegarde physique à chaud. On peut l'utiliser comme un simple mode de sauvegarde supplémentaire, mais elle permet bien d'autres possibilités, d'où le nom de PITR (*Point In Time Recovery*).

7.1.1 AU MENU

- Mettre en place la sauvegarde PITR
  - sauvegarde : manuelle, ou avec `pg_basebackup`
  - archivage : manuel, ou avec `pg_receivewal`
- Restaurer une sauvegarde PITR
- Des outils

Ce module fait le tour de la sauvegarde PITR, de la mise en place de l'archivage (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. Nous évoquerons très rapidement quelques outils externes pour faciliter ces sauvegardes.

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

7.2 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 continu grâce à l'archivage des journaux de transactions. De ce fait, ce type de sauvegarde diminue très fortement la fenêtre de perte de données.

Bien qu'elle se fasse à chaud, la sauvegarde est cohérente.

7.2.1 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
  - dans l'ordre
  - entièrement
  - ou 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). Ils doivent impérativement être rejoués dans l'ordre de leur écriture (et donc de leur nom).

7.2.2 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 toute incohérence par leur application.

Il est possible de rejouer un très grand nombre de journaux (une journée, une semaine, un mois, etc.). Évidemment, plus il y a de journaux à appliquer, plus cela prendra du temps. Mais il n'y a pas de limite au nombre de journaux à rejouer.

Dernier avantage, c'est le système de sauvegarde qui occasionnera le moins de perte de données. 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.

7.2.3 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
  - …avec arrêt si `pg_wal` plein !
- Restauration de l'instance complète
- Impossible de changer d'architecture (même OS conseillé)
- 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*). Il est même

fortement conseillé de restaurer dans la même version du même système d'exploitation, sous peine de devoir réindexer l'instance (différence de définition des locales notamment).

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 donc surveiller la taille du `pg_wal`. En cas de saturation, PostgreSQL s'arrête !

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

7.3 COPIE PHYSIQUE À CHAUD PONCTUELLE AVEC PG_BASEBACKUP

- Réalise les différentes étapes d'une sauvegarde
  - via 1 ou 2 connexions de réplication + slots de réplication
  - base backup + journaux
- Copie intégrale,
  - image de la base à la **fin** du backup
- Pas d'incrémental
- Configuration : *streaming* (rôle, droits, slots)

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

`pg_basebackup` est un produit qui a beaucoup évolué dans les dernières versions de PostgreSQL. De plus, le paramétrage par défaut depuis la version 10 le rend immédiatement utilisable.

Il permet de réaliser toute la sauvegarde de la base, à distance, via deux connexions de réplication : une pour les données, une pour les journaux de transactions qui sont générés pendant la copie.

Il est donc simple à mettre en place et à utiliser, et permet d'éviter de nombreuses étapes que nous verrons par la suite.

Par contre, il ne permet pas de réaliser une sauvegarde incrémentale, et ne permet pas de continuer à archiver les journaux, contrairement aux outils de PITR classiques. Cependant,

ceux-ci peuvent l'utiliser pour réaliser la première copie des fichiers d'une instance.

`pg_basebackup` nécessite des connexions de réplication. Par défaut, tout est en place pour cela dans PostgreSQL 10 et suivants pour une connexion en local.

En général, on veut aussi que `pg_basebackup` puisse poser un slot de réplication temporaire :

```
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
```

Ensuite, il faut configurer le fichier `pg_hba.conf` pour accepter la connexion du serveur où est exécutée `pg_basebackup`. Dans notre cas, il s'agit du même serveur avec un utilisateur dédié :

```
host  replication  sauve  127.0.0.1/32  scram-sha-256
```

Enfin, il faut créer un utilisateur dédié à la réplication (ici `sauve`) qui sera le rôle créant la connexion et lui attribuer un mot de passe :

```
CREATE ROLE sauve LOGIN REPLICATION;
\password sauve
```

Dans un but d'automatisation, le mot de passe finira souvent dans un fichier `.pgpass` ou équivalent.

Il ne reste plus qu'à :

- lancer `pg_basebackup`, ici en lui demandant une archive au format `tar` ;
- archiver les journaux en utilisant une connexion de réplication par *streaming* ;
- forcer le *checkpoint*.

Cela donne la commande suivante, ici pour une sauvegarde en local :

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/

644320/644320 kB (100%), 1/1 tablespace
```

Le résultat est ici un ensemble des deux archives : les journaux sont à part et devront être dépaquetés dans le `pg_wal` à la restauration.

```
$ ls -l /var/lib/postgresql/backups/
total 4163772
-rw------- 1 postgres postgres 659785216 Oct  9 11:37 base.tar
-rw------- 1 postgres postgres  16780288 Oct  9 11:37 pg_wal.tar
```

La cible doit être vide. En cas d'arrêt avant la fin, il faudra tout recommencer de zéro, c'est une limite de l'outil.

Pour restaurer, il suffit de remplacer le PGDATA corrompu par le contenu de l'archive, ou de créer une nouvelle instance et de remplacer son PGDATA par cette sauvegarde. Au démarrage, l'instance repérera qu'elle est une sauvegarde restaurée et réappliquera les journaux. L'instance contiendra les données telles qu'elles étaient à la **fin** du `pg_basebackup`.

Noter que les fichiers de configuration ne sont PAS inclus s'ils ne sont pas dans le PGDATA, notamment sur Debian et ses versions dérivées.

À partir de la v10, un slot temporaire sera créé par défaut pour garantir que le serveur gardera les journaux jusqu'à leur copie intégrale.

À partir de la version 13, la commande `pg_basebackup` crée un fichier manifeste contenant la liste des fichiers sauvegardés, leur taille et une somme de contrôle. Cela permet de vérifier la sauvegarde avec l'outil `pg_verifybackup`.

Lisez bien la documentation de `pg_basebackup`documentation[33] pour votre version précise de PostgreSQL, des options ont changé de nom au fil des versions.

> Même avec un serveur un peu ancien, il est possible d'installer un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

L'outil est développé plus en détail dans notre module I4[34] .

7.4 SAUVEGARDE PITR

2 étapes :

- Archivage des journaux de transactions
  - archivage interne
  - ou outil `pg_receivewal`
- Sauvegarde des fichiers
  - `pg_basebackup`
  - ou manuellement (outils de copie classiques)

Même si la mise en place est plus complexe qu'un `pg_dump`, la sauvegarde PITR demande peu d'étapes. La première chose à faire est de mettre en place l'archivage des journaux de transactions. Un choix est à faire entre un archivage classique et l'utilisation de l'outil `pg_receivewal`.

[33] https://docs.postgresql.fr/current/app-pgbasebackup.html
[34] https://dali.bo/i4_html

Lorsque cette étape est réalisée (et fonctionnelle), il est possible de passer à la seconde : la sauvegarde des fichiers. Là-aussi, il y a différentes possibilités : soit manuellement, soit `pg_basebackup`, soit son propre script ou un outil extérieur.

7.4.1 MÉTHODES D'ARCHIVAGE

- Deux méthodes :
  - processus interne `archiver`
  - outil `pg_receivewal` (flux de réplication)

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 autre méthode existe : `pg_receivewal`. Cet outil livré aussi avec PostgreSQL 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.

7.4.2 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
```

7.4.3 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
- Code retour `archive_command` entre 0 (ok) et 125

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` (le défaut jusque PostgreSQL 9.6), 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 la nécessité du niveau `replica` (valeur par défaut à partir de PostgreSQL 10).

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`. (ou `always` si l'on archive depuis un secondaire).

Enfin, la commande d'archivage s'indique au niveau du paramètre `archive_command`. `archive_command` sert à archiver un seul fichier à chaque appel. PostgreSQL l'appelle une fois pour chaque fichier WAL, dans l'ordre.

PostgreSQL laisse le soin à l'administrateur de définir la méthode d'archivage des journaux de transactions suivant son contexte. Si vous utilisez un outil de sauvegarde, la commande vous sera probablement fournie. Une simple commande de copie suffit dans la plupart des cas. La directive `archive_command` peut alors être positionnée comme suit :

```
archive_command = 'cp %p /mnt/nfs1/archivage/%f'
```

Le joker `%p` est remplacé par le chemin complet vers le journal de transactions à archiver, alors que le joker `%f` correspond au nom du journal de transactions une fois archivé.

En toute rigueur, une copie du fichier ne suffit pas. Par exemple, dans le cas de la commande `cp`, le nouveau fichier n'est pas immédiatement écrit sur disque. La copie est effectuée dans le cache disque du système d'exploitation. En cas de crash 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.

La commande d'archivage suivante est donnée dans la documentation officielle à titre d'exemple :

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
```

Cette commande a deux inconvénients. Elle ne garantit pas que les données seront synchronisées sur disque. De plus si le fichier existe ou a été copié partiellement à cause d'une erreur précédente, la copie ne s'effectuera pas. Cette protection est une bonne chose. Cependant, il faut être vigilant lorsque l'on rétablit le fonctionnement de l'*archiver* suite à un incident ayant provoqué des écritures partielles dans le répertoire d'archive, comme une saturation de l'espace disque.

Il est aussi possible d'y placer le nom d'un script bash, perl ou autre. L'intérêt est de pouvoir faire plus qu'une simple copie. On peut y ajouter la demande de synchronisation du cache sur disque. 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 le code retour de la commande est compris entre 1 et 125, PostgreSQL va tenter périodiquement d'archiver le fichier jusqu'à ce que la commande réussisse (autrement dit, renvoie 0).
Tant qu'un fichier journal n'est pas considéré comme archivé avec succès, PostgreSQL ne le supprimera ou recyclera pas !
Il ne cherchera pas non plus à archiver les fichiers suivants.

Il est donc important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur (disque plein, changement de bande, etc.).

Attention, si le code retour de la commande est supérieur à 125, cela va provoquer un redémarrage du processus `archiver`, et l'erreur ne sera pas comptabilisée dans la vue `pg_stat_archiver` !
Ce cas de figure inclut les erreurs de type `command not found` associées aux codes retours

126 et 127, ou le cas de `rsync`, qui renvoie un code retour 255 en cas d'erreur de syntaxe ou de configuration du ssh.

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[35] 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 ce nombre d'octets (16 Mo par défaut) 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.

7.4.4 PROCESSUS ARCHIVER : LANCEMENT & SUPERVISION

- Redémarrage de PostgreSQL
  - si modification de `wal_level` et/ou `archive_mode`
- ou rechargement de la configuration
- Supervision avec `pg_stat_archiver` ou `archive_status`

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.

La supervision se fait de trois manières complémentaires. D'abord la vue système `pg_stat_archiver` indique les derniers journaux archivés et les dernières erreurs. Dans l'exemple suivant, il y a eu un problème pendant quelques secondes, d'où 6 échecs, avant que l'archivage reprenne :

```
# SELECT * FROM pg_stat_archiver \gx
```

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

```
-[ RECORD 1 ]------+-----------------------------
archived_count     | 156
last_archived_wal  | 000000020000000100000000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count       | 6
last_failed_wal    | 000000020000000100000000D7
last_failed_time   | 2020-01-17 18:24:24.463038+00
stats_reset        | 2020-01-17 16:08:37.980214+00
```

Pour que cette supervision soit fiable, il faut s'assurer que la commande exécutée renvoie un code retour inférieur ou égal à 125. Dans le cas contraire le processus archiver redémarre et l'erreur n'apparaît pas dans la vue.

On peut vérifier les messages d'erreurs dans les traces (qui dépendent bien sûr du script utilisé) :

```
2020-01-17 18:24:18.427 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:18.427 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/000000020000000100000000D7 /opt/pgsql/archives/000000020000000100000000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:19.456 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:19.456 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/000000020000000100000000D7 /opt/pgsql/archives/000000020000000100000000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:20.463 UTC [15431] LOG:  archive command failed with exit code 3
```

Enfin, on peut monitorer les fichiers présents dans `pg_wal/archive_status` : ceux en `.ready` indiquent un fichier à archiver, et ne doivent pas s'accumuler :

```
postgres=# SELECT * FROM pg_ls_dir ('pg_wal/archive_status') ORDER BY 1;

           pg_ls_dir
-------------------------------
 000000020000000100000000DE.done
 000000020000000100000000DF.done
 000000020000000100000000E0.done
 000000020000000100000000E1.ready
 000000020000000100000000E2.ready
 000000020000000100000000E3.ready
 000000020000000100000000E4.ready
 000000020000000100000000E5.ready
 000000020000000100000000E6.ready
 00000002.history.done
```

PostgreSQL archive les journaux impérativement dans l'ordre. S'il y a un problème d'archivage d'un journal, les suivants ne seront pas archivés non plus, et vont s'accumuler dans `pg_wal` !

7.4.5 PG_RECEIVEWAL

- Archivage via le protocole de réplication
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
  - toujours au plus près du primaire
- Slots de réplication obligatoires

`pg_receivewal` est un outil permettant de se faire passer pour un serveur secondaire utilisant la réplication en flux (*streaming replication*) dans le but d'archiver en continu les journaux de transactions. Il fonctionne habituellement sur un autre serveur, où seront archivés les journaux. C'est une alternative à l'`archiver`.

Comme il utilise le protocole de réplication, les journaux archivés ont un retard bien inférieur à celui induit par la configuration du paramètre `archive_command`, les journaux de transactions étant écrits au fil de l'eau avant d'être complets. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le serveur primaire. On peut même utiliser une réplication synchrone (paramètres `synchronous_commit` et `synchronous_standby_names`) pour ne perdre aucune transaction au prix d'une certaine latence.

Cet outil utilise les mêmes options de connexion que la plupart des outils PostgreSQL, avec en plus l'option `-D` pour spécifier le répertoire où sauvegarder les journaux de transactions. L'utilisateur spécifié doit bien évidemment avoir les attributs `LOGIN` et `REPLICATION`.

Comme il s'agit de conserver toutes les modifications effectuées par le serveur dans le cadre d'une sauvegarde permanente, il est nécessaire de s'assurer qu'on ne puisse pas perdre des journaux de transactions. Il n'y a qu'un seul moyen pour cela : utiliser la technologie des slots de réplication. En utilisant un slot de réplication, `pg_receivewal` s'assure que le serveur ne va pas recycler des journaux dont `pg_receivewal` n'aurait pas reçu les enregistrements. On retrouve donc le risque d'accumulation des journaux sur le serveur principal si `pg_receivewal` ne fonctionne pas.

Voici l'aide de cet outil en v13 :

```
$ pg_receivewal --help
pg_receivewal reçoit le flux des journaux de transactions PostgreSQL.
```

```
Usage :
  pg_receivewal [OPTION]...

Options :
  -D, --directory=RÉP          reçoit les journaux de transactions dans ce
                               répertoire
  -E, --endpos=LSN             quitte après avoir reçu le LSN spécifié
      --if-not-exists          ne pas renvoyer une erreur si le slot existe
                               déjà lors de sa création
  -n, --no-loop                ne boucle pas en cas de perte de la connexion
      --no-sync                n'attend pas que les modifications soient
                               proprement écrites sur disque
  -s, --status-interval=SECS   durée entre l'envoi de paquets de statut au
                               (par défaut 10)
  -S, --slot=NOMREP            slot de réplication à utiliser
      --synchronous            vide le journal de transactions immédiatement
                               après son écriture
  -v, --verbose                affiche des messages verbeux
  -V, --version                affiche la version puis quitte
  -Z, --compress=0-9           compresse la sortie tar avec le niveau de
                               compression indiqué
  -?, --help                   affiche cette aide puis quitte

Options de connexion :
  -d, --dbname=CONNSTR         chaîne de connexion
  -h, --host=NOMHÔTE           hôte du serveur de bases de données ou
                               répertoire des sockets
  -p, --port=PORT              numéro de port du serveur de bases de
                               données
  -U, --username=NOM           se connecte avec cet utilisateur
  -w, --no-password            ne demande jamais le mot de passe
  -W, --password               force la demande du mot de passe
                               (devrait arriver automatiquement)

Actions optionnelles :
      --create-slot            créer un nouveau slot de réplication
                               (pour le nom du slot, voir --slot)
      --drop-slot              supprimer un nouveau slot de réplication
                               (pour le nom du slot, voir --slot)

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
```

7.4.6 PG_RECEIVEWAL - CONFIGURATION SERVEUR

- `postgresql.conf` (si < v10) :

```
max_wal_senders = 10
max_replication_slots = 10
```

- `pg_hba.conf` :

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

- Utilisateur de réplication :

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

Le paramètre `max_wal_senders` indique le nombre maximum de connexions de réplication sur le serveur. Logiquement, une valeur de 1 serait suffisante, mais il faut compter sur quelques soucis réseau qui pourraient faire perdre la connexion à `pg_receivewal` sans que le serveur primaire n'en soit mis au courant, et du fait que certains autres outils peuvent utiliser la réplication. `max_replication_slots` indique le nombre maximum de slots de réplication. Pour ces deux paramètres, le défaut est 10 à partir de PostgreSQL 10, mais 0 sur les versions précédentes, et il faudra les modifier.

Si l'on modifie un de ces paramètres, il est nécessaire de redémarrer le serveur PostgreSQL.

Les connexions de réplication nécessitent une configuration particulière au niveau des accès. D'où la modification du fichier `pg_hba.conf`. Le sous-réseau (192.168.0.0/24) est à modifier suivant l'adressage utilisé. Il est d'ailleurs préférable de n'indiquer que le serveur où est installé `pg_receivewal` (plutôt que l'intégralité d'un sous-réseau).

L'utilisation d'un utilisateur de réplication n'est pas obligatoire mais fortement conseillée pour des raisons de sécurité.

7.4.7 PG_RECEIVEWAL - REDÉMARRAGE DU SERVEUR

- Redémarrage de PostgreSQL
- Slot de réplication

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

Pour que les modifications soient prises en compte, nous devons redémarrer le serveur.

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`.

7.4.8 PG_RECEIVEWAL - LANCEMENT DE L'OUTIL

- Exemple de lancement

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

- Journaux créés en temps réel dans le répertoire de stockage
- Mise en place d'un script de démarrage
- S'il n'arrive pas à joindre le serveur primaire
  - Au démarrage de l'outil : `pg_receivewal` s'arrête
  - En cours d'exécution : `pg_receivewal` tente de se reconnecter
- Nombreuses options

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 000000010000000000000000E*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000000F*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000010.partial*
```

En cas d'incident sur le serveur primaire, il est alors possible de partir d'une sauvegarde physique et de rejouer les journaux de transactions disponibles (sans oublier de supprimer l'extension `.partial` du dernier journal).

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

7.4.9 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` ou suite a un

`pg_stop_backup`), 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. Cette méthode nécessite donc une configuration plus importante du serveur PostgreSQL. Par contre, elle 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.

7.5 SAUVEGARDE PITR MANUELLE

- 3 étapes :
  - `pg_start_backup()`
  - copie des fichiers par outil externe
  - `pg_stop_backup()`
- Exclusive : simple… & obsolète !
- Concurrente : plus complexe à scripter
- Aucun impact pour les utilisateurs ; pas de verrou
- Préférer des outils dédiés qu'un script maison

Une fois l'archivage en place, une sauvegarde à chaud a lieu en trois temps :

- l'appel à la fonction `pg_start_backup()` ;
- la copie elle-même par divers outils externes (PostgreSQL ne s'en occupe pas) ;
- l'appel à `pg_stop_backup()`.

La sauvegarde exclusive est la plus simple, et le choix par défaut. Il ne peut y en avoir qu'une à la fois. Elle ne fonctionne que depuis un primaire.

La sauvegarde concurrente, apparue avec PostgreSQL 9.6, peut être lancée plusieurs fois en parallèle. C'est utile pour créer des secondaires alors qu'une sauvegarde physique tourne, par exemple. Elle est nettement plus complexe à gérer par script. Elle peut être exécutée depuis un serveur secondaire, ce qui allège la charge sur le primaire.

Les deux sauvegardes diffèrent par les paramètres envoyés aux fonctions et les fichiers générés dans le PGDATA ou parmi les journaux de transactions.

Dans les deux cas, l'utilisateur ne verra aucun impact (à part peut-être la conséquence d'I/O saturées pendant la copie) : aucun verrou n'est posé. Lectures, écritures, suppression et création de tables, archivage de journaux et autres opérations continuent comme si de rien n'était.

La sauvegarde exclusive fonctionne encore mais est déclarée obsolète, et la fonctionnalité disparaîtra dans une version majeure prochaine. Il est donc conseillé de créer les nouveaux scripts avec des sauvegardes concurrentes.

La description du mécanisme qui suit est essentiellement destinée à la compréhension et l'expérimentation. En production, un script maison reste une possibilité, mais préférez des outils dédiés et fiables : `pg_basebackup`, pgBackRest…

7.5.1 SAUVEGARDE MANUELLE - 1/3 : PG_START_BACKUP

`SELECT pg_start_backup (`

- `un_label` : texte
- `fast` : forcer un checkpoint ?
- `exclusive` : sauvegarde exclusive
  - `backup_label`, `tablespace_map`

`)`

L'exécution de `pg_start_backup()` peut se faire depuis n'importe quelle base de données de l'instance. Le label (le texte en premier argument) n'a aucune importance pour PostgreSQL (il ne sert qu'à l'administrateur, pour reconnaître le backup).

Le deuxième argument est un booléen qui permet de demander un *checkpoint* immédiat, si l'on est pressé et si un pic d'I/O n'est pas gênant. Sinon il faudra attendre souvent plusieurs minutes (selon la configuration du déclenchement du prochain checkpoint, dépendant des paramètres `checkpoint_timeout` et `max_wal_size` et de la rapidité d'écriture imposée par `checkpoint_completion_target`).

Le troisième argument permet de demander une sauvegarde exclusive.

Si la sauvegarde est exclusive, l'appel à `pg_start_backup()` va alors créer un fichier `backup_label` dans le répertoire des données de PostgreSQL. Il contient 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. Il empêche l'exécution de deux sauvegardes PITR en parallèle. Voici un exemple de ce fichier `backup_label` en version 13 :

```
$ cat $PGDATA/backup_label
START WAL LOCATION: 0/11000028 (file 000000010000000000000011)
```

```
CHECKPOINT LOCATION: 0/11000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2019-11-25 17:59:29 CET
LABEL: backup_full_2019_11-25
START TIMELINE: 1
```

Il faut savoir qu'en cas de crash pendant une sauvegarde exclusive, la présence d'un `backup_label` peut poser des problèmes : le serveur croit à une restauration depuis une sauvegarde et ne redémarre donc pas. Le message d'erreur invitant à effacer `backup_label` est cependant explicite. C'est un argument pour éviter d'utiliser les sauvegardes exclusives. (Voir les détails dans ce mail de Robert Haas sur pgsql-hackers[36] .)

Un fichier `tablespace_map` dans PGDATA est aussi créé avec les chemins des tablespaces :

```
134141 /tbl/froid
134152 /tbl/quota
```

La sauvegarde sera concurrente si le troisième paramètre à `pg_start_backup()` est à `false`. Les fichiers `backup_label` et `tablespace_map` ne sont alors pas créés.

Les contraintes des sauvegardes en parallèle sont importantes. En effet, la session qui exécute la commande `pg_start_backup()` doit être la même que celle qui exécutera plus tard `pg_stop_backup()`. Si la connexion venait à être interrompue entre-temps, alors la sauvegarde doit être considérée comme invalide. Comme il n'y a plus de fichier `backup_label` c'est la commande `pg_stop_backup()` qui renverra les informations qui s'y trouvaient, comme nous le verrons plus tard.

Si vos scripts doivent gérer la sauvegarde concurrente, 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 au fil des versions, et il est donc recommandé d'utiliser plutôt `pg_basebackup` ou des outils supportant ces fonctionnalités (pitrery, pgBackRest...).

---

[36] https://www.postgresql.org/message-id/CA+TgmoaGvpybE=xvJeg9Jc92c-9ikeVz3k-_Hg9=mdG05u=e=g@mail.gmail.com

7.5.2 SAUVEGARDE MANUELLE - 2/3 : COPIE DES FICHIERS

- Sauvegarde des fichiers **à chaud**
  - répertoire principal des données
  - tablespaces
- Copie forcément incohérente (la restauration des journaux corrigera)
- `rsync` et autres outils
- Ignorer :
  - `postmaster.pid`, `log`, `pg_wal`, `pg_replslot` et quelques autres
- Ne pas oublier : configuration !

La deuxième étape correspond à la sauvegarde des fichiers. Le choix de l'outil dépend de l'administrateur. Cela n'a aucune incidence au niveau de PostgreSQL.

La sauvegarde doit comprendre aussi les tablespaces si l'instance en dispose.

La sauvegarde se fait à chaud : il est donc possible que pendant ce temps des fichiers changent, disparaissent avant d'être copiés ou apparaissent sans être copiés. Cela n'a pas d'importance en soi car les journaux de transactions corrigeront cela (leur archivage doit donc commencer **avant** le début de la sauvegarde et se poursuivre sans interruption jusqu'à la fin).
Il **faut** s'assurer que l'outil de sauvegarde supporte cela, c'est-à-dire qu'il soit capable de différencier les codes d'erreurs dus à « des fichiers ont bougé ou disparu lors de la sauvegarde » des autres erreurs techniques. `tar` par exemple convient : il retourne 1 pour le premier cas d'erreur, et 2 quand l'erreur est critique. `rsync` est très courant également.

Sur les plateformes Microsoft Windows, peu d'outils sont capables de copier des fichiers en cours de modification. Assurez-vous d'en utiliser un possédant cette fonctionnalité. À noter : l'outil tar (ainsi que d'autres issus du projet GNU) est disponible nativement à travers le projet unxutils[37] .

Des fichiers et répertoires sont à ignorer, pour gagner du temps ou faciliter la restauration. Voici la liste exhaustive (disponible aussi dans la documentation officielle[38]) :

- `postmaster.pid`, `postmaster.opts`, `pg_internal.init` ;
- les fichiers de données des tables non journalisées (*unlogged*) ;
- `pg_wal`, ainsi que les sous-répertoires (mais à archiver séparément !) ;
- `pg_replslot` : les slots de réplication seront au mieux périmés, au pire gênants sur l'instance restaurée ;

[37] http://unxutils.sourceforge.net/
[38] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

- `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans` ne doivent pas être copiés (ils contiennent des informations propres à l'instance, ou qui ne survivent pas à un redémarrage) ;
- les fichiers et répertoires commençant par `pgsql_tmp` (fichiers temporaires) ;
- les fichiers autres que les fichiers et les répertoires standards (donc pas les liens symboliques).

On n'oubliera pas les fichiers de configuration s'ils ne sont pas dans le PGDATA.

7.5.3 SAUVEGARDE MANUELLE - 3/3 : PG_STOP_BACKUP

Ne pas oublier !!

`SELECT * FROM pg_stop_backup (`

- `false` : si concurrente
  - fichier `.backup`
- `true` : attente de l'archivage

`)`

La dernière étape correspond à l'exécution de la procédure stockée `SELECT * FROM pg_stop_backup (true|false)`, le `false` étant alors obligatoire si la sauvegarde était concurrente.

N'oubliez pas d'exécuter `pg_stop_backup()`, et de vérifier qu'il finit avec succès !
Cet oubli trop courant rend vos sauvegardes inutilisables !

PostgreSQL va :

- marquer cette fin de backup dans le journal des transactions (étape capitale pour la restauration) ;
- forcer la finalisation du journal de transactions courant et donc son archivage, afin que la sauvegarde (fichiers + archives) soit utilisable même en cas de crash immédiat : si l'archivage est en place, `pg_stop_backup()` ne rendra pas la main (par défaut) tant que ce dernier journal n'aura pas été archivé avec succès ;
- supprimer ou renommer les fichiers `backup_label` et `tablespace_map`.

En cas de sauvegarde concurrente, il faut passer à la fonction `pg_stop_backup` un premier argument booléen à `false`. Dans ce cas, la fonction renvoie :

- le lsn de fin de backup ;
- l'équivalent du contenu du fichier `backup_label` ;
- l'équivalent du contenu du fichier `tablespace_map`.

Ce contenu doit être conservé : lors de la restauration, il servira à recréer les fichiers `backup_label` ou `tablespace_map` qui seront stockés dans le PGDATA.

```
# SELECT * FROM pg_stop_backup(false) \gx

NOTICE:  all required WAL segments have been archived
-[ RECORD 1 ]----------------------------------------------------------
lsn        | 22/2FE5C788
labelfile  | START WAL LOCATION: 22/2B000028 (file 00000001000000220000002B)+
           | CHECKPOINT LOCATION: 22/2B000060                             +
           | BACKUP METHOD: streamed                                      +
           | BACKUP FROM: master                                          +
           | START TIME: 2019-12-16 13:53:41 CET                         +
           | LABEL: rr                                                    +
           | START TIMELINE: 1                                           +
           |
spcmapfile | 134141 /tbl/froid                                           +
           | 134152 /tbl/quota                                           +
           |
```

Ces informations se retrouvent dans un fichier `.backup` mêlé aux journaux :

```
# cat /var/lib/postgresql/12/main/pg_wal/00000001000000220000002B.00000028.backup

START WAL LOCATION: 22/2B000028 (file 00000001000000220000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 00000001000000220000002F)
CHECKPOINT LOCATION: 22/2B000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2019-12-16 13:53:41 CET
LABEL: rr
START TIMELINE: 1
STOP TIME: 2019-12-16 13:54:04 CET
STOP TIMELINE: 1
```

À partir du moment où `pg_stop_backup` rend la main, il est possible de restaurer la sauvegarde obtenue puis de rejouer les journaux de transactions suivants en cas de besoin, sur un autre serveur ou sur ce même serveur.

Tous les journaux archivés avant celui précisé par le champ `START WAL LOCATION` dans le fichier `backup_label` ne sont plus nécessaires pour la récupération de la sauvegarde du système de fichiers et peuvent donc être supprimés. Attention, il y a plusieurs compteurs hexadécimaux différents dans le nom du fichier journal, qui ne sont pas incrémentés de gauche à droite.

7.5.4 SAUVEGARDE COMPLÈTE À CHAUD : PG_BASEBACKUP

- Réalise les différentes étapes d'une sauvegarde
  - via une connexion de réplication
- Crée un fichier manifeste (v13+)
- Backup de base **sans** les journaux :

```
$ pg_basebackup --format=tar --wal-method=none \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

`pg_basebackup` a été décrit plus haut. Il a l'avantage d'être simple à utiliser, de savoir quels fichiers ne pas copier, de fiabiliser la sauvegarde par un slot de réplication. Il ne réclame en général pas de configuration supplémentaire.

Si l'archivage est déjà en place, copier les journaux est inutile (`--wal-method=none`). Nous verrons plus tard comment lui indiquer où les chercher.

Le fichier manifeste permet de vérifier une sauvegarde grâce à l'outil `pg_verifybackup`.

L'inconvénient principal de `pg_basebackup` reste son incapacité à reprendre une sauvegarde interrompue ou à opérer une sauvegarde différentielle ou incrémentale.

7.5.5 FRÉQUENCE DE LA SAUVEGARDE DE BASE

- Dépend des besoins
- De tous les jours à tous les mois
- Plus elles sont espacées, plus la restauration est longue
  - et plus le risque d'un journal corrompu ou absent est important

La fréquence dépend des besoins. Une sauvegarde par jour est le plus commun, mais il est possible d'espacer encore plus la fréquence.

Cependant, il faut conserver à l'esprit que plus la sauvegarde est ancienne, plus la restauration sera longue car un plus grand nombre de journaux seront à rejouer.

7.5.6 SUIVI DE LA SAUVEGARDE DE BASE

- Vue `pg_stat_progress_basebackup`
- À partir de la v13

La version 13 permet de suivre la progression de la sauvegarde de base, quelque soit l'outil utilisé à condition qu'il passe par le protocole de réplication.

Cela permet ainsi de savoir à quelle phase la sauvegarde se trouve, quelle volumétrie a été envoyée, celle à envoyer, etc.

7.6 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.

7.6.1 RESTAURER UNE SAUVEGARDE PITR (1/5)

- 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, par exemple, de récupérer des données qui ont été supprimées par erreur.

Ensuite, il faut supprimer (ou archiver) l'ancien répertoire des données pour pouvoir y placer l'ancienne sauvegarde des fichiers. Écraser l'ancien répertoire n'est pas suffisant, il faut le supprimer, ainsi que les répertoires des tablespaces au cas où l'instance en possède.

7.6.2 RESTAURER UNE SAUVEGARDE PITR (2/5)

- 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)

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 provenance 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.

7.6.3 RESTAURER UNE SAUVEGARDE PITR (3/5)

- Fichier de configuration
  - jusque v11 : `recovery.conf`
  - v12 et au-delà : `postgresql.[auto.]conf` + `recovery.signal`
- Comment restaurer :
  - `restore_command = '... une commande ...'`

Jusqu'en version 11 incluse, la restauration se configure dans un fichier spécifique, appelé `recovery.conf`, impérativement dans le PGDATA.

À partir de la 12, on utilise directement `postgresql.conf`, ou un fichier inclus, ou `postgresql.auto.conf`. Par contre, il faut créer un fichier vide `recovery.signal`.

Ce sont ces fichiers `recovery.signal` ou `recovery.conf` qui permettent à PostgreSQL de savoir qu'il doit se lancer dans une restauration, et n'a pas simplement subi un arrêt brutal

(auquel cas il ne restaurerait que les journaux en place).

Si vous êtes sur une version antérieure à la version 12, vous pouvez vous inspirer du fichier exemple fourni avec PostgreSQL. Pour une version 10 par exemple, sur Red Hat et dérivés, c'est `/usr/pgsql-10/share/recovery.conf.sample`.
Sur Debian et dérivés, c'est `/usr/share/postgresql/10/recovery.conf.sample`. Il contient certains paramètres liés à la mise en place d'un serveur secondaire (*standby*) inutiles ici. Sinon, les paramètres sont dans les différentes parties du `postgresql.conf`.

Le paramètre essentiel est `restore_command`. Il est le pendant du paramètre `archive_command` pour l'archivage. Cette commande est souvent fournie par l'outil de sauvegarde si vous en utilisez un. 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. La restauration se poursuivra jusqu'à l'épuisement de tous les journaux disponibles.

7.6.4 RESTAURER UNE SAUVEGARDE PITR (4/5)

- Jusqu'où restaurer :
  - `recovery_target_name`, `recovery_target_time`
  - `recovery_target_xid`, `recovery_target_lsn`
  - `recovery_target_inclusive`
- Suivi de timeline :
  - `recovery_target_timeline` : `latest` ?
- Et on fait quoi ?
  - `recovery_target_action` : `pause`
  - `pg_wal_replay_resume` pour ouvrir immédiatement
  - ou modifier & redémarrer

Si l'on ne veut pas simplement restaurer tous les journaux, par exemple pour s'arrêter avant une fausse manipulation désastreuse, plusieurs paramètres permettent de préciser le point d'arrêt :

- jusqu'à un certain nom, grâce au paramètre `recovery_target_name` (le nom correspond à un label enregistré précédemment dans les journaux de transactions grâce à la fonction `pg_create_restore_point()`) ;
- jusqu'à une certaine heure, grâce au paramètre `recovery_target_time` ;

- jusqu'à un certain identifiant de transaction, grâce au paramètre `recovery_target_xid`, numéro de transaction qu'il est possible de chercher dans les journaux eux-mêmes grâce à l'utilitaire `pg_waldump` ;
- jusqu'à un certain LSN (*Log Sequence Number*[39]), grâce au paramètre `recovery_target_lsn`, que là aussi on doit aller chercher dans les journaux eux-mêmes.

Avec le paramètre `recovery_target_inclusive` (par défaut à `true`), il est possible de préciser si la restauration se fait en incluant les transactions au nom, à l'heure ou à l'identifiant de transaction demandé, ou en les excluant.

Dans les cas complexes, nous verrons plus tard que choisir la *timeline* (avec `recovery_target_timeline`, en général à `latest`) peut être utile.

Ces restaurations ponctuelles ne sont possibles que si elles correspondent à un état cohérent d'**après** la fin du *base backup*, soit après le moment du `pg_stop_backup`.

Il est possible de demander à la restauration de s'arrêter une fois arrivée au stade voulu avec `recovery_target_action = pause` (ou, jusqu'en 9.4 incluse, `pause_at_recovery_target = true`). C'est même l'action par défaut si une des options d'arrêt ci-dessus a été choisie : cela permet à l'utilisateur de vérifier que le serveur est bien arrivé au point qu'il désirait. Les alternatives sont `promote` et `shutdown`.

Si la cible est atteinte mais que l'on décide de continuer la restauration jusqu'à un autre point (évidemment postérieur), il faut modifier la cible de restauration dans le fichier de configuration, et **redémarrer** PostgreSQL. C'est le seul moyen de rejouer d'autres journaux sans ouvrir l'instance en écriture.

Si l'on est arrivé au point de restauration voulu, un message de ce genre apparaît :

```
LOG:  recovery stopping before commit of transaction 8693270, time 2021-09-02 11:46:35.394345+02
LOG:  pausing at the end of recovery
HINT:  Execute pg_wal_replay_resume() to promote.
```

(Le terme *promote* pour une restauration est un peu abusif.) `pg_wal_replay_resume()` — malgré ce que pourrait laisser croire son nom ! — provoque ici l'arrêt immédiat de la restauration, donc néglige les opérations contenues dans les WALs que l'on n'a pas souhaité restaurer, puis le serveur s'ouvre en écriture sur une nouvelle timeline.

Attention : jusque PostgreSQL 12 inclus, si un `recovery_target` est spécifié mais n'est toujours *pas* atteint à la fin du rejeu des archives, alors le mode *recovery* se termine et le serveur est promu sans erreur, et ce, même si `recovery_target_action` a la valeur `pause` ! (À condition, bien sûr, que le point de cohérence ait tout de même été dépassé.) Il faut donc être vigilant quant aux messages dans le fichier de trace de PostgreSQL !

[39] https://docs.postgresql.fr/current/datatype-pg-lsn.html

À partir de PostgreSQL 13, l'instance détecte le problème et s'arrête avec un message FATAL : la restauration ne s'est pas déroulée comme attendu. S'il manque juste certains journaux de transactions, cela permet de relancer PostgreSQL après correction de l'oubli.

La documentation officielle complète sur le sujet est sur le site du projet[40] .

7.6.5 RESTAURER UNE SAUVEGARDE PITR (5/5)

- Démarrer PostgreSQL
- Rejeu des journaux
- Vérifier que le point de cohérence est atteint !

La dernière étape est particulièrement simple. Il suffit de démarrer PostgreSQL. PostgreSQL va comprendre qu'il doit rejouer les journaux de transactions.

Les journaux doivent se dérouler au moins jusqu'à rencontrer le « point de cohérence », c'est-à-dire la mention du `pg_stop_backup`. Avant cela, il n'est pas possible de savoir si les fichiers issus du *base backup* sont à jour ou pas, et il est impossible de démarrer l'instance avant ce point. Le message apparaît dans les traces et, dans le doute, on doit vérifier sa présence :

```
2020-01-17 16:08:37.285 UTC [15221] LOG: restored log file "000000010000000100000031"…
2020-01-17 16:08:37.789 UTC [15221] LOG: restored log file "000000010000000100000032"…
2020-01-17 16:08:37.949 UTC [15221] LOG: consistent recovery state reached
                                         at 1/32BFDD88
2020-01-17 16:08:37.949 UTC [15217] LOG: database system is ready to accept
                                         read only connections
2020-01-17 16:08:38.009 UTC [15221] LOG: restored log file "000000010000000100000033"…
```

PostgreSQL continue ensuite 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). Par défaut en v12 il vérifiera qu'il n'existe pas une *timeline* supérieure sur laquelle basculer (par exemple s'il s'agit de la deuxième restauration depuis la sauvegarde du PGDATA). Puis il va s'ouvrir en écriture (sauf si vous avez demandé `recovery_target_action = pause`).

```
2020-01-17 16:08:45.938 UTC [15221] LOG: restored log file "00000001000000010000003C"
                                         from archive
2020-01-17 16:08:46.116 UTC [15221] LOG: restored log file "00000001000000010000003D"…
2020-01-17 16:08:46.547 UTC [15221] LOG: restored log file "00000001000000010000003E"…
2020-01-17 16:08:47.262 UTC [15221] LOG: restored log file "00000001000000010000003F"…
```

[40] https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET

```
2020-01-17 16:08:47.842 UTC [15221] LOG: invalid record length at 1/3F0000A0:
                                         wanted 24, got 0
2020-01-17 16:08:47.842 UTC [15221] LOG: redo done at 1/3F000028
2020-01-17 16:08:47.842 UTC [15221] LOG: last completed transaction was
                                         at log time 2020-01-17 14:59:30.093491+00
2020-01-17 16:08:47.860 UTC [15221] LOG: restored log file "000000010000000010000003F"…
cp: cannot stat ‘/opt/pgsql/archives/00000002.history’: No such file or directory
2020-01-17 16:08:47.966 UTC [15221] LOG:  selected new timeline ID: 2
2020-01-17 16:08:48.179 UTC [15221] LOG:  archive recovery complete
cp: cannot stat ‘/opt/pgsql/archives/00000001.history’: No such file or directory
2020-01-17 16:08:51.613 UTC [15217] LOG:  database system is ready
                                          to accept connections
```

À partir de la version 12, le fichier `recovery.signal` est effacé.

Avant la v12, si un fichier `recovery.conf` est présent, il sera renommé en `recovery.done`.

Ne jamais supprimer ou renommer manuellement un `recovery.conf` !

Le fichier `backup_label` d'une sauvegarde exclusive est renommé en `backup_label.old`.

7.6.6 RESTAURATION PITR : DIFFÉRENTES TIMELINES

- Fin de *recovery* => changement de *timeline* :
  - l'historique des données prend une autre voie
  - le nom des WAL change
  - fichier `.history`
- Permet plusieurs restaurations PITR à partir du même *basebackup*
- Choix :`recovery_target_timeline`
  - défaut : `latest` (v12) ou `current` (!) (<v12)

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/
000000010000000010000003C
000000010000000010000003D
000000010000000010000003E
000000010000000010000003F
000000010000000100000040
00000002.history
000000020000000010000003F
000000020000000100000040
000000020000000100000041
```

À 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`.

Par défaut, jusqu'en version 11 comprise, sa valeur est `current` et la restauration se fait dans la même *timeline* que le *base backup*. Si entre-temps il y a eu une bascule ou une précédente restauration, la nouvelle timeline ne sera pas automatiquement suivie !

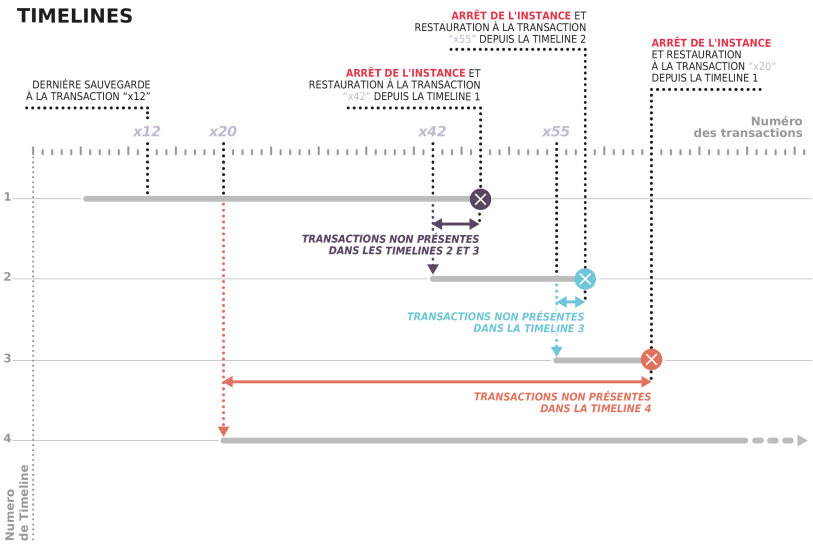
À partir de la version 12, `recovery_target_timeline` est par défaut à `latest` et la restauration suit les changements de *timeline* depuis le moment du *base backup*.

Pour choisir une autre *timeline*, il faut donner le numéro hexadécimal de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. Cela permet d'effectuer

plusieurs restaurations successives à partir du même *base backup* et d'archiver au même endroit sans mélanger les journaux.

Attention, pour restaurer dans une *timeline* précise, il faut que le fichier *history* correspondant soit présent dans les archives, sous peine d'erreur.

7.6.7 RESTAURATION PITR : ILLUSTRATION DES TIMELINES



Ce schéma illustre ce processus de plusieurs restaurations successives, et la création de différentes *timelines* qui en résulte.

On observe ici les éléments suivants avant la première restauration :

- la fin de la dernière sauvegarde se situe en haut à gauche sur l'axe des transactions, à la transaction `x12` ;
- cette sauvegarde a été effectuée alors que l'instance était en activité sur la *timeline* 1.

On décide d'arrêter l'instance alors qu'elle est arrivée à la transaction `x47`, par exemple parce qu'une nouvelle livraison de l'application a introduit un bug qui provoque des pertes de données. L'objectif est de restaurer l'instance avant l'apparition du problème afin de

récupérer les données dans un état cohérent, et de relancer la production à partir de cet état. Pour cela, on restaure les fichiers de l'instance à partir de la dernière sauvegarde, puis on modifie le fichier de configuration pour que l'instance, lors de sa phase de *recovery* :

- restaure les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaure les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x42`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 2, la bifurcation s'effectuant à la transaction `x42`. L'instance étant de nouveau ouverte en écriture, elle va générer de nouveaux WAL, qui seront associés à la nouvelle *timeline* : ils n'écrasent pas les fichiers WAL archivés de la *timeline* 1, ce qui permet de les réutiliser pour une autre restauration en cas de besoin (par exemple si la transaction `x42` utilisée comme point d'arrêt était trop loin dans le passé, et que l'on désire restaurer de nouveau jusqu'à un point plus récent).

Un peu plus tard, on a de nouveau besoin d'effectuer une restauration dans le passé - par exemple, une nouvelle livraison applicative a été effectuée, mais le bug rencontré précédemment n'était toujours pas corrigé. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, puis on configure PostgreSQL pour suivre la *timeline* 2 (paramètre `recovery_target_timeline = 2`) jusqu'à la transaction `x55`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point de la bifurcation (transaction `x42`) ;
- suivre la *timeline* indiquée (2) et rejouer les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x55`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 3, la bifurcation s'effectuant cette fois à la transaction `x55`.

Enfin, on se rend compte qu'un problème bien plus ancien et subtil a été introduit précédemment aux deux restaurations effectuées. On décide alors de restaurer l'instance jusqu'à un point dans le temps situé bien avant, jusqu'à la transaction `x20`. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, et on configure le serveur pour restaurer jusqu'à la transaction `x20`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point précédant immédiatement l' apparition du bug applicatif (transaction `x20`).

Comme la création des deux *timelines* précédentes est archivée dans les fichiers *history*,

l'ouverture de l'instance en écriture va basculer sur une nouvelle *timeline* (4). Suite à cette restauration, toutes les modifications de données provoquées par des transactions effectuées sur la *timeline* 1 après la transaction `x20`, ainsi que celles effectuées sur les *timelines* 2 et 3, ne sont donc pas présentes dans l'instance.

7.7 POUR ALLER PLUS LOIN

- Gagner en place
  - …en compressant les journaux de transactions
- Les outils dédiés à la sauvegarde

7.7.1 COMPRESSER LES JOURNAUX DE TRANSACTIONS

- Objectif : éviter de consommer trop de place disque
- Outils de compression standards : `gzip`, `bzip2`, `lzma`
  - attention à ne pas ralentir l'archivage
- `wal_compression`

L'un des problèmes de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Avec un journal généré toutes les 5 minutes, cela représente 16 Mo (par défaut) toutes les 5 minutes, soit 192 Mo par heure, ou 5 Go par jour. Il n'est pas forcément possible de conserver autant de journaux. Une 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. L'algorithme peut être imposé ou inclus dans l'outil PITR choisi. La compression peut ne pas être très intéressante en terme d'espace disque gagné. Néanmoins, un fichier de 16 Mo aura généralement une taille compressée 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 en cas d'écritures lourdes : une compression élevée mais lente peut être dangereuse.

Noter que la compression des journaux à la source existe aussi (paramètre `wal_compression`, désactivé par défaut), qui s'effectue au niveau de la page, avec un coût en CPU à l'écriture des journaux.

7.7.2 OUTILS DE SAUVEGARDE PITR DÉDIÉS

- Se faciliter la vie avec différents outils
  - pgBackRest
  - barman
  - pitrery (déprécié)
- Fournissent :
  - un outil pour les backups, les purges...
  - une commande pour l'archivage

Il n'est pas conseillé de réinventer la roue et d'écrire soi-même des scripts de sauvegarde, qui doivent prévoir de nombreux cas et bien gérer les erreurs. La sauvegarde concurrente est également difficile à manier. Des outils reconnus existent, dont nous évoquerons brièvement les plus connus. Il en existe d'autres. Ils ne font pas partie du projet PostgreSQL à proprement parler et doivent être installés séparément.

Les outils décrits succintement plus bas fournissent :

- un outil pour procéder aux sauvegardes, gérer la péremption des archives... ;
- un outil qui sera appelé par `archive_command`.

Leur philosophie peut différer, notamment en terme de centralisation ou de compromis entre simplicité et fonctionnalités. Ces dernières s'enrichissent d'ailleurs au fil du temps.

7.7.3 PGBACKREST

- Gère la sauvegarde et la restauration
  - *pull* ou *push*, multidépôts
  - mono ou multi-serveurs
- Indépendant des commandes système
  - utilise un protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multi-thread, sauvegarde depuis un secondaire, asynchrone...
- Projet récent mature

pgBackRest[41] est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

[41] https://pgbackrest.org/

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multi-thread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de `pg_wal` ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser notamment),
- le support intégré de dépôts S3 ou Azure ;
- la sauvegarde depuis un serveur secondaire ;
- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé.

Le projet est récent, très actif, considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest[42] .

7.7.4 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`...
- Spécificité : gestion de `pg_receivewal`

barman est un outil créé par 2ndQuadrant (racheté depuis par EDB). Il a pour but de faciliter la mise en place de sauvegardes PITR. Il gère à la fois la sauvegarde et la restauration.

La commande `barman` dispose de plusieurs actions :

- `list-server`, pour connaître la liste des serveurs configurés ;
- `backup`, pour lancer une sauvegarde de base ;

[42] https://github.com/pgstef/check_pgbackrest/

- `list-backup`, pour connaître la liste des sauvegardes de base ;
- `show-backup`, pour afficher des informations sur une sauvegarde ;
- `delete`, pour supprimer une sauvegarde ;
- `recover`, pour restaurer une sauvegarde (la restauration peut se faire à distance).

Contrairement aux autre outils présentés ici, barman choisit d'utiliser `pg_receivewal`.

Il supporte aussi les dépôts S3 ou blob Azure.

Site web de barman[43]

7.7.5 PITRERY

- Gère la sauvegarde et la restauration
  - mode push
  - mono-serveur
- Multi-commandes
  - `archive_wal`
  - `pitrery`
  - `restore_wal`
- Projet en fin de vie, fin du support en version 14.

pitrery a été créé par la société Dalibo. Il met l'accent sur la simplicité de 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.

Après 10 ans de développement actif, le projet Pitrery est désormais placé en maintenance LTS (*Long Term Support*) jusqu'en novembre 2026. Plus aucune nouvelle fonctionnalité n'y sera ajoutée, les mises à jour concerneront les correctifs de sécurité uniquement. Il est désormais conseillé de lui préférer pgBackRest.

pitrery propose trois exécutables :

- `archive_wal` est appelé par l'`archive_command` et gère l'archivage et la compression des journaux de transactions ;
- `pitrery` gère les sauvegardes et les restaurations ;
- `restore_wal` est appelé par `restore_command` et restaure des journaux archivés par `archive_wal`.

[43] https://www.pgbarman.org/

Note : les versions précédant la version 3 de pitrery utilisent les scripts `archive_xlog` et `restore_xlog`.

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
```

pitrery permet de restaurer une sauvegarde et de préparer la configuration de restauration :

```
$ 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
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.

pitrery est capable de vérifier qu'il possède bien l'intégralité des journaux nécessaires à la restauration depuis les plus anciennes sauvegardes complètes, et que des backups assez récents sont bien là :

```
$ pitrery  check -A -B -g 1d
```

```
2019-03-04 09:12:25 CET INFO: checking local backups in /opt/postgres/archives
2019-03-04 09:12:25 CET INFO: newest backup age: 5d 22h 54min 53s
2019-03-04 09:12:25 CET INFO: number of backups: 3
2019-03-04 09:12:25 CET ERROR: backups are too old
2019-03-04 09:12:25 CET INFO: checking local archives in
                              /opt/postgres/archives/archived_xlog
2019-03-04 09:12:25 CET INFO: oldest backup is:
                              /opt/postgres/archives/2019.02.18_23.39.33
2019-03-04 09:12:25 CET INFO: start wal file is: 00000002000001150000003C
2019-03-04 09:12:25 CET INFO: listing WAL files
2019-03-04 09:12:25 CET INFO: looking for WAL segment size
2019-03-04 09:12:25 CET INFO: WAL segment size: 32MB
2019-03-04 09:12:25 CET INFO: WAL segments per log: 127
2019-03-04 09:12:25 CET INFO: first WAL file checked is: 00000002000001150000003C.gz
2019-03-04 09:12:25 CET INFO: start WAL file found
2019-03-04 09:12:27 CET INFO: looking for 000000020000011800000000
                              (checked 324/1004, 162 segs/s)
2019-03-04 09:12:30 CET INFO: looking for 000000020000011C00000000
                              (checked 836/1004, 167 segs/s)
2019-03-04 09:12:31 CET INFO: last WAL file checked is: 000000020000011D00000023.gz
2019-03-04 09:12:31 CET INFO: all archived WAL files found
```

Pour faciliter la supervision, `pitrery check -B -A -n` permet à pitrery de se comporter comme sa propre sonde Nagios.

Site Web de pitrery[44] .

7.8 CONCLUSION

- Une sauvegarde
  - fiable
  - éprouvée
  - rapide
  - continue
- Mais
  - plus complexe à mettre en place que `pg_dump`
  - qui restaure toute l'instance

Cette méthode de sauvegarde est la seule utilisable dès que les besoins de performance de sauvegarde et de restauration augmentent (*Recovery Time Objective* ou RTO), ou que le volume de perte de données doit être drastiquement réduit (*Recovery Point Objective* ou RPO).

[44] https://dalibo.github.io/pitrery/

7.8.1 QUESTIONS

- N'hésitez pas, c'est le moment !

7.9 QUIZ

- https://dali.bo/i2_quiz

7.10 TRAVAUX PRATIQUES

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.

7.10.1 SAUVEGARDE MANUELLE

But : Mettre en place l'archivage et faire une sauvegarde physique à chaud manuelle avec `pg_start_backup` et `pg_stop_backup`.

> Mettre en place l'archivage des journaux de transactions vers `/opt/pgsql/archives` avec `rsync`.

> Générer de l'activité avec `pgbench` et laisser tourner en arrière-plan.

> Vérifier que les journaux de transactions arrivent bien dans le répertoire d'archivage.

> Indiquer à PostgreSQL qu'une sauvegarde exclusive va être lancée.
> Sauvegarder l'instance à l'aide d'un utilitaire d'archivage (`tar` par exemple) vers `/opt/pgsql/backups/`. Penser à exclure le contenu des répertoires `pg_wal` et `log`.
> À la fin de la sauvegarde, relever la dernière valeur de `pgbench_history.mtime`.
> Ne pas oublier la dernière opération après la sauvegarde.

> Attendre qu'au moins un nouveau journal de transactions soit archivé, puis arrêter l'activité.

7.10.2 RESTAURATION

But : Restaurer une sauvegarde physique.

> Arrêter PostgreSQL.

> Renommer le répertoire `/var/lib/pgsql/14/data` en `/var/lib/pgsql/14/data.old`.

> Comparer le dernier journal de l'instance arrêtée et le dernier archivé, et le déplacer dans le répertoire d'archivage des journaux de transactions au besoin.

> Restaurer le PGDATA de l'instance en utilisant la sauvegarde à chaud : restaurer l'archive, vérifier le contenu de `backup_label`.

> Effacer les journaux éventuellement présents dans ce nouveau PGDATA.

> Créer les fichiers nécessaires à la restauration et renseigner `restore_command`.

> Afficher en continu les traces dans une fenêtre avec `tail -F`.

> Relancer l'instance et attendre la fin de la restauration.
> Vérifier que les journaux sont réappliqués puis que le point de cohérence a été atteint.
> Que se passe-t-il quand le dernier journal a été restauré ?

> Quels sont les fichiers, disparus, modifiés ?

> Vérifier que la connexion peut se faire.
> Générer des journaux : quel est leur nom ?

> Récupérer la dernière valeur de `pgbench_history.mtime`. Vérifier qu'elle est supérieure à celle relevée à la fin de la sauvegarde.

7.10.3 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

But : Créer une sauvegarde physique à chaud à un moment précis de la base avec `pg_basebackup`, et la restaurer.

> Configurer la réplication dans `postgresql.conf` : activer l'archivage, autoriser des connexions en streaming.

> Insérer des données et générer de l'activité avec `pgbench`.
> En parallèle, sauvegarder l'instance avec `pg_basebackup` au format tar, sans oublier les journaux, et avec l'option `--max-rate=16M` pour ralentir la sauvegarde.

> Tout au long de l'exécution de la sauvegarde, surveiller l'évolution de l'activité sur la table `pgbench_history`.

> Une fois la sauvegarde terminée, vérifier son état ainsi que la dernière donnée modifiée dans `pgbench_history`.

> Arrêter l'instance.
> Par sécurité, faites une copie à froid des données (par exemple avec `cp -rfp`).
> Vider le répertoire.
> Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.

> Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

> Quelle est la dernière donnée restaurée ?

> Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction.

7.10.4 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

But : Coupler une sauvegarde à chaud avec `pg_basebackup` et l'archivage

> Remettre en place la copie de l'instance prise précédemment.
> Configurer l'archivage.

> Générer à nouveau de l'activité avec `pgbench`.
> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup`.

> Vérifier que des archives sont bien générées.

> Effacer le PGDATA.
> Restaurer la sauvegarde, configurer la `restore_command` et créer le fichier `recovery.signal`.

> Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

7.11 TRAVAUX PRATIQUES (SOLUTIONS)

7.11.1 SAUVEGARDE MANUELLE

> Mettre en place l'archivage des journaux de transactions vers `/opt/pgsql/archives` avec `rsync`.

D'abord créer le répertoire `/opt/pgsql/archives` avec les droits nécessaire pour l'utilisateur système **postgres** :

```
# mkdir -p /opt/pgsql/archives
# chown -R postgres /opt/pgsql
```

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'
```

Positionner le paramètre `archive_mode` à `on`.

Un redémarrage est nécessaire si le paramètre `archive_mode` a été modifié :

```
# systemctl restart postgresql-14
```

Si seul `archive_command` a été modifié, `systemctl reload postgresql-14` suffira.

> Générer de l'activité avec `pgbench` et laisser tourner en arrière-plan.

Si la base `pgbench` n'existe pas déjà, on peut créer une base d'environ 1 Go ainsi :

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 100 pgbench
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data...
100000 of 10000000 tuples (1%) done (elapsed 0.09 s, remaining 9.39 s)
200000 of 10000000 tuples (2%) done (elapsed 0.36 s, remaining 17.46 s)
…
10000000 of 10000000 tuples (100%) done (elapsed 28.57 s, remaining 0.00 s)
vacuuming...
```

```
creating primary keys...
done.
```

L'activité peut être générée par exemple ainsi pendant 30 min :

```
/usr/pgsql-14/bin/pgbench -n -T1800 -c 10 -j2 pgbench
```

> Vérifier que les journaux de transactions arrivent bien dans le répertoire d'archivage.

La quantité de journaux générés dépend de la machine et des disques. Ils doivent se retrouver dans `/opt/pgsql/archives/` :

```
$ ls -l /opt/pgsql/archives
total 540672
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000BE
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000BF
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C0
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C1
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C2
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C3
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C4
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C5
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C6
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C7
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C8
…
```

> Indiquer à PostgreSQL qu'une sauvegarde exclusive va être lancée.
> Sauvegarder l'instance à l'aide d'un utilitaire d'archivage (`tar` par exemple) vers `/opt/pgsql/backups/`. Penser à exclure le contenu des répertoires `pg_wal` et `log`.
> À la fin de la sauvegarde, relever la dernière valeur de `pgbench_history.mtime`.
> Ne pas oublier la dernière opération après la sauvegarde.

Indiquer à PostgreSQL que la sauvegarde exclusive va démarrer :

```
SELECT pg_start_backup('backup '||current_timestamp, true);
```

Sauvegarder les fichiers de l'instance :

```
$ cd /var/lib/pgsql/14/data/
$ mkdir -p /opt/pgsql/backups
```

```
$ tar -cvhz . -f /opt/pgsql/backups/backup_$(date +%F).tgz \
  --exclude='pg_wal/*' --exclude='log/*'

./
./pg_wal/
./global/
./global/pg_control
./global/1262
./global/2964
./global/1213
./global/1136
…
./pg_ident.conf
./log/
./log/postgresql-Fri.log
./postmaster.opts
./postgresql.conf
./postmaster.pid
./current_logfiles
./backup_label
```

Noter que l'erreur suivante est sans conséquence, les fichiers peuvent être modifiés pendant une copie à chaud :

```
./base/16384/16397
tar: ./base/16384/16397: file changed as we read it
```

Enfin, il ne faut surtout pas oublier d'indiquer à PostgreSQL la fin de la sauvegarde :

```
SELECT pg_stop_backup();

NOTICE:  all required WAL segments have been archived
 pg_stop_backup
----------------
 1/32BFDD88
```

On repère la dernière valeur :

```
$ psql pgbench
pgbench=# SELECT max(mtime) FROM pgbench_history ;

            max
---------------------------
  2020-01-17 14:58:28.423145
```

> Attendre qu'au moins un nouveau journal de transactions soit archivé, puis arrêter l'activité.

Forcer au besoin un changement de journal avec :

```
SELECT pg_switch_wal();
```

7.11.2 RESTAURATION

> Arrêter PostgreSQL.

En tant qu'utilisateur root :

```
# systemctl stop postgresql-14
```

Alternativement, on peut être plus brutal et faire un `kill -9` sur le `postmaster`.

> Renommer le répertoire `/var/lib/pgsql/14/data` en `/var/lib/pgsql/14/data.old`.

```
$ mv /var/lib/pgsql/14/data /var/lib/pgsql/14/data.old
```

> Comparer le dernier journal de l'instance arrêtée et le dernier archivé, et le déplacer dans le répertoire d'archivage des journaux de transactions au besoin.

Avant une restauration, il faut toujours chercher à sauver les toutes dernières transactions. Ici le dernier journal archivé est :

```
$ ls -lrt /opt/pgsql/archives/ | tail -n1
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 000000010000000100000003E
```

On recherche le dernier modifié :

```
$ ls -lr /var/lib/pgsql/14/data.old/pg_wal
total 1048592
drwx------. 2 postgres postgres     8192 17 janv. 14:59 archive_status
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 000000010000000100000007E
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 000000010000000100000007D
…
-rw-------. 1 postgres postgres 16777216 17 janv. 14:53 000000010000000100000043
-rw-------. 1 postgres postgres 16777216 17 janv. 14:52 000000010000000100000042
-rw-------. 1 postgres postgres 16777216 17 janv. 14:52 000000010000000100000041
-rw-------. 1 postgres postgres 16777216 17 janv. 14:54 000000010000000100000040
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 000000010000000100000003F
-rw-------. 1 postgres postgres      358 17 janv. 14:58 000000010000000000000FA.00370610.backup
```

Attention ! Selon l'activité il peut y avoir de nombreux journaux recyclés, mais ne contenant pas de données intéressantes. Il faut regarder la date de modification.

Ci-dessus le dernier journal intéressant est bien le `00000001000000010000003F`, ceux à partir de `000000010000000100000040` sont d'anciens journaux modifiés précédemment et renommés.

PostgreSQL n'a pas eu le temps de terminer et d'archiver ce journal, il faut le récupérer manuellement :

```
$ cp /var/lib/pgsql/14/data.old/pg_wal/00000001000000010000003F  /opt/pgsql/archives/
```

> Restaurer le PGDATA de l'instance en utilisant la sauvegarde à chaud : restaurer l'archive, vérifier le contenu de `backup_label`.

Restaurer l'archive de la dernière sauvegarde :

```
$ cd /var/lib/pgsql/14
$ mkdir data
$ cd data
$ tar xzvf /opt/pgsql/backups/backup_$(date +%F).tgz
…
```

Le `backup_label` a été créé le temps de la sauvegarde par `pg_start_backup` (ce ne serait pas le cas dans une sauvegarde concurrente, PostgreSQL utilise d'autres moyens) et est donc inclus dans la sauvegarde.

```
$ cat backup_label
START WAL LOCATION: 0/FA370610 (file 000000010000000000000FA)
CHECKPOINT LOCATION: 0/FA604098
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2020-01-17 14:52:51 UTC
LABEL: backup 2020-01-17 14:52:48.456761+00
START TIMELINE: 1
```

> Effacer les journaux éventuellement présents dans ce nouveau PGDATA.

Effacer les anciens journaux de l'archive que l'on vient d'extraire, si on avait oublié de les extraire. On peut en profiter pour effacer certains fichiers comme `postmaster.pid` :

```
$ rm -f /var/lib/pgsql/14/data/pg_wal/00*   postmaster.pid
```

> Créer les fichiers nécessaires à la restauration et renseigner `restore_command`.

À partir de la v12 : ajouter la commande de récupération des journaux dans `postgresql.auto.conf` :

```
restore_command = 'cp /opt/pgsql/archives/%f %p'
```

Toujours à partir de la v12 : créer le fichier `recovery.signal` :

```
$ touch /var/lib/pgsql/14/data/recovery.signal
```

Jusqu'en v11 : créer le fichier `recovery.conf` dans le PGDATA, et y préciser `restore_command`. Ce fichier peut aussi être copié depuis un fichier d'exemple fourni avec PostgreSQL (sur Red Hat/CentOS/Rocky Linux : `/usr/pgsql-11/share/recovery.conf.sample`), qui liste toutes les options possibles.

> Afficher en continu les traces dans une fenêtre avec `tail -F`.

Le nom du fichier peut différer suivant le jour de la semaine et le paramétrage. Noter aussi qu'il n'existe pas si `log/*` a bien été exclu du `tar`. `tail -F` l'affichera dès qu'il apparaîtra :

```
$ tail -F /var/lib/pgsql/14/data/log/postgresql-Thu.log
```

> Relancer l'instance et attendre la fin de la restauration.
> Vérifier que les journaux sont réappliqués puis que le point de cohérence a été atteint.
> Que se passe-t-il quand le dernier journal a été restauré ?

Redémarrer le serveur PostgreSQL :

```
# systemctl start postgresql-14
```

La trace indique que PostgreSQL a repéré qu'il était interrompu, et il commence à restaurer les journaux :

```
2020-01-17 16:08:06.373 UTC [15221] LOG:  database system was interrupted;
                                    last known up at 2020-01-17 14:52:51 UTC
cp: cannot stat ‘/opt/pgsql/archives/00000002.history’: No such file or directory
2020-01-17 16:08:06.457 UTC [15221] LOG: starting archive recovery
2020-01-17 16:08:06.524 UTC [15221] LOG: restored log file "0000000100000000000000FA"…
2020-01-17 16:08:06.657 UTC [15221] LOG: redo starts at 0/FA370610
2020-01-17 16:08:06.805 UTC [15221] LOG: restored log file "0000000100000000000000FB"…
2020-01-17 16:08:07.008 UTC [15221] LOG: restored log file "0000000100000000000000FC"…
```

Noter que PostgreSQL a commencé par chercher s'il y avait une *timeline* supérieure à celle sur laquelle il a démarré (le défaut à partir de la v12 est `recovery_target_timeline = latest`).

Ne trouvant pas ce fichier, il a commencé par rejouer ce qu'il a trouvé sur le *timeline* 1, à partir du fichier `000000010000000000000FA` (cela correspond au *checkpoint* indiqué dans `backup_label`).

- S'il y a des problème de droits, les corriger sur le répertoire `data` et les fichiers qu'il contient :

```
$ chown -R postgres /var/lib/pgsql/14/data
$ chmod -R 700 /var/lib/pgsql/14/data
```

- Le message suivant apparaît quand le fichier `backup_label` est présent et que l'on a oublié de créer `recovery.signal`. En effet, PostgreSQL ne peut pas savoir s'il a crashé au milieu d'une sauvegarde et doit redémarrer sur les journaux présents, ou s'il doit restaurer des journaux avec `restore_command`, ou encore s'il doit se considérer comme un serveur secondaire. Le message indique bien la procédure à suivre :

```
2020-01-17 16:01:16.595 UTC [14188] FATAL:  could not locate required checkpoint
                                            record
2020-01-17 16:01:16.595 UTC [14188] HINT:  If you are restoring from a backup,
         touch "/var/lib/pgsql/14/data/recovery.signal" and add required
         recovery options.
        If you are not restoring from a backup,
         try removing the file "/var/lib/pgsql/14/data/backup_label".
        Be careful: removing "/var/lib/pgsql/14/data/backup_label" will result
         in a corrupt cluster if restoring from a backup.
2020-01-17 16:01:16.597 UTC [14185] LOG:  startup process (PID 14188) exited
                                           with exit code 1
```

Il est impératif de vérifier que le point de cohérence a été atteint : il s'agit du moment où `pg_stop_backup` a marqué dans les journaux la fin du *base backup*. On pourrait arrêter le rejeu à partir d'ici au besoin.

```
2020-01-17 16:08:37.789 UTC [15221] LOG: restored log file "000000010000000100000032"…
2020-01-17 16:08:37.949 UTC [15221] LOG: consistent recovery state reached
                                         at 1/32BFDD88
2020-01-17 16:08:37.949 UTC [15217] LOG: database system is ready to accept
                                         read only connections
```

Si le point de cohérence n'est pas atteint, l'instance est inutilisable ! Elle attendra indéfiniment de nouveaux journaux. C'est notamment le cas si `pg_stop_backup` a été oublié, et que PostgreSQL ne sait pas quand il doit s'arrêter !

Il est même possible de voir la présence du point de cohérence à la fin du fichier journal lui-même :

```
$ /usr/pgsql-14/bin/pg_waldump   /opt/pgsql/archives/000000010000000100000032
```

```
rmgr: Heap        len (rec/tot):     79/    79, tx:     270079, lsn: 1/32BFDC48,
                  prev 1/32BFDC00, desc: INSERT off 37 flags 0x00,
                  blkref #0: rel 1663/16384/16399 blk 1766
rmgr: Transaction len (rec/tot):     34/    34, tx:     270082, lsn: 1/32BFDC98,
                  prev 1/32BFDC48, desc: COMMIT 2020-01-17 14:58:16.102856 UTC
rmgr: Heap        len (rec/tot):     79/    79, tx:     270083, lsn: 1/32BFDCC0,
                  prev 1/32BFDC98, desc: INSERT off 41 flags 0x00,
                  blkref #0: rel 1663/16384/16399 blk 1764
rmgr: Transaction len (rec/tot):     34/    34, tx:     270079, lsn: 1/32BFDD10,
                  prev 1/32BFDCC0, desc: COMMIT 2020-01-17 14:58:16.102976 UTC
rmgr: Transaction len (rec/tot):     34/    34, tx:     270083, lsn: 1/32BFDD38,
                  prev 1/32BFDD10, desc: COMMIT 2020-01-17 14:58:16.103000 UTC
rmgr: XLOG        len (rec/tot):     34/    34, tx:          0, lsn: 1/32BFDD60,
                  prev 1/32BFDD38, desc: BACKUP_END 0/FA370610
rmgr: XLOG        len (rec/tot):     24/    24, tx:          0, lsn: 1/32BFDD88,
                  prev 1/32BFDD60, desc: SWITCH
```

Le rejeu des derniers journaux mène aux messages suivants :

```
2020-01-17 16:08:46.116 UTC [15221] LOG: restored log file "00000001000000010000003D"…
2020-01-17 16:08:46.547 UTC [15221] LOG: restored log file "00000001000000010000003E"…
2020-01-17 16:08:47.262 UTC [15221] LOG: restored log file "00000001000000010000003F"…
2020-01-17 16:08:47.842 UTC [15221] LOG: invalid record length at 1/3F0000A0:
                                         wanted 24, got 0
2020-01-17 16:08:47.842 UTC [15221] LOG: redo done at 1/3F000028
2020-01-17 16:08:47.842 UTC [15221] LOG: last completed transaction was at log time
                                         2020-01-17 14:59:30.093491+00
2020-01-17 16:08:47.860 UTC [15221] LOG: restored log file "00000001000000010000003F"…
cp: cannot stat ‘/opt/pgsql/archives/00000002.history’: No such file or directory
2020-01-17 16:08:47.966 UTC [15221] LOG: selected new timeline ID: 2
2020-01-17 16:08:48.179 UTC [15221] LOG: archive recovery complete
cp: cannot stat ‘/opt/pgsql/archives/00000001.history’: No such file or directory
2020-01-17 16:08:51.613 UTC [15217] LOG: database system is ready
                                         to accept connections
```

Cela correspond à la constatation que le dernier journal est incomplet (c'est celui récupéré plus haut à la main), la recherche en vain d'une *timeline* supérieure à suivre (le fichier aurait pu apparaître entre temps), puis la création de la nouvelle *timeline*.

> Quels sont les fichiers, disparus, modifiés ?

Une fois la restauration terminée :

- `backup_label` a été archivé en `backup_label.old` ;
- à partir de la v12 : `recovery.signal` a disparu ;
- jusqu'en v11 comprise : `recovery.conf` a été archivé en `recovery.done`.

Enfin, le fichier `pg_wal/00000002.history` contient le point de divergence entre les deux *timelines* :

```
# cat 00000002.history
1       1/3F0000A0      no recovery target specified
```

Le `postgresql.auto.conf` peut conserver la `recovery_command` si l'on est sûr que ce sera la bonne à la prochaine restauration.

> Vérifier que la connexion peut se faire.
> Générer des journaux : quel est leur nom ?

```
$ psql pgbench
psql (14.1)
Saisissez « help » pour l'aide.

pgbench=# VACUUM ;
```

Les journaux générés appartiennent bien à la *timeline* 2 :

```
$ ls -altr data/pg_wal/

total 737244
-rw-------.  1 postgres postgres 16777216 17 janv. 16:08 000000010000000010000003F
drwx------. 20 postgres postgres     4096 17 janv. 16:08 ..
-rw-------.  1 postgres postgres       42 17 janv. 16:08 00000002.history
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 000000020000000010000003F
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000040
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000041
…
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000064
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000065
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000066
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000067
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000068
```

> Récupérer la dernière valeur de `pgbench_history.mtime`. Vérifier qu'elle est supérieure à celle relevée à la fin de la sauvegarde.

```
SELECT max(mtime) FROM pgbench_history ;

            max
----------------------------
 2020-01-17 14:59:30.092409
```

7.11.3 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

> Configurer la réplication dans `postgresql.conf` : activer l'archivage, autoriser des connexions en streaming.

Si l'archivage est actif, ici on choisit de ne pas archiver réellement et de ne passer que par la réplication :

```
archive_mode = on
archive_command = '/bin/true'
```

Vérifier la configurer de l'autorisation de connexion en réplication dans `pg_hba.conf`. Si besoin, mettre à jour la ligne en fin de fichier :

```
local   replication     all                                     trust
```

Redémarrer PostgreSQL :

```
# systemctl restart postgresql-14
```

> Insérer des données et générer de l'activité avec `pgbench`.
> En parallèle, sauvegarder l'instance avec `pg_basebackup` au format tar, sans oublier les journaux, et avec l'option `--max-rate=16M` pour ralentir la sauvegarde.

```
$ createdb bench
$ /usr/pgsql-14/bin/pgbench -i -s 100 bench
$ vacuumdb -az
$ psql -c "CHECKPOINT;"
$ /usr/pgsql-14/bin/pgbench bench -n -P 5 -T 360

$ pg_basebackup -D /var/lib/pgsql/14/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=16M

1567192/1567192 kB (100%), 1/1 tablespace
```

> Tout au long de l'exécution de la sauvegarde, surveiller l'évolution de l'activité sur la table `pgbench_history`.

```
$ watch -n 5 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'"
```

> Une fois la sauvegarde terminée, vérifier son état ainsi que la dernière donnée modifiée dans `pgbench_history`.

```
$ ls -lha /var/lib/pgsql/14/backups/basebackup
(...)
-rw-------. 1 postgres postgres 86M Nov 29 14:25 base.tar.gz
-rw-------. 1 postgres postgres 32M Nov 29 14:25 pg_wal.tar.gz

$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'

            max
----------------------------
 2019-11-29 14:27:21.673308
```

> Arrêter l'instance.
> Par sécurité, faites une copie à froid des données (par exemple avec `cp -rfp`).
> Vider le répertoire.
> Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.

```
# systemctl stop postgresql-14
# cp -rfp /var/lib/pgsql/14/data /var/lib/pgsql/14/data.old
# rm -rf /var/lib/pgsql/14/data/*
# tar -C /var/lib/pgsql/14/data \
      -xzf /var/lib/pgsql/14/backups/basebackup/base.tar.gz
# tar -C /var/lib/pgsql/14/data/pg_wal \
      -xzf /var/lib/pgsql/14/backups/basebackup/pg_wal.tar.gz
# rm -rf /var/lib/pgsql/14/data/log/*
# systemctl start postgresql-14
```

> Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

```
# tail -f /var/lib/pgsql/14/data/log/postgresql-*.log
2019-11-29 14:29:47.733 CET [7732] LOG: database system was interrupted;
last known up at 2019-11-29 14:24:10 CET
2019-11-29 14:29:47.811 CET [7732] LOG: redo starts at 4/3B000028
2019-11-29 14:29:48.489 CET [7732] LOG:
                                   consistent recovery state reached at 4/4A960188
2019-11-29 14:29:48.489 CET [7732] LOG: redo done at 4/4A960188
2019-11-29 14:29:55.105 CET [7729] LOG:
                                   database system is ready to accept connections
```

> Quelle est la dernière donnée restaurée ?

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
---------------------------
 2019-11-29 14:25:45.698596
```

Grâce aux journaux (`pg_wal`) restaurés, l'ensemble des modifications survenues lors de la sauvegarde ont bien été récupérées.

Toutefois, les données générées après la sauvegarde n'ont, elles, pas été récupérées.

> Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction.

```
# systemctl stop postgresql-14
# rm -rf /var/lib/pgsql/14/data/*
# tar -C /var/lib/pgsql/14/data \
      -xzf /var/lib/pgsql/14/backups/basebackup/base.tar.gz
# rm -rf /var/lib/pgsql/14/data/log/*
# systemctl start postgresql-14

# cat /var/lib/pgsql/14/data/log/postgresql-*.log
2019-11-29 14:44:14.958 CET [7856] LOG:
           database system was shut down in recovery at 2019-11-29 14:43:59 CET
2019-11-29 14:44:14.958 CET [7856] LOG:  invalid checkpoint record
2019-11-29 14:44:14.958 CET [7856]
           FATAL:  could not locate required checkpoint record
2019-11-29 14:44:14.958 CET [7856] HINT:  If you are restoring from a backup,
    touch "/var/lib/pgsql/14/data/recovery.signal" and add required recovery options.
    If you are not restoring from a backup, try removing the file
    "/var/lib/pgsql/14/data/backup_label".
    Be careful: removing "/var/lib/pgsql/14/data/backup_label" will result
    in a corrupt cluster if restoring from a backup.
2019-11-29 14:44:14.959 CET [7853] LOG: startup process (PID 7856) exited
                                       with exit code 1
2019-11-29 14:44:14.959 CET [7853] LOG: aborting startup due to startup
                                       process failure
2019-11-29 14:44:14.960 CET [7853] LOG: database system is shut down
```

PostgreSQL ne trouvant pas les journaux nécessaires à sa restauration à un état cohérent, le service refuse de démarrer. Supprimer le fichier `backup_label` permettrait toutefois de démarrer l'instance MAIS celle-ci serait alors dans un état incohérent.

7.11.4 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

> Remettre en place la copie de l'instance prise précédemment.
> Configurer l'archivage.

```
# systemctl stop postgresql-14
# rm -rf /var/lib/pgsql/14/data
# cp -rfp /var/lib/pgsql/14/data.old /var/lib/pgsql/14/data
```

Créer le répertoire d'archivage s'il n'existe pas déjà :

```
$ mkdir /var/lib/pgsql/14/archives
```

L'utilisateur système `postgres` doit avoir le droit d'y écrire.

Ensuite, dans `postgresql.conf` :

```
archive_command = 'rsync %p /var/lib/pgsql/14/archives/%f'

# systemctl start postgresql-14
# rm -rf /var/lib/pgsql/14/backups/basebackup/*
```

> Générer à nouveau de l'activité avec `pgbench`.
> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup`.

```
$ psql -c "CHECKPOINT;"
$ /usr/pgsql-14/bin/pgbench bench -n -P 5 -T 360
```

En parallèle, lancer à nouveau `pg_basebackup` :

```
$ pg_basebackup -D /var/lib/pgsql/14/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=16M
1567192/1567192 kB (100%), 1/1 tablespace
```

> Vérifier que des archives sont bien générées.

```
$ ls -lha /var/lib/pgsql/14/archives
(...)
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005C
-rw-------. 1 postgres postgres  340 Nov 29 14:55
                                        000000010000000040000005C.00002088.backup
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005D
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005E
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005F
-rw-------. 1 postgres postgres  16M Nov 29 14:54 0000000100000000400000060
-rw-------. 1 postgres postgres  16M Nov 29 14:54 0000000100000000400000061
(...)
```

> Effacer le PGDATA.
> Restaurer la sauvegarde, configurer la `restore_command` et créer le fichier `recovery.signal`.

```
# systemctl stop postgresql-14
# rm -rf /var/lib/pgsql/14/data/*
# tar -C /var/lib/pgsql/14/data \
  -xzf /var/lib/pgsql/14/backups/basebackup/base.tar.gz
# rm -rf /var/lib/pgsql/14/data/log/*
```

Configurer la `restore_command` dans le fichier `postgresql.conf` :

```
restore_command = 'rsync /var/lib/pgsql/14/archives/%f %p'
```

Créer le fichier `recovery.signal` :

```
$ touch /var/lib/pgsql/14/data/recovery.signal
```

Démarrer le service :

```
# systemctl start postgresql-14
```

> Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

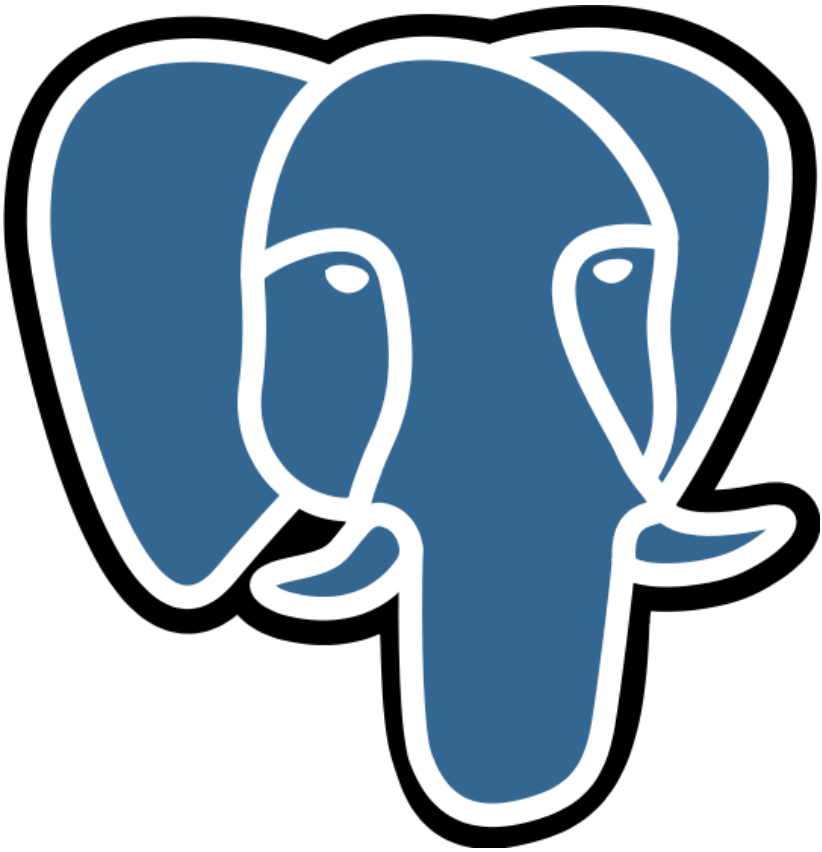
```
# tail -f /var/lib/pgsql/14/data/log/postgresql-*.log
2019-11-29 15:02:34.736 CET [8280] LOG:  database system was interrupted;
last known up at 2019-11-29 14:54:19 CET
2019-11-29 15:02:34.808 CET [8280] LOG: starting archive recovery
2019-11-29 15:02:34.903 CET [8280] LOG: restored log file "0000000100000040000005C"
                                       from archive
2019-11-29 15:02:34.953 CET [8280] LOG: redo starts at 4/5C002088
2019-11-29 15:02:35.083 CET [8280] LOG: restored log file "0000000100000040000005D"…
(...)
2019-11-29 15:02:37.254 CET [8280] LOG: restored log file "00000001000000400000069"…
2019-11-29 15:02:37.328 CET [8280] LOG:
                                    consistent recovery state reached at 4/69AD0728
2019-11-29 15:02:37.328 CET [8277] LOG: database system is ready to accept
                                       read only connections
2019-11-29 15:02:37.430 CET [8280] LOG: restored log file "0000000100000040000006A"…
2019-11-29 15:02:37.614 CET [8280] LOG: restored log file "0000000100000040000006B"…
(...)
2019-11-29 15:03:08.599 CET [8280] LOG: restored log file "00000001000000400000096"…
2019-11-29 15:03:08.810 CET [8280] LOG: redo done at 4/9694CA30
2019-11-29 15:03:08.810 CET [8280] LOG:
            last completed transaction was at log time 2019-11-29 15:00:06.315365+01
```

```
2019-11-29 15:03:09.087 CET [8280] LOG: selected new timeline ID: 2
2019-11-29 15:03:09.242 CET [8280] LOG: archive recovery complete
2019-11-29 15:03:15.571 CET [8277] LOG: database system is ready
                                        to accept connections
```

Cette fois, toutes les données générées après la sauvegarde ont bien été récupérées :

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
            max
----------------------------
 2019-11-29 15:00:06.313512
```

8 POSTGRESQL : GESTION D'UN SINISTRE



8.1 INTRODUCTION

- Une bonne politique de sauvegardes est cruciale
  - mais elle n'empêche pas les incidents
- Il faut être prêt à y faire face

Ce module se propose de faire une description des bonnes et mauvaises pratiques en cas de coup dur :

- crash de l'instance ;
- suppression / corruption de fichiers ;
- problèmes matériels ;
- sauvegardes corrompues…

Seront également présentées les situations classiques de désastres, ainsi que certaines méthodes et outils dangereux et déconseillés.

L'objectif est d'aider à convaincre de l'intérêt qu'il y a à anticiper les problèmes, à mettre en place une politique de sauvegarde pérenne, et à ne pas tenter de manipulation dangereuse sans comprendre précisément à quoi l'on s'expose.

Ce module est en grande partie inspiré de *The Worst Day of Your Life*, une présentation de Christophe Pettus au FOSDEM 2014[45]

8.1.1 AU MENU

- Anticiper les désastres
- Réagir aux désastres
- Rechercher l'origine du problème
- Outils utiles
- Cas type de désastres

8.2 ANTICIPER LES DÉSASTRES

- Un désastre peut toujours survenir
- Il faut savoir le détecter le plus tôt possible
  - et s'être préparé à y répondre

Il est impossible de parer à tous les cas de désastres imaginables.

Le matériel peut subir des pannes, une faille logicielle non connue peut être exploitée, une modification d'infrastructure ou de configuration peut avoir des conséquences imprévues à long terme, une erreur humaine est toujours possible.

Les principes de base de la haute disponibilité (redondance, surveillance…) permettent de mitiger le problème, mais jamais de l'éliminer complètement.

[45] http://thebuild.com/presentations/worst-day-fosdem-2014.pdf

Il est donc extrêmement important de se préparer au mieux, de procéder à des simulations, de remettre en question chaque brique de l'infrastructure pour être capable de détecter une défaillance et d'y réagir rapidement.

8.2.1 DOCUMENTATION

- Documentation complète et à jour
  - emplacement et fréquence des sauvegardes
  - emplacement des traces
  - procédures et scripts d'exploitation
- Sauvegarder et versionner la documentation

Par nature, les désastres arrivent de façon inattendue.

Il faut donc se préparer à devoir agir en urgence, sans préparation, dans un environnement perturbé et stressant — par exemple, en pleine nuit, la veille d'un jour particulièrement critique pour l'activité de la production.

Un des premiers points d'importance est donc de s'assurer de la présence d'une documentation claire, précise et à jour, afin de minimiser le risque d'erreurs humaines.

Cette documentation devrait détailler l'architecture dans son ensemble, et particulièrement la politique de sauvegarde choisie, l'emplacement de celles-ci, les procédures de restauration et éventuellement de bascule vers un environnement de secours.

Les procédures d'exploitation doivent y être expliquées, de façon détaillée mais claire, afin qu'il n'y ait pas de doute sur les actions à effectuer une fois la cause du problème identifié.

La méthode d'accès aux informations utiles (traces de l'instance, du système, supervision…) devrait également être soigneusement documentée afin que le diagnostic du problème soit aussi simple que possible.

Toutes ces informations doivent être organisées de façon claire, afin qu'elles soient immédiatement accessibles et exploitables aux intervenants lors d'un problème.

Il est évidemment tout aussi important de penser à versionner et sauvegarder cette documentation, afin que celle-ci soit toujours accessible même en cas de désastre majeur (perte d'un site).

8.2.2 PROCÉDURES ET SCRIPTS

- Procédures détaillées de restauration / PRA
  - préparer des scripts / utiliser des outils
  - minimiser le nombre d'actions manuelles
- Tester les procédures régulièrement
  - bases de test, développement...
  - s'assurer que chacun les maîtrise
- Sauvegarder et versionner les scripts

La gestion d'un désastre est une situation particulièrement stressante, le risque d'erreur humaine est donc accru.

Un DBA devant restaurer d'urgence l'instance de production en pleine nuit courera plus de risques de faire une fausse manipulation s'il doit taper une vingtaine de commandes en suivant une procédure dans une autre fenêtre (voire un autre poste) que s'il n'a qu'un script à exécuter.

En conséquence, il est important de minimiser le nombre d'actions manuelles à effectuer dans les procédures, en privilégiant l'usage de scripts d'exploitation ou d'outils dédiés (comme pitrery, pgBackRest ou barman pour restaurer une instance PostgreSQL).

Néanmoins, même cette pratique ne suffit pas à exclure tout risque.

L'utilisation de ces scripts ou de ces outils doit également être comprise, correctement documentée, et les procédures régulièrement testées. Le test idéal consiste à remonter fréquemment des environnements de développement et de test ; vos développeurs vous en seront d'ailleurs reconnaissants.

Dans le cas contraire, l'utilisation d'un script ou d'un outil peut aggraver le problème, parfois de façon dramatique — par exemple, l'écrasement d'un environnement sain lors d'une restauration parce que la procédure ne mentionne pas que le script doit être lancé depuis un serveur particulier.

L'aspect le plus important est de s'assurer par des tests réguliers **et manuels** que les procédures sont à jour, n'ont pas de comportement inattendu, et sont maîtrisées par toute l'équipe d'exploitation.

Tout comme pour la documentation, les scripts d'exploitation doivent également être sauvegardés et versionnés.

8.2.3 SUPERVISION ET HISTORISATION

- Tout doit être supervisé
  - réseau, matériel, système, logiciels…
  - les niveaux d'alerte doivent être significatifs
- Les métriques importantes doivent être historisées
  - cela permet de retrouver le moment où le problème est apparu
  - quand cela a un sens, faire des graphes

La supervision est un sujet vaste, qui touche plus au domaine de la haute disponibilité.

Un désastre sera d'autant plus difficile à gérer qu'il est détecté tard. La supervision en place doit donc être pensée pour détecter tout type de défaillance (penser également à superviser la supervision !).

Attention à bien calibrer les niveaux d'alerte, la présence de trop de messages augmente le risque que l'un d'eux passe inaperçu, et donc que l'incident ne soit détecté que tardivement.

Pour aider la phase de diagnostic de l'origine du problème, il faut prévoir d'historiser un maximum d'informations.

La présentation de celles-ci est également importante : il est plus facile de distinguer un pic brutal du nombre de connexions sur un graphique que dans un fichier de traces de plusieurs Go !

8.2.4 AUTOMATISATION

- Des outils existent
  - PAF (Pacemaker), patroni, repmgr…
- Automatiser une bascule est complexe
  - cela peut mener à davantage d'incidents
  - voire à des désastres (*split brain*)

Si on poursuit jusqu'au bout le raisonnement précédent sur le risque à faire effectuer de nombreuses opérations manuelles lors d'un incident, la conclusion logique est que la solution idéale serait de les éliminer complètement, et d'automatiser complètement le déclenchement et l'exécution de la procédure.

Un problème est que toute solution visant à automatiser une tâche se base sur un nombre limité de paramètres et sur une vision restreinte de l'architecture.

De plus, il est difficile à un outil de bascule automatique de diagnostiquer correctement certains types d'incident, par exemple une partition réseau. L'outil peut donc détecter à tort à un incident, surtout s'il est réglé de façon à être assez sensible, et ainsi provoquer lui-même une coupure de service inutile.

Dans le pire des cas, l'outil peut être amené à prendre une mauvaise décision amenant à une situation de désastre, comme un *split brain* (deux instances PostgreSQL se retrouvent ouvertes en écriture en même temps sur les mêmes données).

Il est donc fortement préférable de laisser un administrateur prendre les décisions potentiellement dangereuses, comme une bascule ou une restauration.

8.3 RÉAGIR AUX DÉSASTRES

- Savoir identifier un problème majeur
- Bons réflexes
- Mauvais réflexes

En dépit de toutes les précautions que l'on peut être amené à prendre, rien ne peut garantir qu'aucun problème ne surviendra.

Il faut donc être capable d'identifier le problème lorsqu'il survient, et être prêt à y répondre.

8.3.1 SYMPTÔMES D'UN DÉSASTRE

- Crash de l'instance
- Résultats de requêtes erronnés
- Messages d'erreurs dans les traces
- Dégradation importante des temps d'exécution
- Processus manquants
  - ou en court d'exécution depuis trop longtemps

De très nombreux éléments peuvent aider à identifier que l'on est en situation d'incident grave.

Le plus flagrant est évidemment le crash complet de l'instance PostgreSQL, ou du serveur l'hébergeant, et l'impossibilité pour PostgreSQL de redémarrer.

Les désastres les plus importants ne sont toutefois pas toujours aussi simples à détecter.

Les crash peuvent se produire uniquement de façon ponctuelle, et il existe des cas où l'instance redémarre immédiatement après (typiquement suite au `kill -9` d'un processus backend PostgreSQL).

Cas encore plus délicat, il peut également arriver que les résultats de requêtes soient erronés (par exemple en cas de corruption de fichiers d'index) sans qu'aucune erreur n'apparaisse.

Les symptômes classiques permettant de détecter un problème majeur sont :

- la présence de messages d'erreurs dans les traces de PostgreSQL (notamment des messages `PANIC` ou `FATAL`, mais les messages `ERROR` et `WARNING` sont également très significatifs, particulièrement s'ils apparaissent soudainement en très grand nombre) ;
- la présence de messages d'erreurs dans les traces du système d'exploitation (notamment concernant la mémoire ou le système de stockage) ;
- le constat d'une dégradation importante des temps d'exécution des requêtes sur l'instance ;
- l'absence de certains processus critiques de PostgreSQL ;
- la présence de processus présents depuis une durée inhabituelle (plusieurs semaines, mois…).

8.3.2 BONS RÉFLEXES 1

- Garder la tête froide
- Répartir les tâches clairement
- Minimiser les canaux de communication
- Garder des notes de chaque action entreprise

Une fois que l'incident est repéré, il est important de ne pas foncer tête baissée dans des manipulations.

Il faut bien sûr prendre en considération la criticité du problème, notamment pour définir la priorité des actions (par exemple, en cas de perte totale d'un site, quelles sont les applications à basculer en priorité ?), mais quelle que soit la criticité ou l'impact, il ne faut jamais effectuer une action sans en avoir parfaitement saisi l'impact et s'être assuré qu'elle répondait bien au problème rencontré.

Si le travail s'effectue en équipe, il faut bien faire attention à répartir les tâches clairement, afin d'éviter des manipulations concurrentes ou des oublis qui pourraient aggraver la situation.

Il faut également éviter de multiplier les canaux de communication, cela risque de favoriser la perte d'information, ce qui est critique dans une situation de crise.

Surtout, une règle majeure est de prendre le temps de noter systématiquement toutes les actions entreprises.

Les commandes passées, les options utilisées, l'heure d'exécution, toutes ces informations sont très importantes, déjà pour pouvoir agir efficacement en cas de fausse manipulation, mais également pour documenter la gestion de l'incident après coup, et ainsi en conserver une trace qui sera précieuse si celui-ci venait à se reproduire.

8.3.3 BONS RÉFLEXES 2

- Se prémunir contre une aggravation du problème
  - couper les accès applicatifs
- Si une corruption est suspectée
  - arrêter immédiatement l'instance
  - faire une sauvegarde immédiate des fichiers
  - travailler sur une copie

S'il y a suspicion de potentielle corruption de données, il est primordial de s'assurer au plus vite de couper tous les accès applicatifs vers l'instance afin de ne pas aggraver la situation.

Il est généralement préférable d'avoir une coupure de service plutôt qu'un grand volume de données irrécupérables.

Ensuite, il faut impérativement faire une sauvegarde complète de l'instance avant de procéder à toute manipulation. En fonction de la nature du problème rencontré, le type de sauvegarde pouvant être effectué peut varier (un export de données ne sera possible que si l'instance est démarrée et que les fichiers sont lisibles par exemple). En cas de doute, la sauvegarde la plus fiable qu'il est possible d'effectuer est une copie des fichiers à froid (instance arrêtée) - toute autre action (y compris un export de données) pourrait avoir des conséquences indésirables.

Si des manipulations doivent être tentées pour tenter de récupérer des données, il faut impérativement travailler sur une copie de l'instance, restaurée à partir de cette sauvegarde. Ne jamais travailler directement sur une instance de production corrompue, la moindre action (même en lecture) pourrait aggraver le problème !

Pour plus d'information, voir sur le wiki PostgreSQL[46] .

[46] https://wiki.postgresql.org/wiki/Corruption

8.3.4 BONS RÉFLEXES 3

- Déterminer le moment de démarrage du désastre
- Adopter une vision générale plutôt que focalisée sur un détail
- Remettre en cause chaque élément de l'architecture
  - aussi stable (et/ou coûteux/complexe) soit-il
- Éliminer en priorité les causes possibles côté hardware, système
- Isoler le comportement précis du problème
  - identifier les requêtes / tables / index impliqués

La première chose à identifier est l'instant précis où le problème a commencé à se manifester. Cette information est en effet déterminante pour identifier la cause du problème, et le résoudre — notamment pour savoir à quel instant il faut restaurer l'instance si cela est nécessaire.

Il convient pour cela d'utiliser les outils de supervision et de traces (système, applicatif et PostgreSQL) pour remonter au moment d'apparition des premiers symptômes. Attention toutefois à ne pas confondre les symptômes avec le problème lui-même ! Les symptômes les plus visibles ne sont pas forcément apparus les premiers. Par exemple, la charge sur la machine est un symptôme, mais n'est jamais la cause du problème. Elle est liée à d'autres phénomènes, comme des problèmes avec les disques ou un grand nombre de connexions, qui peuvent avoir commencé à se manifester bien avant que la charge ne commence réellement à augmenter.

Si la nature du problème n'est pas évidente à ce stade, il faut examiner l'ensemble de l'architecture en cause, sans en exclure d'office certains composants (baie de stockage, progiciel…), quels que soient leur complexité / coût / stabilité supposés. Si le comportement observé côté PostgreSQL est difficile à expliquer (crashs plus ou moins aléatoires, nombreux messages d'erreur sans lien apparent…), il est préférable de commencer par s'assurer qu'il n'y a pas un problème de plus grande ampleur (système de stockage, virtualisation, réseau, système d'exploitation).

Un bon indicateur consiste à regarder si d'autres instances / applications / processus rencontrent des problèmes similaires.

Ensuite, une fois que l'ampleur du problème a été cernée, il faut procéder méthodiquement pour en déterminer la cause et les éléments affectés.

Pour cela, les informations les plus utiles se trouvent dans les traces, généralement de PostgreSQL ou du système, qui vont permettre d'identifier précisément les éventuels fichiers ou relations corrompus.

8.3.5 BONS RÉFLEXES 4

- En cas de défaillance matérielle
  - s'assurer de corriger sur du hardware sain et non affecté !
  - baies partagées...

Cette recommandation peut paraître aller de soi, mais si les problèmes sont provoqués par une défaillance matérielle, il est impératif de s'assurer que le travail de correction soit effectué sur un environnement non affecté.

Cela peut s'avérer problématique dans le cadre d'architecture mutualisant les ressources, comme des environnements virtualisés ou utilisant une baie de stockage.

Prendre également la précaution de vérifier que l'intégrité des sauvegardes n'est pas affectée par le problème.

8.3.6 BONS RÉFLEXES 5

- Communiquer, ne pas rester isolé
- Demander de l'aide si le problème est trop complexe
  - autres équipes
  - support
  - forums
  - listes

La communication est très importante dans la gestion d'un désastre.

Il est préférable de minimiser le nombre de canaux de communication plutôt que de les multiplier (téléphone, e-mail, chat, ticket...), ce qui pourrait amener à une perte d'informations et à des délais indésirables.

Il est primordial de rapidement cerner l'ampleur du problème, et pour cela il est généralement nécessaire de demander l'expertise d'autres administrateurs / équipes (applicatif, système, réseau, virtualisation, SAN...). Il ne faut pas rester isolé et risquer que la vision étroite que l'on a des symptômes (notamment en terme de supervision / accès aux traces) empêche l'identification de la nature réelle du problème.

Si la situation semble échapper à tout contrôle, et dépasser les compétences de l'équipe en cours d'intervention, il faut chercher de l'aide auprès de personnes compétentes, par exemple auprès d'autres équipes, du support.

En aucun cas, il ne faut se mettre à suivre des recommandations glanées sur Internet, qui ne se rapporteraient que très approximativement au problème rencontré, voire pas du tout. Si nécessaire, on trouve en ligne des forums et des listes de discussions spécialisées sur lesquels il est également possible d'obtenir des conseils — il est néanmoins indispensable de prendre en compte que les personnes intervenant sur ces médias le font de manière bénévole. Il est déraisonnable de s'attendre à une réaction immédiate, aussi urgent le problème soit-il, et les suggestions effectuées le sont sans aucune garantie.

8.3.7 BONS RÉFLEXES 6

- Dérouler les procédures comme prévu
- En cas de situation non prévue, s'arrêter pour faire le point
  - ne pas hésiter à remettre en cause l'analyse
  - ou la procédure elle-même

Dans l'idéal, des procédures détaillant les actions à effectuer ont été écrites pour le cas de figure rencontré. Dans ce cas, une fois que l'on s'est assuré d'avoir identifié la procédure appropriée, il faut la dérouler méthodiquement, point par point, et valider à chaque étape que tout se déroule comme prévu.

Si une étape de la procédure ne se passe pas comme prévu, il ne faut pas tenter de poursuivre tout de même son exécution sans avoir compris ce qui s'est passé et les conséquences. Cela pourrait être dangereux.

Il faut au contraire prendre le temps de comprendre le problème en procédant comme décrit précédemment, quitte à remettre en cause toute l'analyse menée auparavant, et la procédure ou les scripts utilisés.

C'est également pour parer à ce type de cas de figure qu'il est important de travailler sur une copie et non sur l'environnement de production directement.

8.3.8 BONS RÉFLEXES 7

- En cas de bug avéré
  - tenter de le cerner et de le reproduire au mieux
  - le signaler à la communauté de préférence (configuration, comment reproduire)

Ce n'est heureusement pas fréquent, mais il est possible que l'origine du problème soit liée à un bug de PostgreSQL lui-même.

Dans ce cas, la méthodologie appropriée consiste à essayer de reproduire le problème le plus fidèlement possible et de façon systématique, pour le cerner au mieux.

Il est ensuite très important de le signaler au plus vite à la communauté, généralement sur la liste pgsql-bugs@postgresql.org (cela nécessite une inscription préalable), en respectant les règles définies dans la documentation[47] .

Notamment (liste non exhaustive) :

- indiquer la version précise de PostgreSQL installée, et la méthode d'installation utilisée ;
- préciser la plate-forme utilisée, notamment la version du système d'exploitation utilisé et la configuration des ressources du serveur ;
- signaler uniquement les faits observés, éviter les spéculations sur l'origine du problème ;
- joindre le détail des messages d'erreurs observés (augmenter la verbosité des erreurs avec le paramètre `log_error_verbosity`) ;
- joindre un cas complet permettant de reproduire le problème de façon aussi simple que possible.

Pour les problèmes relevant du domaine de la sécurité (découverte d'une faille), la liste adéquate est security@postgresql.org.

---

[47] https://www.postgresql.org/docs/current/static/bug-reporting.html

8.3.9 BONS RÉFLEXES 8

- Après correction
- Tester complètement l'intégrité des données
  - pour détecter tous les problèmes
  - pour valider après restauration / correction
  - `pg_dumpall > /dev/null` ou `pg_basebackup`
- Reconstruction dans une autre instance
  - `pg_dumpall | psql -h autre serveur`

Une fois les actions correctives réalisées (restauration, recréation d'objets, mise à jour des données...), il faut tester intensivement pour s'assurer que le problème est bien complètement résolu.

Il est donc extrêmement important d'avoir préparé des cas de tests permettant de reproduire le problème de façon certaine, afin de valider la solution appliquée.

En cas de suspicion de corruption de données, il est également important de tenter de procéder à la lecture de la totalité des données depuis PostgreSQL.

Un premier outil pour cela est une sauvegarde avec `pg_basebackup` (voir plus loin).

Alternativement, la commande suivante, exécutée avec l'utilisateur système propriétaire de l'instance (généralement `postgres`) effectue une lecture complète de toutes les tables (mais sans les index ni les vues matérialisées), sans nécessiter de place sur disque supplémentaire :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell, la commande est :

```
PS C:\ pg_dumpall > $null
```

Cette commande ne devrait renvoyer aucune erreur. En cas de problème, notamment une somme de contrôle qui échoue, une erreur apparaîtra :

```
pg_dump: WARNING:  page verification failed, calculated checksum 20565 but expected 17796
pg_dump: erreur : Sauvegarde du contenu de la table « corrompue » échouée :
                  échec de PQgetResult().
pg_dump: erreur : Message d'erreur du serveur :
         ERROR:  invalid page in block 0 of relation base/104818/104828
pg_dump: erreur : La commande était : COPY public.corrompue (i) TO stdout;
pg_dumpall: erreur : échec de pg_dump sur la base de données « corruption », quitte
```

Même si la lecture des données par `pg_dumpall` ou `pg_dump` ne renvoie aucune erreur, il est toujours possible que des problèmes subsistent, par exemple des corruptions silencieuses,

des index incohérents avec les données…

Dans les situations les plus extrêmes (problème de stockage, fichiers corrompus), il est important de tester la validité des données dans une nouvelle instance en effectuant un export/import complet des données.

Par exemple, initialiser une nouvelle instance avec `initdb`, sur un autre système de stockage, voire sur un autre serveur, puis lancer la commande suivante (l'application doit être coupée, ce qui est normalement le cas depuis la détection de l'incident si les conseils précédents ont été suivis) pour exporter et importer à la volée :

```
$ pg_dumpall -h <serveur_corrompu> -U postgres | psql -h <nouveau_serveur> \
                                                    -U postgres postgres
$ vacuumdb --analyze -h <nouveau_serveur> -U postgres postgres
```

D'éventuels problèmes peuvent être détectés lors de l'import des données, par exemple si des corruptions entraînent l'échec de la reconstruction de clés étrangères. Il faut alors procéder au cas par cas.

Enfin, même si cette étape s'est déroulée sans erreur, tout risque n'est pas écarté, il reste la possibilité de corruption de données silencieuses. Sauf si la fonctionnalité de checksum de PostgreSQL a été activée sur l'instance (ce n'est pas activé par défaut !), le seul moyen de détecter ce type de problème est de valider les données fonctionnellement.

Dans tous les cas, en cas de suspicion de corruption de données en profondeur, il est fortement préférable d'accepter une perte de données et de restaurer une sauvegarde d'avant le début de l'incident, plutôt que de continuer à travailler avec des données dont l'intégrité n'est pas assurée.

8.3.10 MAUVAIS RÉFLEXES 1

- Paniquer
- Prendre une décision hâtive
  - exemple, supprimer des fichiers du répertoire `pg_wal`
- Lancer une commande sans la comprendre
  - exemple, `pg_resetwal` ou celles de l'extension `pg_surgery`

Quelle que soit la criticité du problème rencontré, la panique peut en faire quelque chose de pire.

Il faut impérativement garder son calme, et résister au mieux au stress et aux pressions qu'une situation de désastre ne manque pas de provoquer.

Il est également préférable d'éviter de sauter immédiatement à la conclusion la plus évidente. Il ne faut pas hésiter à retirer les mains du clavier pour prendre de la distance par rapport aux conséquences du problème, réfléchir aux causes possibles, prendre le temps d'aller chercher de l'information pour réévaluer l'ampleur réelle du problème.

La plus mauvaise décision que l'on peut être amenée à prendre lors de la gestion d'un incident est celle que l'on prend dans la précipitation, sans avoir bien réfléchi et mesuré son impact. Cela peut provoquer des dégâts irrécupérables, et transformer une situation d'incident en situation de crise majeure.

Un exemple classique de ce type de comportement est le cas où PostgreSQL est arrêté suite au remplissage du système de fichiers contenant les fichiers WAL, `pg_wal`.

Le réflexe immédiat d'un administrateur non averti pourrait être de supprimer les plus vieux fichiers dans ce répertoire, ce qui répond bien aux symptômes observés mais reste une erreur dramatique qui va rendre le démarrage de l'instance impossible.

Quoi qu'il arrive, ne jamais exécuter une commande sans être certain qu'elle correspond bien à la situation rencontrée, et sans en maîtriser complètement les impacts. Même si cette commande provient d'un document mentionnant les mêmes messages d'erreur que ceux rencontrés (et tout particulièrement si le document a été trouvé via une recherche hâtive sur Internet) !

Là encore, nous disposons comme exemple d'une erreur malheureusement fréquente, l'exécution de la commande `pg_resetwal` sur une instance rencontrant un problème. Comme l'indique la documentation, « *[cette commande] ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption* » et « *il ne faut pas perdre de vue que la base de données peut contenir des données incohérentes du fait de transactions partiellement validées* » (documentation[48]). Nous reviendrons ultérieurement sur les (rares) cas d'usage réels de cette commande, mais dans l'immense majorité des cas, l'utiliser va aggraver le problème, en ajoutant des problématiques de corruption logique des données !

Il convient donc de bien s'assurer de comprendre les conséquences de l'exécution de chaque action effectuée.

---

[48] https://docs.postgresql.fr/current/app-pgresetwal.html

8.3.11 MAUVAIS RÉFLEXES 2

- Arrêter le diagnostic quand les symptômes disparaissent
- Ne pas pousser l'analyse jusqu'au bout

Il est important de pousser la réflexion jusqu'à avoir complètement compris l'origine du problème et ses conséquences.

En premier lieu, même si les symptômes semblent avoir disparus, il est tout à fait possible que le problème soit toujours sous-jacent, ou qu'il ait eu des conséquences moins visibles mais tout aussi graves (par exemple, une corruption logique de données).

Ensuite, même si le problème est effectivement corrigé, prendre le temps de comprendre et de documenter l'origine du problème (rapport « post-mortem ») a une valeur inestimable pour prendre les mesures afin d'éviter que le problème ne se reproduise, et retrouver rapidement les informations utiles s'il venait à se reproduire malgré tout.

8.3.12 MAUVAIS RÉFLEXES 3

- Ne pas documenter
  - le résultat de l'investigation
  - les actions effectuées

Après s'être assuré d'avoir bien compris le problème rencontré, il est tout aussi important de le documenter soigneusement, avec les actions de diagnostic et de correction effectuées.

Ne pas le faire, c'est perdre une excellente occasion de gagner un temps précieux si le problème venait à se reproduire.

C'est également un risque supplémentaire dans le cas où les actions correctives menées n'auraient pas suffi à complètement corriger le problème ou auraient eu un effet de bord inattendu.

Dans ce cas, avoir pris le temps de noter le détail des actions effectuées fera là encore gagner un temps précieux.

8.4 RECHERCHER L'ORIGINE DU PROBLÈME

- Quelques pistes de recherche pour cerner le problème
- Liste non exhaustive

Les problèmes pouvant survenir sont trop nombreux pour pouvoir tous les lister, chaque élément matériel ou logiciel d'une architecture pouvant subir de nombreux types de défaillances.

Cette section liste quelques pistes classiques d'investigation à ne pas négliger pour s'efforcer de cerner au mieux l'étendue du problème, et en déterminer les conséquences.

8.4.1 PRÉREQUIS

- Avant de commencer à creuser
  - référencer les symptômes
  - identifier au mieux l'instant de démarrage du problème

La première étape est de déterminer aussi précisément que possible les symptômes observés, sans en négliger, et à partir de quel moment ils sont apparus.

Cela donne des informations précieuses sur l'étendue du problème, et permet d'éviter de se focaliser sur un symptôme particulier, parce que plus visible (par exemple l'arrêt brutal de l'instance), alors que la cause réelle est plus ancienne (par exemple des erreurs IO dans les traces système, ou une montée progressive de la charge sur le serveur).

8.4.2 RECHERCHE D'HISTORIQUE

- Ces symptômes ont-ils déjà été rencontrés dans le passé ?
- Ces symptômes ont-ils déjà été rencontrés par d'autres ?
- Attention à ne pas prendre les informations trouvées pour argent comptant !

Une fois les principaux symptômes identifiés, il est utile de prendre un moment pour déterminer si ce problème est déjà connu.

Notamment, identifier dans la base de connaissances si ces symptômes ont déjà été rencontrés dans le passé (d'où l'importance de bien documenter les problèmes).

Au-delà de la documentation interne, il est également possible de rechercher si ces symptômes ont déjà été rencontrés par d'autres.

Pour ce type de recherche, il est préférable de privilégier les sources fiables (documentation officielle, listes de discussion, plate-forme de support...) plutôt qu'un quelconque document d'un auteur non identifié.

Dans tous les cas, il faut faire très attention à ne pas prendre les informations trouvées pour argent comptant, et ce même si elles proviennent de la documentation interne ou d'une source fiable !

Il est toujours possible que les symptômes soient similaires mais que la cause soit différente. Il s'agit donc ici de mettre en place une base de travail, qui doit être complétée par une observation directe et une analyse.

8.4.3 MATÉRIEL

- Vérifier le système disque (SAN, carte RAID, disques)
- Un `fsync` est-il bien honoré de l'OS au disque ? (batteries !)
- Rechercher toute erreur matérielle
- Firmwares pas à jour
  - ou récemment mis à jour
- Matériel récemment changé

Les défaillances du matériel, et notamment du système de stockage, sont de celles qui peuvent avoir les impacts les plus importants et les plus étendus sur une instance et sur les données qu'elle contient.

Ce type de problème peut également être difficile à diagnostiquer en se contentant d'observer les symptômes les plus visibles. Il est facile de sous-estimer l'ampleur des dégâts.

Parmi les bonnes pratiques, il convient de vérifier la configuration et l'état du système disque (SAN, carte RAID, disques).

Quelques éléments étant une source habituelle de problèmes :

- le système disque n'honore pas les ordres `fsync` ? (SAN ? virtualisation ?) ;
- quel est l'état de la batterie du cache en écriture ?

Il faut évidemment rechercher la présence de toute erreur matérielle, au niveau des disques, de la mémoire, des CPU...

Vérifier également la version des firmwares installés. Il est possible qu'une nouvelle version corrige le problème rencontré, ou à l'inverse que le déploiement d'une nouvelle version soit à l'origine du problème.

Dans le même esprit, il faut vérifier si du matériel a récemment été changé. Il arrive que de nouveaux éléments soient défaillants.

Il convient de noter que l'investigation à ce niveau peut être grandement complexifiée par l'utilisation de certaines technologies (virtualisation, baies de stockage), du fait de la mutualisation des ressources, et de la séparation des compétences et des informations de supervision entre différentes équipes.

8.4.4 VIRTUALISATION

- Mutualisation excessive
- Configuration du stockage virtualisé
- Rechercher les erreurs aussi niveau superviseur
- Mises à jour non appliquées
  - ou appliquées récemment
- Modifications de configuration récentes

Tout comme pour les problèmes au niveau du matériel, les problèmes au niveau du système de virtualisation peuvent être complexes à détecter et à diagnostiquer correctement.

Le principal facteur de problème avec la virtualisation est lié à une mutualisation excessive des ressources.

Il est ainsi possible d'avoir un total de ressources allouées aux VM supérieur à celles disponibles sur l'hyperviseur, ce qui amène à des comportements de fort ralentissement, voire de blocage des systèmes virtualisés.

Si ce type d'architecture est couplé à un système de gestion de bascule automatique (Pacemaker, repmgr…), il est possible d'avoir des situations de bascules impromptues, voire des situations de *split brain*, qui peuvent provoquer des pertes de données importantes. Il est donc important de prêter une attention particulière à l'utilisation des ressources de l'hyperviseur, et d'éviter à tout prix la sur-allocation.

Par ailleurs, lorsque l'architecture inclut une brique de virtualisation, il est important de prendre en compte que certains problèmes ne peuvent être observés qu'à partir de l'hyperviseur, et pas à partir du système virtualisé. Par exemple, les erreurs matérielles ou système risquent d'être invisibles depuis une VM, il convient donc d'être vigilant, et de rechercher toute erreur sur l'hôte.

Il faut également vérifier si des modifications ont été effectuées peu avant l'incident, comme des modifications de configuration ou l'application de mises à jour.

Comme indiqué dans la partie traitant du matériel, l'investigation peut être grandement freinée par la séparation des compétences et des informations de supervision entre différentes équipes. Une bonne communication est alors la clé de la résolution rapide du problème.

8.4.5 SYSTÈME D'EXPLOITATION 1

- Erreurs dans les traces
- Mises à jour système non appliquées
- Modifications de configuration récentes

Après avoir vérifié les couches matérielles et la virtualisation, il faut ensuite s'assurer de l'intégrité du système d'exploitation.

La première des vérifications à effectuer est de consulter les traces du système pour en extraire les éventuels messages d'erreur :

- sous Linux, on trouvera ce type d'informations en sortie de la commande `dmesg`, et dans les fichiers traces du système, généralement situés sous `/var/log` ;
- sous Windows, on consultera à cet effet le journal des événements (les `event logs`).

Tout comme pour les autres briques, il faut également voir s'il existe des mises à jour des paquets qui n'auraient pas été appliquées, ou à l'inverse si des mises à jour, installations ou modifications de configuration ont été effectuées récemment.

8.4.6 SYSTÈME D'EXPLOITATION 2

- Opération d'IO impossible
  - FS plein ?
  - FS monté en lecture seule ?
- Tester l'écriture sur PGDATA
- Tester la lecture sur PGDATA

Parmi les problèmes fréquemment rencontrés se trouve l'impossibilité pour PostgreSQL d'accéder en lecture ou en écriture à un ou plusieurs fichiers.

La première chose à vérifier est de déterminer si le système de fichiers sous-jacent ne serait pas rempli à 100% (commande `df` sous Linux) ou monté en lecture seule (commande `mount` sous Linux).

On peut aussi tester les opérations d'écriture et de lecture sur le système de fichiers pour déterminer si le comportement y est global :

- pour tester une écriture dans le répertoire `PGDATA`, sous Linux :

```
$ touch $PGDATA/test_write
```

- pour tester une lecture dans le répertoire `PGDATA`, sous Linux :

```
$ cat $PGDATA/PGVERSION
```

Pour identifier précisément les fichiers présentant des problèmes, il est possible de tester la lecture complète des fichiers dans le point de montage :

```
$ tar cvf /dev/null $PGDATA
```

8.4.7 SYSTÈME D'EXPLOITATION 3

- Consommation excessive des ressources
  - OOM killer (overcommit !)
- Après un crash, vérifier les processus actifs
  - ne pas tenter de redémarrer si des processus persistent
- Outils : `sar`, `atop`...

Sous Linux, l'installation d'outils d'aide au diagnostic sur les serveurs est très important pour mener une analyse efficace, particulièrement le paquet `systat` qui permet d'utiliser la commande `sar`.

La lecture des traces système et des traces PostgreSQL permettent également d'avancer dans le diagnostic.

Un problème de consommation excessive des ressources peut généralement être anticipée grâce à une supervision sur l'utilisation des ressources et des seuils d'alerte appropriés. Il arrive néanmoins parfois que la consommation soit très rapide et qu'il ne soit pas possible de réagir suffisamment rapidement.

Dans le cas d'une consommation mémoire d'un serveur Linux qui menacerait de dépasser la quantité totale de mémoire allouable, le comportement par défaut de Linux est d'autoriser par défaut la tentative d'allocation.

Si l'allocation dépasse effectivement la mémoire disponible, alors le système va déclencher un processus *Out Of Memory Killer* (OOM Killer) qui va se charger de tuer les processus les plus consommateurs.

Dans le cas d'un serveur dédié à une instance PostgreSQL, il y a de grandes chances que le processus en question appartienne à l'instance.

S'il s'agit d'un *OOM Killer* effectuant un arrêt brutal (`kill -9`) sur un backend, l'instance PostgreSQL va arrêter immédiatement tous les processus afin de prévenir une corruption de la mémoire et les redémarrer.

S'il s'agit du processus principal de l'instance (*postmaster*), les conséquences peuvent être bien plus dramatiques, surtout si une tentative est faite de redémarrer l'instance sans vérifier si des processus actifs existent encore.

Pour un serveur dédié à PostgreSQL, la recommendation est habituellement de désactiver la sur-allocation de la mémoire, empêchant ainsi le déclenchement de ce phénomène.

Voir pour cela les paramètres kernel `vm.overcommit_memory` et `vm.overcommit_ratio` (référence : https://kb.dalibo.com/overcommit_memory).

8.4.8 POSTGRESQL

- Relever les erreurs dans les traces
  - ou messages inhabituels
- Vérifier les mises à jour mineures

Tout comme pour l'analyse autour du système d'exploitation, la première chose à faire est rechercher toute erreur ou message inhabituel dans les traces de l'instance. Ces messages sont habituellement assez informatifs, et permettent de cerner la nature du problème. Par exemple, si PostgreSQL ne parvient pas à écrire dans un fichier, il indiquera précisément de quel fichier il s'agit.

Si l'instance est arrêtée suite à un crash, et que les tentatives de redémarrage échouent avant qu'un message puisse être écrit dans les traces, il est possible de tenter de démarrer l'instance en exécutant directement le binaire `postgres` afin que les premiers messages soient envoyés vers la sortie standard.

Il convient également de vérifier si des mises à jour qui n'auraient pas été appliquées ne corrigeraient pas un problème similaire à celui rencontré.

Identifier les mises à jours appliquées récemment et les modifications de configuration peut également aider à comprendre la nature du problème.

8.4.9 PARAMÉTRAGE DE POSTGRESQL 1

- La désactivation de certains paramètres est dangereuse
  - `fsync`
  - `full_page_write`

Si des corruptions de données sont relevées suite à un crash de l'instance, il convient particulièrement de vérifier la valeur du paramètre `fsync`.

En effet, si celui-ci est désactivé, les écritures dans les journaux de transactions ne sont pas effectuées de façon synchrone, ce qui implique que l'ordre des écritures ne sera pas conservé en cas de crash. Le processus de *recovery* de PostgreSQL risque alors de provoquer des corruptions si l'instance est malgré tout redémarrée.

Ce paramètre ne devrait jamais être positionné à une autre valeur que `on`, sauf dans des cas extrêmement particuliers (en bref, si l'on peut se permettre de restaurer intégralement les données en cas de crash, par exemple dans un chargement de données initial).

Le paramètre `full_page_write` indique à PostgreSQL d'effectuer une écriture complète d'une page chaque fois qu'elle reçoit une nouvelle écriture après un checkpoint, pour éviter un éventuel mélange entre des anciennes et nouvelles données en cas d'écriture partielle.

La désactivation de ce paramètre peut avoir le même type de conséquences que la désactivation de `fsync`.

8.4.10 PARAMÉTRAGE DE POSTGRESQL 2

- Activez les checksums !
  - `initdb --data-checksums`
  - ou `pg_checksums --enable` (v12)
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Vérification lors de `pg_basebackup` (v11)

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Sans mécanisme de sécurité, il est donc possible de modifier un fichier sans que PostgreSQL s'en rende compte, ce qui aboutit à une corruption silencieuse.

Les sommes de contrôles (*checksums*) permettent de se prémunir contre des corruptions silencieuses de données. Leur mise en place est fortement recommandée sur une nouvelle instance. Malheureusement, jusqu'en version 11 comprise, on ne peut le faire qu'à

l'initialisation de l'instance. La version 12 permet de les mettre en place, *base arrêtée*, avec l'utilitaire `pg_checksums`[49] .

À titre d'exemple, créons une instance sans utiliser les *checksums*, et une autre qui les utilisera :

```
$ initdb -D /tmp/sans_checksums/
$ initdb -D /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacun des deux clusters :

```
CREATE TABLE test (name text);
INSERT INTO test (name) VALUES ('toto');
```

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans *checksums* est montré en exemple).

```
SELECT pg_relation_filepath('test');

 pg_relation_filepath
----------------------
 base/12036/16317
```

Instance arrêtée (pour ne pas être gêné par le cache), on va s'attacher à corrompre ce fichier, en remplaçant la valeur « toto » par « goto » avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/12036/16317
$ hexedit /tmp/avec_checksums/base/12036/16399
```

Enfin, on peut ensuite exécuter des requêtes sur ces deux clusters.

Sans *checksums* :

```
TABLE test;

 name
------
 qoto
```

Avec *checksums* :

```
TABLE test;

WARNING:  page verification failed, calculated checksum 16321
          but expected 21348
ERROR:  invalid page in block 0 of relation base/12036/16387
```

Depuis la version 11, les sommes de contrôles, si elles sont là, sont vérifiées par défaut lors d'un `pg_basebackup`. En cas de corruption des données, l'opération sera interrompue.

[49] https://docs.postgresql.fr/current/app-pgchecksums.html

Il est possible de désactiver cette vérification avec l'option `--no-verify-checksums` pour obtenir une copie, aussi corrompue que l'original, mais pouvant servir de base de travail.

En pratique, si vous utilisez PostgreSQL 9.5 au moins et si votre processeur supporte les instructions SSE 4.2 (voir dans `/proc/cpuinfo`), il n'y aura pas d'impact notable en performances. Par contre vous générerez un peu plus de journaux.

8.4.11 ERREUR DE MANIPULATION

- Traces système, traces PostgreSQL
- Revue des dernières manipulations effectuées
- Historique des commandes
- Danger : `kill -9`, `rm -rf`, `rsync`, `find … -exec` ...

L'erreur humaine fait également partie des principales causes de désastre.

Une commande de suppression tapée trop rapidement, un oubli de clause `WHERE` dans une requête de mise à jour, nombreuses sont les opérations qui peuvent provoquer des pertes de données ou un crash de l'instance.

Il convient donc de revoir les dernières opérations effectuées sur le serveur, en commençant par les interventions planifiées, et si possible récupérer l'historique des commandes passées.

Des exemples de commandes particulièrement dangereuses :

- `kill -9`
- `rm -rf`
- `rsync`
- `find` (souvent couplé avec des commandes destructices comme `rm`, `mv`, `gzip`...)

8.5 OUTILS

- Quelques outils peuvent aider
  - à diagnostiquer la nature du problème
  - à valider la correction apportée
  - à appliquer un contournement
- ATTENTION
  - certains de ces outils peuvent corrompre les données !

8.5.1 OUTILS - PG_CONTROLDATA

- Fournit des informations de contrôle sur l'instance
- Ne nécessite pas que l'instance soit démarrée

L'outil `pg_controldata` lit les informations du fichier de contrôle d'une instance PostgreSQL.

Cet outil ne se connecte pas à l'instance, il a juste besoin d'avoir un accès en lecture sur le répertoire `PGDATA` de l'instance.

Les informations qu'il récupère ne sont donc pas du temps réel, il s'agit d'une vision de l'instance telle qu'elle était la dernière fois que le fichier de contrôle a été mis à jour. L'avantage est qu'elle peut être utilisée même si l'instance est arrêtée.

pg_controldata affiche notamment les informations initialisées lors d'initdb, telles que la version du catalogue, ou la taille des blocs, qui peuvent être cruciales si l'on veut restaurer une instance sur un nouveau serveur à partir d'une copie des fichiers.

Il affiche également de nombreuses informations utiles sur le traitement des journaux de transactions et des checkpoints, par exemple :

- positions de l'avant-dernier checkpoint et du dernier checkpoint dans les WAL ;
- nom du WAL correspondant au dernier WAL ;
- timeline sur laquelle se situe le dernier checkpoint ;
- instant précis du dernier checkpoint.

Quelques informations de paramétrage sont également renvoyées, comme la configuration du niveau de WAL, ou le nombre maximal de connexions autorisées.

En complément, le dernier état connu de l'instance est également affiché. Les états potentiels sont :

- `in production` : l'instance est démarrée et est ouverte en écriture ;
- `shut down` : l'instance est arrêtée ;
- `in archive recovery` : l'instance est démarrée et est en mode `recovery` (restauration, `Warm` ou `Hot Standby`) ;
- `shut down in recovery` : l'instance s'est arrêtée alors qu'elle était en mode `recovery` ;
- `shutting down` : état transitoire, l'instance est en cours d'arrêt ;
- `in crash recovery` : état transitoire, l'instance est en cours de démarrage suite à un crash ;
- `starting up` : état transitoire, concrètement jamais utilisé.

Bien entendu, comme ces informations ne sont pas mises à jour en temps réel, elles peuvent être erronées.

Cet asynchronisme est intéressant pour diagnostiquer un problème, par exemple si `pg_controldata` renvoie l'état `in production` mais que l'instance est arrêtée, cela signifie que l'arrêt n'a pas été effectué proprement (*crash* de l'instance, qui sera donc suivi d'un `recovery` au démarrage).

Exemple de sortie de la commande :

```
$ /usr/pgsql-10/bin/pg_controldata /var/lib/pgsql/10/data

pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6451139765284827825
Database cluster state:               in production
pg_control last modified:             Mon 28 Aug 2017 03:40:30 PM CEST
Latest checkpoint location:           1/2B04EC0
Prior checkpoint location:            1/2B04DE8
Latest checkpoint's REDO location:    1/2B04E88
Latest checkpoint's REDO WAL file:    000000010000000100000002
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:1023
Latest checkpoint's NextOID:          41064
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        548
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  1022
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Mon 28 Aug 2017 03:40:30 PM CEST
Fake LSN counter for unlogged rels:   0/1
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
wal_level setting:                    replica
wal_log_hints setting:                off
max_connections setting:              100
max_worker_processes setting:         8
```

```
max_prepared_xacts setting:           0
max_locks_per_xact setting:           64
track_commit_timestamp setting:       off
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Data page checksum version:           0
Mock authentication nonce:            7fb23aca2465c69b2c0f54ccf03e0ece
                                      3c0933c5f0e5f2c096516099c9688173
```

8.5.2 OUTILS - EXPORT/IMPORT DE DONNÉES

- `pg_dump`
- `pg_dumpall`
- `COPY`
- `psql` / `pg_restore`
  - `--section=pre-data` / `data` / `post-data`

Les outils `pg_dump` et `pg_dumpall` permettent d'exporter des données à partir d'une instance démarrée.

Dans le cadre d'un incident grave, il est possible de les utiliser pour :

- extraire le contenu de l'instance ;
- extraire le contenu des bases de données ;
- tester si les données sont lisibles dans un format compréhensible par PostgreSQL.

Par exemple, un moyen rapide de s'assurer que tous les fichiers des tables de l'instance sont lisibles est de forcer leur lecture complète, notamment grâce à la commande suivante :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell :

```
pg_dumpall > $null
```

Attention, les fichiers associés aux index ne sont pas parcourus pendant cette opéra-

tion. Par ailleurs, ne pas avoir d'erreur ne garantit en aucun cas pas l'intégrité fonctionnelle des données : les corruptions peuvent très bien être silencieuses ou concerner les index. Une vérification exhaustive implique d'autres outils comme `pg_checksums` ou `pg_basebackup` (voir plus loin).

Si `pg_dumpall` ou `pg_dump` renvoient des messages d'erreur et ne parviennent pas à exporter certaines tables, il est possible de contourner le problème à l'aide de la commande `COPY`, en sélectionnant exclusivement les données lisibles autour du bloc corrompu.

Il convient ensuite d'utiliser `psql` ou `pg_restore` pour importer les données dans une nouvelle instance, probablement sur un nouveau serveur, dans un environnement non affecté par le problème. Pour parer au cas où le réimport échoue à cause de contraintes non respectées, il est souvent préférable de faire le réimport par étapes :

```
$ pg_restore -1 --section=pre-data --verbose -d cible base.dump
$ pg_restore -1 --section=data --verbose -d cible base.dump
$ pg_restore -1 --section=post-data --exit-on-error --verbose -d cible base.dump
```

En cas de problème, on verra les contraintes posant problème.

Il peut être utile de générer les scripts en pur SQL avant de les appliquer, éventuellement par étape :

```
$ pg_restore --section=post-data -f postdata.sql base.dump
```

Pour rappel, même après un export / import de données réalisé avec succès, des corruptions logiques peuvent encore être présentes. Il faut donc être particulièrement vigilant et prendre le temps de valider l'intégrité fonctionnelle des données.

8.5.3 OUTILS - PAGEINSPECT

- Extension
- Vision du contenu d'un bloc
- Sans le dictionnaire, donc sans décodage des données
- Affichage brut
- Utilisé surtout en debug, ou dans les cas de corruption
- Fonctions de décodage pour les tables, les index (B-tree, hash, GIN, GiST), FSM
- Nécessite de connaître le code de PostgreSQL

Voici quelques exemples.

Contenu d'une page d'une table :

```
SELECT * FROM heap_page_items(get_raw_page('dspam_token_data',0)) LIMIT 2;
```

```
-[ RECORD 1 ]------
lp          | 1
lp_off      | 8152
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,7)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0100000001000000010000000 1000000
-[ RECORD 2 ]------
lp          | 2
lp_off      | 8112
lp_flags    | 1
lp_len      | 40
t_xmin      | 837
t_xmax      | 839
t_field3    | 0
t_ctid      | (0,8)
t_infomask2 | 3
t_infomask  | 1282
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0200000001000000010000000 2000000
```

Et son entête :

```
SELECT * FROM page_header(get_raw_page('dspam_token_data',0));

-[ RECORD 1 ]--------
lsn       | F1A/5A6EAC40
checksum  | 0
flags     | 0
lower     | 56
upper     | 7872
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 839
```

Méta-données d'un index (contenu dans la première page) :

```
SELECT * FROM bt_metap('dspam_token_data_uid_key');

-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 243
level     | 2
fastroot  | 243
fastlevel | 2
```

La page racine est la 243. Allons la voir :

```
SELECT * FROM bt_page_items('dspam_token_data_uid_key',243) LIMIT 10;

 offset |   ctid    | len |nulls|vars|                data
--------+-----------+-----+-----+----+-----------------------------------
      1 | (3,1)     |   8 | f   | f  |
      2 | (44565,1) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (242,1)   |  20 | f   | f  | 77 c6 0d 6f a6 92 db 81 28 00 00 00
      4 | (43569,1) |  20 | f   | f  | 47 a6 aa be 29 e3 13 83 18 00 00 00
      5 | (481,1)   |  20 | f   | f  | 30 17 dd 8e d9 72 7d 84 0a 00 00 00
      6 | (43077,1) |  20 | f   | f  | 5c 3c 7b c5 5b 7a 4e 85 0a 00 00 00
      7 | (719,1)   |  20 | f   | f  | 0d 91 d5 78 a9 72 88 86 26 00 00 00
      8 | (41209,1) |  20 | f   | f  | a7 8a da 17 95 17 cd 87 0a 00 00 00
      9 | (957,1)   |  20 | f   | f  | 78 e9 64 e9 64 a9 52 89 26 00 00 00
     10 | (40849,1) |  20 | f   | f  | 53 11 e9 64 e9 1b c3 8a 26 00 00 00
```

La première entrée de la page 243, correspondant à la donnée `f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00` est stockée dans la page 3 de notre index :

```
SELECT * FROM bt_page_stats('dspam_token_data_uid_key',3);

-[ RECORD 1 ]-+-----
blkno         | 3
type          | i
live_items    | 202
dead_items    | 0
avg_item_size | 19
page_size     | 8192
free_size     | 3312
btpo_prev     | 0
btpo_next     | 44565
btpo          | 1
btpo_flags    | 0

SELECT * FROM bt_page_items('dspam_token_data_uid_key',3) LIMIT 10;

 offset |   ctid    | len |nulls|vars|                data
--------+-----------+-----+-----+----+-----------------------------------
```

```
     1 | (38065,1) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
     2 | (1,1)     |   8 | f   | f  |
     3 | (37361,1) |  20 | f   | f  | 30 fd 30 b8 70 c9 01 80 26 00 00 00
     4 | (2,1)     |  20 | f   | f  | 18 2c 37 36 27 03 03 80 27 00 00 00
     5 | (4,1)     |  20 | f   | f  | 36 61 f3 b6 c5 1b 03 80 0f 00 00 00
     6 | (43997,1) |  20 | f   | f  | 30 4a 32 58 c8 44 03 80 27 00 00 00
     7 | (5,1)     |  20 | f   | f  | 88 fe 97 6f 7e 5a 03 80 27 00 00 00
     8 | (51136,1) |  20 | f   | f  | 74 a8 5a 9b 15 5d 03 80 28 00 00 00
     9 | (6,1)     |  20 | f   | f  | 44 41 3c ee c8 fe 03 80 0a 00 00 00
    10 | (45317,1) |  20 | f   | f  | d4 b0 7c fd 5d 8d 05 80 26 00 00 00
```

Le type de la page est `i`, c'est-à-dire «internal», donc une page interne de l'arbre. Continuons notre descente, allons voir la page 38065 :

```
SELECT * FROM bt_page_stats('dspam_token_data_uid_key',38065);
```

```
-[ RECORD 1 ]-+-----
blkno         | 38065
type          | l
live_items    | 169
dead_items    | 21
avg_item_size | 20
page_size     | 8192
free_size     | 3588
btpo_prev     | 118
btpo_next     | 119
btpo          | 0
btpo_flags    | 65
```

```
SELECT * FROM bt_page_items('dspam_token_data_uid_key',38065) LIMIT 10;
```

```
 offset |    ctid     | len |nulls|vars|                data
--------+-------------+-----+-----+----+-----------------------------------
      1 | (11128,118) |  20 | f   | f  | 33 37 89 95 b9 23 cc 80 0a 00 00 00
      2 | (45713,181) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 0f 00 00 00
      3 | (45424,97)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 26 00 00 00
      4 | (45255,28)  |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 27 00 00 00
      5 | (15672,172) |  20 | f   | f  | f3 4b 2e 8c 39 a3 cb 80 28 00 00 00
      6 | (5456,118)  |  20 | f   | f  | f3 bf 29 a2 39 a3 cb 80 0f 00 00 00
      7 | (8356,206)  |  20 | f   | f  | f3 bf 29 a2 39 a3 cb 80 28 00 00 00
      8 | (33895,272) |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 0a 00 00 00
      9 | (5176,108)  |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 0f 00 00 00
     10 | (5466,41)   |  20 | f   | f  | f3 4b 8e 37 99 a3 cb 80 26 00 00 00
```

Nous avons trouvé une feuille (type `l`). Les ctid pointés sont maintenant les adresses dans la table :

```
SELECT * FROM dspam_token_data WHERE ctid = '(11128,118)';
```

```
 uid |        token         | spam_hits | innocent_hits |  last_hit
-----+----------------------+-----------+---------------+------------
  40 | -6317261189288392210 |         0 |             3 | 2014-11-10
```

8.5.4 OUTILS - PG_RESETWAL

- Efface les WAL courants
- Permet à l'instance de démarrer en cas de corruption d'un WAL
  - comme si elle était dans un état cohérent
  - ...ce qui n'est pas le cas
- **Cet outil est dangereux et mène à des corruptions !!!**
- Pour récupérer ce qu'on peut, et réimporter ailleurs

`pg_resetwal` est un outil fourni avec PostgreSQL. Son objectif est de pouvoir démarrer une instance après un crash si des corruptions de fichiers (typiquement WAL ou fichier de contrôle) empêchent ce démarrage.

Cette action n'est pas une action de réparation ! La réinitialisation des journaux de transactions implique que des transactions qui n'étaient que partiellement validées ne seront pas détectées comme telles, et ne seront donc pas annulées lors du *recovery*.

La conséquence est que les **données de l'instance ne sont plus cohérentes**. Il est fort possible d'y trouver des violations de contraintes diverses (notamment clés étrangères), ou d'autres cas d'incohérences plus difficiles à détecter.

Il s'utilise manuellement, en ligne de commande. Sa fonctionnalité principale est d'effacer les fichiers WAL courants, et il se charge également de réinitialiser les informations correspondantes du fichier de contrôle.

Il est possible de lui spécifier les valeurs à initialiser dans le fichier de contrôle si l'outil ne parvient pas à les déterminer (par exemple, si tous les WAL dans le répertoire `pg_wal` ont été supprimés).

Attention, `pg_resetwal` ne doit **jamais** être utilisé sur une instance démarrée. Avant d'exécuter l'outil, il faut toujours vérifier qu'il ne reste aucun processus de l'instance.

Après la réinitialisation des WAL, une fois que l'instance a démarré, **il ne faut surtout pas ouvrir les accès à l'application** ! Comme indiqué, les données présentent sans aucun doute des incohérences, et toute action en écriture à ce point ne ferait qu'aggraver le problème.

L'étape suivante est donc de faire un export immédiat des données, de les restaurer dans une nouvelle instance initialisée à cet effet (de préférence sur un nouveau serveur, surtout

si l'origine de la corruption n'a pas été clairement identifiée), et ensuite de procéder à une validation méthodique des données.

Il est probable que certaines données incohérentes puissent être identifiées à l'import, lors de la phase de recréation des contraintes : celles-ci échoueront si les données ne les respectent, ce qui permettra de les identifier.

En ce qui concerne les incohérences qui passeront au travers de ces tests, il faudra les trouver et les corriger manuellement, en procédant à une validation fonctionnelle des données.

Il faut donc bien retenir les points suivants :

- `pg_resetwal` n'est pas magique ;
- `pg_resetwal` rend les données incohérentes (ce qui est souvent pire qu'une simple perte d'une partie des données, comme on aurait en restaurant une sauvegarde) ;
- n'utiliser `pg_resetwal` que s'il n'y a aucun autre moyen de faire autrement pour récupérer les données ;
- ne pas l'utiliser sur l'instance ayant subi le problème, mais sur une copie complète effectuée à froid ;
- après usage, exporter toutes les données et les importer dans une nouvelle instance ;
- valider soigneusement les données de la nouvelle instance.

8.5.5 OUTILS - EXTENSION PG_SURGERY

- Extension apparue en v14
- Collection de fonctions permettant de modifier le statut des tuples d'une relation
- Extrèmement dangereuse

Cette extension regroupe des fonctions qui permettent de modifier le satut d'un tuple dans une relation. Il est par exemple possible de rendre une ligne morte ou de rendre visible des tuples qui sont invisibles à cause des informations de visibilité.

Ces fonctions sont dangereuses et peuvent provoquer ou aggraver des corruptions. Elles peuvent pas exemple rendre une table incohérente par rapport à ses index, ou provoquer une violation de contrainte d'unicité ou de clé étrangère. Il ne faut donc les utiliser qu'en dernier recours, sur une copie de votre instance.

8.5.6 OUTILS - VÉRIFICATION D'INTÉGRITÉ

- À froid : `pg_checksums` (à froid, v11)
- Lors d'une sauvegarde : `pg_basebackup` (v11)
- `amcheck` : pure vérification
  - v10 : 2 fonctions pour l'intégrité des index
  - v11 : vérification de la cohérence avec la table (probabiliste)
  - v14 : ajout d'un outil `pg_amcheck`

Depuis la version 11, `pg_checksums` permet de vérifier les sommes de contrôles existantes sur les bases de données **à froid** : l'instance doit être arrêtée proprement auparavant. (En version 11 l'outil s'appelait `pg_verify_checksums`.)

Par exemple, suite à une modification de deux blocs dans une table avec l'outil `hexedit`, on peut rencontrer ceci :

```
$ /usr/pgsql-12/bin/pg_checksums -D /var/lib/pgsql/12/data -c

pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 0:
  calculated checksum 5BF9 but block contains C55D
pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 4438:
  calculated checksum A3 but block contains B8AE
Checksum operation completed
Files scanned:  1282
Blocks scanned: 28484
Bad checksums:  2
Data checksum version: 1
```

À partir de PostgreSQL 12, l'outil `pg_checksums` peut aussi ajouter ou supprimer les sommes de contrôle sur une instance existante arrêtée (donc après le `initdb`), ce qui n'était pas possible dans les versions antérieures.

Une alternative, toujours à partir de la version 11, est d'effectuer une sauvegarde physique avec `pg_basebackup`, ce qui est plus lourd, mais n'oblige pas à arrêter la base.

Le module `amcheck` était apparu en version 10 pour vérifier la cohérence des index et de leur structure interne, et ainsi détecter des bugs, des corruptions dues au système de fichier voire à la mémoire. Il définit deux fonctions :

- `bt_index_check` est destinée aux vérifications de routine, et ne pose qu'un verrou *AccessShareLock* peu gênant ;
- `bt_index_parent_check` est plus minutieuse, mais son exécution gêne les modifications dans la table (verrou *ShareLock* sur la table et l'index) et elle ne peut pas être

exécutée sur un serveur secondaire.

En v11 apparaît le nouveau paramètre `heapallindex`. S'il vaut `true`, chaque fonction effectue une vérification supplémentaire en recréant temporairement une structure d'index et en la comparant avec l'index original. `bt_index_check` vérifiera que chaque entrée de la table possède une entrée dans l'index. `bt_index_parent_check` vérifiera en plus qu'à chaque entrée de l'index correspond une entrée dans la table.

Les verrous posés par les fonctions ne changent pas. Néanmoins, l'utilisation de ce mode a un impact sur la durée d'exécution des vérifications. Pour limiter l'impact, l'opération n'a lieu qu'en mémoire, et dans la limite du paramètre `maintenance_work_mem` (soit entre 256 Mo et 1 Go, parfois plus, sur les serveurs récents). C'est cette restriction mémoire qui implique que la détection de problèmes est probabiliste pour les plus grosses tables (selon la documentation, la probabilité de rater une incohérence est de 2 % si l'on peut consacrer 2 octets de mémoire à chaque ligne). Mais rien n'empêche de relancer les vérifications régulièrement, diminuant ainsi les chances de rater une erreur.

`amcheck` ne fournit aucun moyen de corriger une erreur, puisqu'il détecte des choses qui ne devraient jamais arriver. `REINDEX` sera souvent la solution la plus simple et facile, mais tout dépend de la cause du problème.

Soit `unetable_pkey`, un index de 10 Go sur un entier :

```
CREATE EXTENSION amcheck ;

SELECT bt_index_check('unetable_pkey');
Durée : 63753,257 ms (01:03,753)

SELECT bt_index_check('unetable_pkey', true);
Durée : 234200,678 ms (03:54,201)
```

Ici, la vérification exhaustive multiplie le temps de vérification par un facteur 4.

En version 14, PostgreSQL dispose d'un nouvel outil appelé `pg_amcheck`. Ce dernier facilite l'utilisation de l'extension `amcheck`.

8.6 CAS TYPE DE DÉSASTRES

- Les cas suivants sont assez rares
- Ils nécessitent généralement une restauration
- Certaines manipulations à haut risque sont possibles
  - mais complètement déconseillées !

Cette section décrit quelques-unes des pires situations de corruptions que l'on peut être amené à observer.

Dans la quasi-totalité des cas, la seule bonne réponse est la restauration de l'instance à partir d'une sauvegarde fiable.

8.6.1 AVERTISSEMENT

- Privilégier une solution fiable (restauration, bascule)
- Les actions listées ici sont parfois destructrices
- La plupart peuvent (et vont) provoquer des incohérences
- Travailler sur une copie

La plupart des manipulations mentionnées dans cette partie sont destructives, et peuvent (et vont) provoquer des incohérences dans les données.

Tous les experts s'accordent pour dire que l'utilisation de telles méthodes pour récupérer une instance tend à aggraver le problème existant ou à en provoquer de nouveaux, plus graves. S'il est possible de l'éviter, ne pas les tenter (*ie* : préférer la restauration d'une sauvegarde) !

S'il n'est pas possible de faire autrement (ie : pas de sauvegarde utilisable, données vitales à extraire...), alors TRAVAILLER SUR UNE COPIE.

Il ne faut pas non plus oublier que chaque situation est unique, il faut prendre le temps de bien cerner l'origine du problème, documenter chaque action prise, s'assurer qu'un retour arrière est toujours possible.

8.6.2 CORRUPTION DE BLOCS DANS DES INDEX

- Messages d'erreur lors des accès par l'index ; requêtes incohérentes
- Données différentes entre un indexscan et un seqscan
- Supprimer et recréer l'index : `REINDEX`

Les index sont des objets de structure complexe, ils sont donc particulièrement vulnérables aux corruptions.

Lorsqu'un index est corrompu, on aura généralement des messages d'erreur de ce type :

```
ERROR: invalid page header in block 5869177 of relation base/17291/17420
```

Il peut arriver qu'un bloc corrompu ne renvoie pas de message d'erreur à l'accès, mais que les données elles-mêmes soient altérées, ou que des filtres ne renvoient pas les données attendues.

Ce cas est néanmoins très rare dans un bloc d'index.

Dans la plupart des cas, si les données de la table sous-jacente ne sont pas affectées, il est possible de réparer l'index en le reconstruisant intégralement grâce à la commande `REINDEX`.

8.6.3 CORRUPTION DE BLOCS DANS DES TABLES 1

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

- Cas plus problématique
- Restauration probablement nécessaire

Les corruptions de blocs vont généralement déclencher des erreurs du type suivant :

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

Si la relation concernée est une table, tout ou partie des données contenues dans ces blocs est perdu.

L'apparition de ce type d'erreur est un signal fort qu'une restauration est certainement nécessaire.

8.6.4 CORRUPTION DE BLOCS DANS DES TABLES 2

- `SET zero_damaged_pages = true ;` + `VACUUM FULL`
- Des données vont certainement être perdues

Néanmoins, s'il est nécessaire de lire le maximum de données possibles de la table, il est possible d'utiliser l'option de PostgreSQL `zero_damaged_pages` pour demander au moteur de réinitialiser les blocs invalides à zéro lorsqu'ils sont lus au lieu de tomber en erreur. Il s'agit d'un des très rares paramètres absents de `postgresql.conf`.

Par exemple :

```
> SET zero_damaged_pages = true ;
SET
> VACUUM FULL mycorruptedtable ;
WARNING:  invalid page header in block 32570 of relation base/16390/2663;
          zeroing out page
VACUUM
```

Si cela se termine sans erreur, les blocs invalides ont été réinitialisés.

Les données qu'ils contenaient sont évidemment perdues, mais la table peut désormais être accédée dans son intégralité en lecture, permettant ainsi par exemple de réaliser un export des données pour récupérer ce qui peut l'être.

Attention, du fait des données perdues, le résultat peut être incohérent (contraintes non respectées…).

Par ailleurs, par défaut PostgreSQL ne détecte pas les corruptions logiques, c'est-à-dire n'affectant pas la structure des données mais uniquement le contenu.

Il ne faut donc pas penser que la procédure d'export complet de données suivie d'un import sans erreur garantit l'absence de corruption.

8.6.5 CORRUPTION DE BLOCS DANS DES TABLES 3

- Si la corruption est importante, l'accès au bloc peut faire crasher l'instance
- Il est tout de même possible de réinitialiser le bloc
  - identifier le fichier à l'aide de `pg_relation_filepath()`
  - trouver le bloc avec `ctid` / `pageinspect`
  - réinitialiser le bloc avec `dd`
  - il faut vraiment ne pas avoir d'autre choix

Dans certains cas, il arrive que la corruption soit suffisamment importante pour que le simple accès au bloc fasse crasher l'instance.

Dans ce cas, le seul moyen de réinitialiser le bloc est de le faire manuellement au niveau du fichier, instance arrêtée, par exemple avec la commande `dd`.

Pour identifier le fichier associé à la table corrompue, il est possible d'utiliser la fonction `pg_relation_filepath()` :

```
> SELECT pg_relation_filepath('test_corruptindex') ;

 pg_relation_filepath
----------------------
 base/16390/40995
```

Le résultat donne le chemin vers le fichier principal de la table, relatif au `PGDATA` de l'instance.

Attention, une table peut contenir plusieurs fichiers. Par défaut une instance PostgreSQL sépare les fichiers en segments de 1 Go. Une table dépassant cette taille aura donc des fichiers supplémentaires (`base/16390/40995.1`, `base/16390/40995.2`...).

Pour trouver le fichier contenant le bloc corrompu, il faudra donc prendre en compte le numéro du bloc trouvé dans le champ `ctid`, multiplier ce numéro par la taille du bloc (paramètre `block_size`, 8 ko par défaut), et diviser le tout par la taille du segment.

Cette manipulation est évidemment extrêmement risquée, la moindre erreur pouvant rendre irrécupérables de grandes portions de données.

Il est donc fortement déconseillé de se lancer dans ce genre de manipulations à moins d'être absolument certain que c'est indispensable.

Encore une fois, ne pas oublier de travailler sur une copie, et pas directement sur l'instance de production.

8.6.6 CORRUPTION DES WAL 1

- Situés dans le répertoire `pg_wal`
- Les WAL sont nécessaires au *recovery*
- Démarrage impossible s'ils sont corrompus ou manquants
- Si les fichiers WAL ont été archivés, les récupérer
- Sinon, la restauration est la seule solution viable

Les fichiers WAL sont les journaux de transactions de PostgreSQL.

Leur fonction est d'assurer que les transactions qui ont été effectuées depuis le dernier checkpoint ne seront pas perdues en cas de crash de l'instance.

Si certains sont corrompus ou manquants (rappel : il ne faut JAMAIS supprimer les fichiers WAL, même si le système de fichiers est plein !), alors PostgreSQL ne pourra pas redémarrer.

Si l'archivage était activé et que les fichiers WAL affectés ont bien été archivés, alors il est possible de les restaurer avant de tenter un nouveau démarrage.

Si ce n'est pas possible ou des fichiers WAL archivés ont également été corrompus ou supprimés, l'instance ne pourra pas redémarrer.

Dans cette situation, comme dans la plupart des autres évoquées ici, la seule solution permettant de s'assurer que les données ne seront pas corrompues est de procéder à une restauration de l'instance.

8.6.7 CORRUPTION DES WAL 2

- `pg_resetwal` permet de forcer le démarrage
- ATTENTION !!!
  - cela va provoquer des pertes de données
  - des corruptions de données sont également probables
  - ce n'est pas une action corrective !

L'utilitaire `pg_resetwal` a comme fonction principale de supprimer les fichiers WAL courants et d'en créer un nouveau, avant de mettre à jour le fichier de contrôle pour permettre le redémarrage.

Au minimum, cette action va provoquer la perte de toutes les transactions validées effectuées depuis le dernier checkpoint.

Il est également probable que des incohérences vont apparaître, certaines relativement

simples à détecter via un export/import (incohérences dans les clés étrangères par exemple), certaines complètement invisibles.

L'utilisation de cet utilitaire est extrêmement dangereuse, n'est pas recommandée, et ne peut jamais être considérée comme une action corrective. Il faut toujours privilégier la restauration d'une sauvegarde plutôt que son exécution.

Si l'utilisation de `pg_resetwal` est néanmoins nécessaire (par exemple pour récupérer des données absentes de la sauvegarde), alors il faut travailler sur une copie des fichiers de l'instance, récupérer ce qui peut l'être à l'aide d'un export de données, et les importer dans une autre instance.

Les données récupérées de cette manière devraient également être soigneusement validées avant d'être importée de façon à s'assurer qu'il n'y a pas de corruption silencieuse.

Il ne faut en aucun cas remettre une instance en production après une réinitialisation des WAL.

8.6.8 CORRUPTION DU FICHIER DE CONTRÔLE

- Fichier `global/pg_control`
- Contient les informations liées au dernier checkpoint
- Sans lui, l'instance ne peut pas démarrer
- Recréation avec `pg_resetwal`… parfois
- Restauration nécessaire

Le fichier de contrôle de l'instance contient de nombreuses informations liées à l'activité et au statut de l'instance, notamment l'instant du dernier checkpoint, la position correspondante dans les WAL, le numéro de transaction courant et le prochain à venir…

Ce fichier est le premier lu par l'instance. S'il est corrompu ou supprimé, l'instance ne pourra pas démarrer.

Il est possible de forcer la réinitialisation de ce fichier à l'aide de la commande `pg_resetwal`, qui va se baser par défaut sur les informations contenues dans les fichiers WAL présents pour tenter de « deviner » le contenu du fichier de contrôle.

Ces informations seront très certainement erronées, potentiellement à tel point que même l'accès aux bases de données par leur nom ne sera pas possible :

```
$ pg_isready
/var/run/postgresql:5432 - accepting connections
```

```
$ psql postgres
psql: FATAL:  database "postgres" does not exist
```

Encore une fois, utiliser `pg_resetwal` n'est en aucun cas une solution, mais doit uniquement être considéré comme un contournement temporaire à une situation désastreuse.

Une instance altérée par cet outil ne doit pas être considérée comme saine.

8.6.9 CORRUPTION DU CLOG

- Fichiers dans `pg_xact`
- Statut des différentes transactions
- Son altération risque de causer des incohérences

Le fichier CLOG (*Commit Log*) dans `PGDATA/pg_xact/` contient le statut des différentes transactions, notamment si celles-ci sont en cours, validées ou annulées.

S'il est altéré ou supprimé, il est possible que des transactions qui avaient été marquées comme annulées soient désormais considérées comme valides, et donc que les modifications de données correspondantes deviennent visibles aux autres transactions.

C'est évidemment un problème d'incohérence majeur, tout problème avec ce fichier devrait donc être soigneusement analysé.

Il est préférable dans le doute de procéder à une restauration et d'accepter une perte de données plutôt que de risquer de maintenir des données incohérentes dans la base.

8.6.10 CORRUPTION DU CATALOGUE SYSTÈME

- Le catalogue contient la définition du schéma
- Sans lui, les données sont inaccessibles
- Situation très délicate…

Le catalogue système contient la définition de toutes les relations, les méthodes d'accès, la correspondance entre un objet et un fichier sur disque, les types de données existantes…

S'il est incomplet, corrompu ou inaccessible, l'accès aux données en SQL risque de ne pas être possible du tout.

Cette situation est très délicate, et appelle là encore une restauration.

Si le catalogue était complètement inaccessible, sans sauvegarde la seule solution restante serait de tenter d'extraire les données directement des fichiers data de l'instance, en oubliant toute notion de cohérence, de type de données, de relation...

Personne ne veut faire ça.

8.7 CONCLUSION

- Les désastres peuvent arriver
- Il faut s'y être préparé
- Faites des sauvegardes !
  - et testez-les

8.8 QUIZ

https://dali.bo/i5_quiz

8.9 TRAVAUX PRATIQUES

8.9.1 CORRUPTION D'UN BLOC DE DONNÉES

Nous allons corrompre un bloc et voir certains impacts possibles.

> Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

> Créer une base **pgbench** avec un facteur d'échelle 10 **avec** les clés étrangères entre tables (option `--foreign-keys`).

> Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

> Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

> Arrêter PostgreSQL.

> Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

> Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

> Tenter un `pgdumpall > /dev/null`.

> Arrêter PostgreSQL.
> Voir ce que donne `pg_checksums` (en v12) ou `pg_verify_checksums` (en v11).

> Faire une copie de travail à froid du PGDATA.
> Protéger en écriture le PGDATA original.
> Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

> Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause). Démarrer le cluster sur la copie avec `pg_ctl`.

> Que renvoie `SELECT * FROM pgbench_accounts LIMIT 100` ?

> Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

8.9.2 CORRUPTION D'UN BLOC DE DONNÉES ET INCOHÉRENCES

Nous allons à présent corrompre une table portant une clé étrangère. Nous continuons sur la copie de la base de travail, où les sommes de contrôle ont été désactivées.

> Consulter le format et le contenu de la table `pgbench_branches`.

> Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

> Arrêter PostgreSQL.
> Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
> En toute fin de fichier, remplacer le dernier `01` par un `FF`.
> Redémarrer PostgreSQL.

> Compter le nombre de lignes dans `pgbench_branches`.
> Recompter après `SET enable_seqscan TO off ;`.
> Quelle est la bonne réponse ? Vérifier le contenu de la table.

> Qu'affiche `pageinspect` pour cette table ?

> Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

> Exporter `pgbench_accounts`, définition des index comprise. Supprimer la table (il faudra supprimer `pgbench_history` aussi). Tenter de la réimporter.

8.10 TRAVAUX PRATIQUES (SOLUTION)

8.10.1 CORRUPTION D'UN BLOC DE DONNÉES

> Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

```
# SHOW data_checksums ;

 data_checksums
----------------
 on
```

Si la réponse est `off`, on peut (à partir de la v12) mettre les checksums en place :

```
$ /usr/pgsql-14/bin/pg_checksums -D /var/lib/pgsql/14/data.BACKUP/  --enable --progress
58/58 MB (100%) computed
Checksum operation completed
Files scanned:  964
Blocks scanned: 7524
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster
```

> Créer une base **pgbench** avec un facteur d'échelle 10 **avec** les clés étrangères entre tables (option `--foreign-keys`).

```
$ dropdb --if-exists pgbench ;
$ createdb pgbench ;

$ /usr/pgsql-14/bin/pgbench -i -s 10 -d pgbench --foreign-keys
```

```
...
creating tables...
generating data...
100000 of 1000000 tuples (10%) done (elapsed 0.15 s, remaining 1.31 s)
200000 of 1000000 tuples (20%) done (elapsed 0.35 s, remaining 1.39 s)
..
1000000 of 1000000 tuples (100%) done (elapsed 2.16 s, remaining 0.00 s)
vacuuming...
creating primary keys...
creating foreign keys...
done.
```

> Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

La table fait 128 Mo selon un `\d+`.

La clé `aid` va de 1 à 100000 :

```
# SELECT min(aid), max(aid) FROM pgbench_accounts ;

 min |   max
-----+---------
   1 | 1000000
```

Un `SELECT` montre que les valeurs sont triées mais c'est dû à l'initialisation.

> Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

```
 SELECT pg_relation_filepath('pgbench_accounts') ;

 pg_relation_filepath
----------------------
 base/16454/16489
```

> Arrêter PostgreSQL.

```
 # systemctl restart postgresql-14
```

Cela permet d'être sûr qu'il ne va pas écraser nos modifications lors d'un checkpoint.

> Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

```
 # dnf install hexedit

postgres$ hexedit /var/lib/pgsql/14/data/base/16454/16489
```

Aller par exemple sur la 2è ligne, modifier `80 9F` en `FF FF`. Sortir avec Ctrl-X, confirmer la sauvegarde.

> Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

```
# SELECT * FROM pgbench_accounts ;

WARNING:  page verification failed, calculated checksum 62947 but expected 57715
ERROR:  invalid page in block 0 of relation base/16454/16489
```

> Tenter un `pgdumpall > /dev/null`.

```
$ pg_dumpall > /dev/null
pg_dump: WARNING:  page verification failed, calculated checksum 62947 but expected 57715
pg_dump: error: Dumping the contents of table "pgbench_accounts" failed: PQgetResult() failed.
pg_dump: error: Error message from server:
         ERROR:  invalid page in block 0 of relation base/16454/16489
pg_dump: error: The command was:
         COPY public.pgbench_accounts (aid, bid, abalance, filler) TO stdout;
pg_dumpall: error: pg_dump failed on database "pgbench", exiting
```

> Arrêter PostgreSQL.
> Voir ce que donne `pg_checksums` (en v12) ou `pg_verify_checksums` (en v11).

```
 # systemctl stop postgresql-14

$ /usr/pgsql-14/bin/pg_checksums -D /var/lib/pgsql/14/data/  --check --progress
pg_checksums: error: checksum verification failed in file
"/var/lib/pgsql/14/data//base/16454/16489", block 0:
                              calculated checksum F5E3 but block contains E173
216/216 MB (100%) computed
Checksum operation completed
Files scanned:  1280
Blocks scanned: 27699
Bad checksums:  1
Data checksum version: 1
```

> Faire une copie de travail à froid du PGDATA.

> Protéger en écriture le PGDATA original.
> Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

Dans l'idéal, la copie devrait se faire vers un autre support, une corruption rend celui-ci suspect. Dans le cadre du TP, ceci suffira :

```
$ cp -upR /var/lib/pgsql/14/data/  /var/lib/pgsql/14/data.BACKUP/
$ chmod -R -w /var/lib/pgsql/14/data/
```

Dans `/var/lib/pgsql/14/data.BACKUP/pg_hba.conf` ne doit plus subsister que :

```
local   all             all                                     trust
```

> Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause).

```
$ /usr/pgsql-14/bin/pg_checksums -D /var/lib/pgsql/14/data.BACKUP/  --disable
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums disabled in cluster
```

> Démarrer le cluster sur la copie avec `pg_ctl`.
> Que renvoie `SELECT * FROM pgbench_accounts LIMIT 100` ?

```
/usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data.BACKUP/ start

# SELECT *  FROM pgbench_accounts LIMIT 10;

ERROR:  out of memory
DÉTAIL : Failed on request of size 536888061 in memory context "printtup".
```

Ce ne sera pas forcément cette erreur, plus rien n'est sûr en cas de corruption. L'avantage des sommes de contrôle est justement d'avoir une erreur moins grave et plus ciblée.

Un `pg_dumpall` renverra le même message.

> Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

```
pgbench=# SET zero_damaged_pages TO on ;
SET
pgbench=# VACUUM FULL pgbench_accounts ;
VACUUM

pgbench=# SELECT *  FROM pgbench_accounts LIMIT 100;
```

```
 aid | bid | abalance |                                       filler
-----+-----+----------+-------------------------------------------------------------------------------------
   2 |   1 |        0 |
   3 |   1 |        0 |
   4 |   1 |        0 |
[...]

pgbench=# SELECT min(aid), max(aid), count(aid) FROM pgbench_accounts ;

 min |   max   | count
-----+---------+--------
   2 | 1000000 | 999999
```

Apparemment une ligne a disparu, celle portant la valeur 1 pour la clé. Il est rare que la perte soit aussi évidente !

8.10.2 CORRUPTION D'UN BLOC DE DONNÉES ET INCOHÉRENCES

> Consulter le format et le contenu de la table `pgbench_branches`.

Cette petite table ne contient que 10 valeurs :

```
# SELECT * FROM pgbench_branches ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
   2 |        0 |
   3 |        0 |
   4 |        0 |
   5 |        0 |
   6 |        0 |
   7 |        0 |
   8 |        0 |
   9 |        0 |
  10 |        0 |
(10 lignes)
```

> Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

```
# SELECT pg_relation_filepath('pgbench_branches') ;

 pg_relation_filepath
----------------------
 base/16454/16490
```

> Arrêter PostgreSQL.
> Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
> En toute fin de fichier, remplacer le dernier `01` par un `FF`.
> Redémarrer PostgreSQL.

```
$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data.BACKUP/ stop

$ hexedit /var/lib/pgsql/14/data.BACKUP/base/16454/16490

$ /usr/pgsql-14/bin/pg_ctl -D /var/lib/pgsql/14/data.BACKUP/ start
```

> Compter le nombre de lignes dans `pgbench_branches`.
> Recompter après `SET enable_seqscan TO off ;`.
> Quelle est la bonne réponse ? Vérifier le contenu de la table.

Les deux décomptes sont contradictoires :

```
pgbench=# SELECT count(*) FROM pgbench_branches ;
 count
-------
     9

pgbench=# SET enable_seqscan TO off ;
SET

pgbench=# SELECT count(*) FROM pgbench_branches ;
 count
-------
    10
```

En effet, le premier lit la (petite) table directement, le second passe par l'index, comme un EXPLAIN le montrerait. Les deux objets diffèrent.

Et le contenu de la table est devenu :

```
# SELECT * FROM pgbench_branches ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
   2 |        0 |
   3 |        0 |
   4 |        0 |
```

```
   5 |        0 |
   6 |        0 |
   7 |        0 |
   8 |        0 |
   9 |        0 |
(9 lignes)
```

Le 1 est devenu 255 (c'est notre première modification) mais la ligne 10 a disparu !

Les requêtes peuvent renvoyer un résultat incohérent avec leur critère :

```
pgbench=# SET enable_seqscan TO off;
SET
pgbench=# SELECT * FROM pgbench_branches
          WHERE    bid = 1 ;

 bid | bbalance | filler
-----+----------+--------
 255 |        0 |
```

> Qu'affiche `pageinspect` pour cette table ?

```
pgbench=# CREATE EXTENSION pageinspect ;

pgbench=# SELECT t_ctid, lp_off, lp_len, t_xmin, t_xmax, t_data
          FROM heap_page_items(get_raw_page('pgbench_branches',0));

 t_ctid | lp_off | lp_len | t_xmin | t_xmax |       t_data
--------+--------+--------+--------+--------+--------------------
 (0,1)  |   8160 |     32 |  63726 |      0 | \xff00000000000000
 (0,2)  |   8128 |     32 |  63726 |      0 | \x0200000000000000
 (0,3)  |   8096 |     32 |  63726 |      0 | \x0300000000000000
 (0,4)  |   8064 |     32 |  63726 |      0 | \x0400000000000000
 (0,5)  |   8032 |     32 |  63726 |      0 | \x0500000000000000
 (0,6)  |   8000 |     32 |  63726 |      0 | \x0600000000000000
 (0,7)  |   7968 |     32 |  63726 |      0 | \x0700000000000000
 (0,8)  |   7936 |     32 |  63726 |      0 | \x0800000000000000
 (0,9)  |   7904 |     32 |  63726 |      0 | \x0900000000000000
        |  32767 |    127 |        |        |
(10 lignes)
```

La première ligne indique bien que le 1 est devenu un 255.

La dernière ligne porte sur la première modification, qui a détruit les informations sur le `ctid`. Celle-ci est à présent inaccessible.

> Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

La documentation est sur https://docs.postgresql.fr/current/amcheck.html.

Une vérification complète se fait ainsi :

```
pgbench=# CREATE EXTENSION amcheck ;

pgbench=# SELECT bt_index_check (index => 'pgbench_branches_pkey',
                                 heapallindexed => true);
 bt_index_check
----------------


(1 ligne)

pgbench=# SELECT bt_index_parent_check (index => 'pgbench_branches_pkey',
                                        heapallindexed => true, rootdescend => true);
ERROR:  heap tuple (0,1) from table "pgbench_branches"
        lacks matching index tuple within index "pgbench_branches_pkey"
```

Un seul des problèmes a été repéré.

Un `REINDEX` serait ici une mauvaise idée : c'est la table qui est corrompue ! Les sommes de contrôle, là encore, auraient permis de cibler le problème très tôt.

> Exporter `pgbench_accounts`, définition des index comprise.
> Supprimer la table (il faudra supprimer `pgbench_history` aussi).
> Tenter de la réimporter.

```
$ pg_dump -d pgbench  -t pgbench_accounts -f /tmp/pgbench_accounts.dmp

$ psql pgbench -c 'DROP TABLE pgbench_accounts CASCADE'
NOTICE:  drop cascades to constraint pgbench_history_aid_fkey on table pgbench_history
DROP TABLE

$ psql pgbench < /tmp/pgbench_accounts.dmp
SET
SET
SET
SET
SET
 set_config
------------
```

```
(1 ligne)

SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
COPY 999999
ALTER TABLE
CREATE INDEX
ERROR:  insert or update on table "pgbench_accounts"
        violates foreign key constraint "pgbench_accounts_bid_fkey"
DÉTAIL : Key (bid)=(1) is not present in table "pgbench_branches".
```

La contrainte de clé étrangère entre les deux tables ne peut être respectée : `bid` est à 1 sur de nombreuses lignes de `pgbench_accounts` mais n'existe plus dans la table `pgbench_branches` ! Ce genre d'incohérence doit être recherchée très tôt pour ne pas surgir bien plus tard, quand on doit restaurer pour d'autres raisons.

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.