Formation DBA2

PostgreSQL Avancé

25.03

Table des matières

Sur ce document

Formation	Formation DBA2
Titre	PostgreSQL Avancé
Révision	25.03
ISBN	978-2-38168-133-7
PDF	https://dali.bo/dba2_pdf
EPUB	https://dali.bo/dba2_epub
HTML	https://dali.bo/dba2_html
Slides	https://dali.bo/dba2_slides

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

Marques déposées

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

Versions de PostgreSQL couvertes

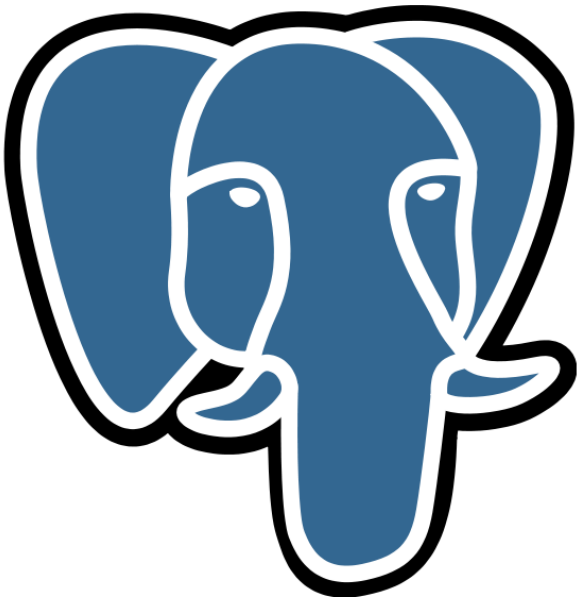
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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

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

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

1/ 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`
Elle est ensuite démarrée.

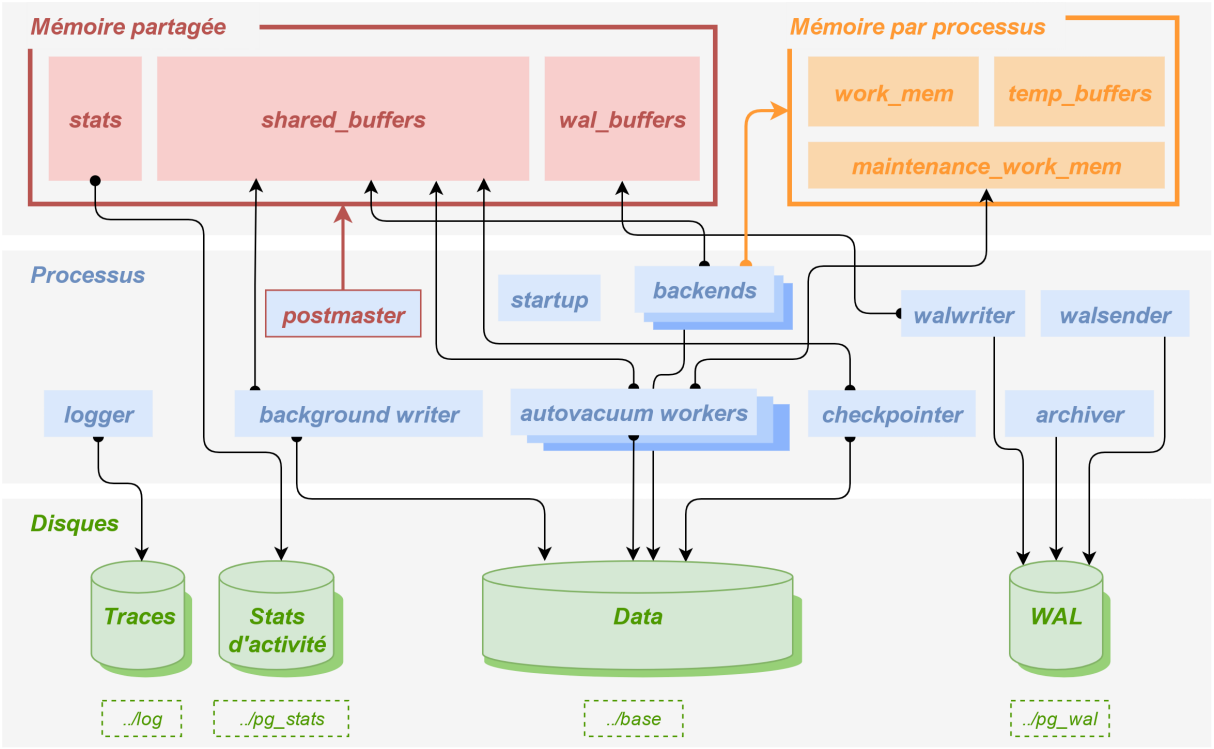
Dans le cas d'une distribution Red Hat, aucune instance n'est créée automatiquement. Il faudra utiliser un script (dont le nom dépend de la version de la distribution) pour créer l'instance, puis nous pourrons utiliser le script de démarrage pour lancer le serveur.

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



1.3.1 Introduction

- PostgreSQL est :
  - multiprocessus (et non multithread)
  - à mémoire partagée
  - client-serveur

L'architecture PostgreSQL est une architecture multiprocessus et non multithread.

Cela signifie que chaque processus de PostgreSQL s'exécute dans un contexte mémoire isolé, et que la communication entre ces processus repose sur des mécanismes systèmes inter-processus : sémaphores, zones de mémoire partagée, sockets. Ceci s'oppose à l'architecture multithread, où l'ensemble du moteur s'exécute dans un seul processus, avec plusieurs threads (contextes) d'exécution, où tout est partagé par défaut.

Le principal avantage de cette architecture multiprocessus est la stabilité : un processus, en cas de problème, ne corrompt que sa mémoire (ou la mémoire partagée), le plantage d'un processus n'affecte

pas directement les autres. Son principal défaut est une allocation statique des ressources de mémoire partagée : elles ne sont pas redimensionnables à chaud.

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

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

1.3.2 Processus d'arrière-plan

```
# ps f -e --format=pid,command | grep -E "postgres|postmaster"
  96122 /usr/pgsql-17/bin/postmaster -D /var/lib/pgsql/15/data/
  96123  \_ postgres: logger
  96125  \_ postgres: checkpointer
  96126  \_ postgres: background writer
  96127  \_ postgres: walwriter
  96128  \_ postgres: autovacuum launcher
  96131  \_ postgres: logical replication launcher
```

(sous Rocky Linux 8)

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

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

1.3.3 Processus d'arrière-plan (suite)

- Les processus présents au démarrage :
  - Un processus père : `postmaster`
  - `background writer`
  - `checkpointer`
  - `walwriter`
  - `autovacuum launcher`
  - `stats collector` (avant v15)
  - `logical replication launcher`
- et d'autres selon la configuration et le moment :
  - dont les *background workers* : parallélisation, extensions…

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

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

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

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

Avant la version 15, le `stats collector` collecte les statistiques d'activité du serveur. À partir de la version 15, ces informations sont conservées en mémoire jusqu'à l'arrêt du serveur où elles sont stockées sur disque jusqu'au prochain démarrage.

Le `logical replication launcher` est un processus dédié à la réplication logique.

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

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

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

Il existe aussi les *background workers* (processus d'arrière-plan), lancés par PostgreSQL, mais aussi par des extensions tierces. Par exemple, la parallélisation des requêtes se base sur la création tem-

poraire de *background workers* épaulant le processus principal de la requête. La réplication logique utilise des *background workers* à plus longue durée de vie. De nombreuses extensions en utilisent pour des raisons très diverses. Le paramètre `max_worker_processes` régule le nombre de ces *workers*. Ne descendez pas en-dessous du défaut (8). Il faudra même parfois monter plus haut.

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
- Le nombre de processus est régi par les paramètres :
  - `max_connections` (défaut : 100) - connexions réservées
  - compromis nombre requêtes actives/nombre cœurs/complexité/mémoire

Pour chaque nouvelle session à l'instance, le processus `postmaster` crée un processus fils qui s'occupe de gérer cette session. Il y a donc un processus dédié à chaque connexion cliente, et ce processus est détruit à fin de cette connexion.

Ce processus dit *backend* reçoit les ordres SQL, les interprète, exécute les requêtes, trie les données, et enfin retourne les résultats. Dans certains cas, il peut demander le lancement d'autres processus pour l'aider dans l'exécution d'une requête en lecture seule (parallélisme).

Le dialogue entre le client et ce processus respecte un protocole réseau bien défini. Le client n'a jamais accès aux données par un autre moyen que par ce protocole.

Le nombre maximum de connexions à l'instance simultanées, actives ou non, est limité par le paramètre `max_connections`. Le défaut est 100.

Certaines connexions sont réservées. Les administrateurs doivent pouvoir se connecter à l'instance même si la limite est atteinte. Quelques connexions sont donc réservées aux superutilisateurs (paramètre `superuser_reserved_connections`, à 3 par défaut). On peut aussi octroyer le rôle `pg_use_reserved_connections` à certains utilisateurs pour leur garantir l'accès à un nombre de connexions réservées, à définir avec le paramètre `reserved_connections` (vide par défaut), cela à partir de PostreSQL 16.

Attention : modifier un de ces paramètres impose un redémarrage complet de l'instance, puisqu'ils ont un impact sur la taille de la mémoire partagée entre les processus PostgreSQL. Il faut donc réfléchir au bon dimensionnement avant la mise en production.

La valeur 100 pour `max_connections` est généralement suffisante. Il peut être intéressant de la diminuer pour se permettre de monter `work_mem` et autoriser plus de mémoire de tri. Il est possible de

monter `max_connections` pour qu'un plus grand nombre de clients puisse se connecter en même temps.

Il s'agit aussi d'arbitrer entre le nombre de requêtes à exécuter à un instant t, le nombre de CPU disponibles, la complexité des requêtes, et le nombre de processus que peut gérer l'OS. Ce dimensionnement est encore compliqué par le parallélisme et la limitation de la bande passante des disques.

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

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

Le plus réputé est PgBouncer, mais il est aussi souvent inclus dans des serveurs d'application (par exemple Tomcat).

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
```

- Une seule instance PostgreSQL doit y accéder !

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

```
SHOW data_directory;

      data_directory
---------------------------
 /var/lib/pgsql/15/data
```

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

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

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

Vous pouvez trouver une description de tous les fichiers et répertoires dans la documentation officielle[1].

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

1.6.2 Fichiers de configuration

- `postgresql.conf` (+ fichiers inclus)
- `postgresql.auto.conf`
- `pg_hba.conf` (+ fichiers inclus (v16))
- `pg_ident.conf` (idem)

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

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

`postgresql.auto.conf` stocke les paramètres de configuration fixés en utilisant la commande `ALTER SYSTEM`. Il surcharge donc `postgresql.conf`. Comme pour `postgresql.conf`, sa modification impose de recharger la configuration ou redémarrer l'instance. Il est techniquement possible, mais déconseillé, de le modifier à la main. `postgresql.auto.conf` est toujours dans le répertoire des données, même si `postgresql.conf` est ailleurs.

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

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

Ces deux derniers fichiers peuvent eux-mêmes inclure d'autres fichiers (depuis PostgreSQL 16). Leur utilisation est décrite dans notre première formation[2].

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`

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

`PG_VERSION` est un fichier. Il contient en texte lisible la version majeure devant être utilisée pour accéder au répertoire (par exemple `15`). On trouve ces fichiers `PG_VERSION` à de nombreux endroits de l'arborescence de PostgreSQL, par exemple dans chaque répertoire de base du répertoire `PGDATA/base/` ou à la racine de chaque tablespace.

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

```
~$ cat /var/lib/postgresql/15/data/postmaster.pid
7771
/var/lib/postgresql/15/data
1503584802
5432
/tmp
localhost
  5432001  54919263
ready

$ ps -HFC postgres
UID  PID    SZ     RSS PSR STIME TIME   CMD
pos 7771 0 42486 16536   3 16:26 00:00  /usr/local/pgsql/bin/postgres
                                          -D /var/lib/postgresql/15/data
pos 7773 0 42486  4656   0 16:26 00:00  postgres: checkpointer
pos 7774 0 42486  5044   1 16:26 00:00  postgres: background writer
pos 7775 0 42486  8224   1 16:26 00:00  postgres: walwriter
pos 7776 0 42850  5640   1 16:26 00:00  postgres: autovacuum launcher
pos 7777 0 42559  3684   0 16:26 00:00  postgres: logical replication launcher

$ ipcs -p |grep  7771
54919263   postgres   7771       10640

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

Le processus père de cette instance PostgreSQL a comme PID le 7771. Ce processus a bien réclamé une sémaphore d'identifiant 54919263. Cette sémaphore correspond à des segments de mémoire partagée pour un total de 56 octets. Le répertoire de données se trouve bien dans `/var/lib/postgresql/15/data`.

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

```
external_pid_file = '/var/run/postgresql/15-main.pid'
```

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

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

```
$ cat $PGDATA/postmaster.opts
/usr/local/pgsql/bin/postgres "-D" "/var/lib/postgresql/15/data"
```

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
```

Depuis la ligne de commande, il existe un utilitaire nommé `oid2name`, dont le but est de faire la liaison entre le nom de fichier et le nom de l'objet PostgreSQL. Il a besoin de se connecter à la base :

```
$ pwd
/var/lib/pgsql/15/data/base/16388

$ /usr/pgsql-17/bin/oid2name -f 16477 -d employes
From database "employes":
  Filenode         Table Name
----------------------------
     16477  employes_big_pkey
```

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

Si nous demandons au sein d'une première session :

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

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

```
SELECT * FROM pg_ls_tmpdir() ;
```

```
       name         |    size    |      modification
--------------------+------------+------------------------
 pgsql_tmp12851.16 | 1073741824 | 2020-09-02 15:43:27+02
 pgsql_tmp12851.11 | 1073741824 | 2020-09-02 15:42:32+02
 pgsql_tmp12851.7  | 1073741824 | 2020-09-02 15:41:49+02
 pgsql_tmp12851.5  | 1073741824 | 2020-09-02 15:41:29+02
 pgsql_tmp12851.9  | 1073741824 | 2020-09-02 15:42:11+02
 pgsql_tmp12851.0  | 1073741824 | 2020-09-02 15:40:36+02
 pgsql_tmp12851.14 | 1073741824 | 2020-09-02 15:43:06+02
 pgsql_tmp12851.4  | 1073741824 | 2020-09-02 15:41:19+02
 pgsql_tmp12851.13 | 1073741824 | 2020-09-02 15:42:54+02
 pgsql_tmp12851.3  | 1073741824 | 2020-09-02 15:41:09+02
 pgsql_tmp12851.1  | 1073741824 | 2020-09-02 15:40:47+02
 pgsql_tmp12851.15 | 1073741824 | 2020-09-02 15:43:17+02
 pgsql_tmp12851.2  | 1073741824 | 2020-09-02 15:40:58+02
 pgsql_tmp12851.8  | 1073741824 | 2020-09-02 15:42:00+02
 pgsql_tmp12851.12 | 1073741824 | 2020-09-02 15:42:43+02
 pgsql_tmp12851.10 | 1073741824 | 2020-09-02 15:42:21+02
 pgsql_tmp12851.6  | 1073741824 | 2020-09-02 15:41:39+02
 pgsql_tmp12851.17 |  546168976 | 2020-09-02 15:43:32+02
```

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

1.6.5 Fichiers liés aux transactions

- `pg_wal/` : journaux de transactions
  - sous-répertoire `archive_status`
  - nom : *timeline*, journal, segment
  - ex : `00000002 00000142 000000FF`
  - créés, initialisés à zéro, recyclés
- `pg_xact/` : état des transactions
- mais aussi : `pg_commit_ts/`, `pg_multixact/`, `pg_serial/` `pg_snapshots/`, `pg_subtrans/`, `pg_twophase/`
- **Ces fichiers sont vitaux !**

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

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

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

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

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

L'ordre d'écriture des journaux est numérique (en hexadécimal), et leur archivage doit suivre cet ordre. Il ne faut pas se fier à la date des fichiers pour le tri : pour des raisons de performances, PostgreSQL recycle généralement les fichiers en les renommant. Dans l'exemple ci-dessus, le dernier journal écrit est `000000020000014300000020` et non `000000020000014300000024`.

Toujours pour des raisons de performance liées aux métadata (voir cette explication d'Andres Freund[3]), les nouveaux journaux sont initialisés par des zéros avant d'être utilisés. Recyclage et initialisation peuvent être désactivés en passant les paramètres `wal_recycle` et `wal_init_zero` à `off`, ce qui n'est conseillé que sur certains systèmes de fichiers comme ZFS ou btrfs.

Dans le cadre d'un archivage PITR et/ou d'une réplication par *log shipping*, le sous-répertoire `pg_wal/archive_status` indique l'état des journaux dans le contexte de l'archivage. Les fichiers `.ready` indiquent les journaux restant à archiver (normalement peu nombreux), les `.done` ceux déjà archivés. Il est possible de connaître facilement le contenu de ce répertoire en utilisant la fonction `pg_ls_archive_statusdir()` :

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

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

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

[3] https://www.postgresql.org/message-id/4xaez4hh3cwkclr77tnkdba6wisueu32vza44iugozai5hklyb%405epf6zpo66aa

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

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

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

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

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

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/` : tablespaces
  - si vraiment nécessaires
  - liens symboliques ou points de jonction
  - totalement optionnels

Dans PGDATA, le sous-répertoire `pg_tblspc` contient les *tablespaces*, c'est-à-dire des espaces de stockage.

1.6.7.1 Sous Linux

Sous Linux, ce sont des liens symboliques vers un simple répertoire extérieur à PGDATA. Chaque lien symbolique a comme nom l'OID du tablespace (table système `pg_tablespace`). PostgreSQL y crée un répertoire lié aux versions de PostgreSQL et du catalogue, et y place les fichiers de données.

```
postgres=# \db+
                          Liste des tablespaces
    Nom     | Propriétaire |      Emplacement      | … | Taille  |…
------------+--------------+-----------------------+---+---------+-
 froid      | postgres     | /mnt/hdd/pg           |   | 3576 kB |
 pg_default | postgres     |                       |   | 6536 MB |
 pg_global  | postgres     |                       |   | 587 kB  |
```

```
sudo ls -R /mnt/hdd/pg
```

```
/mnt/hdd/pg:
PG_15_202209061

/mnt/hdd/pg/PG_15_202209061:
5

/mnt/hdd/pg/PG_15_202209061/5:
142532  142532_fsm  142532_vm
```

1.6.7.2 Sous Windows

Sous Windows, les liens sont à proprement parler des *Reparse Points* (ou *Junction Points*) :

```
postgres=# \db
          Liste des tablespaces
    Nom     | Propriétaire | Emplacement
------------+--------------+-------------
```

```
 pg_default | postgres     |
 pg_global  | postgres     |
 tbl1       | postgres     | T:\TBL1
```

```
PS P:\PGDATA13> dir 'pg_tblspc/*' | ?{$_.LinkType} | select FullName,LinkType,Target

FullName                    LinkType Target
--------                    -------- ------
P:\PGDATA13\pg_tblspc\105921 Junction {T:\TBL1}
```

Par défaut, `pg_tblspc/` est vide. N'existent alors que les tablespaces `pg_global` (sous-répertoire `global/` des objets globaux à l'instance) et `pg_default` (soit `base/`).

La création d'autres tablespaces est totalement optionnelle.

Leur utilité et leur gestion seront abordés plus loin.

1.6.8 Fichiers des statistiques d'activité

Statistiques d'activité :

- `stats collector` (≤v14) & extensions
- `pg_stat_tmp/` : temporaires
- `pg_stat/` : définitif

`pg_stat_tmp` est, jusqu'en version 15, le répertoire par défaut de stockage des statistiques d'activité de PostgreSQL, comme les entrées-sorties ou les opérations de modifications sur les tables. Ces fichiers pouvant générer une grande quantité d'entrées-sorties, l'emplacement du répertoire peut être modifié avec le paramètre `stats_temp_directory`. Par exemple, Debian place ce paramètre par défaut en `tmpfs` :

```
-- jusque v14
SHOW stats_temp_directory;
```

```
           stats_temp_directory
----------------------------------------
 /var/run/postgresql/14-main.pg_stat_tmp
```

À l'arrêt, les fichiers sont copiés dans le répertoire `pg_stat/`.

PostgreSQL gérant ces statistiques en mémoire partagée à partir de la version 15, le collecteur n'existe plus. Mais les deux répertoires sont encore utilisés par des extensions comme `pg_stat_statements` ou `pg_stat_kcache`.

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[4].

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

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

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

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

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

1.7 RÉSUMÉ

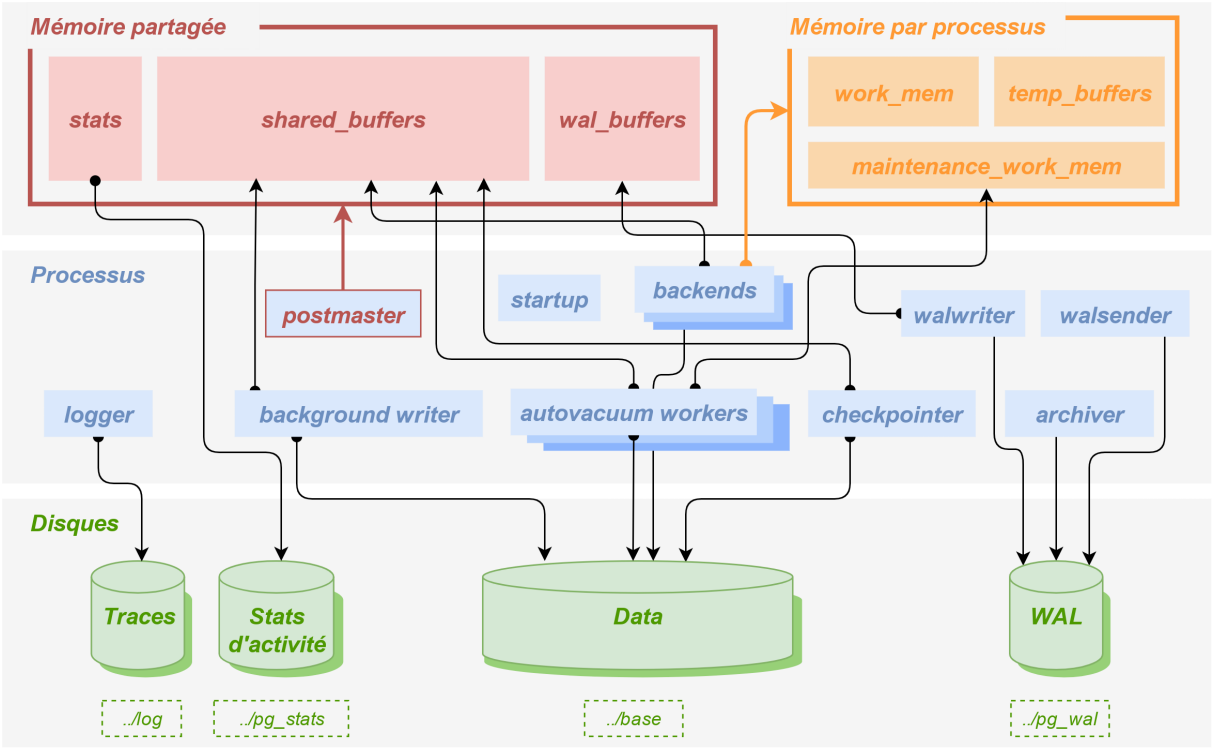


Figure 1/ .1: Architecture de PostgreSQL

Ce schéma ne soit à présent plus avoir aucun secret pour vous.

1.8 CONCLUSION

- PostgreSQL est complexe, avec de nombreux composants
- Une bonne compréhension de cette architecture est la clé d'une bonne administration.
- Pour aller (beaucoup) plus loin :

1.8.1 Questions

N'hésitez pas, c'est le moment !

1.9 QUIZ

https://dali.bo/m1_quiz

1.10 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

1.10.1 Sur Rocky Linux 8 ou 9

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

Installation du dépôt communautaire :

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

- la version majeure de PostgreSQL (ici la 17) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

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

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 17 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql17-server postgresql17-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql17-llvmjit
```

Création d'une première instance :

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

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb
# cat /var/lib/pgsql/17/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/17/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-17/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/17/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

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

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-17
# systemctl stop postgresql-17
# systemctl status postgresql-17
# systemctl reload postgresql-17
# systemctl restart postgresql-17
```

Test rapide de bon fonctionnement et connexion à psql :

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

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

```
# systemctl enable postgresql-17
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

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

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

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

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/17/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-17.service \
        /etc/systemd/system/postgresql-17-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/17/infocentre
```

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

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-17/bin/postgresql-17-setup initdb postgresql-17-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/17/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-17-infocentre
# systemctl [enable|disable] postgresql-17-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

1.10.2 Sur Debian / Ubuntu

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

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

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 17 :

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

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

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

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-17 postgresql-client-17
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/17/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/17/main`
Fichiers de configuration	dans `/etc/postgresql/17/main/`
Traces	dans `/var/log/postgresql/`

Configuration

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

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

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 17 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/17/main/start.conf`.

Ouverture du firewall :

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

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

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

Destruction d'une instance :

```
# pg_dropcluster 17 main
```

Création d'autres instances :

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

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

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

- créer une instance :

```
# pg_createcluster 17 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 17 infocentre \
  --port=12345 \
  --datadir=/PGDATA/17/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/17/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/17/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 17 infocentre start
```

1.10.3 Accès à l'instance depuis le serveur même (toutes distributions)

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

```
$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=#
```

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

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

```
local   all             postgres                                trust
```

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

```
dalibo:~$ psql -U postgres
psql (17.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (17.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (17.0)
Type "help" for help.
postgres=#
```

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

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

Ce fichier doit être protégé des autres utilisateurs :

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

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

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

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/17/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-17

root:~# pg_ctlcluster 17 main reload

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

1.11 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m1_solutions.

1.11.1 Processus

But : voir quelques processus de PostgreSQL

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

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

Se connecter à l'instance PostgreSQL.

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

Créer une nouvelle base de données nommée **b0**.

Se connecter à la base de données **b0** et créer une table `t1` avec une colonne `id` de type `integer`.

Insérer 10 millions de lignes dans la table `t1` avec :

```
INSERT INTO t1 SELECT generate_series(1, 10000000) ;
```

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

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

Redémarrer l'instance PostgreSQL.

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

Se connecter 15 fois à l'instance PostgreSQL sans fermer les sessions, par exemple en lançant plusieurs fois :

```
psql -c 'SELECT pg_sleep(1000)' &
```

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

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

1.11.2 Fichiers

But : voir les fichiers de PostgreSQL

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

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

À quelle base est lié chaque répertoire présent dans le répertoire `base` ? (Voir l'oid de la base dans `pg_database` ou l'utilitaire en ligne de commande `oid2name`)

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

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

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

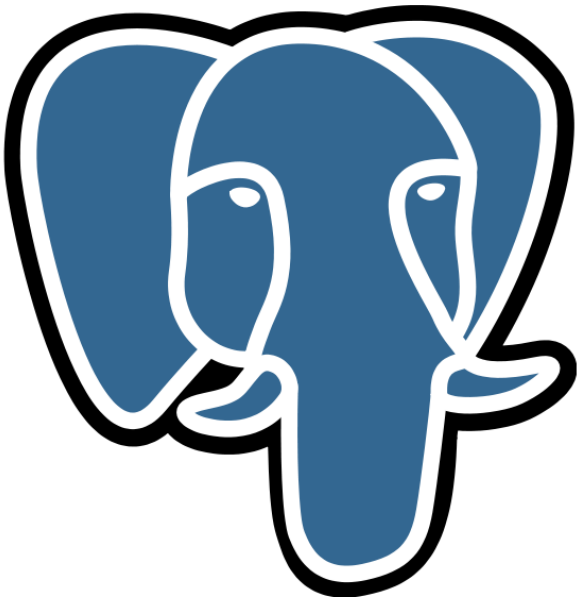
Regarder la taille du fichier correspondant à la table `t1`. Pourquoi est-il vide ?

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

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

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

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
  - risque d'incompatabilité des fichiers, avec les outils
- Tailles de bloc ou de fichier
  - `block_size` : 8 ko
  - `wal_block_size` : 8 ko
  - `segment_size` : 1 Go
  - `wal_segment_size` : 16 Mo (option `--wal-segsize` d'`initdb` en v11)

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

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

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

Recompiler avec une taille de bloc de 32 ko s'est déjà vu sur de très grosses installations (comme le rapporte par exemple Christophe Pettus (San Francisco, 2023)[1]) avec un `shared_buffers` énorme, mais cette configuration est très peu testée, nous la déconseillons dans le cas général.

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

[1] https://thebuild.com/blog/2023/02/08/xtreme-postgresql/

Des tailles non standard vous exposent à rencontrer des problèmes avec des outils s'attendant à des blocs de 8 ko. (Remontez alors le bug.)

2.3 FICHIERS DE CONFIGURATION

- `postgresql.conf` (+ fichiers inclus)
- `postgresql.auto.conf`
- `pg_hba.conf` (+ fichiers inclus (v16))
- `pg_ident.conf` (idem)

Les fichiers de configuration sont habituellement les 4 suivants :

- `postgresql.conf` : il contient une liste de paramètres, sous la forme `paramètre=valeur` ;
- `pg_hba.conf` : il contient les règles d'authentification à la base.
- `pg_ident.conf` : il complète `pg_hba.conf`, quand nous déciderons de nous reposer sur un mécanisme d'authentification extérieur à la base (identification par le système ou par un annuaire par exemple) ;
- `postgresql.auto.conf` : il stocke les paramètres de configuration fixés en utilisant la commande `ALTER SYSTEM` et surcharge donc `postgresql.conf`.

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/15/data/postgresql.conf` par exemple). Debian le place dans `/etc` (`/etc/postgresql/15/main/postgresql.conf` pour l'instance par défaut).

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

```
# SHOW config_file;

               config_file
-------------------------------------------
 /var/lib/postgresql/15/data/postgresql.conf
```

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

```
clé = valeur
```

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

```
data_directory = '/var/lib/postgresql/15/main'
listen_addresses = 'localhost'
port = 5432
shared_buffers = 128MB
```

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

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

2.4.1 Surcharge des paramètres de postgresql.conf

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

Le paramétrage ne dépend pas seulement du contenu de `postgresql.conf`.

Nous pouvons inclure d'autres fichiers depuis `postgresql.conf` grâce à l'une de ces directives :

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

Le ou les fichiers indiqués sont alors inclus à l'endroit où la directive est positionnée. Avec `include`, si le fichier n'existe pas, une erreur `FATAL` est levée ; au contraire la directive `include_if_exists` permet de ne pas s'arrêter si le fichier n'existe pas. Ces directives permettent notamment des ajustements de configuration propres à plusieurs machines d'un ensemble primaire/secondaires dont le `postgresql.conf` de base est identique, ou de gérer la configuration hors de `postgresql.conf`.

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

Ces paramètres peuvent être surchargés par le fichier `postgresql.auto.conf`, qui contient le résultat des commandes de ce type :

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

Ce fichier est principalement utilisé par les administrateurs et les outils qui n'ont pas accès au système de fichiers du serveur. (À partir de PostgreSQL 17, l'utilisation de `ALTER SYSTEM` peut être interdite, en passant le paramètre `allow_alter_system` à `off`. Le but est d'éviter des erreurs de manipulation quand la configuration est gérée par un outil extérieur, comme Ansible ou Patroni. À noter qu'un superutilisateur malveillant saura tout de même contourner cette limite.)

Si des options sont passées directement en arguments à `pg_ctl` (situation rare), elles seront prises en compte en priorité par rapport à celles de ces fichiers de configuration.

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

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

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

Ensuite, un utilisateur peut changer à volonté les valeurs de beaucoup de paramètres au sein d'une session :

```
SET parametre = valeur ;
```

ou même juste au sein d'une transaction :

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

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

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

À titre d'exemple, voici ce qu'il se passe en appliquant la requête suivante :

```
SET client_min_messages = debug2;
```

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

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

```
        name         |  source  | context  | setting  | boot_val | reset_val
---------------------+----------+----------+----------+----------+-----------
 client_min_messages | default  | user     | debug2   | notice   | notice
 log_checkpoints     | default  | sighup   | off      | off      | off
 wal_segment_size    | override | internal | 16777216 | 16777216 | 16777216
```

Nous constatons par exemple que, dans la session ayant effectué la requête, la valeur du paramètre `client_min_messages` a été modifiée à la valeur `debug2`. Nous pouvons aussi voir le contexte dans lequel le paramètre est modifiable : le `client_min_messages` est modifiable par l'utilisateur dans

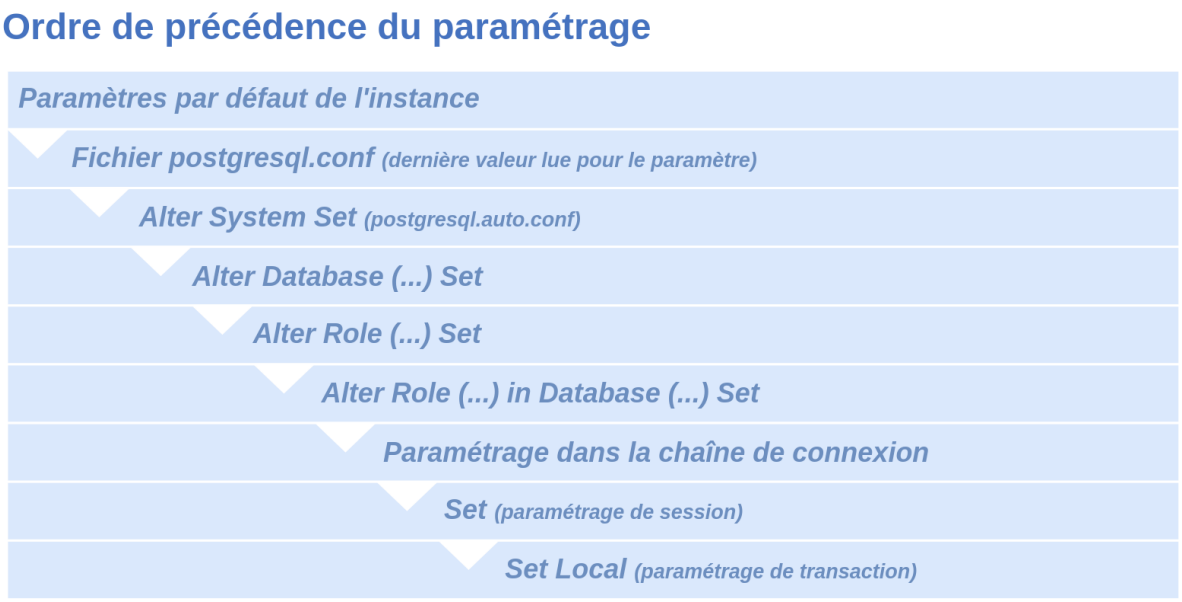
sa session. Le `log_checkpoints` seulement par `sighup`, c'est-à-dire par un `pg_ctl reload`, et le `wal_segment_size` n'est pas modifiable après l'initialisation de l'instance.

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

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

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

SELECT pg_reload_conf() ;

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

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

-[ RECORD 1 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/15/data/postgresql.conf
sourceline | 64
seqno      | 2
name       | max_connections
setting    | 100
applied    | f
error      |
-[ RECORD 2 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/15/data/postgresql.auto.conf
sourceline | 4
seqno      | 17
name       | max_connections
setting    | 200
applied    | f
error      | setting could not be applied
-[ RECORD 3 ]-------------------------------------------------
sourcefile | /var/lib/postgresql/15/data/postgresql.auto.conf
sourceline | 3
seqno      | 16
name       | work_mem
setting    | 16MB
applied    | t
error      |
```

2.4.2 Précédence des paramètres



PostgreSQL offre une certaine granularité dans sa configuration, ainsi certains paramètres peuvent être surchargés par rapport au fichier `postgresql.conf`. Il est utile de connaître l'ordre de précédence. Par exemple, un utilisateur peut spécifier un paramètre dans sa session avec l'ordre `SET`, celui-ci sera prioritaire par rapport à la configuration présente dans le fichier `postgresql.conf`.

2.4.3 Survol de postgresql.conf

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

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

La plupart des paramètres ne sont jamais modifiés. Les défauts sont souvent satisfaisants pour une petite installation. Les plus importants sont supposés acquis (au besoin, voir la formation DBA1[2]).

Les principales sections sont :

Connections and authentication

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

Resource usage (except WAL)

Cette partie fixe des limites à certaines consommations de ressources.

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

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

Write-Ahead Log

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

Tout est prévu pour faciliter la mise en place d'une réplication sans avoir besoin de modifier cette partie sur le primaire.

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

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

Replication

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

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

Query tuning

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

Reporting and logging

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

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

Autovacuum

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

Client connection defaults

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

Lock management

Les paramètres de cette section sont rarement modifiés.

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[3].

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

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)
  - tri sur disque séparé
- Utilisation selon des droits

Par défaut, PostgreSQL se charge du placement des objets sur le disque, dans son répertoire des données, mais il est possible de créer des répertoires de stockage supplémentaires, nommés *tablespaces*.

2.6.0.1 Utilité des tablespaces

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

Pour le système d'exploitation, il s'agit juste d'un répertoire, déclaré ainsi :

```
CREATE TABLESPACE ssd LOCATION '/mnt/ssd/pg';
```

L'idée est de séparer physiquement les objets suivant leur utilisation. Les cas d'utilisation des tablespaces dans PostgreSQL sont :

- l'ajout d'un disque après saturation de la partition du PGDATA sans possibilité de l'étendre au niveau du système (par LVM ou dans la baie de stockage, par exemple) ;
- la répartition des entrées-sorties… si le SAN ou la virtualisation permet encore d'agir à ce niveau ;
- et notamment la séparation des index et des tables, pour répartir les écritures ;
- le déport des fichiers temporaires vers un tablespace dédié, pour la performance ou éviter qu'ils saturent le PGDATA ;
- la séparation entre données froides et chaudes sur des disques de performances différentes, ou encore des index et des tables ;
- les quotas : PostgreSQL ne disposant pas d'un système de quotas, dédier une partition entière d'une taille précise à un tablespace est un contournement ; une transaction voulant étendre un fichier sera alors annulée avec l'erreur `cannot extend file`.

Sans un réel besoin physique, il n'y a pas besoin de créer des tablespaces, et de complexifier l'administration.

Un tablespace n'est pas adapté à une séparation logique des objets. Si vous tenez à distinguer les fichiers de chaque base sans besoin physique, rappelez-vous que PostgreSQL crée déjà un sous-répertoire par base de données dans `PGDATA/base/`.

PostgreSQL ne connaît pas de notion de tablespace en lecture seule, ni de tablespace transportable entre deux bases ou deux instances.

2.6.0.2 Emplacement des tablespaces

Il y a quelques pièges à éviter à la définition d'un tablespace :

Pour des raisons de sécurité et de fiabilité, le répertoire choisi **ne doit pas** être à la racine d'un point de montage. (Cela vaut aussi pour les répertoires PGDATA ou `pg_wal`).

Positionnez toujours les données dans un sous-répertoire, par exemple dans `/mnt/ssd/pg` plutôt que directement dans le point de montage `/mnt/ssd`. (Voir *Utilisation de systèmes de fichiers secondaires*[4] dans la documentation officielle, ou le bug à l'origine de ce conseil[5].)

Surtout, le tablespace doit **impérativement être placé hors de PGDATA**. Certains outils poseraient problème sinon.

Si ce conseil n'est pas suivi, PostgreSQL crée le tablespace mais renvoie un avertissement :

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

Il est aussi déconseillé de mettre le numéro de version de PostgreSQL dans le chemin du tablespace. PostgreSQL le gère déjà à l'intérieur du tablespace. Cela évite des incohérences dans les noms des chemins si vous migrez plus tard avec `pg_upgrade`.

[4] https://doc.postgresql.fr/current/creating-cluster.html#CREATING-CLUSTER-MOUNT-POINTS
[5] https://bugzilla.redhat.com/show_bug.cgi?id=1247477#c1

2.6.1 Tablespaces : mise en place

```
-- déclaration
CREATE TABLESPACE ssd LOCATION '/mnt/ssd/pg';
-- droit pour un utilisateur
GRANT CREATE ON TABLESPACE ssd TO un_utilisateur ;
-- pour toute une base
CREATE DATABASE nomdb TABLESPACE ssd;
ALTER  DATABASE nomdb SET default_tablespace TO ssd ;
-- pour une table
CREATE TABLE une_table (…) TABLESPACE ssd ;
ALTER  TABLE une_table SET TABLESPACE ssd ;  -- verrou !
-- pour un index (pas automatique)
ALTER INDEX une_table_i_idx SET TABLESPACE ssd ;
```

Le répertoire du tablespace doit exister et les accès ouverts et restreints à l'utilisateur système sous lequel tourne l'instance (en général **postgres** sous Linux, **Network Service** sous Windows) :

```
# /mnt/ssd/ doit exister (point de montage d'un SSD par exemple)
 # chown postgres:postgres /mnt/ssd/pg
 # chmod 700 /mnt/ssd/pg
```

Les ordres SQL plus haut permettent de :

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

Quelques choses à savoir :

- La table ou l'index est totalement verrouillé le temps du déplacement.
- Par défaut, les nouveaux index ne sont **pas** créés automatiquement dans le même tablespace que la table, mais en fonction de `default_tablespace`.
- Les index existants ne « suivent » pas automatiquement une table déplacée, il faut les déplacer séparément.
- Depuis PostgreSQL 14, il est possible de préciser un tablespace de réindexation lors d'une réindexation (`REINDEX … TABLESPACE …`).

Les tablespaces des tables sont visibles dans la vue système `pg_tables`, dans `\d+` sous psql, et dans `pg_indexes` pour les index :

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

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

2.6.2 Tablespaces : configuration

```
CREATE TABLESPACE  ssd      LOCATION '/mnt/data_ssd/' ;
CREATE TABLESPACE  ssd_tri1 LOCATION '/mnt/temp1' ;
CREATE TABLESPACE  ssd_tri2 LOCATION '/mnt/temp2' ;
GRANT CREATE ON TABLESPACE  ssd  TO dupont ;
GRANT CREATE ON TABLESPACE  ssd_tri1,ssd_tri2  TO dupont ;
```

- `default_tablespace`

```
default_tablespace = ssd   # postgresql.conf

ALTER DATABASE/ROLE nomdb SET default_tablespace = ssd ;
```

- `temp_tablespaces` :
  - tri & tables temporaires, en alternance
  - protéger le PGDATA

```
ALTER ROLE etl SET temp_tablespaces = ssd_tri1,ssd_tri2;
```

2.6.2.1 Tablespaces de données

Le paramètre `default_tablespace` permet d'utiliser un autre tablespace que celui par défaut dans PGDATA. En plus du `postgresql.conf`, il peut être défini au niveau rôle, base, ou le temps d'une session :

```
ALTER DATABASE erp_prod     SET default_tablespace TO ssd ;   -- base
ALTER DATABASE erp_archives SET default_tablespace TO froid ; -- base
ALTER ROLE etl              SET default_tablespace TO ssd ; -- niveau rôle
ALTER ROLE audit IN DATABASE erp_prod SET default_tablespace TO froid ; -- niveau
 ↪  rôle dans une base
SET default_tablespace TO ssd ; -- session
```

2.6.2.2 Tablespaces de tri

Les opérations de tri et les tables temporaires peuvent être déplacées vers un ou plusieurs tablespaces dédiés grâce au paramètre `temp_tablespaces`. Le premier intérêt est de dédier aux tris une partition rapide (SSD, disque local…). Un autre est de ne plus risquer de saturer la partition du PGDATA en cas de fichiers temporaires énormes dans `base/pgsql_tmp/`.

Ne jamais utiliser de RAM disque (comme `tmpfs`) pour des tablespaces de tri : la mémoire de la machine ne doit servir qu'aux applications et outils, au cache de l'OS, et aux tris en RAM. Favorisez ces derniers en jouant sur `work_mem`.
En cas de redémarrage, ce tablespace ne serait d'ailleurs plus utilisable. Un RAM disque est encore plus dangereux pour les tablespaces de données, bien sûr.

Il faudra ouvrir les droits aux utilisateurs ainsi :

```
GRANT CREATE ON TABLESPACE  ssd_tri1  TO  dupont ;
```

Plusieurs tablespaces temporaires peuvent être paramétrés. Noter que la déclaration se fait sans guillemet. Chaque transaction en choisira un de façon aléatoire à la création d'un objet temporaire, puis utilisera alternativement les autres pour chaque nouveau fichier. C'est un bon moyen de lisser la charge :

- lors d'une requête, un gros fichier de tri peut s'étaler sur plusieurs tablespaces de tri, car il se découpe au besoin en fichiers de 1 Go, voire moins ;
- les différentes tables temporaires d'une même session se répartiront dans les différents tablespaces temporaires ; par contre les différents fichiers d'une grosse table temporaire resteront ensemble dans le même tablespace (une table n'a qu'un tablespace), et ses index avec elle par défaut.

Si un des tablespaces temporaires sature, la requête tombe en erreur immédiatement : PostgreSQL ne regarde pas si autre tablespace temporaire a de la place libre.
Il vaut donc mieux regrouper les espaces disponibles dans un même système de fichiers, et n'avoir qu'un grand tablespace temporaire.

2.6.3 Tablespaces : performance

- Temps d'accès
  - `seq_page_cost` (1)
  - `random_page_cost` (4)
- Opérations simultanées sur le disque
  - `effective_io_concurrency` (1)
  - `maintenance_io_concurrency` (10)

```
ALTER TABLESPACE ssd SET ( random_page_cost = 1 );
ALTER TABLESPACE ssd SET ( effective_io_concurrency    = 500,
                           maintenance_io_concurrency = 500 ) ;
```

Dans le cas de disques de performances différentes, il faut adapter les paramètres concernés aux caractéristiques du tablespace si la valeur par défaut ne convient pas. Ce sont des paramètres classiques qui ne seront pas décrits en détail ici :

- `seq_page_cost` (coût d'accès à un bloc pendant un parcours, défaut 1) ;
- `random_page_cost` (coût d'accès à un bloc isolé, défaut 4) ;
- `effective_io_concurrency` (nombre d'I/O simultanées, défaut 1) et `maintenance_io_concurrency` (idem, pour une opération de maintenance, défaut 10).

Notamment :

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[6], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. Ces valeurs doivent être réduites sur un système très chargé. Une valeur excessive mène au gaspillage de CPU. Le défaut d `effective_io_concurrency` est seulement de 1, et la valeur maximale est 1000.

(Avant la version 13, le principe était le même, mais la valeur exacte de ce paramètre devait être 2 à 5 fois plus basse comme le précise la formule des notes de version de la version 13[7].)

`maintenance_io_concurrency` est similaire à `effective_io_concurrency`, mais pour les opérations de maintenance. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Il faut penser à le monter aussi si on adapte `effective_io_concurrency`.

[6] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[7] https://docs.postgresql.fr/13/release.html

Par exemple, un système paramétré pour des disques classiques aura comme paramètres par défaut :

```
random_page_cost = 4
effective_io_concurrency = 1
maintenance_io_concurrency = 10
```

et un tablespace sur un SSD les surchargera ainsi :

```
ALTER TABLESPACE ssd SET ( random_page_cost = 1 );
ALTER TABLESPACE ssd SET ( effective_io_concurrency    = 500,
                           maintenance_io_concurrency = 500 ) ;
```

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` (v14)

Il faut bien faire la distinction entre session TCP et session de PostgreSQL. Si une session TCP sert

de support à une requête particulièrement longue, laquelle ne renvoie pas de données pendant plusieurs minutes, alors le firewall peut considérer la session inactive, même si le statut du backend dans `pg_stat_activity` est `active`.

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

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

Le mécanisme de keepalive a deux intérêts :

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

Un autre cas peut survenir. Parfois, un client lance une requête. Cette requête met du temps à s'exécuter et le client quitte la session avant de récupérer les résultats. Dans ce cas, le serveur continue à exécuter la requête et ne se rendra compte de l'absence du client qu'au moment de renvoyer les premiers résultats. Depuis la version 14, il est possible d'autoriser la vérification de la connexion pendant l'exécution d'une requête. Il faut pour cela définir la durée d'intervalle entre deux vérifications avec le paramètre `client_connection_check_interval`. Par défaut, cette option est désactivée et sa valeur est de 0.

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[8].

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

2.8 STATISTIQUES SUR L'ACTIVITÉ

- (Ne pas confondre avec statistiques sur les données !)
- Statistiques consultables par des vues systèmes
- Paramètres :
  - `track_activities`, `track_activity_query_size`
  - `track_counts`, `track_io_timing` et `track_functions`
  - `update_process_title`
  - `stats_temp_directory` (< v15)

Les différents processus de PostgreSQL collectent des statistiques d'activité qui ont pour but de mesurer l'activité de la base. Notamment :

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

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

Pour des raisons de performance, ces stastistiques restent en mémoire (ou dans des fichiers temporaires jusque PostgreSQL 14) et ne sont stockées durablement qu'en cas d'arrêt propre. Un arrêt brutal implique donc leur réinitialisation !

Voici les paramètres concernés par cette collecte d'informations.

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

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

`track_activity_query_size` est la taille maximale du texte de requête pouvant être stocké dans `pg_stat_activity`. 1024 caractères est un défaut souvent insuffisant, à monter vers 10 000 si les requêtes sont longues, voire plus ; cette modification nécessite un redémarrage vu qu'elle touche au dimensionnement de la mémoire partagée.

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

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

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

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

Avant la version 15, `stats_temp_directory` servait à indiquer le répertoire de stockage temporaire des statistiques, avant copie dans `pg_stat/` lors d'un arrêt propre. Ce répertoire peut devenir gros, est réécrit fréquemment, et peut devenir source de contention. Il est conseillé de le stocker ailleurs que dans le répertoire de l'instance PostgreSQL, par exemple sur un RAM disque ou `tmpfs` (c'est le défaut sous Debian).

Ce répertoire existe toujours en version 15, notamment si vous utilisez le module `pg_stat_statements`. Cependant, en dehors de ce module, rien d'autre ne l'utilise. Quant au paramètre `stats_temp_directory`, il a disparu.

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_stat_checkpointer` (v17+)
  - `pg_locks`
- Réinitialisation des compteurs : `pg_stat_reset_shared()`

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

Statistiques sur les objets :

Pour les statistiques sur les objets, le système fournit à chaque fois trois vues différentes :

- Une pour tous les objets du type. Elle contient *all* dans le nom, `pg_statio_all_tables` par exemple ;
- Une pour uniquement les objets systèmes. Elle contient *sys* dans le nom, `pg_statio_sys_tables` par exemple ;
- Une pour uniquement les objets non-systèmes. Elle contient *user* dans le nom, `pg_statio_user_tables` par exemple.

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

```
SELECT * FROM pg_stat_user_tables WHERE relname = 'pgbench_accounts' \gx
```

```
-[ RECORD 1 ]-------+------------------------------
relid               | 200784
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 6
last_seq_scan       | 2024-11-14 11:37:16.851753+01
seq_tup_read        | 104077729
idx_scan            | 1
last_idx_scan       | 2024-11-13 15:48:32.962717+01
idx_tup_fetch       | 1319553
n_tup_ins           | 0
n_tup_upd           | 0
```

```
n_tup_del           | 0
n_tup_hot_upd       | 0
n_tup_newpage_upd   | 0
n_live_tup          | 0
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | ø
last_autovacuum     | ø
last_analyze        | ø
last_autoanalyze    | ø
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
```

Les accès physiques aux objets sont visibles dans les vues `pg_statio_xxx_indexes`, `pg_statio_xxx_tables` et `pg_statio_xxx_sequences`. Une vision plus globale est disponible dans `pg_stat_io` (apparue avec PostgreSQL 16).

```
SELECT * FROM pg_statio_user_tables WHERE relname = 'pgbench_accounts' \gx
```

```
-[ RECORD 1 ]---+-----------------
relid           | 200784
schemaname      | public
relname         | pgbench_accounts
heap_blks_read  | 2993285
heap_blks_hit   | 8720337
idx_blks_read   | 277804
idx_blks_hit    | 6114553
toast_blks_read | ø
toast_blks_hit  | ø
tidx_blks_read  | ø
tidx_blks_hit   | ø
```

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

```
SELECT * FROM pg_stat_database WHERE datname ='pgbench' \gx
```

```
-[ RECORD 1 ]-------+------------------------------
relid               | 200784
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 7
last_seq_scan       | 2024-11-14 15:25:08.632708+01
seq_tup_read        | 199954505
idx_scan            | 1001638
last_idx_scan       | 2024-11-14 15:33:15.346497+01
idx_tup_fetch       | 38260013
n_tup_ins           | 0
n_tup_upd           | 91491186
n_tup_del           | 0
n_tup_hot_upd       | 304994
```

```
n_tup_newpage_upd   | 91186192
n_live_tup          | 98976910
n_dead_tup          | 16426
n_mod_since_analyze | 1481762
n_ins_since_vacuum  | 0
last_vacuum         | 2024-11-14 14:41:20.893236+01
last_autovacuum     | 2024-11-14 16:06:01.192891+01
last_analyze        | 2024-11-14 14:41:45.544659+01
last_autoanalyze    | ø
vacuum_count        | 1
autovacuum_count    | 1
analyze_count       | 1
autoanalyze_count   | 0
```

`pg_stat_bgwriter` stocke les statistiques d'écriture des buffers des *background writer*, et du *checkpointer* (jusqu'en version 16 incluse) et des sessions elles-mêmes. On peut ainsi voir si les *backends* écrivent beaucoup ou peu. À partir de PostgreSQL 17, apparaît `pg_stat_checkpointer` qui reprend les champs sur les *checkpoints* et en ajoute quelques-uns. Cette vue permet de vérifier que les *checkpoints* sont réguliers, donc peu gênants.

Exemple (version 17) :

```
TABLE pg_stat_bgwriter \gx

-[ RECORD 1 ]----+------------------------------
buffers_clean    | 3004
maxwritten_clean | 26
buffers_alloc    | 24399160
stats_reset      | 2024-11-05 15:12:27.556173+01

TABLE pg_stat_checkpointer \gx

-[ RECORD 1 ]-------+------------------------------
num_timed           | 282
num_requested       | 2
restartpoints_timed | 0
restartpoints_req   | 0
restartpoints_done  | 0
write_time          | 605908
sync_time           | 3846
buffers_written     | 20656
stats_reset         | 2024-11-05 15:12:27.556173+01
```

Écritures :

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

Requêtes

`pg_stat_activity` est une des vues les plus utilisées et est souvent le point de départ d'une recherche. Elle donne la liste des processus en cours sur l'instance, en incluant entre autres :

- le numéro de processus sur le serveur (`pid`) ;
- la base de données, le nom dutilisateur, l'adresse et le port du client ;

- les dates de début d'ordre, de transaction ou de session ;
- son statut (active ou non) ;
- la requête en cours, ou la dernière requête si la session ne fait rien ;
- le nom de l'application s'il a été renseigné avec le paramètre `application_name` ;
- le type de processus : session d'un utilisateur (*client backend*), processus interne…

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

-[ RECORD 1 ]----+------------------------------------------------------------
datname          | ¤
pid              | 26378
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236776+02
state            | ¤
backend_type     | autovacuum launcher
query            |
-[ RECORD 2 ]----+------------------------------------------------------------
datname          | ¤
pid              | 26380
usename          | postgres
application_name |
backend_start    | 2019-10-24 18:25:28.238157+02
state            | ¤
backend_type     | logical replication launcher
query            |
-[ RECORD 3 ]----+------------------------------------------------------------
datname          | pgbench
pid              | 22324
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.167611+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + -3810 WHERE…
-[ RECORD 4 ]----+------------------------------------------------------------
datname          | postgres
pid              | 22429
usename          | postgres
application_name | psql
backend_start    | 2019-10-28 10:27:09.599426+01
state            | active
backend_type     | client backend
query            | select datname, pid, usename, application_name, backend_start…
-[ RECORD 5 ]----+------------------------------------------------------------
datname          | pgbench
pid              | 22325
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.172585+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 4360 WHERE…
```

```
-[ RECORD 6 ]----+-------------------------------------------------------------
datname          | pgbench
pid              | 22326
usename          | test_performance
application_name | pgbench
backend_start    | 2019-10-28 10:26:51.178514+01
state            | active
backend_type     | client backend
query            | UPDATE pgbench_accounts SET abalance = abalance + 2865 WHERE…
-[ RECORD 7 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26376
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235574+02
state            | ¤
backend_type     | background writer
query            |
-[ RECORD 8 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26375
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.235064+02
state            | ¤
backend_type     | checkpointer
query            |
-[ RECORD 9 ]----+-------------------------------------------------------------
datname          | ¤
pid              | 26377
usename          | ¤
application_name |
backend_start    | 2019-10-24 18:25:28.236239+02
state            | ¤
backend_type     | walwriter
query            |
```

Les textes des requêtes sont tronqués à 1024 caractères : c'est un problème courant. Il est conseillé de monter le paramètre `track_activity_query_size` à plusieurs kilooctets.

Cette vue fournit aussi les *wait events*, qui indiquent ce qu'une session est en train d'attendre. Cela peut être très divers et inclut la levée d'un verrou sur un objet, celle d'un verrou interne, la fin d'une entrée-sortie… L'absence de *wait event* indique que la requête s'exécute. À noter qu'une session avec un *wait event* peut rester en statut `active`.

Les détails sur les champs `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente) sont disponibles dans le tableau des événements d'attente[9]. de la documentation.

À partir de PostgreSQL 17, la vue `pg_wait_events` peut être directement jointe à `pg_stat_activity`, et son champ `description` évite d'aller voir la documentation :

[9] https://docs.postgresql.fr/current/monitoring-stats.html#WAIT-EVENT-TABLE

```
SELECT datname, application_name, pid,
       wait_event_type, wait_event, query, w.description
FROM pg_stat_activity a
  LEFT OUTER JOIN pg_wait_events w
  ON (a.wait_event_type = w.type AND a.wait_event = w.name)
WHERE backend_type='client backend'
AND   wait_event IS NOT NULL
ORDER BY wait_event DESC LIMIT 4 \gx

-[ RECORD 1 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786146
wait_event_type  | LWLock
wait_event       | WALWriteLock
query            | UPDATE pgbench_accounts SET abalance = abalance + 4055 WHERE…
description      | ø
-[ RECORD 2 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786190
wait_event_type  | IO
wait_event       | WalSync
query            | UPDATE pgbench_accounts SET abalance = abalance + -1859  WHERE…
description      | Waiting for a WAL file to reach durable storage
-[ RECORD 3 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786145
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 3553 WHERE…
description      | Waiting for a read from a relation data file
-[ RECORD 4 ]----+-----------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786143
wait_event_type  | IO
wait_event       | DataFileRead
query            | UPDATE pgbench_accounts SET abalance = abalance + 1929 WHERE…
description      | Waiting for a read from a relation data file
```

Le processus de la ligne 2 attend une synchronisation sur disque du journal de transaction (WAL), et les deux suivants une lecture d'un fichier de données. (La description vide en ligne 1 est un souci de la version 17.2).

Pour entrer dans le détail des champs liés aux connexions :

- `backend_type` est le type de processus : on filtrera généralement sur `client backend`, mais on y trouvera aussi des processus de tâche de fond comme `checkpointer`, `walwriter`, `autovacuum launcher` et autres processus de PostgreSQL, ou encore des *workers* lancés par des extensions ;
- `datname` est le nom de la base à laquelle la session est connectée, et `datid` est son identifiant (OID) ;

- `pid` est le processus du *backend*, c'est-à-dire du processus PostgreSQL chargé de discuter avec le client, qui durera le temps de la session (sauf parallélisation) ;
- `usename` est le nom de l'utilisateur connecté, et `usesysid` est son OID dans `pg_roles` ;
- `application_name` est un nom facultatif, et il est recommandé que l'application cliente le renseigne autant que possible avec `SET application_name TO 'nom_outil_client'` ;
- `client_addr` est l'adresse IP du client connecté (`NULL` si connexion sur socket Unix), et `client_hostname` est le nom associé à cette IP, renseigné uniquement si `log_hostname` a été passé à `on` (cela peut ralentir les connexions à cause de la résolution DNS) ;
- `client_port` est le numéro de port sur lequel le client est connecté, toujours s'il s'agit d'une connexion IP.

Une requête parallélisée occupe plusieurs processus, et apparaîtra sur plusieurs lignes de `pid` différents. Le champ `leader_pid` indique le processus principal. Les autres processus disparaîtront dès la requête terminée.

Pour les champs liés aux durées de session, transactions et requêtes :

- `backend_start` est le timestamp de l'établissement de la session ;
- `xact_start` est le timestamp de début de la transaction ;
- `query_start` est le timestamp de début de la requête en cours, ou de la dernière requête exécutée ;
- `status` vaut soit `active`, soit `idle` (la session ne fait rien) soit `idle in transaction` (en attente pendant une transaction) ;
- `backend_xid` est l'identifiant de la transaction en cours, s'il y en a une ;
- `backend_xmin` est l'horizon des transactions visibles, et dépend aussi des autres transactions en cours.

Rappelons qu'une session durablement en statut `idle in transaction` bloque le fonctionnement de l'autovacuum car `backend_xmin` est bloqué. Cela peut mener à des tables fragmentées et du gaspillage de place disque.

Depuis PostgreSQL 14, `pg_stat_activity` peut afficher un champ `query_id`, c'est-à-dire un identifiant de requête normalisée (dépouillée des valeurs de paramètres). Il faut que le paramètre `compute_query_id` soit à `on` ou `auto` (le défaut, et alors une extension peut l'activer). Ce champ est utile pour retrouver une requête dans la vue de l'extension `pg_stat_statements`, par exemple.

Certains champs de cette vue ne sont renseignés que si le paramètre `track_activities` est à `on` (valeur par défaut, qu'il est conseillé de laisser ainsi).

À noter qu'il ne faut pas interroger `pg_stat_activity` au sein d'une transaction, son contenu pourrait sembler figé.

Verrous :

`pg_locks` permet de voir les verrous posés sur les objets (principalement les relations comme les tables et les index). Le processus (`pid`) est la clé commune pour la rapprocher de `pg_stat_activity`.

Archivage et réplication :

`pg_stat_archiver` donne des informations sur l'archivage des journaux de transaction et notamment sur les erreurs d'archivage. L'exemple suivant montre un archivage en erreur :

```
TABLE pg_stat_archiver \gx
```

```
-[ RECORD 1 ]------+------------------------------
archived_count     | 1637
last_archived_wal  | 000000010000000700000E3
last_archived_time | 2024-11-14 16:00:00.418887+01
failed_count       | 13254
last_failed_wal    | 000000010000000700000E4
last_failed_time   | 2024-11-14 16:01:37.347793+01
stats_reset        | 2024-11-05 14:58:00.515774+01
```

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

Si les réplications se font par des slots de réplication (optionnels en réplication physique), `pg_stat_replication_slots` donne des informations sur leur état et leur retard.

Autres vues :

Des vues plus spécialisées existent :

`pg_stat_ssl` donne des informations sur les connexions SSL : version SSL, suite de chiffrement, nombre de bits pour l'algorithme de chiffrement, compression, Distinguished Name (DN) du certificat client.

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

`pg_stat_slru` permet de suivre les accès à différents petits caches internes de PostgreSQL (à partir de PostgreSQL 17).

Réinitialisation :

Ces vues contiennent des compteurs cumulatifs. L'évolution en fonction du temps est souvent gérée par les nombreux outils qui font appel à ces vues. Il existe une fonction pour réinitialiser les compteurs de certaines vues (pas toutes) :

```
SELECT pg_stat_reset_shared('archiver') ;
```

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
- Affiner :
  - Échantillonnage

```
ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300 ;
```

  - Statistiques multicolonnes sur demande

```
CREATE STATISTICS nom ON champ1, champ2… FROM nom_table ;
```

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

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

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

Les statistiques sont collectées sur :

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

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

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

La vue `pg_stats` affiche les statistiques collectées :

```
\d pg_stats
```

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

- `inherited` : la statistique concerne-t-elle un objet utilisant l'héritage (table parente, dont héritent plusieurs tables) ;
- `null_frac` : fraction d'enregistrements dont la colonne vaut NULL ;
- `avg_width` : taille moyenne de cet attribut dans l'échantillon collecté ;
- `n_distinct` : si positif, c'est le nombre de valeurs distinctes ; si négatif, c'est la fraction de valeurs distinctes pour cette colonne dans la table. Il est possible de forcer la valeur de ce champ s'il est constaté que la collecte des statistiques le calcule mal. Par exemple, pour indiquer à l'optimiseur que chaque valeur apparaît statistiquement deux fois :

```
ALTER TABLE matable ALTER COLUMN yyy SET (n_distinct = -0.5) ;
ANALYZE matable ;
```

- `most_common_vals` et `most_common_freqs` : les valeurs les plus fréquentes de la table, et leur fréquence. Le nombre de valeurs collectées est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target`. Le défaut de 100 échantillons sur 30 000 lignes peut être modifié comme ci-après (sachant que le temps de planification augmente exponentiellement avec ce paramètre, et qu'il vaut mieux ne pas dépasser la valeur 1000) :

```
ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300 ;
```

- `histogram_bounds` : les limites d'histogramme sur la colonne. Les histogrammes permettent d'évaluer la sélectivité d'un filtre par rapport à sa valeur précise. Ils permettent par exemple à l'optimiseur de déterminer que 4,3 % des enregistrements d'une colonne `noms` commencent par un A, ou 0,2 % par AL. Le principe est de regrouper les enregistrements triés dans des groupes de tailles approximativement identiques, et de stocker les limites de ces groupes (on ignore les `most_common_vals`, pour lesquelles il y a déjà une mesure plus précise). Le nombre d'`histogram_bounds` est calculé de la même façon que les `most_common_vals` ;
- `correlation` : le facteur de corrélation statistique entre l'ordre physique et l'ordre logique des enregistrements de la colonne. Il vaudra par exemple `1` si les enregistrements sont physiquement stockés dans l'ordre croissant, `-1` si ils sont dans l'ordre décroissant, ou `0` si ils sont totalement aléatoirement répartis. Ceci sert à affiner le coût d'accès aux enregistrements ;

- `most_common_elems` et `most_common_elems_freqs` : les valeurs les plus fréquentes si la colonne est un tableau (NULL dans les autres cas), et leur fréquence. Le nombre de valeurs collectées est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target` ;
- `elem_count_histogram` : les limites d'histogramme sur la colonne si elle est de type tableau.

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

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 classique à plateaux. Ceci prend déjà en considération un potentiel effet du cache. Sur une base fortement en cache, il est donc possible d'être tenté d'abaisser le `random_page_cost` à 3, voire 2,5, ou des valeurs encore bien moindres dans le cas de bases totalement en mémoire.

Le cas des disques SSD est particulièrement intéressant. Ces derniers n'ont pas à proprement parler

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

```
# pour un SSD
seq_page_cost = 1
random_page_cost = 1.1
```

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Seuls les parcours *Bitmap Scan* sont impactés par ce paramètre. Selon la documentation[10], pour un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1, n-1 s'il s'agit d'un RAID 5 ou 6, n/2 s'il s'agit d'un RAID 10). Avec du SSD, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. Ces valeurs doivent être réduites sur un système très chargé. Une valeur excessive mène au gaspillage de CPU. Le défaut d `effective_io_concurrency` est seulement de 1, et la valeur maximale est 1000.

(Avant la version 13, le principe était le même, mais la valeur exacte de ce paramètre devait être 2 à 5 fois plus basse comme le précise la formule des notes de version de la version 13[11].)

Le paramètre `maintenance_io_concurrency` a le même sens que `effective_io_concurrency`, mais pour les opérations de maintenance. Celles-ci peuvent ainsi se voir accorder plus de ressources qu'une simple requête. Le défaut est de 10, et il faut penser à le monter aussi comme `effective_io_concurrency` sur des disques plus rapides qu'un simple disque magnétique.

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

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

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

Certaines autres informations permettent de nuancer les valeurs précédentes. `effective_cache_size` est la taille totale du cache. Il permet à PostgreSQL de modéliser plus finement le coût réel d'une opération disque, en prenant en compte la probabilité que cette information se trouve dans le cache du système d'exploitation ou dans celui de l'instance, et soit donc moins coûteuse à accéder.

Le parcours de l'espace des solutions est un parcours exhaustif. Sa complexité est principalement liée au nombre de jointures de la requête et est de type exponentiel. Par exemple, planifier de façon ex-

[10] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[11] https://docs.postgresql.fr/13/release.html

haustive une requête à une jointure dure 200 microsecondes environ, contre 7 secondes pour 12 jointures. Une autre stratégie, l'optimiseur génétique, est donc utilisée pour éviter le parcours exhaustif quand le nombre de jointure devient trop élevé.

Pour plus de détails, voir l'article sur les coûts de planification[12] issu de notre base de connaissance.

2.10.2 Nombre de tables considérées par le planificateur

- Réordonne les tables :
  - `join_collapse_limit`
  - `from_collapse_limit`
  - défaut 8, parfois besoin de plus
  - attention au temps de planification, selon le besoin
- GEQO :
  - optimiseur génétique
  - rapide, mais non optimal
  - `geqo` & `geqo_threshold` (≥ 12 tables)

Les paramètres suivants peuvent être modifiés par session.

Nombre de tables considérées :

Les paramètres `join_collapse_limit` et `from_collapse_limit` sont trop peu connus, mais peuvent améliorer radicalement les performances si vous joignez souvent plus de huit tables.

En principe, le planificateur ne tient pas compte de l'ordre des tables dans la clause `FROM` et peut décider de commencer la requête par n'importe laquelle. Cependant, la complexité des plans d'exécution à étudier explose avec le nombre de tables et de combinaisons de jointures, avec toutes leurs possibilités. Les paramètres `from_collapse_limit` et `join_collapse_limit` définissent le nombre de tables prises en compte à la fois.

- `join_collapse_limit` (à 8 par défaut) définit le nombre de tables prises en compte dans le `FROM` (vision simplifiée). S'il y a plus de tables, elles seront jointes au premier résultat dans un deuxième temps. Cela peut donner des résultats catastrophiques si le critère de filtrage le plus utile est sur la neuvième table du `FROM` !

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

- `from_collapse_limit` remonte les tables dans une sous-requête dans le `FROM`, pourvu de ne pas dépasser cette valeur. On le définit habituellement à la même valeur que `join_collapse_limit`.

Comme exemple de plan désastreux à cause d'un critère de filtrage sur une table non prise en compte, voir ce plan[13]. Monter `join_collapse_limit` de 8 à 9 bascule vers un plan bien meilleur[14]. Une autre solution aurait été de remonter la table `INNER JOIN contacts`, qui porte le critère, plus haut dans le `FROM`. Mais il n'est pas toujours possible de modifier le code, qui d'ailleurs peut être dynamique.

Pour éviter de se soucier de ce problème, il est fréquent de monter les valeurs de `join_collapse_limit` et `from_collapse_limit` à 10 ou 12 quand on utilise parfois autant de tables. Des valeurs plus élevées (jusque 20 ou plus) sont plus dangereuses, car le temps de planification peut monter très haut (centaines de millisecondes, voire pire). Mais elles se justifient dans certains domaines. Il est alors préférable de positionner ces valeurs élevées uniquement au niveau de la session, de l'écran, ou de l'utilisateur avec `SET` ou `ALTER ROLE … SET …`.

À l'inverse, descendre `join_collapse_limit` à 1 permet de dicter l'ordre d'exécution au planificateur[15], une mesure de dernier recours.

Parallèment, il ne sera pas inutile de juger de la pertinence d'une requête avec autant de tables.

Optimiseur génétique :

Ce qui suit suppose que `join_collapse_limit` dépasse 12.

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

Le code source de PostgreSQL décrit le principe[16], résumé aussi dans ce schéma :

[13] https://explain.dalibo.com/plan/D0
[14] https://explain.dalibo.com/plan/EQN
[15] https://docs.postgresql.fr/current/explicit-joins.html
[16] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f

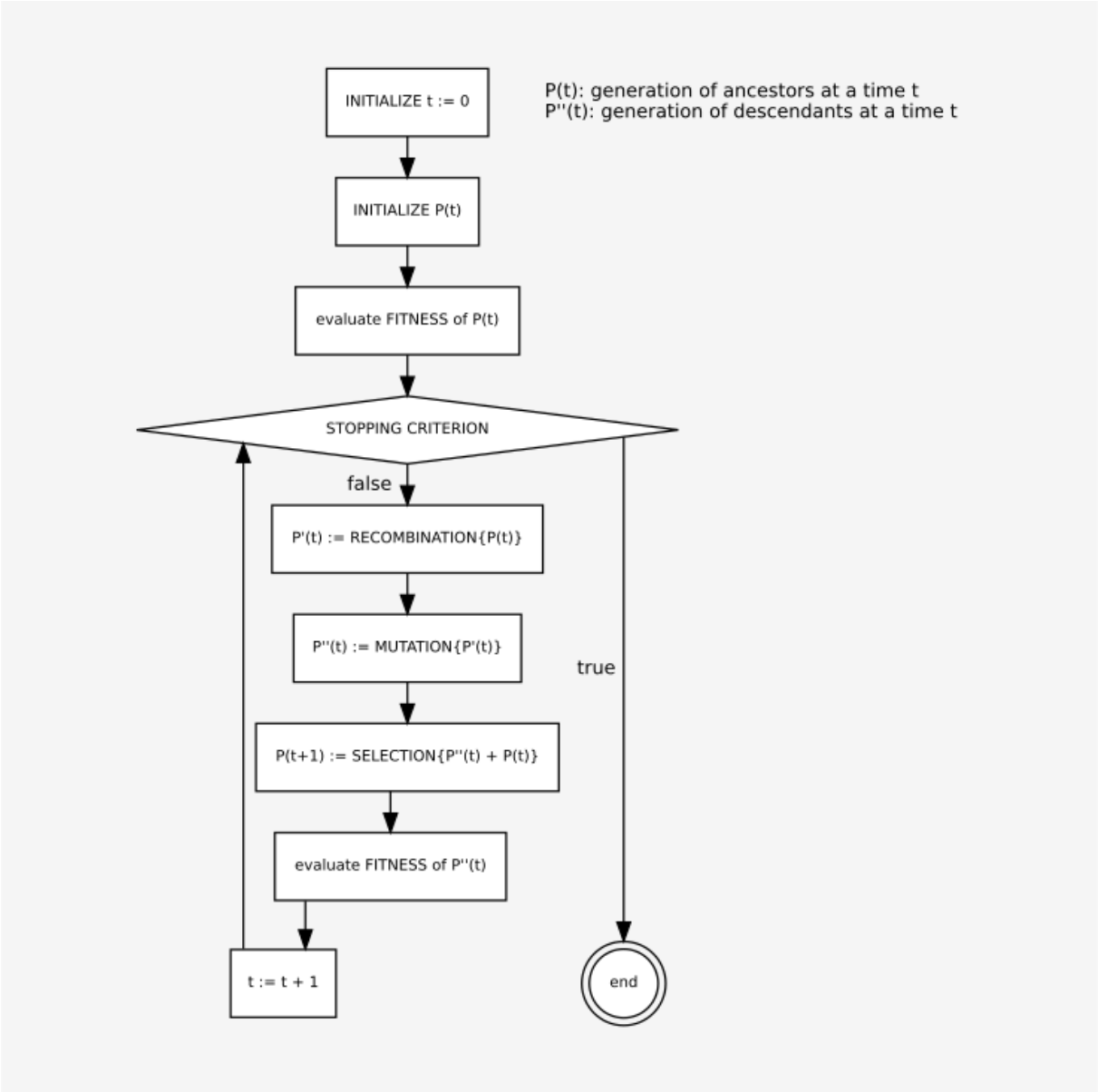


Figure 2/ .1: Principe d'un algorithme génétique (schéma de la documentation officielle, licence PostgreSQL)

Ce mécanisme est configuré par des paramètres dont le nom commence par `geqo`. Exceptés ceux ci-dessous, il est déconseillé d'y toucher sans une bonne connaissance des algorithmes génétiques.

- `geqo`, par défaut à `on`, permet d'activer/désactiver GEQO ;
- `geqo_threshold`, par défaut à 12, est le nombre d'éléments minimum à joindre dans un `FROM` avant d'optimiser celui-ci par GEQO au lieu du planificateur exhaustif ;
- `geqo_seed` (la « graine » pour la génération aléatoire, 0 par défaut) permet de forcer la re-

cherche d'autres plans.

À `geqo_seed` identique, les plans générés seront toujours les mêmes[17] (toutes autres choses identiques par ailleurs).

En production, on ne montera pas `geqo_threshold` ou à peine, et en limitant dans une session. Le nombre de plans à étudier sans l'optimisation du GEQO peut devenir colossal et consommer beaucoup de RAM et de CPU.

2.10.3 Paramètres supplémentaires de l'optimiseur

- Requêtes préparées
  - `plan_cache_mode`
- Partitionnement
  - `constraint_exclusion`
  - `enable_partition_pruning`
- Curseurs
  - `cursor_tuple_fraction`
- Mutualiser les entrées-sorties
  - `synchronize_seqscans`

Tous les paramètres suivants peuvent être modifiés par session.

Requêtes préparées :

Pour les requêtes préparées, par défaut l'optimiseur génère des plans personnalisés pour les cinq premières exécutions d'une requête préparée, puis il bascule sur un plan générique dès que celui-ci devient plus intéressant que la moyenne des plans personnalisés précédents. Le paramètre de configuration `plan_cache_mode` permet de modifier cela :

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

[17] https://www.postgresql.org/docs/current/geqo-pg-intro.html#GEQO-PG-INTRO-GEN-POSSIBLE-PLANS

Partitionnement :

Avant la version 10, PostgreSQL ne connaissait qu'un partitionnement par héritage, où l'on crée une table parente et des tables filles héritent de celle-ci, possédant des contraintes `CHECK` comme critères de partitionnement, par exemple `CHECK (date >='2011-01-01' and date < '2011-02-01')` pour une table fille d'un partitionnement par mois. Afin que PostgreSQL ne parcourt que les partitions correspondant à la clause `WHERE` d'une requête, le paramètre `constraint_exclusion` doit valoir `partition` (la valeur par défaut) ou `on`. `partition` est moins coûteux dans un contexte d'utilisation classique car les contraintes d'exclusion ne seront examinées que dans le cas de requêtes `UNION ALL`, qui sont les requêtes générées par le partitionnement.

Pour le partitionnement déclaratif (à favoriser à partir de PostgreSQL 10), `enable_partition_pruning`, activé par défaut, est le paramètre équivalent.

Curseurs :

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

Synchronisation des parcours de table :

Quand plusieurs requêtes souhaitent accéder séquentiellement à la même table, les processus se rattachent à ceux déjà en cours de parcours, afin de profiter des entrées-sorties que ces processus effectuent, le but étant que le système se comporte comme si un seul parcours de la table était en cours, et réduise donc fortement la charge disque. Le seul problème de ce mécanisme est que les processus se rattachant ne parcourent pas la table dans son ordre physique : elles commencent leur parcours de la table à l'endroit où se trouve le processus auquel elles se rattachent, puis rebouclent sur le début de la table. Les résultats n'arrivent donc pas forcément toujours dans le même ordre, ce qui n'est normalement pas un problème (on est censé utiliser `ORDER BY` dans ce cas). Mais il est toujours possible de désactiver ce mécanisme en passant `synchronize_seqscans` à `off`.

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

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m2_solutions.

2.13.1 Tablespace

But : Ajouter un tablespace

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

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

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

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

2.13.2 Statistiques d'activités, tables et vues système

But : Consulter les statistiques d'activité

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

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

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

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

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

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

Pour voir les sessions connectées :

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

2.13.3 Statistiques sur les données

But : Consulter les statistiques sur les données

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

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

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

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

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

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

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

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

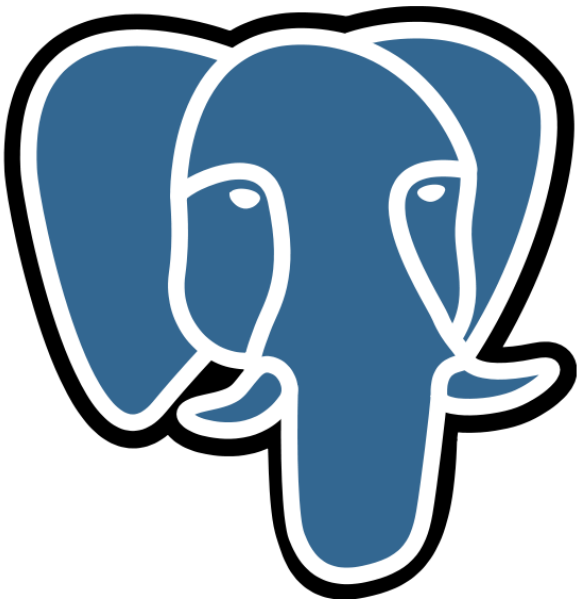
Modifier le contenu de la table `t4` avec `UPDATE t4 SET id = 100000;`

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

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

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

3/ Mémoire et journalisation dans PostgreSQL

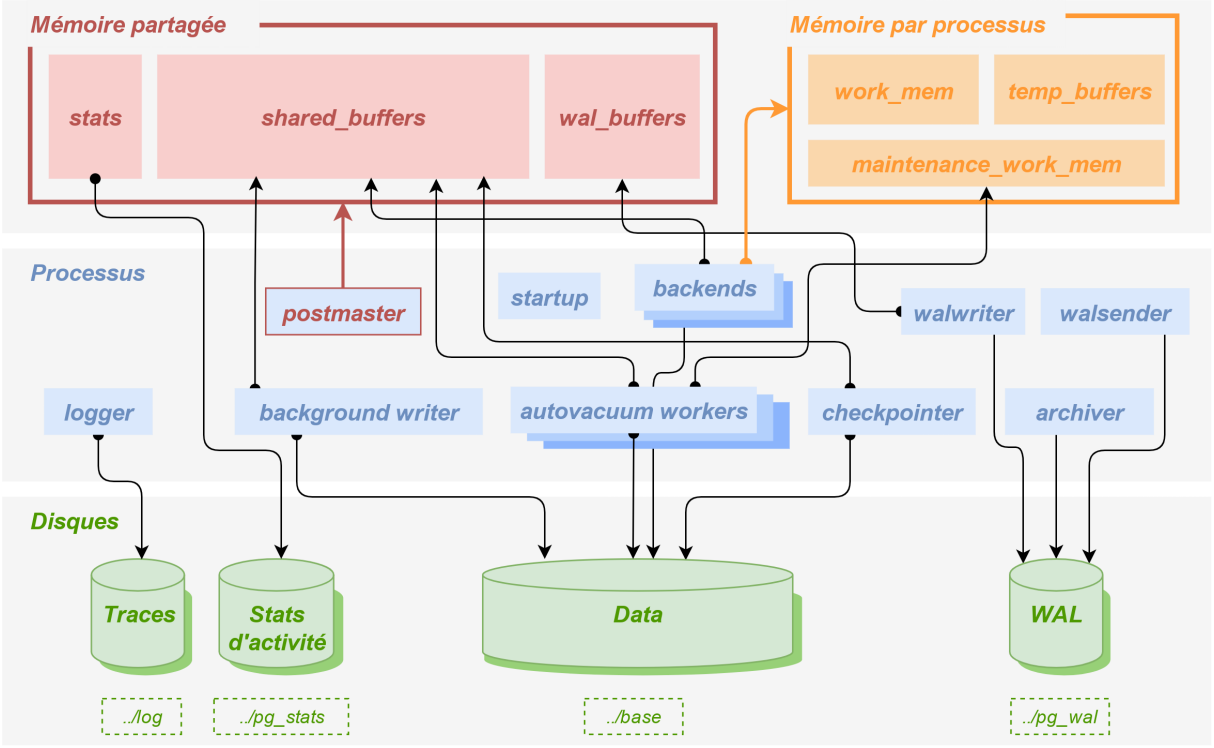


3.1 AU MENU

La mémoire & PostgreSQL :

- Mémoire partagée
- Mémoire des processus
- Les *shared buffers* & la gestion du cache
- La journalisation

3.2 RAPPEL DE L'ARCHITECTURE DE POSTGRESQL



3.3 MÉMOIRE PARTAGÉE

3.3.1 Zones de la mémoire partagée

- `shared_buffers`
  - cache disque des fichiers de données
- `wal_buffers`
  - cache disque des journaux de transactions
- `max_connections`
  - 100… ou plus ?
- `track_activity_query_size`
  - à monter
- verrous
  - `max_connections`, `max_locks_per_transaction`
- SLRU, etc…
- Modification → redémarrage

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

Les principales zones de mémoire partagées décrites ici sont fixes, et les tailles calculées en fonction de paramètres. Nous verrons en détail l'utilité de certaines de ces zones dans les chapitres suivants.

Shared buffers :

Les *shared buffers* sont le cache des fichiers de données présents sur le disque. Ils représentent de loin la volumétrie la plus importante.

Paramètre associé : `shared_buffers` (à adapter systématiquement)

Wal buffers :

Les *wal buffers* sont le cache des journaux de transaction.

Paramètre associé : `wal_buffers` (rarement modifié)

Données liées aux sessions :

Cet espace mémoire sert à gérer les sessions ouvertes, celles des utilisateurs, mais aussi celles ouvertes par de nombreux processus internes.

Principaux paramètres associés :

- `max_connections`, soit le nombre de connexions simultanées possibles (défaut : 100, souvent suffisant mais à adapter) ;
- `track_activity_query_size` (défaut : 1024 ; souvent monté à 10 000 s'il y a des requêtes de grande taille) ;

Verrous :

La gestion des verrous est décrite dans un autre module[1]. Lors des mises à jour ou suppressions dans les tables, les verrous sur les lignes sont généralement stockés dans les lignes mêmes ; mais de nombreux autres verrous sont stockés en mémoire partagée. Les paramètres associés pour le dimensionnement sont surtout :

- `max_connections` à nouveau ;
- `max_locks_per_transaction`, soit le nombre de verrous possible pour une transaction (défaut : 64, généralement suffisant) ;
- `max_pred_locks_per_relation`, nombre de verrous possible pour une table si le niveau d'isolation « sérialisation » est choisi ;
- mais encore les paramètres liés aux nombres de divers processus internes, ou de transactions préparées.

Les SLRU :

Les SLRU (*Simple Least Recently Used Buffers*) sont de petits caches pour certaines métadonneés de diverses natures : transactions et sous-transactions en cours, horodatage des *commits*, notifications par `NOTIFY`…

Le mécanisme a été revu en PostgreSQL 17 et, depuis, il est possible de modifier la taille des SLRU avec les paramètres `commit_timestamp_buffers`, `multixact_member_buffers`, `multixact_offset_buffers`, `notify_buffers`, `serializable_buffers`, `subtransaction_buffers` et `transaction_buffers`.

Les valeurs par défaut ne font parfois que 16 blocs. Les augmenter peut être intéressant si l'on repère une contention sur l'un d'eux. Un indice est la présence récurrente de *wait events* avec « SLRU » dans le nom. (Pour la liste des *wait events*, voir la table `pg_wait_events`, à partir de la version 17 également. Ils se voient dans `pg_stat_activity`. Il existe aussi une vue `pg_stat_slru`.)

Modification :

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

[1] https://dali.bo/m4_html#verrouillage-et-mvcc

3.3.2 Taille de la mémoire partagée

```
-- v15+
SHOW shared_memory_size ;
SHOW shared_memory_size_in_huge_pages ;
```

À partir de la version 15, le paramètre `shared_memory_size` permet de connaître la taille complète de mémoire partagée allouée (un peu supérieure à `shared_buffers` en pratique). Dans le cadre de l'utilisation des *Huge Pages*, il est possible de consulter le paramètre `shared_memory_size_in_huge_pages` pour connaître le nombre de pages mémoires nécessaires (mais on ne peut savoir ici si elles sont utilisées) :

```
postgres=# \dconfig shared*
              Liste des paramètres de configuration
            Paramètre             |             Valeur
----------------------------------+---------------------------------
 shared_buffers                   | 12GB
 shared_memory_size               | 12835MB
 shared_memory_size_in_huge_pages | 6418
 ...
```

Des zones de mémoire partagée non statiques peuvent exister : par exemple, à l'exécution d'une requête parallélisée, les processus impliqués utilisent de la mémoire partagée dynamique. Depuis PostgreSQL 14, une partie peut être pré-allouée avec le paramètre `min_dynamic_shared_memory` (0 par défaut).

3.4 MÉMOIRE PAR PROCESSUS

3.4.1 work_mem, maintenance_work_mem

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

Les processus de PostgreSQL ont accès à la mémoire partagée, définie principalement par `shared_buffers`, mais ils ont aussi leur mémoire propre. Cette mémoire n'est utilisable que par le processus l'ayant allouée.

- Le paramètre le plus important est `work_mem`, qui définit la taille maximale de la mémoire de travail d'un `ORDER BY`, de certaines jointures, pour la déduplication… que peut utiliser un processus sur un nœud de requête, principalement lors d'opérations de tri ou regroupement.
- Autre paramètre capital, `maintenance_work_mem` qui est la mémoire utilisable pour les opérations de maintenance lourdes : `VACUUM`, `CREATE INDEX`, `REINDEX`, ajouts de clé étrangère…

Cette mémoire liée au processus est rendue immédiatement après la fin de l'ordre concerné.

- Il existe aussi `logical_decoding_work_mem` (défaut : 64 Mo), utilisable pour chacun des flux de réplication logique (s'il y en a, ils sont rarement nombreux).

Opérations de maintenance & maintenance_work_mem :

`maintenance_work_mem` peut être monté à 256 Mo à 1 Go, voire plus sur les machines récentes, car il concerne des opérations lourdes (indexation, nettoyage des index par `VACUUM`…). Leurs consommations de RAM s'additionnent, mais en pratique ces opérations sont rarement exécutées plusieurs fois simultanément.

Monter au-delà de 1 Go n'a d'intérêt que pour la création ou la réindexation de très gros index.

`autovacuum_work_mem` est la mémoire que s'autorise à prendre l'autovacuum pour les nettoyages d'index, et ce pour chaque *worker* de l'autovacuum (3 maximum par défaut). Par défaut, ce paramètre reprend `maintenance_work_mem`, et est généralement laissé tel quel.

Paramétrage de work_mem :

Pour `work_mem`, c'est beaucoup plus compliqué.

Si `work_mem` est trop bas, beaucoup d'opérations de tri, y compris nombre de jointures, ne s'effectueront pas en RAM. Par exemple, si une jointure par hachage impose d'utiliser 100 Mo en mémoire, mais que `work_mem` vaut 10 Mo, PostgreSQL écrira des dizaines de Mo sur disque à chaque appel de la jointure. Si, par contre, le paramètre `work_mem` vaut 120 Mo, aucune écriture n'aura lieu sur disque, ce qui accélérera forcément la requête.

Trop de fichiers temporaires peuvent ralentir les opérations, voire saturer le disque. Un `work_mem` trop bas peut aussi contraindre le planificateur à choisir des plans d'exécution moins optimaux.

Par contre, si `work_mem` est trop haut, et que trop de requêtes le consomment simultanément, le danger est de saturer la RAM. Il n'existe en effet pas de limite à la consommation des sessions de PostgreSQL, ni globalement ni par session !
Or le paramétrage de l'overcommit sous Linux est par défaut très permissif, le noyau ne bloquera rien. La première conséquence de la saturation de mémoire est l'assèchement du cache système (complémentaire de celui de PostgreSQL), et la dégradation des performances. Puis le système va se mettre à swapper, avec à la clé un ralentissement général et durable. Enfin le noyau, à court de mémoire, peut être amené à tuer un processus de PostgreSQL. Cela mène à l'arrêt de l'instance, ou plus fréquemment à son redémarrage brutal avec coupure de toutes les connexions et requêtes en cours.

Toutefois, si l'administrateur paramètre correctement l'overcommit (voir https://dali.bo/j1_html#configuration-de-la-surréservation-mémoire), Linux refusera d'allouer la RAM et la requête tombera en erreur, mais le cache système sera préservé, et PostgreSQL ne tombera pas.

Suivant la complexité des requêtes, il est possible qu'un processus utilise plusieurs fois `work_mem` (par exemple si une requête fait une jointure et un tri, ou qu'un nœud est parallélisé). À l'inverse, beaucoup de requêtes ne nécessitent aucune mémoire de travail.

La valeur de `work_mem` dépend donc beaucoup de la mémoire disponible, des requêtes et du nombre de connexions actives.

Si le nombre de requêtes simultanées est important, `work_mem` devra être faible. Avec peu de requêtes simultanées, `work_mem` pourra être augmenté sans risque.

Il n'y a pas de formule de calcul miracle. Une première estimation courante, bien que très conservatrice, peut être :

> `work_mem` = mémoire / `max_connections`

On obtient alors, sur un serveur dédié avec 16 Go de RAM et 200 connexions autorisées :

```
work_mem = 80MB
```

Mais `max_connections` est fréquemment surdimensionné, et beaucoup de sessions sont inactives. `work_mem` est alors sous-dimensionné.

Plus finement, Christophe Pettus propose en première intention[2] :

> `work_mem` = 4 × mémoire libre / `max_connections`

Soit, pour une machine dédiée avec 16 Go de RAM, donc 4 Go de *shared buffers*, et 200 connections :

```
work_mem = 240MB
```

Dans l'idéal, si l'on a le temps pour une étude, on montera `work_mem` jusqu'à voir disparaître l'essentiel des fichiers temporaires dans les traces, tout en restant loin de saturer la RAM lors des pics de charge.

En pratique, le défaut de 4 Mo est très conservateur, souvent insuffisant. Généralement, la valeur varie entre 10 et 100 Mo. Au-delà de 100 Mo, il y a souvent un problème ailleurs : des tris sur de trop gros volumes de données, une mémoire insuffisante, un manque d'index (utilisés pour les tris), etc. Des valeurs vraiment grandes ne sont valables que sur des systèmes d'infocentre.

Augmenter globalement la valeur du `work_mem` peut parfois mener à une consommation excessive de mémoire. Il est possible de ne la modifier que le temps d'une session pour les besoins d'une requête ou d'un traitement particulier :

```
SET work_mem TO '30MB' ;
```

hash_mem_multiplier :

`hash_mem_multiplier` est un paramètre multiplicateur, qui peut s'appliquer à certaines opérations (le hachage, lors de jointures ou agrégations). Par défaut, il vaut 1 en versions 13 et 14, et 2 à partir de la 15. Le seuil de consommation fixé par `work_mem` est alors multiplié d'autant. `hash_mem_multiplier` permet de donner plus de RAM à ces opérations sans augmenter globalement `work_mem`. Il peut lui aussi être modifié dans une session.

Tables temporaires

Les tables temporaires (et leurs index) sont locales à chaque session, et disparaîtront avec elle. Elles sont tout de même écrites sur disque dans le répertoire de la base.

Le cache dédié à ces tables pour minimiser les accès est séparé des *shared buffers*, parce qu'il est propre à la session. Sa taille dépend du paramètre `temp_buffers`. La valeur par défaut (8 Mo) peut

[2] https://thebuild.com/blog/2023/03/13/everything-you-know-about-setting-work_mem-is-wrong/

être insuffisante dans certains cas pour éviter les accès aux fichiers de la table. Elle doit être augmentée dans la session *avant* la création de la table temporaire.

Il ne faut pas laisser inutilement ouvertes des sessions ayant créé des tables temporaires, sinon la mémoire n'est jamais rendue.

3.5 SHARED BUFFERS

3.5.1 Utilité des shared buffers

Shared buffers ou blocs de mémoire partagée

- partage les blocs entre les processus
- cache en lecture ET écriture
- double emploi partiel avec le cache du système
  - pas de *direct I/O*
- caches importants pour les performances !
  - voir `effective_cache_size` (~⅔ RAM)

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

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

Tous les processus accèdent à ce cache unique. C'est la zone la plus importante, par la taille, de la mémoire partagée. Toute modification de données est tracée dans le journal de transaction, **puis** modifiée dans ce cache. Elle n'est donc pas écrite sur le disque par le processus effectuant la modification, sauf en dernière extrémité (voir *Synchronisation en arrière plan*).

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

Les lectures et écritures de PostgreSQL passent toutefois toujours par le cache du système. Il n'y a pas de *direct I/O* comme dans certains SGBD concurrents. Les deux caches risquent donc de stocker les mêmes informations. Les algorithmes d'éviction sont différents entre le système et PostgreSQL, PostgreSQL disposant de davantage d'informations sur l'utilisation des données, et le type d'accès. La redondance est habituellement limitée, mais il existe des cas problématiques. En conséquence, des travaux sont en cours pour contourner le cache du système quand cela est pertinent. Cela implique de réimplémenter certaines fonctionnalités fournies par le noyau (*prefetching*, *read-ahead*), en

les optimisant pour PostgreSQL. Une première étape apparaît en version 17. PostgreSQL sait depuis longtemps lire plusieurs blocs d'un coup (fonction `pread()`), mais ce n'est pas très bien adapté à la nature fragmentée de son cache. Avec l'utilisation des vectored I/O[3], PostgreSQL peut lire plusieurs blocs contigus sur le système de fichiers, et les écrire à des positions *disjointes* en mémoire virtuelle, avec un seul appel système `preadv()` (voir la page de manuel[4]), (Avec un OS qui ne connait pas la fonction, comme Windows, PostgreSQL se contente d'appeler `pread()` plusieurs fois.) Avec la valeur par défaut d'un nouveau paramètre, `io_combine_limit`, et des blocs de taille standard, on a jusqu'à 16 fois moins d'appels système pour la lecture des blocs en dehors du cache PostgreSQL. En version 17, seule l'implémentation du parcours séquentiel, celle de `ANALYZE`, et celle de `pg_prewarm` utilisent cette nouvelle API. Nous conseillons de ne pas changer la valeur de ce paramètre sans tests sérieux montrant qu'un changement est bénéfique.

3.5.2 Dimensionnement des shared buffers

- En première intention & avant tests :

```
shared_buffers = 25 % RAM généralement
```

- Si > 8 Go :
  - *Huge Pages*,
  - `max_wal_size`, `checkpoint_timeout`…

Dimensionner correctement ce cache est important pour de nombreuses raisons.

Un cache trop petit :

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

Un cache trop grand :

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

[3] https://en.wikipedia.org/wiki/Vectored_I/O
[4] https://man.freebsd.org/cgi/man.cgi?query=pread&sektion=2

La documentation officielle[5] conseille ceci pour dimensionner `shared_buffers` :

> Un bon point de départ est 25 % de la mémoire vive totale. Ne pas dépasser 40 %, car le cache du système d'exploitation est aussi utilisé.

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

```
shared_buffers = 8GB
```

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

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

Le cache système limite la plupart du temps l'impact d'un mauvais paramétrage de `shared_buffers`, et il est moins grave de sous-dimensionner un peu `shared_buffers` que de le sur-dimensionner.

> Attention : une valeur élevée de `shared_buffers` (au-delà de 8 Go) nécessite de paramétrer finement le système d'exploitation (*Huge Pages* notamment) et d'autres paramètres liés aux journaux et *checkpoints* comme `max_wal_size`. Il faut aussi s'assurer qu'il restera de la mémoire pour le reste des opérations (tri…) et donc adapter `work_mem`.

> Modifier `shared_buffers` impose de redémarrer l'instance.

Un cache supplémentaire est disponible pour PostgreSQL : celui du système d'exploitation. Il est donc intéressant de préciser à PostgreSQL la taille approximative du cache, ou du moins de la part du cache qu'occupera PostgreSQL. Le paramètre `effective_cache_size` n'a pas besoin d'être très précis, mais il permet une meilleure estimation des coûts par le moteur. Il est paramétré habituellement aux alentours des ⅔ de la taille de la mémoire vive du système d'exploitation, pour un serveur dédié.

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

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

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

[5] https://docs.postgresql.fr/current/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-MEMORY

3.5.3 Notions essentielles de gestion du cache

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

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

Buffer pin

Un processus voulant accéder à un bloc du cache (*buffer*) doit d'abord épingler (*to pin*) ce bloc pour forcer son maintien en cache. Pour ce faire, il incrémente le compteur *buffer pin*, puis le décrémente quand il a fini. Un buffer dont le pin est différent de 0 est donc utilisé et ne peut être recyclé.

Buffer dirty/clean

Un buffer est *dirty* (sale) si son contenu a été modifié en mémoire, mais pas encore sur disque. Le fichier de données n'est donc plus à jour (mais a bien été pérennisée sur disque dans les journaux de transactions). Il faudra écrire ce bloc avant de pouvoir le supprimer du cache. Nous verrons plus loin comment.

Un buffer est *clean* (propre) s'il n'a pas été modifié. Le bloc peut être supprimé du cache sans nécessiter le coût d'une écriture sur disque.

Compteur d'utilisation

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

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

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

Clocksweep (ou algorithme de balayage)

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

L'algorithme *clocksweep* parcourt la liste des buffers de façon cyclique à la recherche d'un buffer *unpinned* dont le compteur d'utilisation est à zéro. Tout buffer visité voit son compteur décrémenté de 1. Le système effectue autant de passes que nécessaire sur tous les blocs jusqu'à trouver un buffer à 0. Ce *clocksweep* est effectué par chaque processus, au moment où ce dernier a besoin d'un nouveau buffer.

3.5.4 Ring buffer

But : Le *ring buffer* permet de ne pas purger le cache à cause :

- des grandes tables
- de certaines opérations
  - *Seq Scan*
  - `VACUUM` (écritures)
  - `COPY`, `CREATE TABLE AS SELECT…`
  - etc.
- À partir de PG 16 : `vacuum_buffer_usage_limit`

```
VACUUM (ANALYZE, BUFFER_USAGE_LIMIT '16MB') ;
```

```
vacuumdb --analyze --buffer-usage-limit='16MB'
```

Une table peut être plus grosse que les *shared buffers*. Sa lecture intégrale (lors d'un parcours complet ou d'une opération de maintenance) ne doit pas mener à l'éviction de tous les blocs du cache. PostgreSQL utilise donc plutôt un *ring buffer* quand la taille de la relation dépasse ¼ de `shared_buffers`. Un *ring buffer* est une zone de mémoire gérée à l'écart des autres blocs du cache.

Pour un parcours complet d'une table, cette zone est de 256 ko (taille choisie pour tenir dans un cache L2). Si un bloc y est modifié (`UPDATE`…), il est traité hors du *ring buffer* comme un bloc sale normal.

Pour un `VACUUM` ou un `ANALYZE`, la même technique est utilisée, mais les écritures se font dans le *ring buffer*. Sa taille est de 2 Mo (256 ko seulement jusque PostgreSQL 16 inclus). À partir de PostgreSQL 16, elle peut être augmentée pour accélérer les opérations avec le paramètre `vacuum_buffer_usage_limit` ou ponctuellement ainsi :

```
vacuumdb --buffer-usage-limit='16MB'
```

ou ainsi :

```
VACUUM (ANALYZE, BUFFER_USAGE_LIMIT '16MB') ;
```

En montant à quelques mégaoctets, l'accélération de la vitesse du `VACUUM` peut être notable[6]. C'est aussi très intéressant pour accélérer `ANALYZE`.

Pour les écritures en masse (notamment `COPY` ou `CREATE TABLE AS SELECT`), une technique similaire utilise un *ring buffer* de 16 Mo.

Le site *The Internals of PostgreSQL*[7] et un README[8] dans le code de PostgreSQL entrent plus en détail sur tous ces sujets tout en restant lisibles.

[6] https://blog.dalibo.com/2024/03/26/strategies-acces.html
[7] https://www.interdb.jp/pg/pgsql08.html
[8] https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README

3.5.5 Contenu du cache

2 extensions en « contrib » :

- `pg_buffercache`
- `pg_prewarm`

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

`pg_buffercache` permet de consulter le contenu du cache (à utiliser de manière très ponctuelle). La requête suivante indique les objets non système de la base en cours, présents dans le cache et s'ils sont *dirty* ou pas :

```
pgbench=# CREATE EXTENSION pg_buffercache ;

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

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

L'extension `pg_prewarm` permet de précharger un objet dans le cache de PostgreSQL (si le cache est assez gros, bien sûr) :

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

Il permet même de recharger dès le démarrage le contenu du cache lors d'un arrêt (voir la documentation[9]).

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

Ces deux outils sont décrits dans le module de formation X2[10].

3.5.6 Synchronisation en arrière-plan

Pour synchroniser les blocs « dirty » :

- **Checkpointer** essentiellement :
  - lors des checkpoints (surtout périodiques)
  - synchronise toutes les pages *dirty*
- **Background writer** :
  - de façon anticipée, selon l'activité
  - une portion des pages
- **Backends**
  - en dernière extrémité

Afin de limiter les attentes des sessions interactives, PostgreSQL dispose de deux processus complémentaires, le `background writer` et le `checkpointer`. Tous deux ont pour rôle d'écrire sur le disque les buffers *dirty* (« sales ») de façon asynchrone. Le but est de ne pas impacter les temps de traitement des requêtes des utilisateurs, donc que les écritures soient lissées sur de grandes plages de temps, pour ne pas saturer les disques, mais en nettoyant assez souvent le cache pour qu'il y ait toujours assez de blocs libérables en cas de besoin.

Le `checkpointer` écrit généralement l'essentiel des blocs *dirty*. Lors des *checkpoints*, il synchronise sur disque tous les blocs modifiés. Son rôle est de lisser cette charge sans saturer les disques.

Comme le `checkpointer` n'est pas censé passer très souvent, le `background writer` anticipe les besoins des sessions, et écrit lui-même une partie des blocs *dirty*.

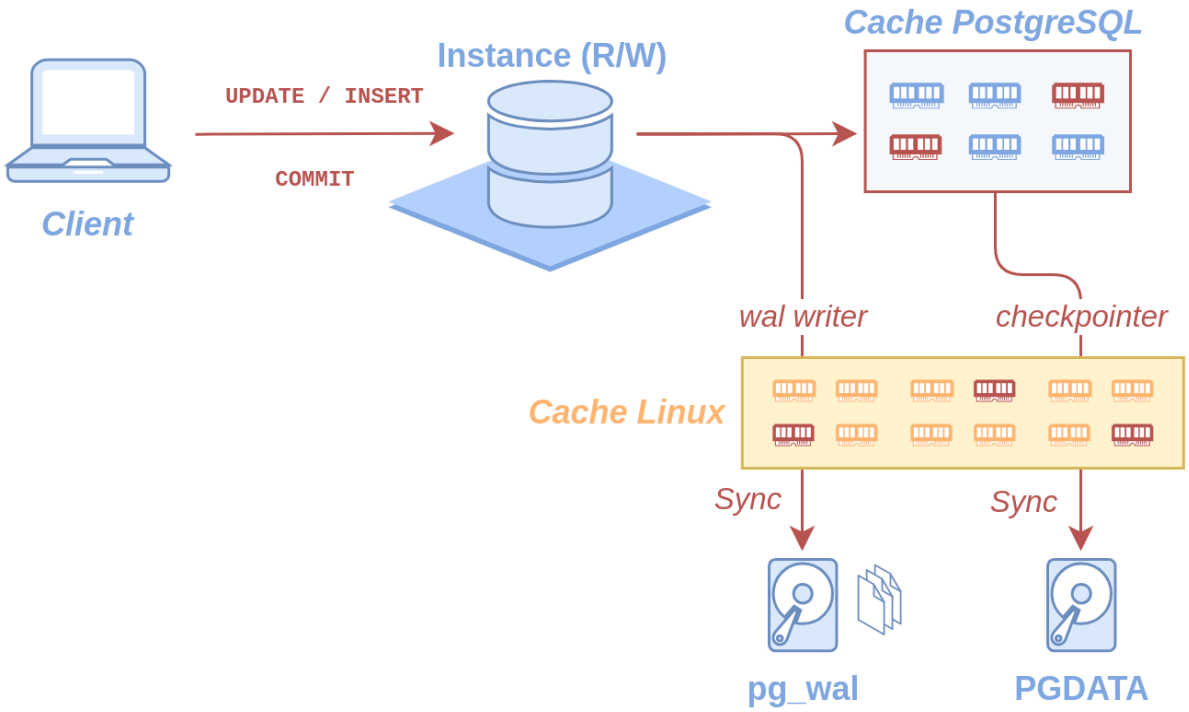
Lors d'écritures intenses, il est possible que ces deux mécanismes soient débordés. Les processus *backend* ne trouvent alors plus en cache de bloc libre ou libérable, et doivent alors écrire eux-mêmes dans les fichiers de données (après les journaux de transaction, bien sûr). Cette situation est évidemment à éviter, ce qui implique généralement de rendre le `background writer` plus agressif.

Nous verrons plus loin les paramètres concernés.

[10] https://dali.bo/x2_html

3.6 JOURNALISATION

3.6.1 Principe de la journalisation



3.6.2 Intégrité & durabilité

- Intégrité : la base reste cohérente malgré :
  - arrêt brutal des processus
  - crash machine
  - …
- Durabilité garantie si `COMMIT`
- Écriture des modifications dans un journal **avant** les fichiers de données
- WAL : *Write Ahead Logging*

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

- l'intégrité : la base reste cohérente quoi qu'il arrive ; un arrêt d'urgence ne corrompra pas la base.
- la durabilité : toute donnée validée (`COMMIT`) est écrite physiquement, et un arrêt brutal immédiatement ne va pas la faire disparaître (excepté la perte de tous les disques de stockage et réplicas, bien sûr).

Pour cela, le mécanisme est relativement simple : toute modification affectant un fichier sera d'abord écrite dans le journal. Les modifications affectant les vrais fichiers de données ne sont écrites qu'en mémoire, dans les *shared buffers*.

Les écritures dans le journal, bien que synchrones, sont relativement performantes, car elles sont séquentielles (moins de déplacement de têtes pour les disques magnétiques). Il n'y a que le fichier en cours à synchroniser à chaque `COMMIT`.

Ce n'est généralement que bien plus tard que les modificationss seront écrites de façon asynchrone, soit par un processus recherchant un buffer libre, soit par le `background writer`, soit par le `checkpointer`. Ce dernier processus sait étaler la charge en écriture dans les fichiers de données sur plusieurs minutes, et dans l'idéal il est seul à s'en charger.

Il existe plusieurs configurations et astuces pour arbitrer entre le niveau de durabilité des données exigé et les contraintes de performances : secondaire en réplication synchrone pour une sécurité maximale, désactivation partielle du mécanisme d'enregistrement synchrone des journaux dans une session, tables de travail non journalisées (*unlogged*), regroupement des insertions pour réduire l'impact des `COMMIT`… Aucune ne remet en cause l'intégrité définie dans les modèle de données.

3.6.3 Journaux de transaction (rappels)

Essentiellement :

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

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

```
$ ls -l
total 2359320
...
```

```
-rw------- 1 postgres postgres 33554432 Mar 26 16:28 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 !

3.6.4 Que contiennent les journaux de transactions ?

- Outils : `pg_waldump` ou `pg_walinspect`

```
 start_lsn  | record_type |     description      |    block_ref
------------+-------------+----------------------+----------------
89/3B6B6868 | INSERT_LEAF | off: 1               | …1663/5/1673829
89/3B6B68A8 | COMMIT      | 2025-01-20 16:40:30. | ø
89/3B6B68D0 | INSERT      | off: 2, flags: 0x00  | …1663/5/1673826
89/3B6B6910 | INSERT_LEAF | off: 2               | …1663/5/1673829
89/3B6B6950 | ABORT       | 2025-01-20 16:40:30. | ø
89/3B6B6978 | UPDATE      | old_xmax: 34696089,  | …1663/5/1673826
89/3B6B69C0 | INSERT_LEAF | off: 3               | …1663/5/1673829
89/3B6B6A00 | COMMIT      | 2025-01-20 16:40:30. | ø
```

Il est peu courant de consulter le contenu des journaux, mais ce peut être utile pour chercher le moment exact d'une mauvaise manipulation au niveau applicatif, et savoir à quel moment restaurer l'instance (par exemple à cause d'une table supprimée[11]).

Les journaux de transactions sont des fichiers binaires. Deux outils existent pour les lire :

- `pg_walinspect` est une extension présente depuis PostgreSQL 15, utilisable en SQL, qui nécessite qu'un journal soit encore dans le `pg_wal/` de PostgreSQL pour être étudié (il doit donc être récent) ;
- `pg_waldump` est un outil en ligne de commande, un peu moins souple à manipuler, qui peut être utilisé sur tout journal qu'on a pu récupérer (dans l'archive PITR par exemple).

Exemple de résultat de `pg_walinspect` sur quelques lignes :

[11] https://blog.hagander.net/locating-the-recovery-point-just-before-a-dropped-table-230/

```
/* LSN de départ */
SELECT pg_current_wal_lsn () AS lsn1 \gset

/* Une nouvelle table, une insertion,
   une insertion annulée, une mise à jour*/
CREATE TABLE demo (i int PRIMARY KEY);
INSERT INTO demo SELECT 1 ;
BEGIN ;
  INSERT INTO demo SELECT 2 ;
ROLLBACK ;
UPDATE demo SET i = 3 WHERE i=1;

CHECKPOINT ;

/* LSN final */
SELECT pg_current_wal_lsn () AS lsn2 \gset

SELECT :'lsn1' AS lsn1 ,  :'lsn2' AS lsn2 ;

    lsn1     |    lsn2
-------------+-------------
 89/3B69CC20 | 89/3B6B6AF8

CREATE EXTENSION IF NOT EXISTS pg_walinspect ;
/* Affichag des enregistrements de journaux des opérations ci-dessus */
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2') \gx
```

Ce petit exemple génère 80 enregistrements dans les journaux, donc seul un petit extrait figure plus haut. L'essentiel de ces lignes correspondent à des enregistrements dans diverses tables systèmes. Il n'y a que quelques enregistrements concernant directement les données insérées ou modifiées.

Noter les numéros de LSN et de transaction (`xid`) qui peuvent servir de point de restauration.

On trouvera par exemple 3 enregistrements de type `COMMIT`, avec leur heure exacte : le premier correspond à la création de la table, les deux suivants aux insertions d'une ligne validées.

```
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2')
WHERE record_type = 'COMMIT' \gx

-[ RECORD 1 ]----------------------------------------------------------------
start_lsn        | 89/3B6B6508
end_lsn          | 89/3B6B6790
prev_lsn         | 89/3B6B6448
xid              | 34696086
resource_manager | Transaction
record_type      | COMMIT
record_length    | 645
main_data_length | 616
fpi_length       | 0
description      | 2025-01-20 16:40:30.534955+01; inval msgs: catcache 80 catcache 79
 ↪  catcache 80 catcache 79 catcache 55 catcache 54 catcache 7 catcache 6 catcache 7
 ↪  catcache 6 catcache 7 catcache 6 catcache 7 catcache 6 catcache 7 catcache 6
 ↪  catcache 7 catcache 6 catcache 7 catcache 6 catcache 55 catcache 54 catcache 7
 ↪  catcache 6 catcache 32 catcache 19 catcache 55 catcache 54 catcache 55 catcache
 ↪  54 snapshot 2608 relcache 1673826 relcache 1673829 relcache 1673826 snapshot
 ↪  2608 relcache 1673826 relcache 1673829
```

```
block_ref        | ø
-[ RECORD 2 ]----------------------------------------------------------------
start_lsn        | 89/3B6B68A8
end_lsn          | 89/3B6B68D0
prev_lsn         | 89/3B6B6868
xid              | 34696087
resource_manager | Transaction
record_type      | COMMIT
record_length    | 34
main_data_length | 8
fpi_length       | 0
description      | 2025-01-20 16:40:30.547764+01
block_ref        | ø
-[ RECORD 3 ]----------------------------------------------------------------
start_lsn        | 89/3B6B6A00
end_lsn          | 89/3B6B6A28
prev_lsn         | 89/3B6B69C0
xid              | 34696089
resource_manager | Transaction
record_type      | COMMIT
record_length    | 34
main_data_length | 8
fpi_length       | 0
description      | 2025-01-20 16:40:30.549943+01
block_ref        | ø
```

De même on trouvera un enregistrement de type `ABORT` pour la transaction terminée par `ROLLBACK`.

Il y a non pas un, mais deux, enregistrements de type `CREATE` :

```
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2')
WHERE record_type = 'CREATE' \gx
```

```
-[ RECORD 1 ]----+---------------
start_lsn        | 89/3B69CC50
end_lsn          | 89/3B69CC80
prev_lsn         | 89/3B69CC20
xid              | 34696086
resource_manager | Storage
record_type      | CREATE
record_length    | 42
main_data_length | 16
fpi_length       | 0
description      | base/5/1673826
block_ref        | ø
-[ RECORD 2 ]----+---------------
start_lsn        | 89/3B6AD2C0
end_lsn          | 89/3B6AD2F0
prev_lsn         | 89/3B6AB7C0
xid              | 34696086
resource_manager | Storage
record_type      | CREATE
record_length    | 42
main_data_length | 16
fpi_length       | 0
```

```
description      | base/5/1673829
block_ref        | ø
```

En effet, la table est accompagnée d'un index pour sa clé primaire, et le champ `description` stocke les chemins physiques des objets :

```
SELECT pg_relation_filepath ('demo') as demo_node,
       pg_relation_filepath ('demo_pkey') AS pk_node ;

   demo_node    |    pk_node
----------------+----------------
 base/5/1673826 | base/5/1673829
```

Dans les insertions, ce sont aussi des informations physiques qui vont apparaître, comme la relation (`rel`), la partie de la table (`main`, c'est-à-dire la partie principale), le bloc, l'offset dans le bloc…

```
SELECT * FROM pg_get_wal_records_info (:'lsn1', :'lsn2')
WHERE record_type LIKE 'INSERT%' AND block_ref LIKE '%1673826%' \gx

…
-[ RECORD 7 ]----+-----------------------------------------------
start_lsn        | 89/3B6B68D0
end_lsn          | 89/3B6B6910
prev_lsn         | 89/3B6B68A8
xid              | 34696088
resource_manager | Heap
record_type      | INSERT
record_length    | 59
main_data_length | 3
fpi_length       | 0
description      | off: 2, flags: 0x00
block_ref        | blkref #0: rel 1663/5/1673826 fork main blk 0
```

On trouvera également un enregistrement de type `UPDATE`, et de nombreux enregistrements de type `INSERT_LEAF` correspondant à des ajouts dans les feuilles d'index. Noter que tous ces enregistrements figurent même pour la transaction annulée, celle-ci alourdit donc inutilement les journaux.

3.6.5 Checkpoint

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

PostgreSQL trace les modifications de données dans les journaux WAL. Ceux-ci sont générés au fur et à mesure des écritures.

Si le système ou l'instance sont arrêtés brutalement, il faut que PostgreSQL puisse appliquer le contenu des journaux non traités sur les fichiers de données. Il a donc besoin de savoir à partir d'où rejouer ces données. Ce point est ce qu'on appelle un *checkpoint*, ou « point de reprise ».

Les principes sont les suivants :

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

Tout fichier de journal antérieur au dernier point de reprise valide **peut être supprimé** ou recyclé, car il n'est plus nécessaire à la récupération.

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

> **Un checkpoint n'est pas un « instantané » cohérent de l'ensemble des fichiers.** C'est simplement l'endroit à partir duquel les journaux doivent être rejoués. Il faut donc pouvoir garantir que tous les blocs modifiés dans le cache *au démarrage du checkpoint* auront été synchronisés sur le disque quand le checkpoint sera terminé, et marqué comme dernier checkpoint valide. Un checkpoint peut donc durer plusieurs minutes, sans que cela ne bloque l'activité.

C'est le processus `checkpointer` qui est responsable de l'écriture des buffers devant être synchronisés durant un checkpoint.

3.6.6 Déclenchement & comportement des checkpoints - 1

- Déclenchement périodique (idéal)
  - `checkpoint_timeout`
- ou : Quantité de journaux
  - `max_wal_size` (pas un plafond !)
- ou : `CHECKPOINT`
- À la fin :
  - sync
  - recyclage des journaux
- Espacer les checkpoints peut réduire leur volumétrie

Plusieurs paramètres influencent le comportement des checkpoints.

Dans l'idéal les checkpoints sont périodiques. Le temps maximum entre deux checkpoints est fixé par `checkpoint_timeout` (par défaut 300 secondes). C'est parfois un peu court pour les instances actives.

Le checkpoint intervient aussi quand il y a beaucoup d'écritures et que le volume des journaux dépasse le seuil défini par le paramètre `max_wal_size` (1 Go par défaut). Un checkpoint est alors déclenché.

L'ordre `CHECKPOINT` déclenche aussi un *checkpoint* sans attendre. En fait, il sert surtout à des utilitaires.

Une fois le checkpoint terminé, les journaux sont à priori inutiles. Ils peuvent être effacés pour redescendre en-dessous de la quantité définie par `max_wal_size`. Ils sont généralement « recyclés », c'est-à-dire renommés, et prêt à être réécris.

Cependant, les journaux peuvent encore être retenus dans `pg_wal/` si l'archivage a été activé et que certains n'ont pas été sauvegardés, ou si l'on garde des journaux pour des serveurs secondaires.

À cause de cela, le volume de l'ensemble des fichiers WAL peut largement dépasser la taille fixée par `max_wal_size`. Ce n'est **pas** une valeur plafond !

Il existe un paramètre `min_wal_size` (défaut : 80 Mo) qui fixe la quantité minimale de journaux à tout moment, même sans activité en écriture. Ils seront donc vides et prêts à être remplis en cas d'écriture

imprévue. Bien sûr, s'il y a des grosses écritures, PostgreSQL créera au besoin des journaux supplémentaires, jusque `max_wal_size`, voire au-delà. Mais il lui faudra les créer et les remplir intégralement de zéros avant utilisation.

Après un gros pic d'activité suivi d'un checkpoint et d'une période calme, la quantité de journaux va très progressivement redescendre de `max_wal_size` à `min_wal_size`.

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

Des checkpoints espacés ont aussi pour effet de réduire la quantité totale de journaux écrits. En effet, par défaut, un bloc modifié est intégralement écrit dans les journaux à sa première modification après un checkpoint, mais par la suite seules les modifications de ce bloc sont journalisées. Espacer les checkpoints peut économiser beaucoup de place disque quand les journaux sont archivés, et du réseau s'ils sont répliqués. Par contre, un écart plus grand entre checkpoints peut allonger la restauration après un arrêt brutal, car il y aura plus de journaux à rejouer.

En pratique, une petite instance se contentera du paramétrage de base ; une plus grosse montera `max_wal_size` à plusieurs gigaoctets.

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

Pour `min_wal_size`, rien n'interdit de prendre une valeur élevée pour mieux absorber les montées d'activité brusques.

Enfin, le checkpoint comprend un *sync* sur disque final. Toujours pour éviter des à-coups d'écriture, PostgreSQL demande au système d'exploitation de forcer un vidage du cache quand `checkpoint_flush_after` a déjà été écrit (par défaut 256 ko). Avant PostgreSQL 9.6, ceci se paramétrait au niveau de Linux en abaissant les valeurs des *sysctl* `vm.dirty_*`. Il y a un intérêt à continuer de le faire, car PostgreSQL n'est pas seul à écrire de gros fichiers (exports `pg_dump`, copie de fichiers…).

3.6.7 Déclenchement & comportement des checkpoints - 2

- Dilution des écritures
  - `checkpoint_completion_target` × durée moy. entre 2 checkpoints
- Surveillance :
  - `checkpoint_warning`
  - `log_checkpoints`
  - Gardez de la place ! sinon crash…

Quand le checkpoint démarre, il vise à lisser au maximum le débit en écriture. La durée d'écriture des données se calcule à partir d'une fraction de la durée d'exécution des précédents checkpoints, fraction fixée par le paramètre `checkpoint_completion_target`. Sa valeur par défaut est celle préconisée par la documentation pour un lissage maximum, soit 0,9 (depuis la version 14, et auparavant le défaut de 0,5 était fréquemment corrigé). Par défaut, PostgreSQL prévoit donc une durée maximale de 300 × 0,9 = 270 secondes pour opérer son checkpoint, mais cette valeur pourra évoluer ensuite suivant la durée réelle des checkpoints précédents.

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

Enfin, répétons que `max_wal_size` n'est pas une limite en dur de la taille de `pg_wal/`.

La partition de `pg_wal/` doit être taillée généreusement. Sa saturation entraîne l'arrêt immédiat de l'instance !

3.6.8 Paramètres du background writer

Nettoyage selon l'activité, en plus du `checkpointer` :

- `bgwriter_delay`
- `bgwriter_lru_maxpages`
- `bgwriter_lru_multiplier`
- `bgwriter_flush_after`

Comme le `checkpointer` ne s'exécute pas très souvent, et ne s'occupe pas des blocs salis depuis son exécution courante, il est épaulé par le `background writer`. Celui-ci pour but de nettoyer une partie des blocs *dirty* pour faire de la place à d'autres. Il s'exécute beaucoup plus fréquemment que le `checkpointer` mais traite moins de blocs à chaque fois.

À intervalle régulier, le `background writer` synchronise un nombre de buffers proportionnel à l'activité sur l'intervalle précédent. Quatre paramètres régissent son comportement :

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

Pour les paramètres `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier`, *lru* signifie *Least Recently Used* (« moins récemment utilisé »). Ainsi le `background writer` synchronisera les pages du cache qui ont été utilisées le moins récemment.

Ces paramètres permettent de rendre le `background writer` plus agressif si la supervision montre que les processus *backends* écrivent trop souvent les blocs de données eux-mêmes, faute de blocs libres dans le cache, ce qui évidemment ralentit l'exécution du point de vue du client.

Évidemment, certaines écritures massives ne peuvent être absorbées par le `background writer`. Elles provoquent des écritures par les processus *backend* et des checkpoints déclenchés.

3.6.9 WAL buffers : journalisation en mémoire

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

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

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

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

Ce mécanisme est géré par ces paramètres, rarement modifiés :

- `wal_buffers` : taille des *WAL buffers*, soit par défaut 1/32e de `shared_buffers` avec un maximum de 16 Mo (la taille d'un segment), des valeurs supérieures (par exemple 128 Mo[12]) pouvant être intéressantes pour les très grosses charges ;
- `wal_writer_delay` (défaut : 200 ms) : intervalle auquel le *walwriter* se réveille pour écrire les buffers non synchronisés ;
- `wal_writer_flush_after` (défaut : 1 Mo) : au-delà de cette valeur, les journaux écrits sont synchronisés sur disque pour éviter l'accumulation dans le cache de l'OS.

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

[12] https://thebuild.com/blog/2023/02/08/xtreme-postgresql/

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

3.6.10 Compression des journaux

- `wal_compression`
  - compression des enregistrements
  - moins de journaux
  - un peu de CPU
  - `off` (défaut)
  - `pglz` (`on`), `lz4`, `zstd` (v15)

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

Comme il y a moins de journaux, leur rejeu est aussi plus rapide, ce qui accélère la réplication et la reprise après un crash. Le prix est une augmentation de la consommation en CPU.

Les détails et un exemple figurent dans ce billet du blog Dalibo[13].

Depuis PostgreSQL 15, on peut même choisir l'algorithme : `pglz`, `lz4` ou `zstd`. `on` est le synonyme de `pglz`… qui est sans doute le moins bon des trois (voir ce petit test[14]), surtout en terme de consommation CPU.

[13] https://blog.dalibo.com/2024/01/05/cambouis.html
[14] https://www.postgresql.org/message-id/YMmlvyVyAFlxZ%2B/H%40paquier.xyz

3.6.11 Limiter le coût de la journalisation

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

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

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

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

Il existe aussi `commit_delay` (défaut : `0`) et `commit_siblings` (défaut : `5`) comme mécanisme de regroupement de transactions[15]. S'il y au moins `commit_siblings` transactions en cours, PostgreSQL attendra jusqu'à `commit_delay` (en microsecondes) avant de valider une transaction pour permettre à d'autres transactions de s'y rattacher. Ce mécanisme, désactivé par défaut, accroît la latence de certaines transactions afin que plusieurs soient écrites ensembles, et n'apporte un gain de performance global qu'avec de nombreuses petites transactions en parallèle, et des disques classiques un peu lents. (En cas d'arrêt brutal, il n'y a pas à proprement parler de perte de données puisque les transactions délibérément retardées n'ont pas été signalées comme validées.)

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

3.7 AU-DELÀ DE LA JOURNALISATION

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

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

3.7.1 L'archivage des journaux

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

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

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

Il suffit de rejouer tous les journaux depuis le checkpoint précédent la sauvegarde jusqu'à la fin de la sauvegarde, ou même à un point précis dans le temps. L'application de ces journaux permet de rendre

à nouveau cohérents les fichiers de données, même si ils ont été sauvegardés en cours de modification.

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

Tout ceci est vu dans le module *Point In Time Recovery*[16].

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

wal_level :

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

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

Le niveau `logical` est destiné à la réplication logique[17].

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

archive_mode & **archive_command**/**archive_library** :

Il faut qu'`archive_mode` soit à `on` pour activer l'archivage. Les journaux sont alors copiés grâce à une commande shell à fournir dans `archive_command` ou grâce à une bibliothèque partagée indiquée dans `archive_library` (version 15 ou postérieure). En général on y indiquera ce qu'exige un outil de sauvegarde dédié (par exemple pgBackRest ou barman) dans sa documentation.

3.7.2 Réplication

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

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

- envoyés régulièrement vers le secondaire, qui les rejouera : c'est le principe de la réplication par *log shipping* ;

[16] https://dali.bo/i2_html
[17] https://dali.bo/w5_html

- envoyés par fragments vers cet autre serveur : c'est la réplication par *streaming*.

Ces thèmes ne seront pas développés ici. Signalons juste que la réplication par *log shipping* implique un archivage actif sur le primaire, et l'utilisation de `restore_command` (et d'autres pour affiner) sur le secondaire. Le *streaming* permet de se passer d'archivage, même si coupler *streaming* et sauvegarde PITR est une bonne idée. Sur un PostgreSQL récent, le primaire a par défaut le nécessaire activé pour se voir doté d'un secondaire : `wal_level` est à `replica` ; `max_wal_senders` permet d'ouvrir des processus dédiés à la réplication ; et l'on peut garder des journaux en paramétrant `wal_keep_size` (ou `wal_keep_segments` avant la version 13) pour limiter les risques de décrochage du secondaire.

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

3.8 CONCLUSION

Mémoire et journalisation :

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

3.8.1 Questions

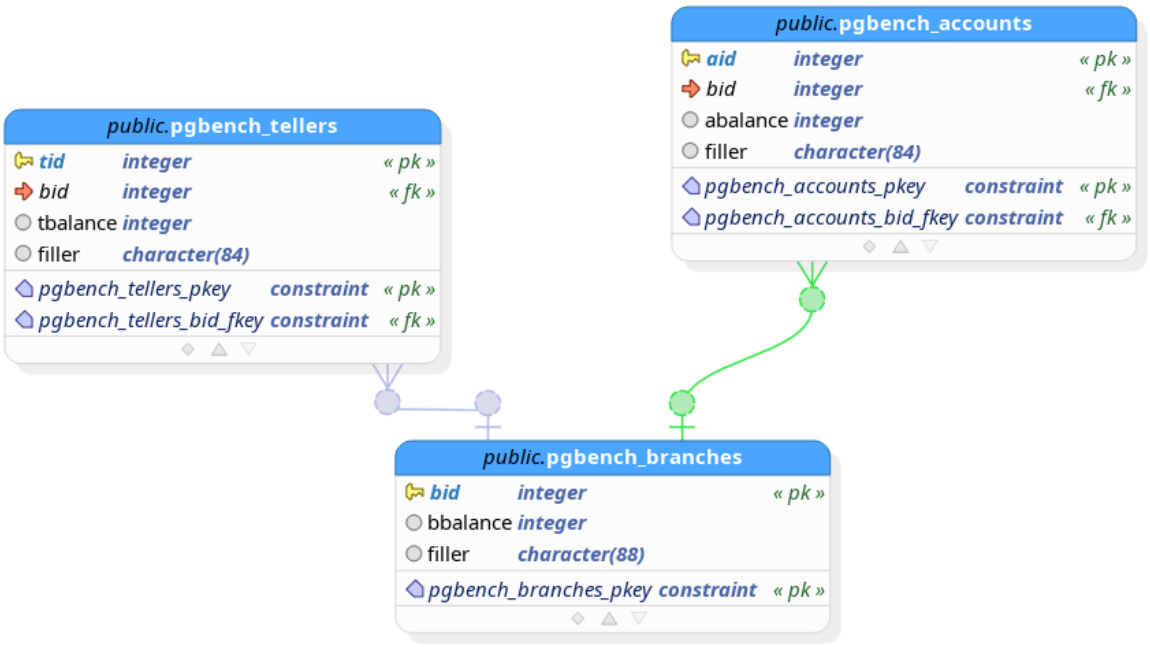
N'hésitez pas, c'est le moment !

3.9 QUIZ

https://dali.bo/m3_quiz

3.10 INTRODUCTION À PGBENCH

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



La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[18].

3.10.1 Installation

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

Sur les distributions à paquets RPM (RockyLinux…), l'outil n'est pas dans le chemin par défaut, il faudra fournir le chemin complet (qui ne sera pas répété ici):

```
/usr/pgsql-17/bin/pgbench
```

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

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

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

Le `pg_hba.conf` doit éventuellement être adapté. La base par défaut s'initialise ainsi (ajouter `--port` et `--host` au besoin) :

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

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

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

3.10.2 Générer de l'activité

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

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

Pour afficher, ajouter `--debug` :

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

À la fin, s'affichent notamment le nombre de transactions (avec et sans le temps de connexion) et la durée moyenne d'exécution du point de vue du client (`latency`) :

```
 scaling factor: 100
 query mode: simple
 number of clients: 20
 number of threads: 4
 duration: 10 s
 number of transactions actually processed: 20433
 latency average = 9.826 ms
 tps = 2035.338395 (including connections establishing)
 tps = 2037.198912 (excluding connections establishing)
```

Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS='-c synchronous_commit=off -c commit_siblings=20' \
            pgbench -U pgbench -c 20 -j 4 -T100 pgbench  2>/dev/null
```

```
latency average = 6.992 ms
tps = 2860.465176 (including connections establishing)
tps = 2862.964803 (excluding connections establishing)
```

Des tests rigoureux doivent durer bien sûr beaucoup plus longtemps que 100 s, par exemple pour tenir compte des effets de cache, des checkpoints périodiques, etc.

3.11 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m3_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`.

Insérer 500 lignes dans la table `t2` avec `generate_series`.

Pour réinitialiser les statistiques de `t2` :

- utiliser la fonction `pg_stat_reset_single_table_counters`
- l'OID en paramètre est dans la table des relations `pg_class`, ou peut être trouvé avec `'t2'::regclass`

Afin de vider le cache, redémarrer l'instance PostgreSQL.

Se connecter à la base de données **b0** et lire les données de la table `t2`.

Récupérer les statistiques IO pour la table `t2` dans la vue système `pg_statio_user_tables`. Qu'observe-t-on ?

Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

Lire de nouveau les données de la table `t2` et consulter ses statistiques. Qu'observe-t-on ?

3.11.2 Mémoire de tri

But : constater l'influence de la mémoire de tri

Ouvrir un premier terminal et laisser défiler le fichier de traces.

Dans un second terminal, activer la trace des fichiers temporaires ainsi que l'affichage du niveau LOG pour le client (il est possible de le faire sur la session uniquement).

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Activer le chronométrage dans la session (`\timing on`). Lire les données de la table `t2` en triant par la colonne `id` Qu'observe-t-on ?

Configurer la valeur du paramètre `work_mem` à `100MB` (il est possible de le faire sur la session uniquement).

Lire de nouveau les données de la table `t2` en triant par la colonne `id`. Qu'observe-t-on ?

3.11.3 Cache disque de PostgreSQL

But : constater l'effet du cache de PostgreSQL

Se connecter à la base de données `b1`. Installer l'extension `pg_buffercache`.

Créer une table `t2` avec une colonne `id` de type `integer`.

Insérer un million de lignes dans la table `t2` avec `generate_series`.

Pour vider le cache de PostgreSQL, redémarrer l'instance.

Pour vider le cache du système d'exploitation, sous **root** :

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

Se connecter à la base de données **b1**. En utilisant l'extension `pg_buffercache`, que contient le cache de PostgreSQL ? (Compter les blocs pour chaque table ; au besoin s'inspirer de la requête du cours.)

Activer l'affichage de la durée des requêtes. Lire les données de la table `t2`, en notant la durée d'exécution de la requête. Que contient le cache de PostgreSQL ?

Lire de nouveau les données de la table `t2`. Que contient le cache de PostgreSQL ?

Configurer la valeur du paramètre `shared_buffers` à un quart de la RAM.

Redémarrer l'instance PostgreSQL.

Se connecter à la base de données **b1** et extraire de nouveau toutes les données de la table `t2`. Que contient le cache de PostgreSQL ?

Modifier le contenu de la table `t2`, par exemple avec :

```
UPDATE t2 SET id = 0 WHERE id < 1000 ;
```

Que contient le cache de PostgreSQL ?

Exécuter un checkpoint. Que contient le cache de PostgreSQL ?

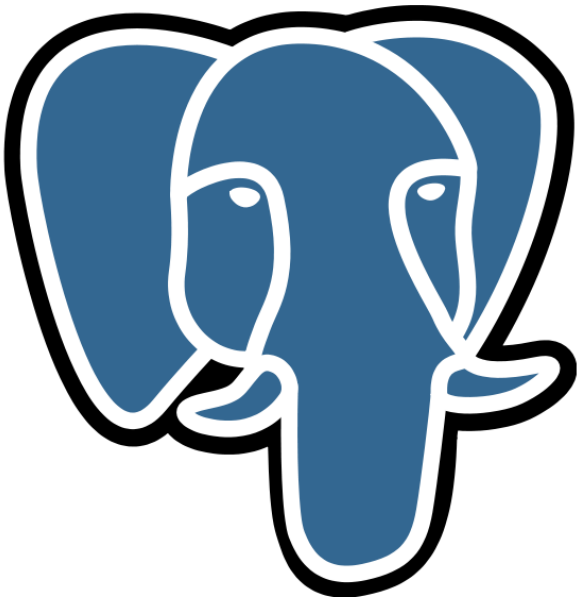
3.11.4 Journaux

But : Observer la génération de journaux

> Insérer 10 millions de lignes dans la table `t2` avec `generate_series`. Que se passe-t-il au niveau du répertoire `pg_wal` ?

> Exécuter un checkpoint. Que se passe-t-il au niveau du répertoire `pg_wal` ?

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

4.3.1 Définitions

- *MultiVersion Concurrency Control*
- Contrôle de Concurrence Multi-Version
- Plusieurs versions du même enregistrement
- Granularité : l'enregistrement (pas le champ !)

MVCC est un sigle signifiant *MultiVersion Concurrency Control*, ou « contrôle de concurrence multiversion ».

Le principe est de faciliter l'accès concurrent de plusieurs utilisateurs (sessions) à la base en disposant en permanence de plusieurs versions différentes d'un même enregistrement. Chaque session peut travailler simultanément sur la version qui s'applique à son contexte (on parle d'« instantané » ou de *snapshot*).

Par exemple, une transaction modifiant un enregistrement va créer une nouvelle version de cet enregistrement. Mais celui-ci ne devra pas être visible des autres transactions tant que le travail de modification n'est pas validé en base. Les autres transactions *verront* donc une ancienne version de cet enregistrement. La dénomination technique est « lecture cohérente » (*consistent read* en anglais).

Précisons que la granularité des modifications est bien l'enregistrement (ou ligne) d'une table. Modifier un champ (colonne) revient à modifier la ligne. Deux transactions ne peuvent pas modifier deux champs différents d'un même enregistrement sans entrer en conflit, et les verrous portent toujours sur des lignes entières.

4.3.2 Alternative à MVCC : un seul enregistrement en base

- Verrouillage en lecture et exclusif en écriture
- Nombre de verrous ?
- Contention ?
- Cohérence ?
- Annulation ?

Avant d'expliquer en détail MVCC, voyons l'autre solution de gestion de la concurrence qui s'offre à nous, afin de comprendre le problème que MVCC essaye de résoudre.

Une table contient une liste d'enregistrements.

- Une transaction voulant consulter un enregistrement doit le verrouiller (pour s'assurer qu'il n'est pas modifié) de façon partagée, le consulter, puis le déverrouiller.
- Une transaction voulant modifier un enregistrement doit le verrouiller de façon exclusive (personne d'autre ne doit pouvoir le modifier ou le consulter), le modifier, puis le déverrouiller.

Cette solution a l'avantage de la simplicité : il suffit d'un gestionnaire de verrous pour gérer l'accès concurrent aux données. Elle a aussi l'avantage de la performance, dans le cas où les attentes de verrous sont peu nombreuses, la pénalité de verrouillage à payer étant peu coûteuse.

Elle a par contre des inconvénients :

- Les verrous sont en mémoire. Leur nombre est donc probablement limité. Que se passe-t-il si une transaction doit verrouiller 10 millions d'enregistrements ? Des mécanismes de promotion de verrou sont implémentés. Les verrous lignes deviennent des verrous bloc, puis des verrous table. **Le nombre de verrous est limité, et une promotion de verrou peut avoir des conséquences dramatiques** ;
- Un processus devant lire un enregistrement devra attendre la fin de la modification de celui-ci. Ceci entraîne rapidement de gros problèmes de contention. **Les écrivains bloquent les lecteurs, et les lecteurs bloquent les écrivains**. Évidemment, les écrivains se bloquent entre eux, mais cela est normal (il n'est pas possible que deux transactions modifient le même enregistrement simultanément, chacune sans conscience de ce qu'a effectué l'autre) ;
- Un ordre SQL (surtout s'il dure longtemps) n'a aucune garantie de voir des données cohérentes du début à la fin de son exécution : si, par exemple, durant un `SELECT` long, un écrivain modifie à la fois des données déjà lues par le `SELECT`, et des données qu'il va lire, le `SELECT` n'aura pas une vue cohérente de la table. Il pourrait y avoir un total faux sur une table comptable par exemple, le `SELECT` ayant vu seulement une partie des données validées par une nouvelle transaction ;
- Comment annuler une transaction ? Il faut un moyen de défaire ce qu'une transaction a effectué, au cas où elle ne se terminerait pas par une validation mais par une annulation.

4.3.3 Implémentation de MVCC par *undo*

- MVCC par *undo* :
  - une version de l'enregistrement dans la table
  - sauvegarde des anciennes versions
  - l'adresse physique d'un enregistrement ne change pas
  - la lecture cohérente est complexe
  - l'*undo* est complexe à dimensionner… et parfois insuffisant
  - l'annulation est lente
- Exemple : Oracle

C'est l'implémentation d'Oracle, par exemple. Un enregistrement, quand il doit être modifié, est recopié précédemment dans le tablespace d'UNDO. La nouvelle version de l'enregistrement est ensuite écrite par-dessus. Ceci implémente le MVCC (les anciennes versions de l'enregistrement sont toujours disponibles), et présente plusieurs avantages :

- Les enregistrements ne sont pas dupliqués dans la table. Celle-ci ne grandit donc pas suite à une mise à jour (si la nouvelle version n'est pas plus grande que la version précédente) ;
- Les enregistrements gardent la même adresse physique dans la table. Les index correspondant à des données non modifiées de l'enregistrement n'ont donc pas à être modifiés eux-mêmes, les index permettant justement de trouver l'adresse physique d'un enregistrement par rapport à une valeur.

Elle a aussi des défauts :

- La gestion de l'*undo* est très complexe : comment décider ce qui peut être purgé ? Il arrive que la purge soit trop agressive, et que des transactions n'aient plus accès aux vieux enregistrements (erreur `SNAPSHOT TOO OLD` sous Oracle, par exemple) ;
- La lecture cohérente est complexe à mettre en œuvre : il faut, pour tout enregistrement modifié, disposer des informations permettant de retrouver l'image avant modification de l'enregistrement (et la bonne image, il pourrait y en avoir plusieurs). Il faut ensuite pouvoir le reconstituer en mémoire ;
- Il est difficile de dimensionner correctement le fichier d'*undo*. Il arrive d'ailleurs qu'il soit trop petit, déclenchant l'annulation d'une grosse transaction. Il est aussi potentiellement une source de contention entre les sessions ;
- L'annulation (`ROLLBACK`) est très lente : il faut, pour toutes les modifications d'une transaction, défaire le travail, donc restaurer les images contenues dans l'*undo*, les réappliquer aux tables (ce qui génère de nouvelles écritures). Le temps d'annulation peut être supérieur au temps de traitement initial devant être annulé.

4.3.4 L'implémentation MVCC de PostgreSQL

- *Copy On Write* (duplication à l'écriture)
- Une version d'enregistrement n'est jamais modifiée
- Toute modification entraîne une nouvelle version
- Pas d'*undo* : pas de contention, `ROLLBACK` instantané

Dans une table PostgreSQL, un enregistrement peut être stocké dans plusieurs versions. Une modification d'un enregistrement entraîne l'écriture d'une nouvelle version de celui-ci. Une ancienne version ne peut être recyclée que lorsqu'aucune transaction ne peut plus en avoir besoin, c'est-à-dire qu'aucune transaction n'a un instantané de la base plus ancien que l'opération de modification de cet enregistrement, et que cette version est donc invisible pour tout le monde. Chaque version

d'enregistrement contient bien sûr des informations permettant de déterminer s'il est visible ou non dans un contexte donné.

Les avantages de cette implémentation stockant plusieurs versions dans la table principale sont multiples :

- La lecture cohérente est très simple à mettre en œuvre : à chaque session de lire la version qui l'intéresse. La visibilité d'une version d'enregistrement est simple à déterminer ;
- Il n'y a pas d'*undo*. C'est un aspect de moins à gérer dans l'administration de la base ;
- Il n'y a pas de contention possible sur l'*undo* ;
- Il n'y a pas de recopie dans l'*undo* avant la mise à jour d'un enregistrement. La mise à jour est donc moins coûteuse ;
- L'annulation d'une transaction est instantanée : les anciens enregistrements sont toujours disponibles.

Cette implémentation a quelques défauts :

- Il faut supprimer régulièrement les versions obsolètes des enregistrements ;
- Il y a davantage de maintenance d'index (mais moins de contentions sur leur mise à jour) ;
- Les enregistrements embarquent des informations de visibilité, qui les rendent plus volumineux.

4.4 NIVEAUX D'ISOLATION

4.4.1 Principe des niveaux d'isolation

- Chaque transaction (et donc session) est isolée à un certain point :
  - elle ne voit pas les opérations des autres
  - elle s'exécute indépendamment des autres
- Le niveau d'isolation au démarrage d'une transaction peut être spécifié :
  - `BEGIN ISOLATION LEVEL xxx`;

Chaque transaction, en plus d'être atomique, s'exécute séparément des autres. Le niveau de séparation demandé est un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction). Quatre niveaux sont définis, ils ne sont pas tous implémentés par PostgreSQL.

4.4.2 Niveau READ UNCOMMITTED

- Non disponible sous PostgreSQL
  - si demandé, s'exécute en `READ COMMITTED`
- Lecture de données modifiées par d'autres transactions **non** validées
- Aussi appelé *dirty reads*
- Dangereux
- Pas de blocage entre les sessions

Ce niveau d'isolation n'est disponible que pour les SGBD non-MVCC. Il est très dangereux : il est possible de lire des données invalides, ou temporaires, puisque tous les enregistrements de la table sont lus, quels que soient leurs états. Il est utilisé dans certains cas où les performances sont cruciales, au détriment de la justesse des données.

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

4.4.3 Niveau READ COMMITTED

- Niveau d'isolation par défaut
- La transaction ne lit que les données validées en base
- Un ordre SQL s'exécute dans un instantané (les tables semblent figées sur la durée de l'ordre)
- L'ordre suivant s'exécute dans un instantané différent

Ce mode est le mode par défaut, et est suffisant dans de nombreux contextes. PostgreSQL étant MVCC, les écrivains et les lecteurs ne se bloquent pas mutuellement, et chaque ordre s'exécute sur un instantané de la base (ce n'est pas un prérequis de `READ COMMITTED` dans la norme SQL). Il n'y a plus de lectures d'enregistrements non valides (*dirty reads*). Il est toutefois possible d'avoir deux problèmes majeurs d'isolation dans ce mode :

- Les lectures non-répétables (*non-repeatable reads*) : une transaction peut ne pas voir les mêmes enregistrements d'une requête sur l'autre, si d'autres transactions ont validé des modifications entre temps ;
- Les lectures fantômes (*phantom reads*) : des enregistrements peuvent ne plus satisfaire une clause `WHERE` entre deux requêtes d'une même transaction.

4.4.4 Niveau REPEATABLE READ

- Instantané au début de la transaction
- Ne voit donc plus les modifications des autres transactions
- Voit toujours ses propres modifications
- Peut entrer en conflit avec d'autres transactions si modification des mêmes enregistrements

Ce mode, comme son nom l'indique, permet de ne plus avoir de lectures non-répétables. Deux ordres SQL consécutifs dans la même transaction retourneront les mêmes enregistrements, dans la même version. Ceci est possible car la transaction voit une image de la base figée. L'image est figée non au démarrage de la transaction, mais à la première commande non TCL (*Transaction Control Language*) de la transaction, donc généralement au premier `SELECT` ou à la première modification.

Cette image sera utilisée pendant toute la durée de la transaction. En lecture seule, ces transactions ne peuvent pas échouer. Elles sont entre autres utilisées pour réaliser des exports des données : c'est ce que fait `pg_dump`.

Dans le standard, ce niveau d'isolation souffre toujours des lectures fantômes, c'est-à-dire de lecture d'enregistrements différents pour une même clause `WHERE` entre deux exécutions de requêtes. Cependant, PostgreSQL est plus strict et ne permet pas ces lectures fantômes en `REPEATABLE READ`. Autrement dit, un même `SELECT` renverra toujours le même résultat.

En écriture, par contre (ou `SELECT FOR UPDATE`, `FOR SHARE`), si une autre transaction a modifié les enregistrements ciblés entre temps, une transaction en `REPEATABLE READ` va échouer avec l'erreur suivante :

```
ERROR: could not serialize access due to concurrent update
```

Il faut donc que l'application soit capable de la rejouer au besoin.

4.4.5 Niveau SERIALIZABLE

- Niveau d'isolation le plus élevé
- Chaque transaction se croit seule sur la base
  - sinon annulation d'une transaction en cours
- Avantages :
  - pas de « lectures fantômes »
  - évite des verrous, simplifie le développement
- Inconvénients :
  - pouvoir rejouer les transactions annulées
  - toutes les transactions impliquées doivent être sérialisables

PostgreSQL fournit un mode d'isolation appelé `SERIALIZABLE` :

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
…
COMMIT / ROLLBACK ;
```

Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base, et comme si elles se déroulaient les unes à la suite des autres. Dès que cette garantie ne peut plus être apportée, PostgreSQL annule celle qui entraînera le moins de perte de données.

Le niveau `SERIALIZABLE` est utile quand le résultat d'une transaction peut être influencé par une transaction tournant en parallèle, par exemple quand des valeurs de lignes dépendent de valeurs d'autres lignes : mouvements de stocks, mouvements financiers… avec calculs de stocks. Autrement dit, si une transaction lit des lignes, elle a la garantie que leurs valeurs ne seront pas modifiées jusqu'à son `COMMIT`, y compris par les transactions qu'elle ne voit pas — ou bien elle tombera en erreur.

Au niveau `SERIALIZABLE` (comme en `REPEATABLE READ`), il est donc essentiel de pouvoir rejouer une transaction en cas d'échec. Par contre, nous simplifions énormément tous les autres points du développement. Il n'y a plus besoin de `SELECT FOR UPDATE`, solution courante mais très gênante pour les transactions concurrentes. Les triggers peuvent être utilisés sans soucis pour valider des opérations.

Ce mode doit être mis en place globalement, car toute transaction non sérialisable peut en théorie s'exécuter n'importe quand, ce qui rend inopérant le mode sérialisable sur les autres.

La sérialisation utilise le « verrouillage de prédicats ». Ces verrous sont visibles dans la vue `pg_locks` sous le nom `SIReadLock`, et ne gênent pas les opérations habituelles, du moins tant que la sérialisation est respectée. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation.

Le wiki PostgreSQL[1], et la documentation officielle[2] donnent des exemples, et ajoutent quelques conseils pour l'utilisation de transactions sérialisables. Afin de tenter de réduire les verrous et le nombre d'échecs :

- faire des transactions les plus courtes possibles (si possible uniquement ce qui a trait à l'intégrité) ;
- limiter le nombre de connexions actives ;
- utiliser les transactions en mode `READ ONLY` dès que possible, voire en `SERIALIZABLE READ ONLY DEFERRABLE` (au risque d'un délai au démarrage) ;
- augmenter certains paramètres liés aux verrous, c'est-à-dire augmenter la mémoire dédiée ; car si elle manque, des verrous de niveau ligne pourraient être regroupés en verrous plus larges et plus gênants ;
- éviter les parcours de tables (*Seq Scan*), et donc privilégier les accès par index.

[1] https://wiki.postgresql.org/wiki/SSI/fr
[2] https://docs.postgresql.fr/current/transaction-iso.html#XACT-SERIALIZABLE

4.5 BLOCS & LIGNES

4.5.1 Structure d'un bloc

- 1 bloc = 8 ko
- `ctid` = (bloc, item dans le bloc)

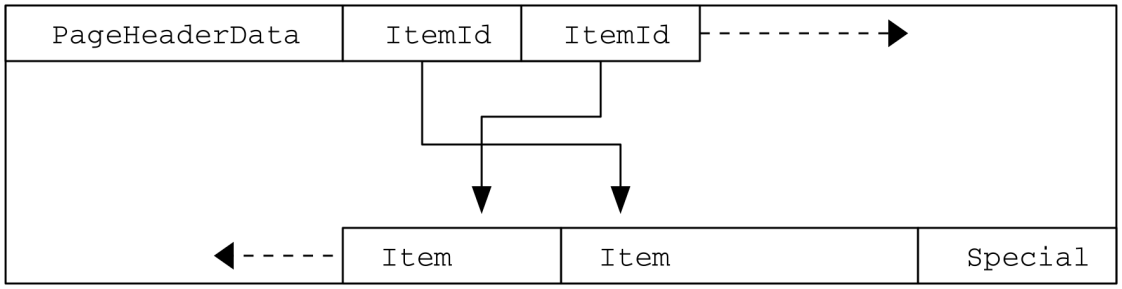


Figure 4/ .1: Répartition des lignes au sein d'un bloc (schéma de la documentation officielle, licence PostgreSQL)

Le bloc (ou page) est l'unité de base de transfert pour les I/O, le cache mémoire… Il fait généralement 8 ko (ce qui ne peut être modifié qu'en recompilant). Les lignes y sont stockées avec des informations d'administration telles que décrites dans le schéma ci-dessus. Une ligne ne fait jamais partie que d'un seul bloc (si cela ne suffit pas, un mécanisme que nous verrons plus tard, nommé TOAST, se déclenche).

Nous distinguons dans ce bloc :

- un entête de page avec diverses informations, notamment la somme de contrôle (si activée) ;
- des identificateurs de 4 octets, pointant vers les emplacements des lignes au sein du bloc ;
- les lignes, stockées à rebours depuis la fin du bloc ;
- un espace spécial, vide pour les tables ordinaires, mais utilisé par les blocs d'index.

Le `ctid` identifie une ligne, en combinant le numéro du bloc (à partir de 0) et l'identificateur dans le bloc (à partir de 1). Comme la plupart des champs administratifs liés à une ligne, il suffit de l'inclure dans un `SELECT` pour l'afficher. L'exemple suivant affiche les premiers et derniers éléments des deux blocs d'une table et vérifie qu'il n'y a pas de troisième bloc :

```
# CREATE TABLE deuxblocs AS SELECT i, i  AS j FROM generate_series(1, 452) i;
SELECT 452

# SELECT ctid, i,j FROM deuxblocs
WHERE ctid in ( '(1, 1)', '(0, 226)', '(1, 1)', '(1, 226)', '(1, 227)', '(2, 0)' );


  ctid   |  i  |  j
---------+-----+-----
 (0,1)   |   1 |   1
 (0,226) | 226 | 226
 (1,1)   | 227 | 227
 (1,226) | 452 | 452
```

Un `ctid` ne doit jamais servir à désigner une ligne de manière pérenne et ne doit pas être utilisé dans des requêtes ! Il peut changer n'importe quand, notamment en cas d'`UPDATE` ou de `VACUUM FULL` !

La documentation officielle[3] contient évidemment tous les détails.

Pour observer le contenu d'un bloc, à titre pédagogique, vous pouvez utiliser l'extension pageinspect[4].

4.5.2 xmin & xmax

Table initiale :

xmin	xmax	Nom	Solde
100		M. Durand	1500
100		Mme Martin	2200

PostgreSQL stocke des informations de visibilité dans chaque version d'enregistrement.

- `xmin` : l'identifiant de la transaction créant cette version.
- `xmax` : l'identifiant de la transaction invalidant cette version.

Ici, les deux enregistrements ont été créés par la transaction 100. Il s'agit peut-être, par exemple, de la transaction ayant importé tous les soldes à l'initialisation de la base.

[3] https://docs.postgresql.fr/current/storage-page-layout.html
[4] https://docs.postgresql.fr/current/pageinspect.html#PAGEINSPECT-HEAP-FUNCS

4.5.3 xmin & xmax (suite)

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

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

Nous décidons d'enregistrer un virement de 200 € du compte de M. Durand vers celui de Mme Martin. Ceci doit être effectué dans une seule transaction : l'opération doit être atomique, sans quoi de l'argent pourrait apparaître ou disparaître de la table.

Nous allons donc tout d'abord démarrer une transaction (ordre `SQL BEGIN`). PostgreSQL fournit donc à notre session un nouveau numéro de transaction (150 dans notre exemple). Puis nous effectuerons :

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

4.5.4 xmin & xmax (suite)

```
UPDATE soldes SET solde = solde + 200 WHERE nom = 'Mme Martin';
```

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

Puis nous effectuerons :

```
UPDATE soldes SET solde = solde + 200 WHERE nom = 'Mme Martin';
```

Nous avons maintenant deux versions de chaque enregistrement. Notre session ne voit bien sûr plus que les nouvelles versions de ces enregistrements, sauf si elle décidait d'annuler la transaction, auquel cas elle reverrait les anciennes données.

Pour une autre session, la version visible de ces enregistrements dépend de plusieurs critères :

- La transaction 150 a-t-elle été validée ? Sinon elle est invisible ;
- La transaction 150 a-t-elle été validée après le démarrage de la transaction en cours, et sommes-nous dans un niveau d'isolation (*repeatable read* ou *serializable*) qui nous interdit de voir les modifications faites depuis le début de notre transaction ? ;
- La transaction 150 a-t-elle été validée après le démarrage de la requête en cours ? Une requête, sous PostgreSQL, voit un instantané cohérent de la base, ce qui implique que toute transaction validée après le démarrage de la requête doit être ignorée.

Dans le cas le plus simple, 150 ayant été validée, une transaction 160 ne verra pas les premières versions : `xmax` valant 150, ces enregistrements ne sont pas visibles. Elle verra les secondes versions, puisque `xmin` = 150, et pas de `xmax`.

4.5.5 xmin & xmax (suite)

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

- Comment est effectuée la suppression d'un enregistrement ?
- Comment est effectuée l'annulation de la transaction 150 ?

- La suppression d'un enregistrement s'effectue simplement par l'écriture d'un `xmax` dans la version courante de la ligne ;
- Le `COMMIT` ou le `ROLLBACK` consiste à écrire le statut de la transaction dans le CLOG (*Commit Log*), une zone mémoire dédiée au statut des transactions.

4.5.6 CLOG

- Le CLOG (*Commit Log*) enregistre l'état des transactions.
- Chaque transaction occupe 2 bits de CLOG (4 statuts)
- `COMMIT` ou `ROLLBACK` très rapide
- Référence pour savoir si une ligne est visible ou pas
- Puis reporté dans les lignes (*hint bits*)

Le CLOG permet à PostgreSQL de savoir si les numéros de transaction portés par une ligne sont validés ou non. Le CLOG est stocké dans une série de fichiers de 256 ko, stockés dans le répertoire `pg_xact/` de PGDATA (répertoire racine de l'instance PostgreSQL). Chaque transaction est créée dans ce fichier dès son démarrage et est encodée sur deux bits puisqu'une transaction peut avoir quatre états :

- `TRANSACTION_STATUS_IN_PROGRESS` signifie que la transaction est en cours, c'est l'état initial ;
- `TRANSACTION_STATUS_COMMITTED` signifie que la la transaction a été validée ;
- `TRANSACTION_STATUS_ABORTED` signifie que la transaction a été annulée ;
- `TRANSACTION_STATUS_SUB_COMMITTED` signifie que la transaction comporte des sous-transactions, afin de valider l'ensemble des sous-transactions de façon atomique.

Nous avons donc un million d'états de transactions par fichier de 256 ko.

Ainsi, annuler une transaction (`ROLLBACK`) est quasiment instantané sous PostgreSQL : il suffit d'écrire `TRANSACTION_STATUS_ABORTED` dans l'entrée de CLOG correspondant à la transaction. Les lignes insérées par cette transaction seront simplement ignorées, de même que des versions de lignes créées par un `UPDATE`. Les versions de lignes marquées comme périmées par ce même `UPDATE` dans la transaction resteront visibles.

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

Pour éviter de se référer au CLOG trop souvent, chaque ligne porte des *hint bits* contenant, entre autres, le statut des numéros de transaction portés par la ligne. Mais ce n'est pas la requête qui a modifié les lignes qui les renseigne. La première requête qui lit une ligne après une modification, même un `SELECT`, lit le CLOG et met à jour ses *hint bits*, ce qui occasionne de nouvelles écritures. (Et génèrent même de nouveaux journaux si `wal_log_hints` est `on` ou si les sommes de contrôle sont activées, ce qui est recommandé.)

4.6 AVANTAGES & INCONVÉNIENTS DU MVCC DE POSTGRESQL

4.6.1 Avantages du MVCC de PostgreSQL

- Avantages classiques de MVCC (concurrence d'accès)
- Implémentation simple et performante
- Peu de sources de contention
- Verrouillage simple d'enregistrement
- `ROLLBACK` instantané
- Données conservées aussi longtemps que nécessaire

Reprenons les avantages du MVCC tel qu'implémenté par PostgreSQL :

- Les lecteurs ne bloquent pas les écrivains, ni les écrivains les lecteurs ;
- Le code gérant les instantanés est simple, ce qui est excellent pour la fiabilité, la maintenabilité et les performances ;
- Les différentes sessions ne se gênent pas pour l'accès à une ressource commune (l'*undo*) ;
- Un enregistrement est facilement identifiable comme étant verrouillé en écriture : il suffit qu'il ait une version ayant un `xmax` correspondant à une transaction en cours ;
- L'annulation est instantanée : il suffit d'écrire le nouvel état de la transaction annulée dans la CLOG. Pas besoin de restaurer les valeurs précédentes, elles sont toujours là ;
- Les anciennes versions restent en ligne aussi longtemps que nécessaire. Elles ne pourront être effacées de la base qu'une fois qu'aucune transaction ne les considérera comme visibles.

(Précisons toutefois que ceci est une vision un peu simplifiée pour les cas courants. La signification du `xmax` est parfois altérée par des bits positionnés dans des champs systèmes inaccessibles par l'utilisateur. Cela arrive, par exemple, quand des transactions insèrent des lignes portant une clé étrangère, pour verrouiller la ligne pointée par cette clé, laquelle ne doit pas disparaître pendant la durée de cette transaction.)

4.6.2 Inconvénients du MVCC de PostgreSQL

- Nettoyage des enregistrements
  - `VACUUM`
  - automatisation : autovacuum
- Tables plus volumineuses
- Écritures amplifiées
- Pas de visibilité des lignes dans les index
- Les colonnes supprimées impliquent reconstruction

Comme toute solution complexe, l'implémentation MVCC de PostgreSQL est un compromis. Les avantages cités précédemment sont obtenus au prix de concessions. Bien sûr, divers mécanismes tentent de mitiger ces soucis.

4.6.2.1 VACUUM

Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande `VACUUM`. Il a un avantage sur la technique de l'*undo* : le nettoyage n'est pas effectué par un client faisant des mises à jour (et créant donc des enregistrements morts), et le ressenti est donc meilleur.

`VACUUM` peut se lancer à la main, mais dans le cas général on s'en remet à l'autovacuum, un démon qui lance les `VACUUM` (et bien plus) en arrière-plan quand il le juge nécessaire. Tout cela sera traité en détail par la suite.

4.6.2.2 Bloat

Les tables sont forcément plus volumineuses que dans l'implémentation par *undo*, pour deux raisons :

- les informations de visibilité y sont stockées, il y a donc un surcoût d'une douzaine d'octets par enregistrement ;
- il y a toujours des enregistrements morts dans une table, une sorte de *fond de roulement*, qui se stabilise quand l'application est en régime stationnaire.

Ces enregistrements sont recyclés à chaque passage de `VACUUM`.

4.6.2.3 Amplification des écritures

Toute écriture a tendance à en déclencher d'autres. D'abord, toute modification est écrite dans les journaux de transaction, comme le fait tout moteur de base de données relationnelle fiable, et éventuellement archivée et/ou répliquée.

Des suppressions, insertions ou mises à jour, même suivies d'un `ROLLBACK`, écrivent toujours dans les fichiers de données eux-mêmes. (Ce sont les numéros de transaction et le CLOG qui définiront la visibilité des lignes.) Le `VACUUM` nettoie les lignes, et cela genère aussi des écritures et des journaux. Nous verrons également le coût du problème de *wraparound*. La mise à jour des *hints bits*, même lors d'un `SELECT`, génère de nouvelles écritures voire des journaux.

Comme les index pointent vers un bloc et une ligne, les index sont également souvent à modifier suite à une écriture (et cela est aussi journalisé).

4.6.2.4 Visibilité

Les index n'ont pas d'information de visibilité. Il est donc nécessaire d'aller vérifier dans la table associée que l'enregistrement trouvé dans l'index est bien visible. Cela a un impact sur le temps d'exécution de requêtes comme `SELECT count(*)` sur une table : dans le cas le plus défavorable, il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles. La *visibility map* permet de limiter cette vérification aux données les plus récentes. Il existe également des mécanismes de nettoyage des index quand une requête rencontre des *tuples* périmés.

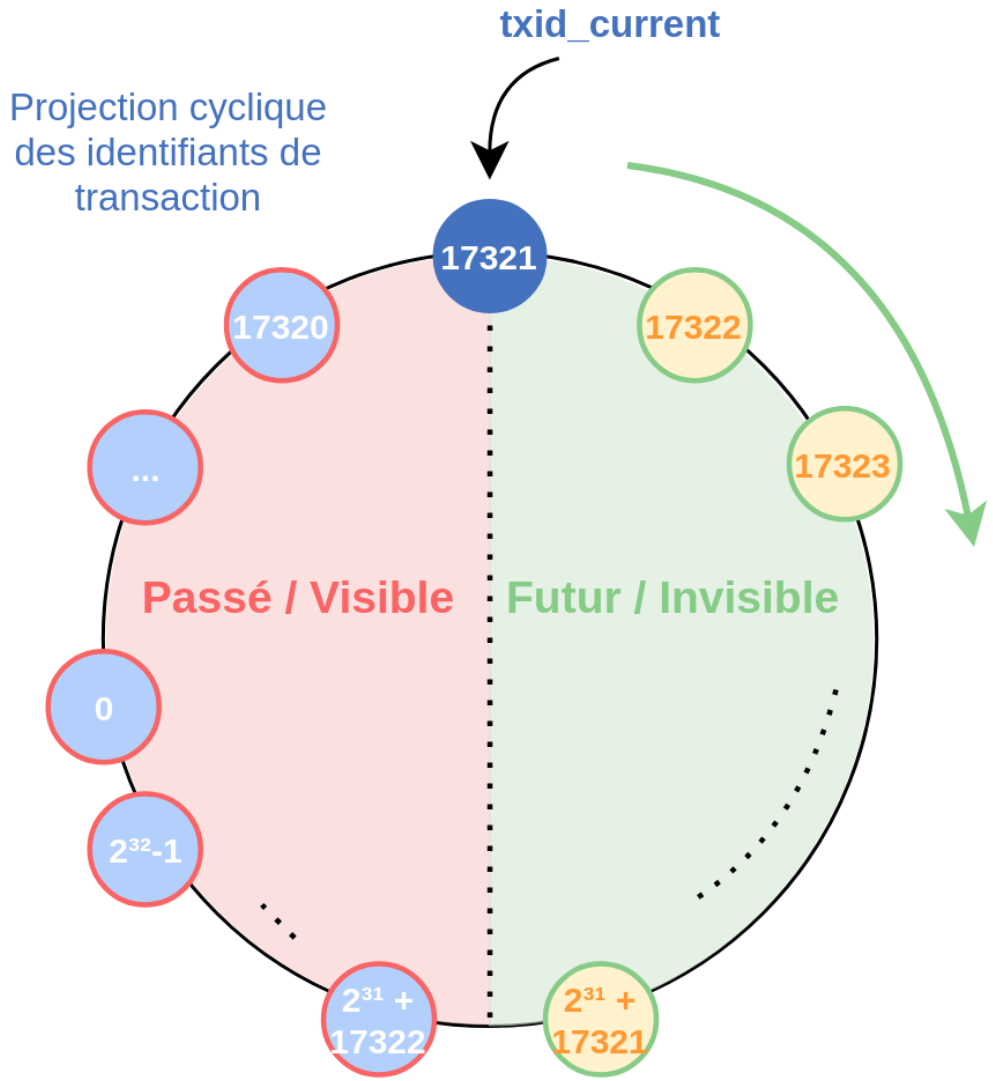
4.6.2.5 Colonnes supprimées

Un `VACUUM` ne s'occupe pas de l'espace libéré par des colonnes supprimées (fragmentation verticale). Un `VACUUM FULL` est nécessaire pour reconstruire la table.

4.7 LE WRAPAROUND

4.7.1 Le wraparound (1)

- *Wraparound* : bouclage du compteur de `xmin` / `xmax`
- 32 bits ~ 4 milliards



Le numéro de transaction stocké sur chaque ligne est sur 32 bits, même si PostgreSQL utilise en interne 64 bits. Il y aura donc dépassement de ce compteur au bout de 4 milliards de transactions. Sur les

machines actuelles, avec une activité soutenue, cela peut être atteint relativement rapidement.

En fait, ce compteur est cyclique. Cela peut se voir ainsi :

```
SELECT pg_current_xact_id(), *
FROM pg_control_checkpoint()\gx
```

```
-[ RECORD 1 ]--------+-------------------------
pg_current_xact_id   | 10549379902
checkpoint_lsn       | 62/B902F128
redo_lsn             | 62/B902F0F0
redo_wal_file        | 000000010000006200000B9
timeline_id          | 1
prev_timeline_id     | 1
full_page_writes     | t
next_xid             | 2:1959445310
next_oid             | 24614
next_multixact_id    | 1
next_multi_offset    | 0
oldest_xid           | 1809610092
oldest_xid_dbid      | 16384
oldest_active_xid    | 1959445310
oldest_multi_xid     | 1
oldest_multi_dbid    | 13431
oldest_commit_ts_xid | 0
newest_commit_ts_xid | 0
checkpoint_time      | 2023-12-29 16:39:25+01
```

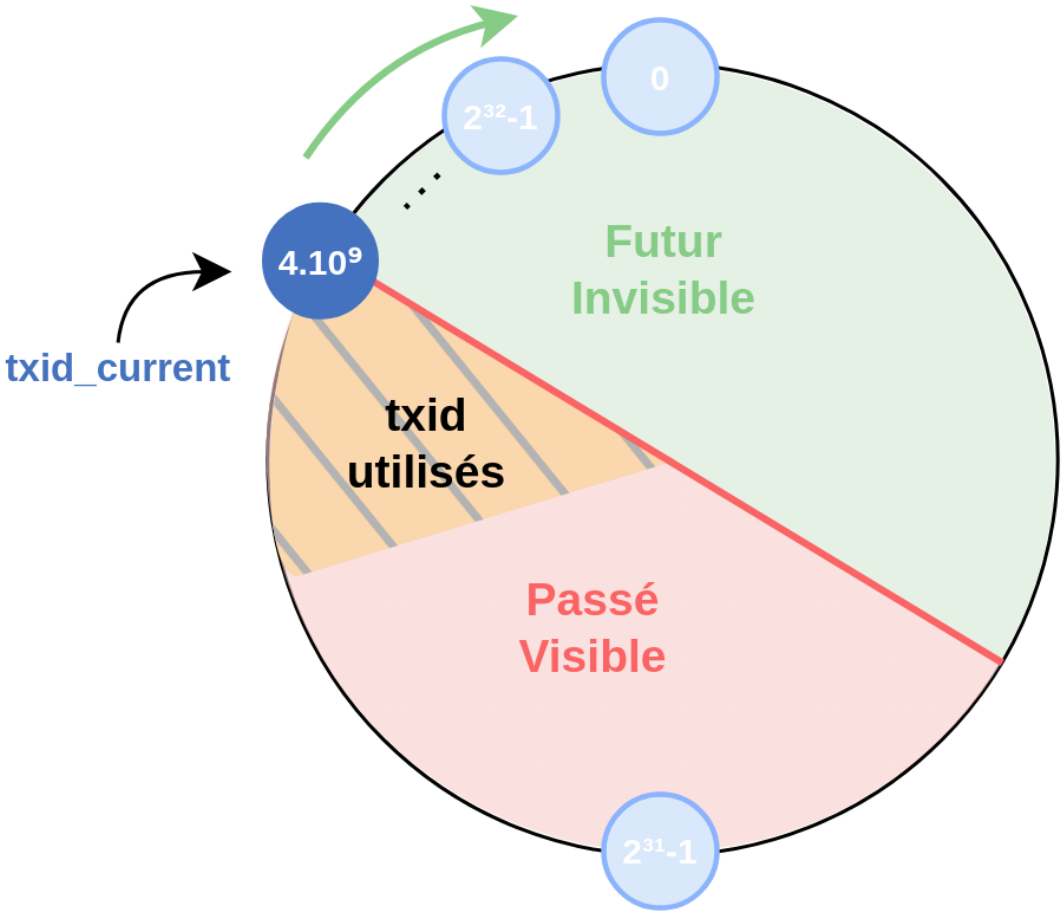
`pg_current_xact_id()` renvoie le numéro de transaction en cours, soit 10 549 379 902 (sur 64 bits). Le premier chiffre de la ligne `next_xid` indique qu'il y a déjà eu deux « époques », et le numéro de transaction sur 32 bits dans ce troisième cycle est 1 959 445 310. On vérifie que 10 549 379 902 = 1 959 445 310 + 2×2^{32}.

Le nombre d'époques n'est enregistré ni dans les tables ni dans les lignes. Pour savoir quelles transactions sont visibles depuis la transaction en cours (ou exactement depuis le « *snapshot* » de l'état de la base associé), PostgreSQL considère que les 2 milliards de numéros de transactions supérieurs à celle en cours correspondent à des transactions futures, et les 2 milliards de numéros inférieurs à des transactions dans le passé.

Si on se contentait de boucler, une fois dépassé le numéro de transaction 2^{31}-1 (environ 2 milliards), de nombreux enregistrements deviendraient invisibles, car validés par des transactions à présent situées dans le futur. Il y a donc besoin d'un mécanisme supplémentaire.

4.7.2 Le wraparound (2)

Après 4 milliards de transactions :



PostgreSQL se débrouille afin de ne jamais laisser dans les tables des numéros de transactions visibles situées dans ce qui serait le futur de la transaction actuelle.

En temps normal, les numéros de transaction visibles se situent tous dans des valeurs un peu inférieures à la transaction en cours. Les lignes plus vieilles sont « gelées », c'est-à-dire marquées comme assez anciennes pour qu'on ignore le numéro de transaction. (Cela implique bien sûr de réécrire la ligne sur le disque.) Il n'y a alors plus de risque de présence de numéros de transaction qui serait abusivement dans le futur de la transaction actuelle. Les numéros de transaction peuvent être réutilisés

sans risque de mélange avec ceux issus du cycle précédent.

Ce cycle des numéros de transactions peut se poursuivre indéfiniment.

Ces recyclages sont visibles dans `pg_control_checkpoint()`, comme vu plus haut. La même fonction renvoie `oldest_xid`, qui est le numéro de la transaction (32 bits) la plus ancienne dans l'instance, qui est par définition celui de la base la plus vieille, ou plutôt de la ligne la plus vieille dans la base :

```
SELECT datname, datfrozenxid, age (datfrozenxid)
FROM pg_database ORDER BY 3 DESC ;
```

```
  datname   | datfrozenxid |    age
------------+--------------+-----------
 pgbench    |   1809610092 | 149835222
 template0 |   1957896953 |   1548361
 template1 |   1959012415 |    432899
 postgres  |   1959445305 |         9
```

La ligne la plus ancienne dans la base **pgbench** a moins de 150 millions de transactions d'âge, il n'y a pas de confusion possible.

Un peu plus de 50 millions de transactions plus tard, la même requête renvoie :

```
  datname   | datfrozenxid |   age
------------+--------------+----------
 template0 |   1957896953 | 52338522
 template1 |   1959012415 | 51223060
 postgres  |   1959445305 | 50790170
 pgbench    |   1959625471 | 50610004
```

Les lignes les plus anciennes de `pgbench` semblent avoir disparu.

4.7.3 Le wraparound (3)

Concrètement ?

- `VACUUM FREEZE`
  - géré par l'autovacuum
  - au pire, d'office
  - potentiellement beaucoup d'écritures

Concrètement, l'opération de « gel » des lignes qui possèdent des identifiants de transaction suffisamment anciens est appelée `VACUUM FREEZE`.

Ce dernier peut être déclenché manuellement, mais il fait aussi partie des tâches de maintenance habituellement gérées par le démon `autovacuum`, en bonne partie en même temps que les `VACUUM`

habituels. Un `VACUUM FREEZE` n'est pas bloquant, mais les verrous sont parfois plus gênants que lors d'un `VACUUM` simple. Les écritures générées sont très variables, et parfois gênantes.

Si cela ne suffit pas, le moteur déclenche automatiquement un `VACUUM FREEZE` sur une table avec des lignes trop âgées, et ce, même si autovacuum est désactivé.

Ces opérations suffisent dans l'immense majorité des cas. Dans le cas contraire, l'`autovacuum` devient de plus en plus agressif. Si le stock de transactions disponibles descend en-dessous de 40 millions (10 millions avant la version 14), des messages d'avertissements apparaissent dans les traces.

Dans le pire des cas, après bien des messages d'avertissements, le moteur refuse toute nouvelle transaction dès que le stock de transactions disponibles se réduit à 3 millions (1 million avant la version 14 ; valeurs codées en dur). Il faudra alors lancer un `VACUUM FREEZE` manuellement. Ceci ne peut arriver qu'exceptionnellement (par exemple si une transaction préparée a été oubliée depuis 2 milliards de transactions et qu'aucune supervision ne l'a détectée).

`VACUUM FREEZE` sera développé dans le module VACUUM et autovacuum[5]. La documentation officielle[6] contient aussi un paragraphe sur ce sujet.

[5] https://dali.bo/m5_html
[6] https://docs.postgresql.fr/current/maintenance.html

4.8 OPTIMISATIONS DE MVCC

- HOT
- Free Space Map
- Visibility Map

Les fonctionnalités que nous allons décrire visent à mitiger les inconvénients du MVCC de PostgreSQL.

4.8.1 Mise à jour jour HOT

HOT = *Heap-Only Tuples*

- Si place dans le bloc
- Si aucune colonne indexée modifiée
- Alors la mise à jour se fait dans le même bloc
  - gain en mise à jour des index
  - gain en écritures et en utilisation du cache
- Favorisée par un *fillfator* < 100

Le mécanisme des *Heap-Only Tuples* (HOT) permet de stocker, sous condition, plusieurs versions du même enregistrement dans le même bloc, avec une forme de chaînage entre plusieurs versions d'une ligne dans le bloc. Ceci permet au fur et à mesure des mises à jour de supprimer automatiquement les anciennes versions, sans attendre `VACUUM`. Cela permet aussi de ne pas toucher aux index, qui pointent donc grâce à cela sur plusieurs versions du même enregistrement. Il y a donc un gain en écriture comme en utilisation du cache. Les conditions sont les suivantes :

- Aucune colonne indexée n'a été modifiée par l'opération : l'index n'a pas besoin d'être modifié et peut continuer à pointer vers le même bloc ;
- Le bloc contient assez de place pour la nouvelle version (les enregistrements ne sont pas chaînés entre plusieurs blocs).

Afin que cette dernière condition ait plus de chance d'être vérifiée, il peut être utile de baisser la valeur du paramètre `fillfactor` pour une table donnée (cf documentation officielle[7]). Par défaut, le `fillfactor` vaut 100% pour économiser la place. Sacrifier un peu d'espace en le baissant peut être rentable car le mécanisme HOT peut réduire la fragmentation.

[7] https://docs.postgresql.fr/current/sql-createtable.html#SQL-CREATETABLE-STORAGE-PARAMETERS

Le nombre de mises à jour HOT sur une table peut être suivi en comparant les champs `n_tup_hot_upd` (décompte des mises à jour HOT) et `n_tup_upd` (toutes les mises à jour) dans la vue pg_stat_user_tables[8].

Documentation officielle : Heap-Only Tuples (HOT)[9]

4.8.2 Free Space Map

- Fichier pointant les espaces libres des blocs
- Optimisation des insertions

Chaque table possède une *Free Space Map* avec une liste des espaces libres de chaque bloc, organisée en arbre. PostgreSQL l'utilise pour savoir où insérer de nouvelles lignes ou versions de lignes. Elle est stockée dans les fichiers `*_fsm` associés à chaque table.

Documentation officielle : Carte des espaces libres[10]

4.8.3 Visibility Map

- Quels blocs sont intégralement visibles ?
- Mise à jour par `VACUUM`
- Utilisation :
  - Accélérer `VACUUM` et `VACUUM FREEZE`
  - *Index Only Scan*

La *Visibility Map* permet de savoir si tous les enregistrements d'un bloc sont visibles de toutes les sessions en cours. C'est le cas si toutes les transactions de modifications sont terminées, et si les lignes mortes ont été nettoyées. Il n'y a donc plus besoin de consulter les numéros de transaction et la CLOG pour toutes les lignes du bloc. En cas de doute, ou d'enregistrement non visible, le bloc n'est pas marqué comme totalement visible.

Cela accélère grandement le `VACUUM` : sa phase 1 ne parcourt pas toute la table, mais uniquement les enregistrements pour lesquels la *Visibility Map* est à *faux*, car des données sont potentiellement

[8] https://docs.postgresql.fr/current/monitoring-stats.html#MONITORING-PG-STAT-ALL-TABLES-VIEW
[9] https://docs.postgresql.fr/current/storage-hot.html
[10] https://docs.postgresql.fr/current/storage-fsm.html

obsolètes dans le bloc. Les blocs intégralement visibles peuvent être ignorés. Après nettoyage d'un bloc, `VACUUM` positionne sa réference dans la *Visibility Map* à *vrai*, si toutes les lignes sont bien visibles pour toutes les sessions.

À l'inverse, un *Index Only Scan* (parcours d'index seuls) utilise cette *Visibility Map* pour savoir si les données de l'index suffisent ou s'il faut aller vérifier la visibilité de la ligne dans la table. Dans ce dernier cas, le plan indique des *Heap Fetches*, qui dégradent les performances. Un `VACUUM` fréquent améliore donc les performances si des *Index Only Scans* sont utilisés. (Les *Index Scans* ou *Bitmap Scans* ne sont pas concernés.)

La même *Visibility Map* sert aussi à noter les bloc entièrement « gelés » pour accélérer les `VACUUM FREEZE`, là encore pour ne pas avoir à les relire.

Documentation officielle : Carte de visibilité[11]

[11] https://docs.postgresql.fr/current/storage-vm.html#STORAGE-VM

4.9 VERROUS

4.9.1 Verrouillage et MVCC

La gestion des verrous est liée à l'implémentation de MVCC

- Verrouillage d'objets en mémoire
- Verrouillage d'objets sur disque
- Paramètres

4.9.2 Le gestionnaire de verrous

PostgreSQL possède un gestionnaire de verrous

- Verrous d'objet
- Niveaux de verrouillage
- Empilement des verrous
- *Deadlock*
- Vue `pg_locks`

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

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler. Si le verrou ne peut être obtenu immédiatement, par défaut PostgreSQL attendra indéfiniment qu'il soit libéré.

Ce verrou en attente peut lui-même imposer une attente à d'autres sessions qui s'intéresseront au même objet. Si ce verrou en attente est bloquant (cas extrême : un `VACUUM FULL` sans `SKIP_LOCKED` lui-même bloqué par une session qui tarde à faire un `COMMIT`), il est possible d'assister à un phénomène d'empilement de verrous en attente.

Les noms des verrous peuvent prêter à confusion : `ROW SHARE` par exemple est un verrou de table, pas un verrou d'enregistrement. Il signifie qu'on a pris un verrou sur une table pour y faire des `SELECT FOR UPDATE` par exemple. Ce verrou est en conflit avec les verrous pris pour un `DROP TABLE`, ou pour un `LOCK TABLE`.

Le gestionnaire de verrous détecte tout verrou mortel (*deadlock*) entre deux sessions. Un *deadlock* est

la suite de prise de verrous entraînant le blocage mutuel d'au moins deux sessions, chacune étant en attente d'un des verrous acquis par l'autre.

Il est possible d'accéder aux verrous actuellement utilisés sur une instance par la vue `pg_locks`.

4.9.3 Verrous sur enregistrement

- Le gestionnaire de verrous possèdes des verrous sur enregistrements
  - transitoires
  - le temps de poser le `xmax`
- Utilisation de verrous sur disque
  - pas de risque de pénurie
- Les verrous entre transaction se font sur leurs ID

Le gestionnaire de verrous fournit des verrous sur enregistrement. Ceux-ci sont utilisés pour verrouiller un enregistrement le temps d'y écrire un `xmax`, puis libérés immédiatement.

Le verrouillage réel est implémenté comme suit :

- D'abord, chaque transaction verrouille son objet « identifiant de transaction » de façon exclusive.
- Une transaction voulant mettre à jour un enregistrement consulte le `xmax`. Si ce `xmax` est celui d'une transaction en cours, elle demande un verrou exclusif sur l'objet « identifiant de transaction » de cette transaction, qui ne lui est naturellement pas accordé. La transaction est donc placée en attente.
- Enfin, quand l'autre transaction possédant le verrou se termine (`COMMIT` ou `ROLLBACK`), son verrou sur l'objet « identifiant de transaction » est libéré, débloquant ainsi l'autre transaction, qui peut reprendre son travail.

Ce mécanisme ne nécessite pas un nombre de verrous mémoire proportionnel au nombre d'enregistrements à verrouiller, et simplifie le travail du gestionnaire de verrous, celui-ci ayant un nombre bien plus faible de verrous à gérer.

Le mécanisme exposé ici est évidemment simplifié.

4.9.4 La vue pg_locks

- `pg_locks` :
  - visualisation des verrous en place
  - tous types de verrous sur objets
- Complexe à interpréter :
  - verrous sur enregistrements pas directement visibles

C'est une vue globale à l'instance.

```
# \d pg_locks
                    Vue « pg_catalog.pg_locks »
      Colonne       |   Type                   | Collationnement | NULL-able | …
--------------------+--------------------------+-----------------+-----------+-
 locktype           | text                     |                 |           |
 database           | oid                      |                 |           |
 relation           | oid                      |                 |           |
 page               | integer                  |                 |           |
 tuple              | smallint                 |                 |           |
 virtualxid         | text                     |                 |           |
 transactionid      | xid                      |                 |           |
 classid            | oid                      |                 |           |
 objid              | oid                      |                 |           |
 objsubid           | smallint                 |                 |           |
 virtualtransaction | text                     |                 |           |
 pid                | integer                  |                 |           |
 mode               | text                     |                 |           |
 granted            | boolean                  |                 |           |
 fastpath           | boolean                  |                 |           |
 waitstart          | timestamp with time zone |                 |           |
```

- `locktype` est le type de verrou, les plus fréquents étant `relation` (table ou index), `transactionid` (transaction), `virtualxid` (transaction virtuelle, utilisée tant qu'une transaction n'a pas eu à modifier de données, donc à stocker des identifiants de transaction dans des enregistrements) ;
- `database` est la base dans laquelle ce verrou est pris ;
- `relation` est l'OID de la relation cible si locktype vaut `relation` (ou `page` ou `tuple`) ;
- `page` est le numéro de la page dans une relation (pour un verrou de type `page` ou `tuple`) cible ;
- `tuple` est le numéro de l'enregistrement cible (quand verrou de type `tuple`) ;
- `virtualxid` est le numéro de la transaction virtuelle cible (quand verrou de type `virtualxid`) ;

- `transactionid` est le numéro de la transaction cible ;
- `classid` est le numéro d'OID de la classe de l'objet verrouillé (autre que relation) dans `pg_class`. Indique le catalogue système, donc le type d'objet, concerné. Aussi utilisé pour les advisory locks ;
- `objid` est l'OID de l'objet dans le catalogue système pointé par `classid` ;
- `objsubid` correspond à l'ID de la colonne de l'objet `objid` concerné par le verrou ;
- `virtualtransaction` est le numéro de transaction virtuelle possédant le verrou (ou tentant de l'acquérir si `granted` vaut `f`) ;
- `pid` est le PID (l'identifiant de processus système) de la session possédant le verrou ;
- `mode` est le niveau de verrouillage demandé ;
- `granted` signifie si le verrou est acquis ou non (donc en attente) ;
- `fastpath` correspond à une information utilisée surtout pour le débogage (*fastpath* est le mécanisme d'acquisition des verrous les plus faibles) ;
- `waitstart` indique depuis quand le verrou est en attente.

La plupart des verrous sont de type relation, `transactionid` ou `virtualxid`. Une transaction qui démarre prend un verrou virtualxid sur son propre `virtualxid`. Elle acquiert des verrous faibles (`ACCESS SHARE`) sur tous les objets sur lesquels elle fait des `SELECT`, afin de garantir que leur structure n'est pas modifiée sur la durée de la transaction. Dès qu'une modification doit être faite, la transaction acquiert un verrou exclusif sur le numéro de transaction qui vient de lui être affecté. Tout objet modifié (table) sera verrouillé avec `ROW EXCLUSIVE`, afin d'éviter les `CREATE INDEX` non concurrents, et empêcher aussi les verrouillages manuels de la table en entier (`SHARE ROW EXCLUSIVE`).

4.9.5 Verrous - Paramètres

- Nombre :
  - `max_locks_per_transaction` (+ paramètres pour la sérialisation)
- Durée :
  - `lock_timeout` (éviter l'empilement des verrous)
  - `deadlock_timeout` (défaut 1 s)
- Trace :
  - `log_lock_waits`

Nombre de verrous :

`max_locks_per_transaction` sert à dimensionner un espace en mémoire partagée destinée aux verrous sur des objets (notamment les tables). Le nombre de verrous est :

```
max_locks_per_transaction × max_connections
```

Si les transactions préparées (liées aux transactions en deux phases[12]) sont activées, en montant `max_prepared_transactions` au-delà de 0, le nombre de verrous devient :

```
max_locks_per_transaction × (max_connections + max_prepared_transactions)
```

La valeur par défaut de 64 est largement suffisante la plupart du temps. Il peut arriver qu'il faille le monter, par exemple si l'on utilise énormément de partitions, mais le message d'erreur est explicite.

Le nombre maximum de verrous d'une session n'est pas limité à `max_locks_per_transaction`. C'est une valeur moyenne. Une session peut acquérir autant de verrous qu'elle le souhaite pourvu qu'au total la table de hachage interne soit assez grande. Les verrous de lignes sont stockés sur les lignes et donc potentiellement en nombre infini.

Pour la sérialisation, les verrous de prédicat possèdent des paramètres spécifiques. Pour économiser la mémoire, les verrous peuvent être regroupés par bloc ou relation (voir `pg_locks` pour le niveau de verrouillage). Les paramètres respectifs sont :

- `max_pred_locks_per_transaction` (64 par défaut) ;
- `max_pred_locks_per_page` (par défaut 2, donc 2 lignes verrouillées entraînent le verrouillage de tout le bloc, du moins pour la sérialisation) ;
- `max_pred_locks_per_relation` (voir la documentation[13] pour les détails).

Durées maximales de verrou :

Si une session attend un verrou depuis plus longtemps que `lock_timeout`, la requête est annulée. Il est courant de poser cela avant un ordre assez intrusif, même bref, sur une base utilisée. Par exemple, il faut éviter qu'un `VACUUM FULL`, s'il est bloqué par une transaction un peu longue, ne bloque lui-même toutes les transactions suivantes (phénomène d'empilement des verrous) :

```
postgres=# SET lock_timeout TO '3s' ;
SET
postgres=# VACUUM FULL t_grosse_table ;
ERROR:  canceling statement due to lock timeout
```

Il faudra bien sûr retenter le `VACUUM FULL` plus tard, mais la production n'est pas bloquée plus de 3 secondes.

PostgreSQL recherche périodiquement les *deadlocks* entre transactions en cours. La périodicité par défaut est de 1 s (paramètre `deadlock_timeout`), ce qui est largement suffisant la plupart du temps : les *deadlocks* sont assez rares, alors que la vérification est quelque chose de coûteux. L'une des transactions est alors arrêtée et annulée, pour que les autres puissent continuer :

```
postgres=*# DELETE FROM t_centmille_int WHERE i < 50000;
```

[12] https://docs.postgresql.fr/current/two-phase.html
[13] https://docs.postgresql.fr/current/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-RELATION

```
ERROR:  deadlock detected
DÉTAIL : Process 453259 waits for ShareLock on transaction 3010357;
blocked by process 453125.
Process 453125 waits for ShareLock on transaction 3010360;
blocked by process 453259.
ASTUCE : See server log for query details.
CONTEXTE : while deleting tuple (0,1) in relation "t_centmille_int"
```

Trace des verrous :

Pour tracer les attentes de verrous un peu longue, il est fortement conseillé de passer `log_lock_waits` à `on` (le défaut est `off`).

Le seuil est également défini par `deadlock_timeout` (1 s par défaut) Ainsi, une session toujours en attente de verrou au-delà de cette durée apparaîtra dans les traces :

```
LOG:  process 457051 still waiting for ShareLock on transaction 35373775
      after 1000.121 ms
DETAIL:  Process holding the lock: 457061. Wait queue: 457051.
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
```

S'il ne s'agit pas d'un *deadlock*, la transaction continuera, et le moment où elle obtiendra son verrou sera également tracé :

```
LOG:  process 457051 acquired ShareLock on transaction 35373775 after
      18131.402 ms
CONTEXT:  while deleting tuple (221,55) in relation "t_centmille_int"
STATEMENT:  DELETE FROM t_centmille_int ;
LOG:  duration: 18203.059 ms  statement: DELETE FROM t_centmille_int ;
```

4.10 DURÉES DE SESSIONS, TRANSACTIONS & ORDRES

4.10.1 Quelle durée pour les sessions & transactions ?

Divers seuils possibles, jamais globalement.

```
SET …_timeout TO '5s' ;
ALTER ROLE … IN DATABASE … SET ..._timeout TO '…s'
```

Paramètre	Cible du seuil
`lock_timeout`	Attente de verrou
`statement_timeout`	Ordre en cours
`idle_session_timeout`	Session inactive
`idle_in_transaction_session_timeout`	Transaction en cours, inactive
`transaction_timeout` (v17)	Transaction en cours

PostgreSQL possède divers seuils pour éviter des ordres, transactions ou sessions trop longues. Le dépassement d'un seuil provoque la fin et l'annulation de l'ordre, transaction ou session. Par défaut, aucun n'est activé.

Ne définissez jamais des paramètres globalement, les ordres de maintenance ou les batchs de nuit seraient aussi concernés, tout comme les superutilisateurs. Utilisez un ordre `SET` dans la session, ou `ALTER ROLE … IN DATABASE … SET …`.

Il est important que l'application sache gérer l'arrêt de connexion ou l'annulation d'un ordre ou d'une transaction et réagir en conséquence (nouvelle tentative, abandon avec erreur…). Dans le cas contraire, l'application pourrait rester déconnectée, et suivant les cas des données pourraient être perdues.

`lock_timeout` permet d'interrompre une requête qui mettrait trop de temps à acquérir son verrou. Il faut l'activer par précaution avant une opération lourde (un `VACUUM FULL` notamment) pour éviter un « empilement des verrous ». En effet, si l'ordre ne peut pas s'exécuter immédiatement, il bloque toutes les autres requêtes sur la table.

`statement_timeout` est la durée maximale d'un ordre. Défini au niveau d'un utilisateur ou d'une session, cet ordre permet d'éviter que des requêtes aberrantes chargent la base pendant des heures ou des jours ; ou encore que des requêtes « s'empilent » sur une base totalement saturée ou avec trop de contention. Autre exemple : les utilisateurs de supervision et autres utilisateurs non critiques.

À noter : l'extrait suivant d'une sauvegarde par `pg_dump` montre que l'outil inhibe ces paramètres par précaution, afin que la restauration n'échoue pas.

```
-- Dumped from database version 17.2 (Debian 17.2-1.pgdg120+1)
-- Dumped by pg_dump version 17.2 (Debian 17.2-1.pgdg120+1)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
…
```

L'idée peut être reprise pour des scripts batchs, par exemple.

4.10.2 Quelle durée pour une session ?

- Courte
  - coût & temps des connexions
  - pooler ?
- Longue
  - risque de saturation du nombre de connexions
  - (rare) gaspillage mémoire par les backends

Il est généralement conseillé d'utiliser des sessions assez longues, car la création d'une connexion à PostgreSQL est une opération lourde. Si l'applicatif ne cesse de se connecter/déconnecter, il faudra penser à un pooler (pgBouncer, ou côté applicatif).

Des sessions très longues et inactives ne sont en général pas un souci. Un pooler garde justement ses sessions longtemps. Mais si les sessions ne se ferment jamais, le nombre maximal de sessions (`max_connections`) peut être atteint, et de nouvelles connexions refusées.

De plus, pour les performances, il n'est pas très bon qu'il y ait des milliers de sessions ouvertes, malgré les progrès des versions récentes de PostgreSQL. Enfin, si la consommation mémoire du *backend* associé à une session est raisonnable, il arrive, exceptionnellement, qu'il se mette à occuper durablement de la mémoire (par exemple en cas d'accès à des milliers de tables et index durant son existence).

La durée maximale des sessions peut être réglée par `idle_session_timeout`, ou au niveau du pooler s'il y en a un.

4.10.3 Quelle durée pour une transaction ?

- Courte
  - synchronisation fréquente coûteuse
- Longue
  - verrous bloquants

Les transactions ouvertes et figées durablement sans faire de `COMMIT` ou `ROLLBACK` sont un problème sérieux, car l'autovacuum ne pourra pas nettoyer de ligne que cette session pourrait encore voir. Si l'applicatif ne peut être corrigé, ou si des utilisateurs ouvrent un outil quelconque sans le refermer, une solution est de définir `idle_in_transaction_session_timeout`, par exemple à une ou plusieurs heures.

Pour les performances, il faut éviter les sessions trop courtes car le coût de la synchronisation sur disque des journaux lors d'un `COMMIT` est coûteux (sauf table *unlogged* ou `synchronous_commit`). Typiquement, un traitement de batch ou de chargement regroupera de nombreuses opérations en une seule transaction.

D'un autre côté, chaque transaction maintient ses verrous jusqu'à sa fin, et peut donc bloquer d'autres transactions et ralentir toute l'application. En utilisation transactionnelle, il vaut donc mieux que les transactions soient courtes et n'en fassent pas plus que ce qui est dicté par le besoin fonctionnel, et si possible le plus tard possible dans la transaction.

`transaction_timeout` (à partir de PostgreSQL 17) peut alors servir comme sécurité en cas de problème. À noter que cette limite ne concerne pas les transactions préparées, liées au transactions en deux phases[14], et dont la longueur est parfois un souci.

[14] https://docs.postgresql.fr/current/two-phase.html

4.11 TOAST

4.11.1 Mécanisme TOAST

TOAST : *The Oversized-Attribute Storage Technique*
Que faire si une ligne dépasse d'un bloc ?

- Compresser
- Déporter dans une table
- Ou les deux
- Inutile de le faire dans l'applicatif
- Politique par champ
  - `PLAIN` / `MAIN` / `EXTERNAL` / `EXTENDED`

Principe :

Une ligne ne peut déborder d'un bloc, et un bloc fait 8 ko par défaut. Cela ne suffit pas pour certains champs qui peuvent être beaucoup plus longs, comme certains textes, mais aussi des types composés (`json`, `jsonb`, `hstore`), ou binaires (`bytea`), et même `numeric`.

Le mécanisme TOAST consiste en deux mécanismes. Le premier compresse le contenu des champs, mais ça ne suffit pas forcément. Le second consiste à découper les champs trop longs et à déporter les morceaux dans une autre table associée à la table principale, table gérée de manière transparente pour l'utilisateur. La combinaison des deux mécanismes est possible. Ces deux mécanismes permettent au final d'éviter qu'un enregistrement ne dépasse la taille d'un bloc.

Politiques de stockage :

Chaque champ possède une propriété de stockage :

```
CREATE TABLE demotoast (i int, t text, png bytea, j jsonb);
```

```
# \d+ unetable
                    Table « public.demotoast »
 Colonne |  Type   |  Col…  | NULL-able | Par défaut | Stockage | …
---------+---------+--------+-----------+------------+----------+--
 i       | integer |        |           |            | plain    |
 t       | text    |        |           |            | extended |
 png     | bytea   |        |           |            | extended |
 j       | jsonb   |        |           |            | extended |
Méthode d'accès : heap
```

Les différentes politiques de stockage sont :

- `PLAIN` : stockage uniquement dans la table, sans compression (champs numériques ou dates notamment) ;

- `MAIN` : stockage dans la table autant que possible, éventuellement compressé (politique rarement utilisée) ;
- `EXTERNAL` : stockage au besoin dans la table TOAST, sans chercher à compresser ;
- `EXTENDED` : stockage au besoin dans la table TOAST, éventuellement compressé (cas général des champs texte ou binaire).

Il est rare d'avoir à modifier ce paramétrage, mais cela arrive. Par exemple, certains longs champs (souvent binaires, par exemple du JPEG, des PNG, des PDF…) sont déjà compressés et il ne vaut pas la peine de gaspiller du CPU dans cette tâche. Dans le cas extrême où le champ compressé est plus grand que l'original, PostgreSQL revient à la valeur originale, mais du CPU a été consommé inutilement. Il peut alors être intéressant de passer de `EXTENDED` à `EXTERNAL`, pour un gain de temps parfois non négligeable :

```
ALTER TABLE demotoast ALTER COLUMN png SET STORAGE EXTERNAL ;
```

Lors d'un changement de mode de stockage, les données existantes ne sont pas modifiées. Il faut modifier les champs d'une manière ou d'une autre pour forcer le changement.

Performances :

Le découpage et la compression restent des opérations coûteuses. Il reste déconseillé de stocker des données binaires de grande taille dans une base de données ! De plus, cela surcharge les journaux de transaction et les sauvegardes.

Il faut rappeler qu'accéder aux champs « toastés » sans en avoir besoin est une mauvaise pratique :

```
INSERT INTO demotoast
SELECT i, rpad('0',100000,'0') AS t,
pg_read_binary_file ('/tmp/slonik.png'),
(SELECT jsonb_agg(to_json(rq)) FROM (SELECT * FROM pg_stat_all_tables) AS rq)
FROM generate_series (1,1000) i;

EXPLAIN (ANALYZE,VERBOSE,BUFFERS,SERIALIZE) SELECT i FROM demotoast ;

                            QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on public.demotoast  (cost=0.00..177.00 rows=1000 width=4) (actual
↪  time=0.008..0.178 rows=1000 loops=1)
   Output: i
   Buffers: shared hit=167
 Query Identifier: -2257406617252911738
 Planning Time: 0.029 ms
 Serialization: time=0.073 ms  output=9kB  format=text
 Execution Time: 0.321 ms

EXPLAIN (ANALYZE,VERBOSE,BUFFERS,SERIALIZE) SELECT * FROM demotoast ;

                            QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on public.demotoast  (cost=0.00..177.00 rows=1000 width=1196) (actual
↪  time=0.012..0.182 rows=1000 loops=1)
```

```
  Output: i, t, png, j
  Buffers: shared hit=167
Query Identifier: 698565989149231362
Planning Time: 0.055 ms
Serialization: time=397.480 ms  output=307084kB  format=text
  Buffers: shared hit=14500
Execution Time: 397.776 ms
```

L'effet du `SELECT *` est ici désastreux, avec 14 500 blocs lus supplémentaires. (Noter que l'effet n'est visible dans `EXPLAIN (ANALYZE)` qu'avec l'option `SERIALIZE`, qui n'est disponible qu'à partir de PostgreSQL 17. Sans cette option, `EXPLAIN(ANALYZE)` affiche un résultat trompeur par rapport aux requêtes réelles !)

4.11.2 TOAST & table de débordement

- Table de débordement `pg_toast_XXX`
  - masquée, transparente
- Jusqu'à 1 Go par champ (déconseillé)
  - texte, JSON, binaire…
  - compression optionnelle
- Une raison de plus d'éviter les `SELECT *`

Principe des tables de débordement :

Le débordement dans des tables séparées de la table principale a d'autres intérêts :

- la partie principale d'une table ayant des champs très longs est moins grosse, alors que les « gros champs » ont moins de chance d'être accédés systématiquement par le code applicatif ;
- ces champs peuvent aussi être compressés de façon transparente, avec souvent de gros gains en place ;
- si un `UPDATE` ne modifie pas un de ces champs « toastés », la table TOAST n'est pas mise à jour : le pointeur vers l'enregistrement de cette table est juste « cloné » dans la nouvelle version de l'enregistrement.

Les tables pg_toast_XXX :

À chaque table utilisateur se voit associée une table TOAST, et ce dès sa création si la table possède un champ « toastable ». Les enregistrements y sont stockés en morceaux (*chunks*) d'un peu moins de 2 ko. Tous les champs « toastés » d'une table se retrouvent dans la même table `pg_toast_XXX`, dans un espace de nommage séparé nommé `pg_toast`.

Pour l'utilisateur, les tables TOAST sont totalement transparentes. Un développeur doit juste savoir qu'il n'a pas besoin de déporter des champs texte (ou JSON, ou binaires…) imposants dans une table séparée pour des raisons de volumétrie de la table principale : PostgreSQL le fait déjà, et de manière efficace ! Il est donc souvent inutile de se donner la peine de compresser les données au niveau applicatif juste pour réduire le stockage.

La présence de ces tables n'apparaît guère que dans `pg_class`, par exemple ainsi :

```
SELECT * FROM pg_class c
WHERE c.relname = 'demotoast'
OR c.oid = (SELECT reltoastrelid FROM pg_class
            WHERE relname = 'demotoast');
```

```
-[ RECORD 1 ]-------+-----------------
oid                 | 1315757
relname             | pg_toast_1315754
relnamespace        | 99
reltype             | 0
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 1315757
reltablespace       | 0
relpages            | 9500
reltuples           | 38000
relallvisible       | 9500
reltoastrelid       | 0
relhasindex         | t
…
-[ RECORD 2 ]-------+-----------------
oid                 | 1315754
relname             | demotoast
relnamespace        | 2200
reltype             | 1315756
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 1315754
reltablespace       | 0
relpages            | 167
reltuples           | 1000
relallvisible       | 0
reltoastrelid       | 1315757
relhasindex         | f
…
```

La partie TOAST est une table à part entière, avec une clé primaire. On ne peut ni ne doit y toucher !

```
\d+ pg_toast.pg_toast_1315754

Table TOAST « pg_toast.pg_toast_1315754 »
  Colonne   |  Type   | Stockage
------------+---------+----------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
```

```
Table propriétaire : « public.demotoast »
Index :
    "pg_toast_1315754_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Méthode d'accès : heap
```

L'index de la table TOAST est toujours utilisé pour accéder aux *chunks*.

La volumétrie des différents éléments (partie principale, TOAST, index éventuels) peut se calculer grâce à cette requête dérivée du wiki[15] :

```
SELECT
    oid AS table_oid,
    c.relnamespace::regnamespace || '.' || relname AS TABLE,
    reltoastrelid,
    reltoastrelid::regclass::text AS table_toast,
    reltuples AS nb_lignes_estimees,
    pg_size_pretty(pg_table_size(c.oid)) AS " Table (dont TOAST)",
    pg_size_pretty(pg_relation_size(c.oid)) AS "  Heap",
    pg_size_pretty(pg_relation_size(reltoastrelid)) AS "  Toast",
    pg_size_pretty(pg_indexes_size(reltoastrelid)) AS  "  Toast (PK)",
    pg_size_pretty(pg_indexes_size(c.oid)) AS " Index",
    pg_size_pretty(pg_total_relation_size(c.oid)) AS "Total"
FROM  pg_class c
WHERE relkind = 'r'
AND   relname = 'demotoast'
\gx
```

```
-[ RECORD 1 ]-------+-------------------------
table_oid           | 1315754
table               | public.demotoast
reltoastrelid       | 1315757
table_toast         | pg_toast.pg_toast_1315754
nb_lignes_estimees  | 1000
 Table (dont TOAST) | 76 MB
  Heap              | 1336 kB
  Toast             | 74 MB
  Toast (PK)        | 816 kB
 Index              | 0 bytes
Total               | 76 MB
```

On constate que la plus grande partie de la volumétrie de la table est en fait dans la partie TOAST.

La taille des éventuels index sur les champs susceptibles d'être toastés est comptabilisée avec tous les index de la table (la clé primaire de la table TOAST est à part).

Détails du mécanisme TOAST :

Les détails techniques du mécanisme TOAST sont dans la documentation officielle[16]. En résumé, le mécanisme TOAST est déclenché sur un enregistrement quand la taille d'un enregistrement dépasse 2 ko. Les champs « toastables » peuvent alors être compressés pour que la taille de l'enregistrement redescende en-dessous de 2 ko. Si cela ne suffit pas, des champs sont alors découpés et déportés vers la table TOAST. Dans ces champs de la table principale, l'enregistrement ne contient plus qu'un pointeur vers la table TOAST associée.

[15] https://wiki.postgresql.org/wiki/Disk_Usage
[16] https://doc.postgresql.fr/current/storage-toast.html

Un champ MAIN peut tout de même être stocké dans la table TOAST, si l'enregistrement, même compressé, dépasse 2 ko : mieux vaut « toaster » que d'empêcher l'insertion.

Cette valeur de 2 ko convient généralement. Au besoin, on peut l'augmenter en utilisant le paramètre de stockage `toast_tuple_target` ainsi :

```
ALTER TABLE demotoast SET (toast_tuple_target = 3000);
```

mais cela est rarement utile.

Le mécanisme peut différer entre deux lignes pour un même champ, en fonction de la taille réelle de celui-ci sur la ligne.

Il est possible d'avoir le détail champ par champ :

```
SELECT i,
        pg_column_compression (png) AS png_c,
        pg_column_toast_chunk_id (png) AS ck_png,
        pg_column_size(png) AS png_size,
        pg_column_compression (j) AS j_c,
        pg_column_toast_chunk_id (j) AS ck_j,
        pg_column_size (j) AS j_size
FROM demotoast
LIMIT 3 ;
```

```
 i | png_c | ck_png  | png_size | j_c  |  ck_j   | j_size
---+-------+---------+----------+------+---------+--------
 1 |       | 1315760 |    58635 | pglz | 1315759 |  14418
 2 |       | 1315762 |    58635 | pglz | 1315761 |  14418
 3 |       | 1315764 |    58635 | pglz | 1315763 |  14418
```

(La fonction `pg_column_compression()` existe depuis PostgreSQL 14 et indique l'algorithme de compression éventuel (voir plus bas), et `pg_column_toast_chunk_id()` depuis PostgreSQL 17.)

On constate que le champ `png` n'est pas compressé (puisqu'on l'a défini comme `EXTERNAL`), et que le champ de `j` de type `jsonb` est compressé avec l'algorithme indiqué. Chaque champ de chaque ligne a son `chunk_id` propre qui permet de compter le nombre de *chunks*

```
SELECT chunk_id, count(*)
  FROM pg_toast.pg_toast_1315754
 WHERE chunk_id IN ( 1315760, 1315759 )
GROUP BY 1;
```

```
 chunk_id | count
----------+-------
  1315759 |     8
  1315760 |    30
```

En multipliant par 2 ko, on retrouve à peu près les tailles sur disque (compressées ou pas) renvoyées par `pg_column_size()`.

<!– TODO : faire une requête pour savoir le % de compressé et de déporté dans chaque champ de chaque table ? ça réclame du dynamique… – > KB ? –>

Administration des tables TOAST :

Les tables TOAST restent forcément dans le même tablespace que la table principale. Leur maintenance (notamment le nettoyage par `autovacuum`) s'effectue en même temps que la table principale, comme le montre un `VACUUM (VERBOSE)`. Une variante de l'ordre permet d'ailleurs de passer outre le nettoyage d'une table TOAST pour accélérer un nettoyage urgent (depuis PostgreSQL 14) :

```
VACUUM (VERBOSE, PROCESS_TOAST off) longs_textes ;
```

4.11.3 TOAST & compression

- `pglz` (`zlib`) : défaut
- `lz4` à préférer
  - généralement plus rapide
  - compression équivalente (à vérifier)
- Mise en place :

```
default_toast_compression = lz4
```

ou :

```
ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4 ;
```

Depuis la version 14, il est possible de modifier l'algorithme de compression. Ceci est défini par le nouveau paramètre `default_toast_compression` dont la valeur par défaut est :

```
SHOW default_toast_compression ;
```

```
 default_toast_compression
---------------------------
 pglz
```

c'est-à-dire que PostgreSQL utilise la zlib, seule compression disponible jusqu'en version 13 incluse.

À partir de la version 14, un nouvel algorithme `lz4` est disponible (et intégré dans les paquets distribués par le PGDG).

De manière générale, l'algorithme `lz4` ne compresse pas mieux les données courantes que `pglz`, mais cela dépend des usages. Surtout, `lz4` est **beaucoup** plus rapide à compresser, et parfois à décompresser.
Nous conseillons d'utiliser `lz4` pour la compression de vos TOAST, même si, en toute rigueur, l'arbitrage entre consommations CPU en écriture ou lecture et place disque ne peut se faire qu'en testant soigneusement avec les données réelles.

Pour changer l'algorithme, vous pouvez :

- soit changer la valeur par défaut dans `postgresql.conf` :

```
default_toast_compression = lz4
```

- soit déclarer la méthode de compression à la création de la table :

```
CREATE TABLE t1 (
  c1 bigint GENERATED ALWAYS AS identity,
  c2 text   COMPRESSION lz4
  ) ;
```

- soit après création :

```
ALTER TABLE t1 ALTER COLUMN c2 SET COMPRESSION lz4 ;
```

`lz4` accélère aussi les restaurations logiques comportant beaucoup de données toastées et compressées. Si `lz4` n'a pas été activé par défaut, il peut être utilisé dès le chargement :

```
$ PGOPTIONS='-c default_toast_compression=lz4'  pg_restore …
```

Des lignes d'une même table peuvent être compressées de manières différentes. En effet, l'utilisation de `SET COMPRESSION` sur une colonne préexistante ne recompresse pas les données de la table TOAST associée. De plus, des données toastées lues par une requête, puis réinsérées sans être modifiées, sont recopiées vers les champs cibles telles quelles, sans étapes de décompression/recompression, et ce même si la compression de la cible est différente, même s'il s'agit d'une autre table. Même un `VACUUM FULL` sur la table principale réécrit la table TOAST, en recopiant simplement les champs compressés tels quels.

Pour forcer la recompression de toutes les données d'une colonne, il faut modifier leur contenu. Et en fait, c'est peu intéressant car l'écart de volumétrie est généralement faible. Noter qu'il existe une fonction `pg_column_compression (nom_champ)`[17] pour consulter la compression d'un champ sur chaque ligne concernée.

Pour aller plus loin :

- Blog Dalibo : Les mains dans le cambouis #2 - Le mécanisme de TOAST[18] (2024)
- Blog Fujitsu PostgreSQL : What is the new LZ4 TOAST compression in PostgreSQL 14, and how fast is it?[19] (2021), avec notamment un benchmark montrant l'intérêt de la compression `lz4`.

[17] https://docs.postgresql.fr/current/functions-admin.html#FUNCTIONS-ADMIN-DBOBJECT
[18] https://blog.dalibo.com/2024/02/27/toast.html
[19] https://www.postgresql.fastware.com/blog/what-is-the-new-lz4-toast-compression-in-postgresql-14

4.12 CONCLUSION

- PostgreSQL dispose d'une implémentation MVCC complète, permettant :
  - que les lecteurs ne bloquent pas les écrivains
  - que les écrivains ne bloquent pas les lecteurs
  - que les verrous en mémoire soient d'un nombre limité
- Cela impose par contre une mécanique un peu complexe, dont les parties visibles sont la commande `VACUUM` et le processus d'arrière-plan autovacuum.

4.12.1 Questions

N'hésitez pas, c'est le moment !

4.13 QUIZ

https://dali.bo/m4_quiz

4.14 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m4_solutions.

4.14.1 Niveaux d'isolation READ COMMITTED et REPEATABLE READ

But : Découvrir les niveaux d'isolation

Créer une nouvelle base de données nommée **b2**.

Dans la base de données **b2**, créer une table `t1` avec deux colonnes `c1` de type integer et `c2` de type `text`.

Insérer 5 lignes dans table `t1` avec des valeurs de `(1, 'un')` à `(5, 'cinq')`.

Ouvrir une transaction.

Lire les données de la table `t1`.

Depuis une autre session, mettre en majuscules le texte de la troisième ligne de la table `t1`.

Revenir à la première session et lire de nouveau toute la table `t1`.

Fermer la transaction et ouvrir une nouvelle transaction, cette fois-ci en `REPEATABLE READ`.

Lire les données de la table `t1`.

Depuis une autre session, mettre en majuscules le texte de la quatrième ligne de la table `t1`.

Revenir à la première session et lire de nouveau les données de la table `t1`. Que s'est-il passé ?

4.14.2 Niveau d'isolation SERIALIZABLE (Optionnel)

But : Découvrir le niveau d'isolation *serializable*

Une table de comptes bancaires contient 1000 clients, chacun avec 3 lignes de crédit et 600 € au total :

```
CREATE TABLE mouvements_comptes
(client int,
 mouvement numeric NOT NULL DEFAULT 0
 );
CREATE INDEX on mouvements_comptes (client) ;

-- 3 clients, 3 lignes de +100, +200, +300 €
INSERT INTO mouvements_comptes (client, mouvement)
SELECT i, j * 100
FROM generate_series(1, 1000) i
CROSS JOIN generate_series(1, 3) j ;
```

Chaque mouvement donne lieu à une ligne de crédit ou de débit. Une ligne de crédit correspondra à l'insertion d'une ligne avec une valeur `mouvement` positive. Une ligne de débit correspondra à l'insertion d'une ligne avec une valeur `mouvement` négative. **Nous exigeons que le client ait toujours un solde positif**. Chaque opération bancaire se déroulera donc dans une transaction, qui se terminera par l'appel à cette procédure de test :

```
CREATE PROCEDURE verifie_solde_positif (p_client int)
LANGUAGE plpgsql
AS $$
DECLARE
    solde   numeric ;
BEGIN
    SELECT  round(sum (mouvement), 0)
    INTO    solde
    FROM    mouvements_comptes
    WHERE   client = p_client ;
    IF solde < 0 THEN
        -- Erreur fatale
        RAISE EXCEPTION 'Client % - Solde négatif : % !', p_client, solde ;
    ELSE
        -- Simple message
        RAISE NOTICE 'Client % - Solde positif : %', p_client, solde ;
    END IF ;
END ;
$$ ;
```

Au sein de trois transactions successives :

- insérer successivement 3 mouvements de **débit** de 300 € pour le client **1**
- chaque transaction doit finir par `CALL verifie_solde_positif (1);` avant le `COMMIT`

- la sécurité fonctionne-t-elle ?

Pour le client **2**, ouvrir deux transactions en parallèle :

- dans chacune, procéder à retrait de 500 € ;
- dans chacune, appeler `CALL verifie_solde_positif (2)` ;
- dans chacune, valider la transaction ;
- la règle du solde positif est-elle respectée ?

- Reproduire avec le client **3** le même scénario de deux débits parallèles de 500 €, mais avec des transactions sérialisables : (`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE`).
- Avant chaque `COMMIT`, consulter la vue `pg_locks` pour la table `mouvements_comptes` :

```
SELECT locktype, mode, pid, granted FROM pg_locks
WHERE relation = (SELECT oid FROM pg_class WHERE relname = 'mouvements_comptes')
 ↪  ;
```

4.14.3 Effets de MVCC

But : Voir l'effet du MVCC dans les lignes

Créer une nouvelle table `t2` avec deux colonnes : un entier et un texte.

Insérer 5 lignes dans la table `t2`, de `(1, 'un')` à `(5, 'cinq')`.

Lire les données de la table `t2`.

Commencer une transaction. Mettre en majuscules le texte de la troisième ligne de la table `t2`.

Lire les données de la table `t2`. Que faut-il remarquer ?

Ouvrir une autre session et lire les données de la table `t2`. Que faut-il observer ?

Afficher `xmin` et `xmax` lors de la lecture des données de la table `t2`, dans chaque session.

Récupérer maintenant en plus le `ctid` lors de la lecture des données de la table `t2`, dans chaque session.

Valider la transaction.

Installer l'extension `pageinspect`.

La documentation[a] indique comment décoder le contenu du bloc 0 de la table `t2` :

```
SELECT * FROM heap_page_items(get_raw_page('t2', 0)) ;
```

Que faut-il remarquer ?

[a] https://docs.postgresql.fr/current/pageinspect.html

- Lancer `VACUUM` sur `t2`.
- Relancer la requête avec `pageinspect`.
- Comment est réorganisé le bloc ?

Pourquoi l'autovacuum n'a-t-il pas nettoyé encore la table ?

4.14.4 Verrous

But : Trouver des verrous

Ouvrir une transaction et lire les données de la table `t1`. Ne pas terminer la transaction.

Ouvrir une autre transaction, et tenter de supprimer la table `t1`.

Lister les processus du serveur PostgreSQL. Que faut-il remarquer ?

Depuis une troisième session, récupérer la liste des sessions en attente avec la vue `pg_stat_activity`.

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

Récupérer le nom de l'objet dont le verrou n'est pas récupéré.

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

Retrouver les informations sur la session bloquante.

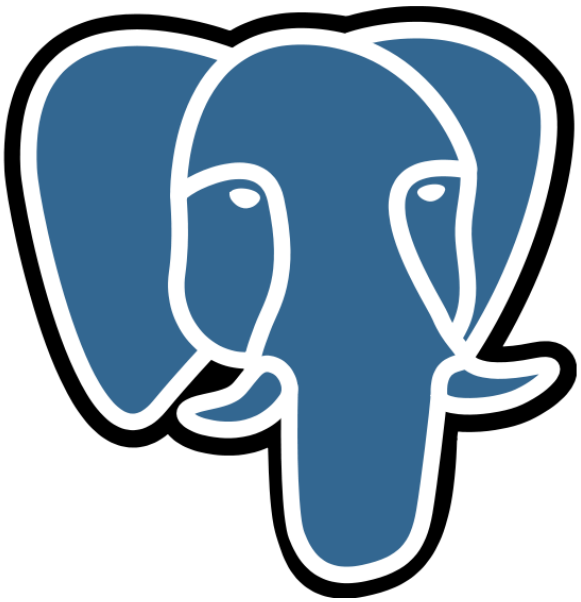
Retrouver cette information avec la fonction `pg_blocking_pids`.

Détruire la session bloquant le `DROP TABLE`.

Pour créer un verrou, effectuer un `LOCK TABLE` dans une transaction qu'il faudra laisser ouverte.

Construire une vue `pg_show_locks` basée sur `pg_stat_activity`, `pg_locks`, `pg_class` qui permette de connaître à tout moment l'état des verrous en cours sur la base : processus, nom de l'utilisateur, âge de la transaction, table verrouillée, type de verrou.

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 `vacuumdb` (shell, pour appels en masse)
  - manuel par `VACUUM` (SQL)
  - par le démon `autovacuum` (seuils)

`VACUUM` est né du besoin de nettoyer les lignes mortes. Au fil du temps il a été couplé à d'autres ordres (`ANALYZE`, `VACUUM FREEZE`) et s'est occupé d'autres opérations de maintenance (création de la *visibility map* par exemple). Des options permettent de réguler son activité. Son paramétrage n'est donc pas toujours très clair.

L'outil `vacuumdb` est un outil qui génère des ordres `VACUUM` après s'être connecté à PostgreSQL. Il reprend simplement les options de `VACUUM` (voir sa page de manuel[1]). Il est plutôt destiné aux appels depuis le système ou comme tâche planifiée. Par rapport à un appel en SQL, `vacuumdb` facilite les appels en masse, avec notamment ces options :

- `--all` pour nettoyer toutes les bases de données les unes après les autres ;
- `--jobs` pour paralléliser sur plusieurs sessions.

`autovacuum` est un processus de l'instance PostgreSQL. Il est activé par défaut, et il fortement conseillé de le conserver ainsi. Dans le cas général, son fonctionnement convient et il ne gênera pas les utilisateurs. Au contraire, il faudra parfois le rendre plus agressif.

L'autovacuum ne gère pas toutes les variantes de `VACUUM` (notamment pas le `FULL`).

[1] https://docs.postgresql.fr/current/app-vacuumdb.html

5.3 FONCTIONNEMENT DE VACUUM

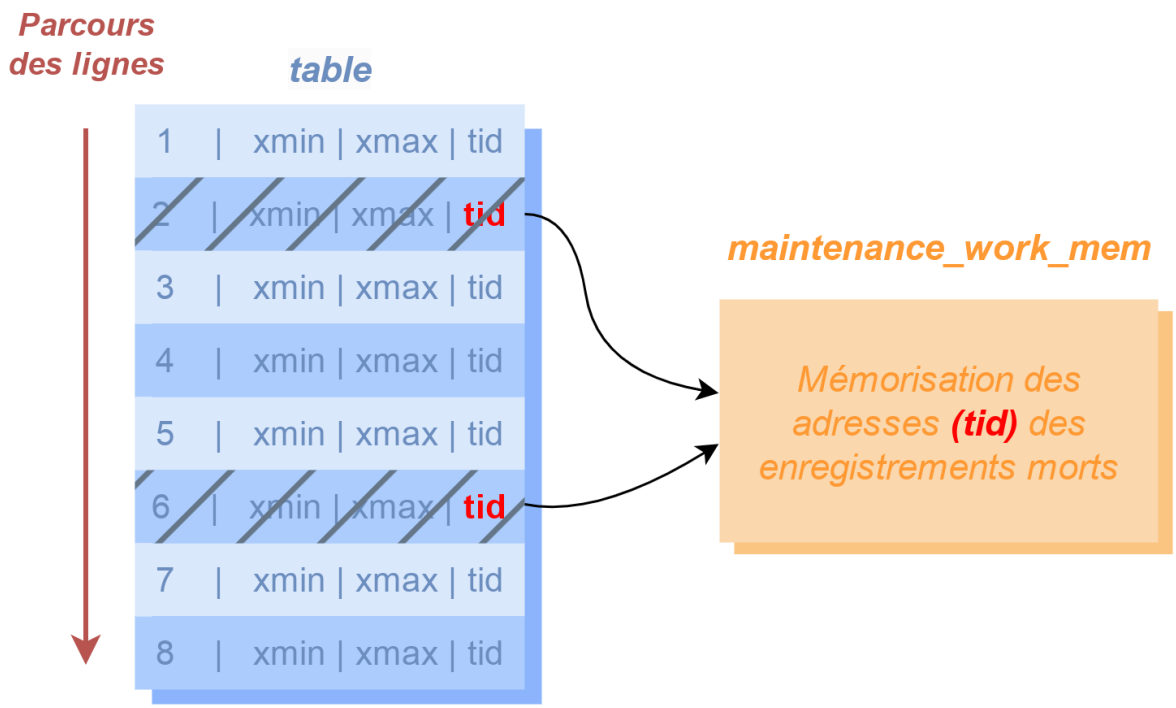


Figure 5/ .1: Phase 1/3 : recherche des enregistrements morts

Un ordre `VACUUM` vise d'abord à nettoyer les lignes mortes.

Le traitement `VACUUM` se déroule en trois passes. Cette première passe parcourt la table à nettoyer, à la recherche d'enregistrements morts. Un enregistrement est mort s'il possède un `xmax` qui correspond à une transaction validée, et que cet enregistrement n'est plus visible dans l'instantané d'aucune transaction en cours sur la base. D'autres lignes mortes portent un `xmin` d'une transaction annulée.

L'enregistrement mort ne peut pas être supprimé immédiatement : des enregistrements d'index pointent vers lui et doivent aussi être nettoyés. La session effectuant le `VACUUM` garde en mémoire la liste des adresses des enregistrements morts, à hauteur d'une quantité indiquée par le paramètre `maintenance_work_mem`. Si cet espace est trop petit pour contenir tous les enregistrements morts, `VACUUM` effectue plusieurs séries de ces trois passes.

5.3.1 Fonctionnement de VACUUM (suite)

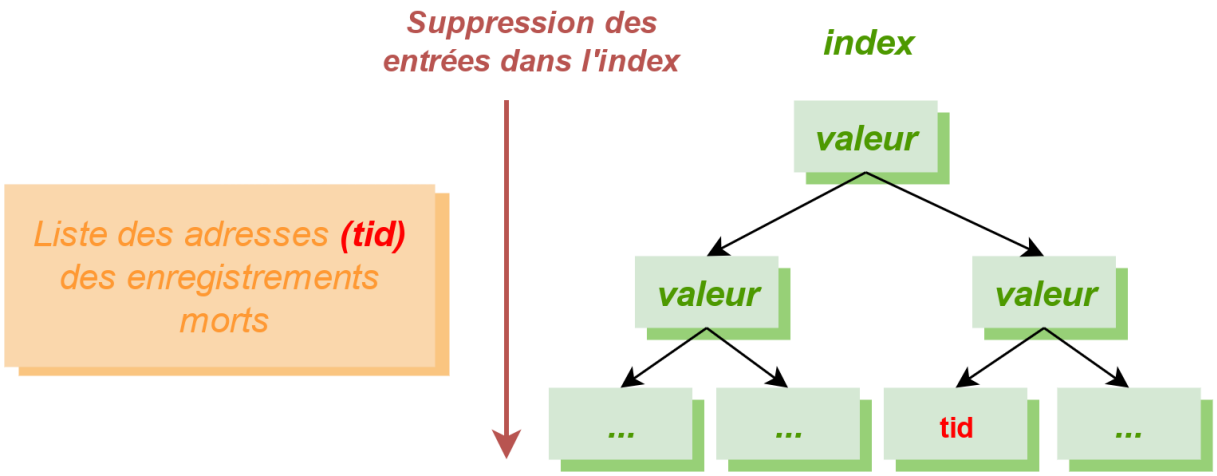


Figure 5/ .2: Phase 2/3 : nettoyage des index

La seconde passe se charge de nettoyer les entrées d'index. `VACUUM` possède une liste de `tid` (`tuple id`) à invalider. Il parcourt donc tous les index de la table à la recherche de ces `tid` et les supprime. En effet, les index sont triés afin de mettre en correspondance une valeur de clé (la colonne indexée par exemple) avec un `tid`. Il n'est par contre pas possible de trouver un `tid` directement. Les pages entièrement vides sont supprimées de l'arbre et stockées dans la liste des pages réutilisables, la *Free Space Map* (FSM).

Pour gagner du temps si le temps presse, cette phase peut être ignorée de deux manières. La première est de désactiver l'option `INDEX_CLEANUP` :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) nom_table ;
```

À partir de la version 14, un autre mécanisme, automatique cette fois, a été ajouté. Le but est toujours d'exécuter rapidement le `VACUUM`, mais uniquement pour éviter le *wraparound*. Quand la table atteint l'âge, très élevé, de 1,6 milliard de transactions (défaut des paramètres `vacuum_failsafe_age` et `vacuum_multixact_failsafe_age`), un `VACUUM` simple va automatiquement désactiver le nettoyage des index pour nettoyer plus rapidement la table et permettre d'avancer l'identifiant le plus ancien de la table.

Cette phase de nettoyage des index peut être parallélisée (clause `PARALLEL`), chaque index pouvant être traité par un CPU.

5.3.2 Fonctionnement de VACUUM (suite)

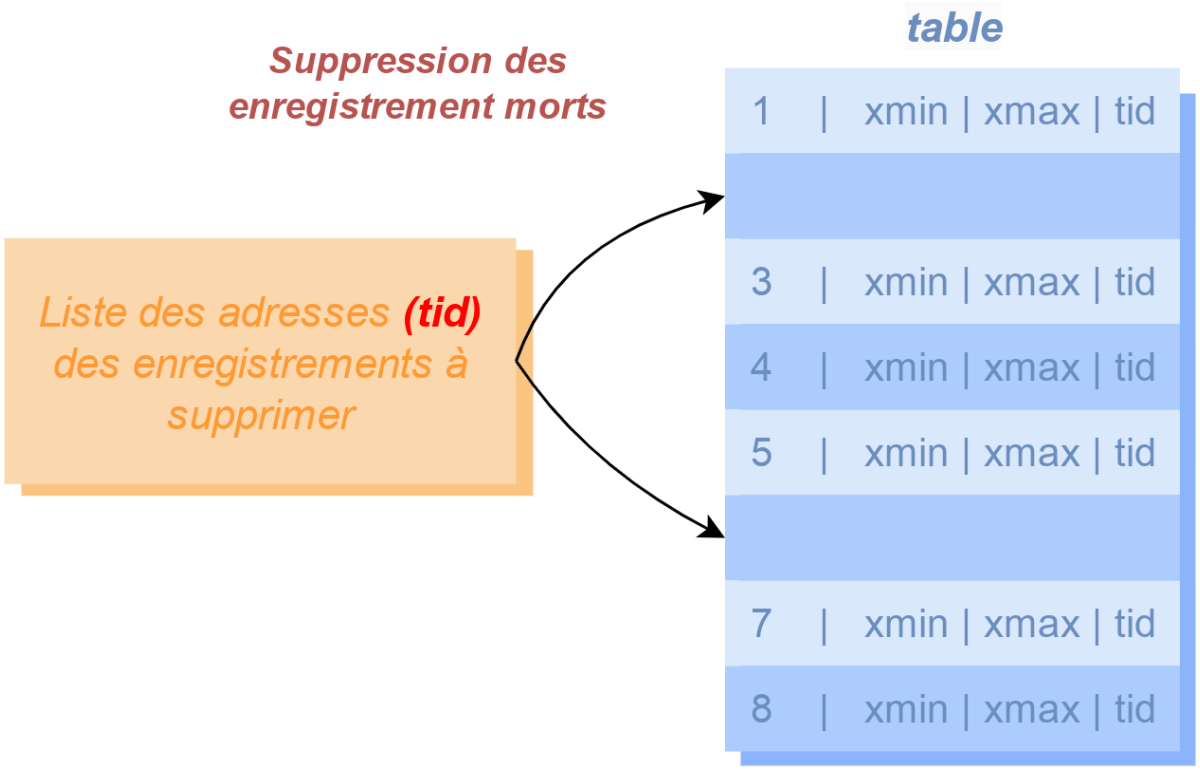


Figure 5/ .3: Phase 3/3 : suppression des enregistrements morts

NB : L'espace est rarement rendu à l'OS !

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements morts identifiés, ceux-ci peuvent disparaître. C'est le rôle de cette passe. Quand un enregistrement est supprimé d'un bloc, ce bloc est complètement réorganisé afin de consolider l'espace libre. Cet espace est renseigné dans la *Free Space Map* (FSM).

Une fois cette passe terminée, si le parcours de la table n'a pas été terminé lors de la passe précédente, le travail reprend où il en était du parcours de la table.

Si les derniers blocs de la table sont vides, ils sont rendus au système (si le verrou nécessaire peut être obtenu, et si l'option `TRUNCATE` n'est pas `off`). C'est le seul cas où `VACUUM` réduit la taille de la table. Les espaces vides (et réutilisables) au milieu de la table constituent le *bloat* (littéralement « boursouflure » ou « gonflement », que l'on peut aussi traduire par fragmentation).

Les statistiques d'activité sont aussi mises à jour.

5.4 LES OPTIONS DE VACUUM

- Quelle tâche ?
- Comment améliorer les performances ?
- Quelles options en cas d'urgence ?
- Autres options

5.4.1 Tâches d'un VACUUM

Ne pas confondre :

- `VACUUM` seul
  - nettoyage des lignes mortes, *visibility map*, *hint bits*
- `ANALYZE`
  - statistiques sur les données
- `VACUUM (ANALYZE)`
  - nettoyage & statistiques
- `VACUUM (FREEZE)`
  - gel des lignes
  - parfois gênant ou long
- `VACUUM FULL`
  - bloquant !
  - jamais lancé par l'autovacuum

VACUUM

Par défaut, `VACUUM` procède principalement au nettoyage des lignes mortes. Pour que cela soit efficace, il met à jour la *visibility map*, et la crée au besoin. Au passage, il peut geler certaines lignes rencontrées.

L'autovacuum le déclenchera sur les tables en fonction de l'activité.

Le verrou `SHARE UPDATE EXCLUSIVE` posé protège la table contre les modifications simultanées du schéma, et ne gêne généralement pas les opérations, sauf les plus intrusives (il empêche par exemple

un `LOCK TABLE`). L'autovacuum arrêtera spontanément un `VACUUM` qu'il aurait lancé et qui gênerait ; mais un `VACUUM` lancé manuellement continuera jusqu'à la fin.

VACUUM ANALYZE

`ANALYZE` existe en tant qu'ordre séparé, pour rafraîchir les statistiques sur un échantillon des données, à destination de l'optimiseur. L'autovacuum se charge également de lancer des `ANALYZE` en fonction de l'activité.

L'ordre `VACUUM ANALYZE` (ou `VACUUM (ANALYZE)`) force le calcul des statistiques sur les données en même temps que le `VACUUM`.

VACUUM FREEZE

`VACUUM FREEZE` procède au « gel » des lignes visibles par toutes les transactions en cours sur l'instance, afin de parer au problème du *wraparound* des identifiants de transaction.

Un ordre `FREEZE` n'existe pas en tant que tel.

Préventivement, lors d'un `VACUUM` simple, l'autovacuum procède au gel de certaines des lignes rencontrées. De plus, il lancera un `VACUUM FREEZE` sur une table dont les plus vieilles transactions dépassent un certain âge. Ce peut être très long, et très lourd en écritures si une grosse table doit être entièrement gelée d'un coup. Autrement, l'activité n'est qu'exceptionnellement gênée (voir plus bas).

L'opération de gel sera détaillée plus loin.

VACUUM FULL

L'ordre `VACUUM FULL` permet de reconstruire la table sans les espaces vides. C'est une opération très lourde, risquant de bloquer d'autres requêtes à cause du verrou exclusif qu'elle pose (on ne peut même plus lire la table !), mais il s'agit de la seule option qui permet de réduire la taille de la table au niveau du système de fichiers de façon certaine.

Il faut prévoir l'espace disque (la table est reconstruite à côté de l'ancienne, puis l'ancienne est supprimée). Les index sont reconstruits au passage. Un `VACUUM FULL` gèle agressivement les lignes, et effectue donc au passage l'équivalent d'un `FREEZE`.

L'autovacuum ne lancera jamais un `VACUUM FULL` !

Il existe aussi un ordre `CLUSTER`, qui permet en plus de trier la table suivant un des index.

5.4.2 Options de performance de VACUUM

- Index :
  - `PARALLEL`
- Taille du *buffer ring* (v16+)
  - `VACUUM (BUFFER_USAGE_LIMIT 2MB)`
  - paramètre `vacuum_buffer_usage_limit`
  - 256 ko ou 2 Mo par défaut, à monter
- `SKIP_DATABASE_STATS`, `ONLY_DATABASE_STATS` (v16+)
- Éviter les verrous
  - `SKIP_LOCKED`
  - `SET lock_timeout = '1s'`

PARALLEL :

L'option `PARALLEL` permet le traitement parallélisé des index. Le nombre indiqué après `PARALLEL` précise le niveau de parallélisation souhaité. Par exemple :

```
VACUUM (VERBOSE, PARALLEL 4) matable ;

INFO:  vacuuming "public.matable"
INFO:  launched 3 parallel vacuum workers for index cleanup (planned: 3)
```

SKIP_DATABASE_STATS, ONLY_DATABASE_STATS :

En fin d'exécution d'un `VACUUM`, même sur une seule table, le champ `pg_database.datfrozenxid` est mis à jour. Il contient le numéro de la transaction la plus ancienne non encore gélée dans toute la base de données. Cette opération impose de parcourir `pg_class` pour récupérer le plus ancien des numéros de transaction de chaque table (`relfrozenxid`), Or cette mise à jour n'est utile que pour l'autovacuum et le `VACUUM FREEZE`, et a rarement un caractère d'urgence.

Depuis la version 16, l'option `SKIP_DATABASE_STATS` demande au `VACUUM` d'ignorer la mise à jour de l'identifiant de transaction. Le principe est d'activer cette option pour les nettoyages en masse. À l'inverse, l'option `ONLY_DATABASE_STATS` demande de ne faire que la mise à jour du `datfrozenxid`, ce qui peut être fait une seule fois en fin de traitement.

L'outil `vacuumdb` procède ainsi automatiquement si le serveur est en version 16 minimum. Par exemple :

```
$ vacuumdb --echo --all
```

```
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « pgbench »
RESET search_path;
SELECT c.relname, ns.nspname FROM pg_catalog.pg_class c
 JOIN pg_catalog.pg_namespace ns ON c.relnamespace OPERATOR(pg_catalog.=) ns.oid
 LEFT JOIN pg_catalog.pg_class t ON c.reltoastrelid OPERATOR(pg_catalog.=) t.oid
 WHERE c.relkind OPERATOR(pg_catalog.=) ANY (array['r', 'm'])
 ORDER BY c.relpages DESC;
SELECT pg_catalog.set_config('search_path', '', false);
VACUUM (SKIP_DATABASE_STATS) public.pgbench_accounts;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_proc;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_attribute;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_description;
VACUUM (SKIP_DATABASE_STATS) pg_catalog.pg_statistic;
…
…
VACUUM (ONLY_DATABASE_STATS);
SELECT pg_catalog.set_config('search_path', '', false);
vacuumdb : exécution de VACUUM sur la base de données « postgres »
…
…
VACUUM (ONLY_DATABASE_STATS);
```

BUFFER_USAGE_LIMIT :

Cette option apparue en version 16 permet d'augmenter la taille de la mémoire partagée que peuvent utiliser `VACUUM`, et `ANALYZE`. Par défaut, cet espace nommé *buffer ring* n'est que de 256 ko de mémoire partagée, une valeur assez basse, cela pour protéger le cache (*shared buffers*). Si cette mémoire ne suffit pas, PostgreSQL doit recycler certains de ces buffers, d'où une écriture possible de journaux sur disque, avec un ralentissement à la clé.

Monter la taille du *buffer ring* avec `BUFFER_USAGE_LIMIT` permet une exécution plus rapide de VACUUM, et génère moins de journaux. De manière globale, on peut aussi modifier le paramètre `vacuum_buffer_usage_limit`. Les valeurs vont de 128 ko à 16 Go ; 0 désactive le *buffer ring*, il n'y a alors aucune limite en terme d'utilisation du cache. La taille par défaut n'est que de 2 Mo (et même seulement 256 ko jusque PostgreSQL 16 inclus).

Les machines modernes permettent de monter sans problème ce paramètre à quelques mégaoctets pour un gain en vitesse d'écriture très appréciable[2]. Il peut y avoir un impact négatif sur les autres requêtes si le débit en lecture ou écriture du `VACUUM` augmente trop. La valeur 0 est envisageable dans le cas d'une plage de maintenance où une purge du cache de PostgreSQL n'aurait pas de gros impact.

Exemples d'utilisation :

```
ANALYZE (BUFFER_USAGE_LIMIT '8MB');

VACUUM (BUFFER_USAGE_LIMIT '8MB');

VACUUM (ANALYZE, BUFFER_USAGE_LIMIT 0) ;  # sans limite

vacuumdb --analyze --buffer-usage-limit=8MB --echo -d pgbench
```

[2] https://blog.dalibo.com/2024/03/26/strategies-acces.html

```
…
VACUUM (SKIP_DATABASE_STATS, ANALYZE, BUFFER_USAGE_LIMIT '8MB')
       public.pgbench_accounts;
…
```

Les verrous, SKIP_LOCKED et lock_timeout :

L'option `SKIP_LOCKED` permet d'ignorer toute table pour laquelle la commande `VACUUM` ne peut pas obtenir immédiatement son verrou. Cela évite de bloquer le `VACUUM` sur une table, et permet d'éviter un empilement des verrous derrière celui que le `VACUUM` veut poser, surtout en cas de `VACUUM FULL`. La commande passe alors à la table suivante à traiter. Exemple :

```
VACUUM (FULL, SKIP_LOCKED) t_un_million_int, t_cent_mille_int ;

WARNING:  skipping vacuum of "t_un_million_int" --- lock not available
VACUUM
```

Une technique un peu différente est de paramétrer dans la session un petit délai avant abandon en cas de problème de verrou. Là encore, cela vise à limiter les empilements de verrou sur une base active. Par contre, comme l'ordre tombe immédiatement en erreur après le délai, il est plus adapté aux ordres ponctuels sur une table.

```
SET lock_timeout TO '100ms' ;
-- Un LOCK TABLE a été fait dans une autre session
VACUUM (verbose) pgbench_history,pgbench_tellers;

ERROR:  canceling statement due to lock timeout
Durée : 1000,373 ms (00:01,000)
RESET lock_timeout ;
```

5.4.3 Options pour un VACUUM en urgence

```
VACUUM (SKIP_DATABASE_STATS,    /* PG 16+ */
        INDEX_CLEANUP off,
        PROCESS_TOAST off,      /* PG 14+ */
        TRUNCATE      off,
        BUFFER_USAGE_LIMIT '1GB'  /* voire 0 (PG 16+) */
        ) ;
VACUUM (ONLY_DATABASE_STATS);   /* PG 16+ */
```

Ces options sont surtout destinées à désactiver certaines étapes d'un `VACUUM` quand le temps presse vraiment.

INDEX_CLEANUP :

L'option `INDEX_CLEANUP` (par défaut à `on` jusque PostgreSQL 13 compris) déclenche systématiquement le nettoyage des index. La commande `VACUUM` va supprimer les enregistrements de l'index qui

pointent vers des lignes mortes de la table. Quand il faut nettoyer des lignes mortes urgemment dans une grosse table, la valeur `off` fait gagner beaucoup de temps :

```
VACUUM (VERBOSE, INDEX_CLEANUP off) unetable ;
```

Les index peuvent être nettoyés plus tard lors d'un autre `VACUUM`, ou reconstruits (`REINDEX`).

Cette option existe aussi sous la forme d'un paramètre de stockage (`vacuum_index_cleanup`) propre à la table pour que l'autovacuum en tienne aussi compte.

En version 14, le nouveau défaut est `auto`, qui indique que PostgreSQL décide de faire ou non le nettoyage des index suivant la quantité d'entrées à nettoyer. Il faut au minimum 2 % d'éléments à nettoyer pour que le nettoyage ait lieu.

PROCESS_TOAST :

À partir de la version 14, cette option active ou non le traitement de la partie TOAST associée à la table (parfois la partie la plus volumineuse d'une table). Elle est activée par défaut. Son utilité est la même que pour `INDEX_CLEANUP`.

```
VACUUM (VERBOSE, PROCESS_TOAST off) unetable ;
```

TRUNCATE :

L'option `TRUNCATE` (à `on` par défaut) permet de tronquer les derniers blocs vides d'une table. `TRUNCATE off` évite d'avoir à poser un verrou exclusif certes court, mais parfois gênant.

Cette option existe aussi sous la forme d'un paramètre de stockage de table (`vacuum_truncate`).

BUFFER_USAGE_LIMIT :

Le *ring buffer* du `VACUUM` étant par défaut très réduit, une augmentation, même modeste, accélère les écritures. À la limite, s'il n'y a pas d'autre activité, on peut lui octroyer tout le cache de PostgreSQL (valeur `0`).

5.4.4 Autres options de VACUUM

- `VERBOSE`
- Ponctuellement :
  - `DISABLE_PAGE_SKIPPING`

VERBOSE :

Cette option affiche un grand nombre d'informations sur ce que fait la commande. En général, c'est une bonne idée de l'activer :

```
VACUUM (VERBOSE) pgbench_accounts_5 ;

INFO:  vacuuming "public.pgbench_accounts_5"
INFO:  scanned index "pgbench_accounts_5_pkey" to remove 9999999 row versions
DÉTAIL : CPU: user: 12.16 s, system: 0.87 s, elapsed: 18.15 s
INFO:  "pgbench_accounts_5": removed 9999999 row versions in 163935 pages
DÉTAIL : CPU: user: 0.16 s, system: 0.00 s, elapsed: 0.20 s
INFO:  index "pgbench_accounts_5_pkey" now contains 100000000 row versions in 301613
 ↪  pages
DÉTAIL : 9999999 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "pgbench_accounts_5": found 10000001 removable,
       10000051 nonremovable row versions in 327870 out of 1803279 pages
DÉTAIL : 0 dead row versions cannot be removed yet, oldest xmin: 1071186825
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 1475409 frozen pages.
0 pages are entirely empty.
CPU: user: 13.77 s, system: 0.89 s, elapsed: 19.81 s.
VACUUM
```

DISABLE_PAGE_SKIPPING :

Par défaut, PostgreSQL ne traite que les blocs modifiés depuis le dernier `VACUUM`, ce qui est un gros gain en performance (l'information est stockée dans la *Visibility Map*, qui est généralement un tout petit fichier).

Activer l'option `DISABLE_PAGE_SKIPPING` force l'analyse de tous les blocs de la table. La table est intégralement reparcourue. Ce peut être utile en cas de problème, notamment pour reconstruire cette *Visibility Map*.

Mélange des options :

Il est possible de mélanger toutes ces options presque à volonté, et de préciser plusieurs tables à nettoyer :

```
VACUUM (VERBOSE, ANALYZE, INDEX_CLEANUP off, TRUNCATE off,
        DISABLE_PAGE_SKIPPING) bigtable, smalltable ;
```

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`
- Manuel ou via autovacuum
- Pour `VACUUM FULL`
  - vue `pg_stat_progress_cluster`

La vue `pg_stat_progress_vacuum` contient une ligne par `VACUUM` (simple ou `FREEZE`) en cours d'exécution :

```
TABLE pg_stat_progress_vacuum \gx
```

```
-[ RECORD 1 ]------+--------------
pid                | 2603780
datid              | 1308955
datname            | grossetable
relid              | 1308962
phase              | scanning heap
heap_blks_total    | 163935
heap_blks_scanned  | 3631
```

```
heap_blks_vacuumed   | 0
index_vacuum_count   | 0
max_dead_tuple_bytes | 67108864
dead_tuple_bytes     | 0
num_dead_item_ids    | 0
indexes_total        | 0
indexes_processed    | 0
```

Dans cet exemple, le `VACUUM` exécuté par le PID 4299 a parcouru 86 665 blocs (soit 68 % de la table), et en a traité 86 664.

À noter que le suivi du `VACUUM` dans les index (les deux derniers champs) nécessite au moins PostgreSQL 17. C'est souvent la partie la plus longue d'un `VACUUM`.

Dans le cas d'un `VACUUM ANALYZE`, la seconde partie de recueil des statistiques pourra être suivie dans `pg_stat_progress_analyze` :

```
SELECT * FROM pg_stat_progress_analyze ;
```

```
-[ RECORD 1 ]-------------+---------------------------------
pid                       | 1938258
datid                     | 748619
datname                   | grossetable
relid                     | 748698
phase                     | acquiring inherited sample rows
sample_blks_total         | 1875
sample_blks_scanned       | 1418
ext_stats_total           | 0
ext_stats_computed        | 0
child_tables_total        | 16
child_tables_done         | 6
current_child_table_relid | 748751
```

Les vues précédentes affichent aussi bien les opérations lancées manuellement que celles décidées par l'autovacuum.

Par contre, pour un `VACUUM FULL`, il faudra suivre la progression au travers de la vue `pg_stat_progress_cluster`. Cette vue est utilisable aussi avec l'ordre `CLUSTER`, d'où le nom. Exemple :

```
SELECT * FROM pg_stat_progress_cluster \gx
```

```
-[ RECORD 1 ]-------+------------------
pid                 | 21157
datid               | 13444
datname             | postgres
relid               | 16384
command             | VACUUM FULL
phase               | seq scanning heap
cluster_index_relid | 0
heap_tuples_scanned | 13749388
heap_tuples_written | 13749388
heap_blks_total     | 199105
heap_blks_scanned   | 60839
index_rebuild_count | 0
```

Ces vues n'affichent que les opérations en cours, elles n'historisent rien. Si aucun `VACUUM` n'est en cours, elles n'afficheront rien.

5.5.2 Droit de lancer un VACUUM

- Propriétaire
- Superutilisateur
- Inclus dans droit de maintenance (v17)

```
GRANT MAINTAIN ON matable TO dba ; -- granulaire
GRANT pg_maintain         TO dba ; -- global
```

`VACUUM` est une opération de maintenance ne pouvant être effectuée que par :

- le propriétaire de la table ;
- un superutilisateur ;
- à partir de PostgreSQL 17, un utilisateur possédant le droit de maintenir la table, soit via le rôle `pg_maintain` ou par un `GRANT MAINTAIN` sur la table. (Ce droit de maintenance inclut d'autres droits, comme les droits de `ANALYZE`, `CLUSTER`, `LOCK TABLE`, `REFRESH MATERIALIZED VIEW` et `REINDEX`, mais pas celui de lire les données.)

Sans droit de maintenance, le `VACUUM` ne fonctionne pas pour un utilisateur non propriétaire de la table :

```
$ vacuumdb --username maintenance --table t1
vacuumdb: vacuuming database "b3"
WARNING:  permission denied to vacuum "t1", skipping it
```

Si **maintenance** devient membre du rôle `pg_maintain`, tout fonctionne :

```
$ psql --username postgres --command 'GRANT pg_maintain TO maintenance'
GRANT ROLE

$ vacuumdb --username maintenance
vacuumdb: vacuuming database "b3"
```

Avant la version 17, il est toujours possible d'avoir un groupe propriétaire de l'objet et d'ajouter un rôle de maintenance comme membre de ce groupe. Il aura le droit de lire les données. Mais si jamais un objet est créé par quelqu'un sans transférer la propriété au groupe, le rôle de maintenance ne peut faire d'opération de maintenance sur cet objet.

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 sans doute augmenter certaines ressources allouées au nettoyage (paramètre `autovacuum_vacuum_cost_limit`, voir plus bas), car les *workers* se les partagent ; sauf `maintenance_work_mem` qui, lui, est une quantité de RAM utilisable par chaque *worker* indépendamment.

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[3].)

Ces seuils pourront être adaptés table par table.

[3] https://docs.postgresql.fr/current/routine-vacuuming.html#AUTOVACUUM

5.6.3 Déclenchement de l'autovacuum (suite)

- Pour `VACUUM`
  - `autovacuum_vacuum_scale_factor` (20 %)
  - `autovacuum_vacuum_threshold` (50)
  - `autovacuum_vacuum_insert_threshold` (1000)
  - `autovacuum_vacuum_insert_scale_factor` (20 %)
- Pour `ANALYZE`
  - `autovacuum_analyze_scale_factor` (10 %)
  - `autovacuum_analyze_threshold` (50)
- Adapter pour une grosse table :

```
ALTER TABLE table_name SET (autovacuum_vacuum_scale_factor = 0.1);
```

Pour le `VACUUM`, si on considère les enregistrements morts (supprimés ou anciennes versions de lignes), la condition de déclenchement est :

```
nb_enregistrements_morts (pg_stat_all_tables.n_dead_tup) >=
    autovacuum_vacuum_threshold
  + autovacuum_vacuum_scale_factor × nb_enregs (pg_class.reltuples)
```

où, par défaut :

- `autovacuum_vacuum_threshold` vaut 50 lignes ;
- `autovacuum_vacuum_scale_factor` vaut 0,2 soit 20 % de la table.

Donc, par exemple, dans une table d'un million de lignes, modifier 200 050 lignes provoquera le passage d'un `VACUUM`.

Pour les grosses tables avec de l'historique, modifier 20 % de la volumétrie peut être extrêmement long. Quand l'`autovacuum` lance enfin un `VACUUM`, celui-ci a donc beaucoup de travail et peut durer longtemps et générer beaucoup d'écritures. Il est donc fréquent de descendre la valeur de `autovacuum_vacuum_scale_factor` à quelques pour cent sur les grosses tables. (Une alternative est de monter `autovacuum_vacuum_threshold` à un nombre de lignes élevé et de descendre `autovacuum_vacuum_scale_factor` à 0, mais il faut alors calculer le nombre de lignes qui déclenchera le nettoyage, et cela dépend fortement de la table et de sa fréquence de mise à jour.)

S'il faut modifier un paramètre, il est préférable de ne pas le faire au niveau global mais de cibler les tables où cela est nécessaire. Par exemple, l'ordre suivant réduit à 5 % de la table le nombre de lignes

à modifier avant que l'`autovacuum` y lance un `VACUUM` :

```
ALTER TABLE nom_table SET (autovacuum_vacuum_scale_factor = 0.05);
```

À partir de PostgreSQL 13, le `VACUUM` est aussi lancé quand il n'y a que des insertions, avec deux nouveaux paramètres et un autre seuil de déclenchement :

```
nb_enregistrements_insérés (pg_stat_all_tables.n_ins_since_vacuum) >=
    autovacuum_vacuum_insert_threshold
  + autovacuum_vacuum_insert_scale_factor × nb_enregs (pg_class.reltuples)
```

Pour l'`ANALYZE`, le principe est le même. Il n'y a que deux paramètres, qui prennent en compte toutes les lignes modifiées *ou* insérées, pour calculer le seuil :

```
nb_insert + nb_updates + nb_delete (n_mod_since_analyze) >=
    autovacuum_analyze_threshold + nb_enregs × autovacuum_analyze_scale_factor
```

où, par défaut :

- `autovacuum_analyze_threshold` vaut 50 lignes ;
- `autovacuum_analyze_scale_factor` vaut 0,1, soit 10 %.

Dans notre exemple d'une table, modifier 100 050 lignes provoquera le passage d'un `ANALYZE`.

Là encore, il est fréquent de modifier les paramètres sur les grosses tables pour rafraîchir les statistiques plus fréquemment.

Les insertions ont toujours été prises en compte pour `ANALYZE`, puisqu'elles modifient le contenu de la table. (Par contre, jusque PostgreSQL 12 inclus, `VACUUM` ne tenait pas compte des lignes insérées pour déclencher son nettoyage. Or, cela avait des conséquences pour les tables à insertion seule : gel de lignes retardé, *Index Only Scan* impossibles… Pour cette raison, à partir de la version 13, les insertions sont aussi prises en compte pour déclencher un `VACUUM`.)

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`
  - montés souvent à ½ à 1 Go
- Impact
  - `VACUUM`
  - construction d'index

`maintenance_work_mem` est la quantité de mémoire qu'un processus effectuant une opération de maintenance (c'est-à-dire n'exécutant pas des requêtes classiques comme `SELECT`, `INSERT`, `UPDATE`…) est autorisé à allouer pour sa tâche de maintenance.

Cette mémoire est utilisée lors de la construction d'index ou l'ajout de clés étrangères. et, dans le contexte de `VACUUM`, pour stocker les adresses des enregistrements pouvant être recyclés. Cette mémoire est remplie pendant la phase 1 du processus de `VACUUM`, tel qu'expliqué plus haut. Rappelons qu'une adresse d'enregistrement (`tid`, pour `tuple id`) a une taille de 6 octets et est composée du numéro dans la table, et du numéro d'enregistrement dans le bloc, par exemple `(0,1)`, `(3164,98)` ou `(5351510,42)`.

Le défaut de 64 Mo est assez faible. Si tous les enregistrements morts d'une table ne tiennent pas dans `maintenance_work_mem`, `VACUUM` est obligé de faire plusieurs passes de nettoyage, donc plusieurs parcours complets de chaque index. Une valeur assez élevée de `maintenance_work_mem` est donc conseillée : s'il est déjà possible de stocker plusieurs dizaines de millions d'enregistrements à effacer dans 256 Mo, 1 Go peut être utile lors de très grosses purges.

PostgreSQL 17 améliore beaucoup la consommation mémoire et la vitesse de nettoyage des index, et doit rendre rarissime les nettoyages d'index en plusieurs passes.

PostgreSQL 17 fait aussi disparaître une limite de 1 Go pour le nettoyage des index : `VACUUM` ne sait pas en utiliser plus jusque PostgreSQL 16. Par contre, l'indexation de grosses tables pourra toujours bénéficier d'une valeur supérieure à 1 Go.

Rappelons que plusieurs `VACUUM` ou `autovacuum` peuvent fonctionner simultanément et consommer chacun un `maintenance_work_mem` ! (Voir `autovacuum_max_workers` plus haut.)

`autovacuum_work_mem` permet de surcharger `maintenance_work_mem` spécifiquement pour l'autovacuum. Par défaut les deux sont identiques, et l'on conserve généralement cette configuration. Au besoin, `maintenance_work_mem` peut être surchargé le temps d'une session.

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 ou 2 /20)
  - jusque total de : `vacuum_cost_limit` (200)
  - pause : `vacuum_cost_delay` (en manuel : 0 ms !)
- Surcharge pour l'autovacuum
  - `autovacuum_vacuum_cost_limit` (identique)
  - `autovacuum_vacuum_cost_delay` (2 ms)
  - => débit en écriture max : ~40 Mo/s
- Pour accélérer : augmenter la limite

Principe :

Les paramètres suivant permettent d'éviter qu'un `VACUUM` ne gêne les autres sessions en saturant le disque. Le principe est de provoquer une pause après qu'une certaine activité a été réalisée.

Paramètres de coûts :

Ces trois paramètres « quantifient » l'activité de `VACUUM`, affectant un coût arbitraire à chaque fois qu'une opération est réalisée :

- `vacuum_cost_page_hit` : coût d'accès à chaque page présente dans le cache de PostgreSQL (valeur : 1) ;
- `vacuum_cost_page_miss` : coût d'accès à chaque page hors de ce cache (valeur : 2 à partir de la PostgreSQL 14, 10 auparavant) ;
- `vacuum_cost_page_dirty` : coût de modification d'une page, et donc de son écriture (valeur : 20).

Il est déconseillé de modifier ces paramètres de coût.

Pause :

Quand le coût cumulé atteint un seuil, l'opération de nettoyage marque une pause. Elle est gouvernée par deux paramètres :

- `vacuum_cost_limit` : coût cumulé à atteindre avant de déclencher la pause (défaut : 200) ;
- `vacuum_cost_delay` : temps à attendre (défaut : 0 ms !)

En conséquence, les `VACUUM` lancés manuellement (en ligne de commande ou via `vacuumdb`) ne sont **pas** freinés par ce mécanisme et peuvent donc entraîner de fortes écritures ! Mais c'est généralement ce que l'on veut dans un batch ou en urgence, et il vaut mieux alors être le plus rapide possible.

(Pour les urgences, rappelons que l'option `INDEX_CLEANUP off` ou `PROCESS_TOAST off` permettent aussi d'ignorer le nettoyage des index ou des TOAST.)

Paramétrage pour le VACUUM manuel :

Il est conseillé de ne pas toucher au paramétrage par défaut de `vacuum_cost_limit` et `vacuum_cost_delay`.

Si on doit lancer un `VACUUM` manuellement en limitant son débit, procéder comme suit dans une session :

```
-- Reprise pour le VACUUM du paramétrage d'autovacuum
SET vacuum_cost_limit = 200 ;
SET vacuum_cost_delay = '2ms' ;
VACUUM (VERBOSE) matable ;
```

Avec `vacuumdb`, il faudra passer par la variable d'environnement `PGOPTIONS`.

Paramétrage pour l'autovacuum :

Les `VACUUM` d'autovacuum, eux, sont par défaut limités en débit pour ne pas gêner l'activité normale de l'instance. Deux paramètres surchargent les précédents :

- `autovacuum_cost_limit` vaut par défaut -1, donc reprend la valeur 200 de `vacuum_cost_limit` ;
- `autovacuum_vacuum_cost_delay` vaut par défaut 2 ms.

Un (`autovacuum_`) `vacuum_cost_limit` à 200 consiste à traiter au plus 200 blocs lus en cache (car `vacuum_cost_page_hit` = 1), soit 1,6 Mo, avant de faire la pause de 2 ms. Si ces blocs doivent être écrits, on descend en-dessous de 10 blocs traités avant chaque pause (`vacuum_cost_page_dirty` = 20) avant la pause de 2 ms, d'où un débit en écriture maximal de l'autovacuum de 40 Mo/s Cela s'observe aisément par exemple avec `iotop`.

Ce débit est partagé équitablement entre les différents *workers* lancés par l'autovacuum (sauf paramétrage spécifique au niveau de la table).

Pour rendre l'`autovacuum` plus rapide, il est préférable d'augmenter `autovacuum_vacuum_cost_limit` au-delà de 200, plutôt que de réduire `autovacuum_vacuum_cost_delay` qui n'est qu'à 2 ms, pour ne pas monopoliser le disque. (Exception : les versions antérieures à la 12, car `autovacuum_vacuum_cost_delay` valait alors 20 ms et le débit en écriture saturait à 4 Mo/s ! La valeur 2 ms tient mieux compte des disques actuels.).

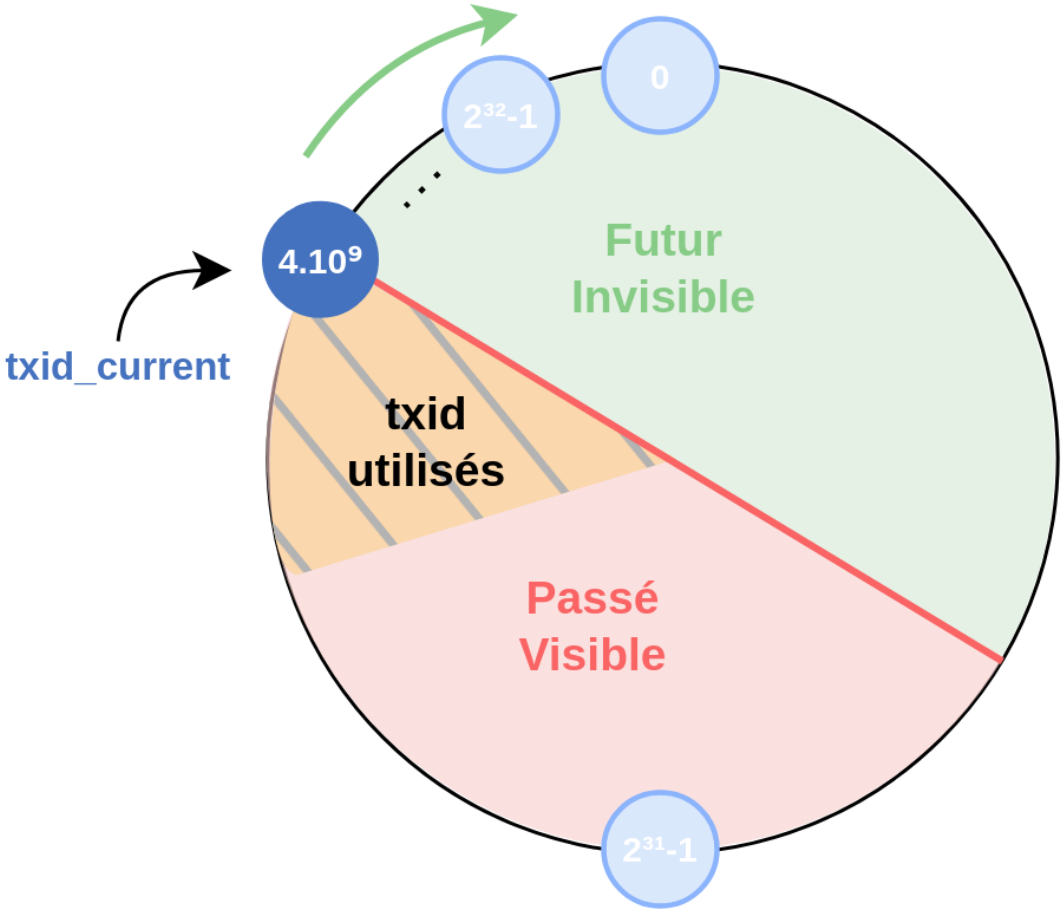
La prise en compte de la nouvelle valeur de la limite par les *workers* en cours sur les tables est automatique à partir de PostgreSQL 16. Dans les versions précédentes, il faut arrêter les *workers* en cours (avec `pg_cancel_backend()`) et attendre que l'`autovacuum` les relance. Quand `autovacuum_max_workers` est augmenté, prévoir aussi d'augmenter la limite. Il faut pouvoir assumer le débit supplémentaire pour les disques.

Sur le sujet, voir la conférence de Robert Haas à PGconf.EU 2023 à Prague[4].

[4] https://www.postgresql.eu/events/pgconfeu2023/sessions/session/4847/slides/432/Understanding%20and%20Fixing%20Autovacuum%20-%20PGCONF.EU%202023.pdf#page=14

5.7.4 Paramétrage du FREEZE (1)

Le but est de geler les numéros de transaction assez vite :



Rappelons que les numéros de transaction stockés sur les lignes ne sont stockés que sur 32 bits, et sont recyclés. Il y a donc un risque de mélanger l'avenir et le futur des transactions lors du rebouclage (*wraparound*). Afin d'éviter ce phénomène, l'opération `VACUUM FREEZE` « gèle » les vieux enregistrements, afin que ceux-ci ne se retrouvent pas brusquement dans le futur.

Concrètement, il s'agit de positionner un *hint bit* dans les entêtes des lignes concernées, indiquant qu'elle est plus vieille que tous les numéros de transactions actuellement actifs. (Avant la 9.4, la colonne système `xmin` était simplement remplacée par un `FrozenXid`, ce qui revient au même).

5.7.5 Paramétrage du FREEZE (2)

Quand le `VACUUM` gèle-t-il les lignes ?

- « Âge » d'une table : `age ( pgclass.relfrozenxid )`
  - Les blocs nettoyés/gelés sont notés dans la *visibility map*
- `vacuum_freeze_min_age` (50 Mtrx)
  - âge des lignes *rencontrées* à geler
- `vacuum_freeze_table_age` (150 Mtrx)
  - agressif (toute la table)
- Au plus tard, par l'autovacuum sur toute la table :
  - `autovacuum_freeze_max_age` (200 Mtrx)
- Attention après un import massif/migration logique !
  - `VACUUM FREEZE` préventif en période de maintenance

Geler une ligne ancienne implique de réécrire le bloc et donc des écritures dans les journaux de transactions et les fichiers de données. Il est inutile de geler trop tôt une ligne récente, qui sera peut-être bientôt réécrite.

Paramétrage :

Plusieurs paramètres règlent ce fonctionnement. Leurs valeurs par défaut sont satisfaisantes pour la plupart des installations et ne sont pour ainsi dire jamais modifiées. Par contre, il est important de bien connaître le fonctionnement pour ne pas être surpris.

Le numéro de transaction le plus ancien connu au sein d'une table est porté par `pgclass.relfrozenxid`, et est sur 32 bits. Il faut utiliser la fonction `age()` pour connaître l'écart par rapport au numéro de transaction courant (géré sur 64 bits en interne).

```
SELECT relname, relfrozenxid, round(age(relfrozenxid) /1e6,2) AS "age_Mtrx"
FROM pg_class c
WHERE relname LIKE 'pgbench%' AND relkind='r'
ORDER BY age(relfrozenxid) ;
```

```
       relname        | relfrozenxid | age_Mtrx
----------------------+--------------+----------
 pgbench_accounts_7   |    882324041 |     0.00
 pgbench_accounts_8   |    882324041 |     0.00
 pgbench_accounts_2   |    882324041 |     0.00
 pgbench_history      |    882324040 |     0.00
 pgbench_accounts_5   |    848990708 |    33.33
```

```
pgbench_tellers     |    832324041 |    50.00
pgbench_accounts_3  |    719860155 |   162.46
pgbench_accounts_9  |    719860155 |   162.46
pgbench_accounts_4  |    719860155 |   162.46
pgbench_accounts_6  |    719860155 |   162.46
pgbench_accounts_1  |    719860155 |   162.46
pgbench_branches    |    719860155 |   162.46
pgbench_accounts_10 |    719860155 |   162.46
```

Une partie du gel se fait lors d'un `VACUUM` normal. Si ce dernier rencontre un enregistrement plus vieux que `vacuum_freeze_min_age` (par défaut 50 millions de transactions écoulées), alors le *tuple* peut et doit être gelé. Cela ne concerne que les lignes dans des blocs qui ont des lignes mortes à nettoyer : les lignes dans des blocs un peu statiques y échappent. (Y échappent aussi les lignes qui ne sont pas forcément visibles par toutes les transactions ouvertes.)

`VACUUM` doit donc périodiquement déclencher un nettoyage plus agressif de toute la table (et non pas uniquement des blocs modifiés depuis le dernier `VACUUM`), afin de nettoyer tous les vieux enregistrements. C'est le rôle de `vacuum_freeze_table_age` (par défaut 150 millions de transactions). Si la table a atteint cet âge, un `VACUUM` (manuel ou automatique) lancé dessus deviendra « agressif » :

```
VACUUM (VERBOSE) pgbench_tellers ;
INFO:  aggressively vacuuming "public.pgbench_tellers"
```

C'est équivalent à l'option `DISABLE_PAGE_SKIPPING` : les blocs ne contenant que des lignes vivantes seront tout de même parcourus. Les lignes non gelées qui s'y trouvent et plus vieilles que `vacuum_freeze_min_age` seront alors gelées. Ce peut être long, ou pas, en fonction de l'efficacité de l'étape précédente.

À côté des numéros de transaction habituels, les identifiants `multixact`, utilisés pour supporter le verrouillage de lignes par des transactions multiples évitent aussi le *wraparound* avec des paramètres spécifiques (`vacuum_multixact_freeze_min_age`, `vacuum_multixact_freeze_table_age`) qui ont les mêmes valeurs que leurs homologues.

Enfin, il faut traiter le cas de tables sur lesquelles un `VACUUM` complet ne s'est pas déclenché depuis très longtemps. L'autovacuum y veille : `autovacuum_freeze_max_age` (par défaut 200 millions de transactions) est l'âge maximum que doit avoir une table. S'il est dépassé, un `VACUUM` agressif est automatiquement lancé sur cette table. Il est visible dans `pg_stat_activity` avec la mention caractéristique *to prevent wraparound* :

```
autovacuum: VACUUM public.pgbench_accounts (to prevent wraparound)
```

Ce traitement est lancé même si `autovacuum` est désactivé (c'est-à-dire à `off`).

En fait, un `VACUUM FREEZE` lancé manuellement équivaut à un `VACUUM` avec les paramètres `vacuum_freeze_table_age` (âge minimal de la table) et `vacuum_freeze_min_age` (âge minimal des lignes pour les geler) à 0. Il va geler toutes les lignes qui le peuvent, même « jeunes ».

Charge induite par le gel :

Le gel des lignes peut être très lourd s'il y a beaucoup de lignes à geler, ou très rapide si l'essentiel du travail a été fait par les nettoyages précédents. Si la table a déjà été entièrement gelée (parfois depuis des centaines de millions de transactions), il peut juste s'agir d'une mise à jour du `relfrozenxid`.

Les blocs déjà entièrement gelés sont recensés dans la *visibility map* (qui recense aussi les blocs sans ligne morte). ils ne seront pas reparcourus s'ils ne sont plus modifiés. Cela accélère énormément le `FREEZE` sur les grosses tables (avant PostgreSQL 9.6, il y avait forcément au moins un parcours complet de la table !) Si le `VACUUM` est interrompu, ce qu'il a déjà gelé n'est pas perdu, il ne faut donc pas hésiter à l'interrompre au besoin.

L'âge de la table peut dépasser `autovacuum_freeze_max_age` si le nettoyage est laborieux, ce qui explique la marge par rapport à la limite fatidique des 2 milliards de transactions.

Quelques problèmes possibles sont évoqués plus bas.

Âge d'une base :

Nous avons vu que l'âge d'une base est en fait l'âge de la table la plus ancienne, qui se calcule à partir de la colonne `pg_database.datfrozenxid` :

```
SELECT datname, datfrozenxid, age (datfrozenxid)
FROM pg_database ORDER BY 3 DESC ;
```

```
  datname  | datfrozenxid |    age
-----------+--------------+-----------
 pgbench   |   1809610092 | 149835222
 template0 |   1957896953 |   1548361
 template1 |   1959012415 |    432899
 postgres  |   1959445305 |         9
```

Concrètement, on verra l'âge d'une base de données approcher peu à peu des 200 millions de transactions, ce qui correspondra à l'âge des plus « vieilles » tables, souvent celles sur lesquelles l'autovacuum ne passe jamais. L'âge des tables évolue même si l'essentiel de leur contenu, voire la totalité, est déjà gelé (car il peut rester le `pg_class.relfrozenxid` à mettre à jour, ce qui sera bien sûr très rapide). Cet âge va retomber quand un gel sera forcé sur ces tables, puis remonter, etc.

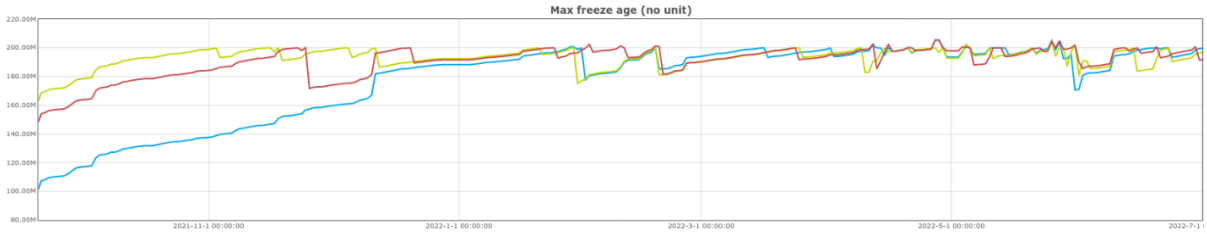


Figure 5/ .4: Évolution de l'âge des bases sur une instance

Rappelons que le `FREEZE` réécrit tous les blocs concernés. Il peut être quasi-instantané, mais le déclenchement inopiné d'un `VACUUM FREEZE` sur l'intégralité d'une très grosse table assez statique est une mauvaise surprise assez fréquente. La table est réécrite, les entrées-sorties sont chargées, la sauvegarde PITR enfle, évidemment à un moment où la base est chargée.
Une base chargée en bloc et peu modifiée peut même voir le `FREEZE` se déclencher sur toutes les tables en même temps. Le délai avant le déclenchement du gel par l'autovacuum dépend de la consommation des numéros de transaction sur l'instance migrée, et varie de quelques semaines à des années. Sont concernés tous les imports massifs et les migrations et restauration logiques (`pg_restore`, réplication logique, migration de bases depuis d'autres bases de données), mais pas les migrations et restaurations physiques.
Des ordres `VACUUM FREEZE` sur les plus grosses tables à des moments calmes permettent d'étaler ces écritures. Si ces ordres sont interrompus, l'essentiel de ce qu'ils auront pu geler n'est plus à re-geler plus tard.

Résumé :

Que retenir de ce paramétrage complexe ?

- le `VACUUM` gèlera une partie des lignes un peu anciennes lors de son fonctionnement habituel ;
- un bloc gelé non modifié ne sera plus à regeler ;
- de grosses tables statiques peuvent engendrer soudainement une grosse charge en écriture ; il vaut mieux être proactif.

5.8 AUTRES PROBLÈMES COURANTS

5.8.1 L'autovacuum dure trop longtemps

- Fréquence de passage ?
- Débit ?
- Nombre de workers ?
- Taille vraiment trop grosse ?

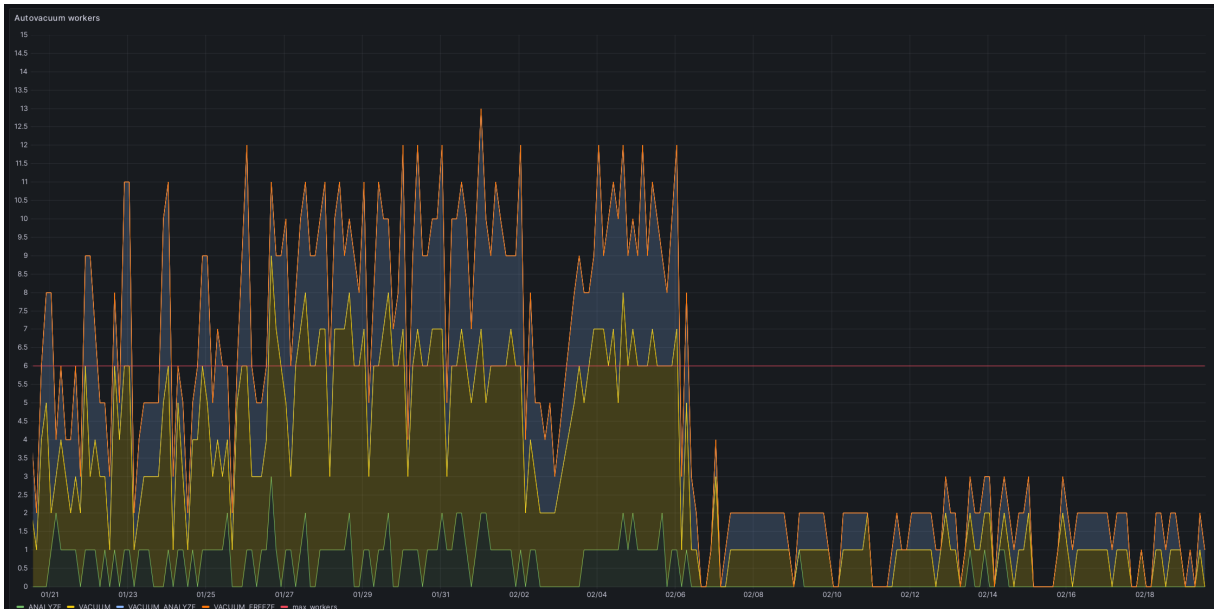


Figure 5/ .5: Nombre de workers avant et après la réduction d'autovacuum_vacuum_cost_delay

Nous avons vu que le paramétrage de l'autovacuum vise à limiter la charge sur la base. Le nettoyage d'une grosse table peut donc être parfois très long. Ce n'est pas forcément un problème majeur si l'opération arrive à terme dans un délai raisonnable, mais il vaut mieux savoir pourquoi. Il peut y avoir plusieurs causes, qui ne s'excluent pas mutuellement.

Il est fréquent que les grosses tables soient visitées trop peu souvent. Rappelons que la propriété `autovacuum_vacuum_scale_factor` de chaque table est par défaut à 20 % : lorsque l'autovacuum se déclenche, il doit donc traiter une volumétrie importante. Il est conseillé de réduire la valeur de ce paramètre (ou de jouer sur `autovacuum_vacuum_threshold`) pour un nettoyage plus fréquent.

Le débit en écriture peut être insuffisant (c'est fréquent sur les anciennes versions), auquel cas, avec des disques corrects, on peut baisser `autovacuum_vacuum_cost_delay` ou monter

`autovacuum_vacuum_cost_limit`. Sur le graphique ci-dessus, issu d'un cas réel, les trois workers semblaient en permanence occupés. Il risquait donc d'y avoir un retard pour nettoyer certaines tables, ou calculer des statistiques. La réduction de `autovacuum_vacuum_cost_delay` de 20 à 2 ms a mené à une réduction drastique de la durée de traitement de chaque worker.

Rappelons qu'un `VACUUM` manuel (ou planifié) n'est soumis à aucun bridage.

Le nombre de workers peut être trop bas, notamment s'il y a de nombreuses tables. Auquel cas ils semblent tous activés en permanence (comme ci-dessus). Monter `autovacuum_max_workers` au-delà de 3 nécessite d'augmenter le débit autorisé avec les paramètres ci-dessus.

Pour des grandes tables, le partitionnement permet de paralléliser l'activité de l'autovacuum. Les *workers* peuvent en effet travailler dans la même base, sur des tables différentes.

Un grand nombre de bases actives peut devenir un frein et augmenter l'intervalle entre deux nettoyages d'une base, bien que l'`autovacuum launcher` ignore les bases inutilisées.

Exceptionnellement, l'autovacuum peut tomber en erreur (bloc corrompu, index fonctionnel avec une fonction boguée…) et ne jamais finir (surveiller les traces).

5.8.2 Arrêter un VACUUM ?

- Lancement manuel ou script
  - risque avec certains verrous
- Autovacuum
  - interrompre s'il gêne
- Exception : *to prevent wraparound* lent **et** bloquant
  - `pg_cancel_backend` + `VACUUM FREEZE` manuel

Le cas des `VACUUM` manuels a été vu plus haut : ils peuvent gêner quelques verrous ou opérations DDL. Il faudra les arrêter manuellement au besoin.

C'est différent si l'autovacuum a lancé le processus : celui-ci sera arrêté si un utilisateur pose un verrou en conflit.

La seule exception concerne un `VACUUM FREEZE` lancé quand la table doit être gelée, donc avec la mention *to prevent wraparound* dans `pg_stat_activity` : celui-ci ne sera pas interrompu. Il ne pose qu'un verrou destinée à éviter les modifications de schéma simultanées (SHARE UPDATE EXCLUSIVE). Comme le débit en lecture et écriture est bridé par le paramétrage habituel de l'autovacuum, ce verrou peut durer assez longtemps (surtout avant PostgreSQL 9.6, où toute la table est relue à

chaque `FREEZE`). Cela peut s'avérer gênant avec certaines applications. Une solution est de réduire `autovacuum_vacuum_cost_delay`, surtout avant PostgreSQL 12 (voir plus haut).

Si les opérations sont impactées, on peut vouloir lancer soi-même un `VACUUM FREEZE` manuel, non bridé. Il faudra alors repérer le PID du `VACUUM FREEZE` en cours, l'arrêter avec `pg_cancel_backend`, puis lancer manuellement l'ordre `VACUUM FREEZE` sur la table concernée, (et rapidement avant que l'autovacuum ne relance un processus).

La supervision peut se faire avec `pg_stat_progress_vacuum` et `iotop`.

5.8.3 Ce qui peut bloquer le VACUUM FREEZE

- Causes :
  - sessions *idle in transaction* sur une longue durée
  - slot de réplication en retard/oublié
  - transactions préparées oubliées
  - erreur à l'exécution du `VACUUM`
- Conséquences :
  - processus autovacuum répétés
  - arrêt des transactions
  - mode single…
- Supervision :
  - `check_pg_activity` : `xmin`, `max_freeze_age`
  - surveillez les traces !

Il arrive que le fonctionnement du `FREEZE` soit gêné par un problème qui lui interdit de recycler les plus anciens numéros de transactions. (Ces causes gênent aussi un `VACUUM` simple, mais les symptômes sont alors surtout un gonflement des tables concernées).

Les causes possibles sont :

- des sessions *idle in transactions* durent depuis des jours ou des semaines (voir le statut `idle in transaction` dans `pg_stat_activity`, et au besoin fermer la session) : au pire, elles disparaissent après redémarrage ;
- des slots de réplication pointent vers un secondaire très en retard, voire disparu (consulter `pg_replication_slots`, et supprimer le slot) ;

- des transactions préparées (pas des requêtes préparées !) n'ont jamais été validées ni annulées, (voir `pg_prepared_xacts`, et annuler la transaction) : elles ne disparaissent pas après redémarrage ;
- l'opération de `VACUUM` tombe en erreur : corruption de table ou index, fonction d'index fonctionnel buggée, etc. (voir les traces et corriger le problème, supprimer l'objet ou la fonction, etc.).

Pour effectuer le `FREEZE` en urgence le plus rapidement possible, on peut utiliser :

```
VACUUM (FREEZE, VERBOSE, INDEX_CLEANUP off, TRUNCATE off) ;
```

Cette commande force le gel de toutes les lignes, ignore le nettoyage des index et ne supprime pas les blocs vides finaux (le verrou peut être gênant). Un `VACUUM` classique serait à prévoir ensuite à l'occasion.

En toute rigueur, une version sans l'option `FREEZE` est encore plus rapide : le mode agressif serait déclenché mais les lignes plus récentes que `vacuum_freeze_min_age` (50 millions de transaction) ne seraient pas encore gelées. On peut même monter ce paramètre dans la session pour alléger au maximum la charge sur une table dont les lignes ont des âges bien étalés.

Ne pas oublier de nettoyer toutes les bases de l'instance.

Dans le pire des cas, plus aucune transaction ne devient possible (y compris les opérations d'administration comme `DROP`, ou `VACUUM` sans `TRUNCATE off`) :

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database
↪  "db1"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
You might also need to commit or roll back old prepared transactions,
or drop stale replication slots.
```

En dernière extrémité, il reste un délai de grâce d'un million de transactions, qui ne sont accessibles que dans le très austère mode monoutilisateur[5] de PostgreSQL.

Avec la sonde Nagios check_pgactivity[6], et les services **max_freeze_age** et **oldest_xmin**, il est possible de vérifier que l'âge des bases ne dérive pas, ou de trouver quel processus porte le `xmin` le plus ancien. S'il y a un problème, il entraîne généralement l'apparition de nombreux messages dans les traces : lisez-les régulièrement !

[5] https://docs.postgresql.fr/current/app-postgres.html#APP-POSTGRES-SINGLE-USER
[6] https://github.com/OPMDG/check_pgactivity

5.9 RÉSUMÉ DES CONSEILS SUR L'AUTOVACUUM

> "Vacuuming is like exercising.
> If it hurts, you're not doing it enough!"
>
> (Robert Haas, PGConf.EU 2023, Prague, 13 décembre 2023)

Certains sont frileux à l'idée de passer un `VACUUM`. En général, ce n'est que reculer pour mieux sauter.

5.9.1 Résumé des conseils sur l'autovacuum (1/2)

- Laisser l'autovacuum faire son travail
- Augmenter le débit autorisé
- Surveiller `last_(auto)analyze` / `last_(auto)vacuum`
- Nombre de *workers*
- Grosses tables, par ex :

```
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.01) ;
ALTER TABLE table_name SET (autovacuum_vacuum_threshold = 1000000) ;
```

- Mais ne pas hésiter à planifier un `vacuumdb` quotidien

L'autovacuum fonctionne convenablement pour les charges habituelles. Il ne faut pas s'étonner qu'il fonctionne longtemps en arrière-plan : il est justement conçu pour ne pas se presser. Au besoin, ne pas hésiter à lancer manuellement l'opération, donc sans bridage en débit.

Si les disques sont bons, on peut augmenter le débit autorisé :

- en réduisant la durée de pause (`autovacuum_vacuum_cost_delay`), surtout avant la version 12 ;
- de préférence, en augmentant le coût à atteindre avant une pause (`autovacuum_vacuum_cost_limit`) ;
- en augmentant `vacuum_buffer_usage_limit` (en version 16).

Comme le déclenchement d'`autovacuum` est très lié à l'activité, il faut vérifier qu'il passe assez souvent sur les tables sensibles en surveillant `pg_stat_all_tables.last_autovacuum` et `last_autoanalyze`.

Si les statistiques peinent à se rafraîchir, ne pas hésiter à activer plus souvent l'autovacuum sur les grosses tables problématiques ainsi :

```
-- analyze après 5 % de modification au lieu du défaut de 10 %
ALTER TABLE table_name SET (autovacuum_analyze_scale_factor = 0.05) ;
```

De même, si la fragmentation s'envole, descendre `autovacuum_vacuum_scale_factor`. (On peut préférer utiliser les variantes en `*_threshold` de ces paramètres, et mettre les `*_scale_factor` à 0).

Dans un modèle avec de très nombreuses tables actives, le nombre de *workers* doit parfois être augmenté.

5.9.2 Résumé des conseils sur l'autovacuum (2/2)

- Mode manuel
  - batchs / tables temporaires / tables à insertions seules (<v13)
  - si pressé !
- Danger du `FREEZE` brutal après migration logique ou gros import
  - prévenir
- `VACUUM FULL` : dernière extrémité

L'autovacuum n'est pas toujours assez rapide à se déclencher, par exemple entre les différentes étapes d'un batch : on intercalera des `VACUUM ANALYZE` manuels. Il faudra le faire systématiquement pour les tables temporaires (que l'autovacuum ne voit pas). De plus, avant PostgreSQL 13, pour les tables où il n'y a que des insertions, l'autovacuum ne lance spontanément que l'`ANALYZE` : il faudra effectuer un `VACUUM` explicite pour profiter de certaines optimisations.

Au final, un `vacuumdb` planifié quotidien à un moment calme est généralement une bonne idée : ce qu'il traite ne sera plus à nettoyer en journée, et il apporte la garantie qu'aucune table ne sera négligée. Cela ne dispense pas de contrôler l'activité de l'`autovacuum` sur les grosses tables très sollicitées.

Un point d'attention reste le gel brutal de grosses quantités de données chargées ou modifiées en même temps. Un `VACUUM FREEZE` préventif dans une période calme reste la meilleure solution.

Un `VACUUM FULL` sur une grande table est une opération très lourde, à réserver à la récupération d'une partie significative de son espace, qui ne serait pas réutilisé plus tard.

5.10 CONCLUSION

- `VACUUM` fait de plus en plus de choses au fil des versions
- Convient généralement
- Paramétrage apparemment complexe
  - en fait relativement simple avec un peu d'habitude

5.10.1 Questions

N'hésitez pas, c'est le moment !

5.11 QUIZ

https://dali.bo/m5_quiz

5.12 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/m5_solutions.

5.12.1 Traiter la fragmentation

But : Traiter la fragmentation

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

Désactiver l'autovacuum pour la table `t3`.

Insérer un million de lignes dans la table `t3` avec la fonction `generate_series`.

Récupérer la taille de la table `t3`.

Supprimer les 500 000 premières lignes de la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM VERBOSE` sur la table `t3`. Quelle est l'information la plus importante ?

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Exécuter un `VACUUM FULL VERBOSE` sur la table `t3`.

Récupérer la taille de la table `t3`. Que faut-il en déduire ?

Créer une table `t4` avec une colonne `id` de type `integer`.

Désactiver l'autovacuum pour la table `t4`.

Insérer un million de lignes dans la table `t4` avec `generate_series`.

Récupérer la taille de la table `t4`.

Supprimer les 500 000 **dernières** lignes de la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

Exécuter un `VACUUM` sur la table `t4`.

Récupérer la taille de la table `t4`. Que faut-il en déduire ?

5.12.2 Détecter la fragmentation

But : Détecter la fragmentation

Créer une table `t5` avec deux colonnes : `c1` de type `integer` et `c2` de type `text`.

Désactiver l'autovacuum pour la table `t5`.

Insérer un million de lignes dans la table `t5` avec `generate_series`.

- Installer l'extension `pg_freespacemap` (documentation : https://docs.postgresql.fr/current/pgfreespacemap.html)
- Que rapporte la fonction `pg_freespace()` quant à l'espace libre de la table `t5` ?

- Modifier exactement 200 000 lignes de la table `t5`.

- Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

Exécuter un `VACUUM` sur la table `t5`.

Que rapporte `pg_freespace` quant à l'espace libre de la table `t5` ?

Récupérer la taille de la table `t5`.

Exécuter un `VACUUM (FULL, VERBOSE)` sur la table `t5`.

Récupérer la taille de la table `t5` et l'espace libre rapporté par `pg_freespacemap`. Que faut-il en déduire ?

5.12.3 Gestion de l'autovacuum

But : Voir fonctionner l'autovacuum

Créer une table `t6` avec une colonne `id` de type `integer`.

Insérer un million de lignes dans la table `t6` :

```
INSERT INTO t6(id) SELECT generate_series (1, 1000000) ;
```

Que contient la vue `pg_stat_user_tables` pour la table `t6` ? Il faudra peut-être attendre une minute. (Si la version de PostgreSQL est antérieure à la 13, il faudra lancer un `VACUUM t6`.)

Vérifier le nombre de lignes dans `pg_class.reltuples`.

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.

- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

- Modifier 60 000 lignes supplémentaires de la table `t6` avec :

```
UPDATE t6 SET id=1 WHERE id > 940000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

Descendre le facteur d'échelle de la table `t6` à 10 % pour le `VACUUM`.

- Modifier encore 200 000 autres lignes de la table `t6` :

```
UPDATE t6 SET id=1 WHERE id > 740000 ;
```

- Attendre une minute.
- Que contient la vue `pg_stat_user_tables` pour la table `t6` ?
- Que faut-il en déduire ?

6/ Partitionnement déclaratif (introduction)

- Le partitionnement déclaratif apparaît avec PostgreSQL 10
- Préférer un PostgreSQL récent
- Ne plus utiliser l'ancien partitionnement par héritage.

Ce module introduit le partitionnement déclaratif introduit avec PostgreSQL 10, et amélioré dans les versions suivantes. PostgreSQL 13 au minimum est conseillé pour ne pas être gêné par les limitations des versions précédentes (qui ne sont d'ailleurs plus supportées). Les améliorations à chaque nouvelle version de PostgreSQL doivent mener à favoriser une version récente de PostgreSQL.

Le partitionnement par héritage, au fonctionnement totalement différent, reste utilisable, mais ne doit plus servir aux nouveaux développements, du moins pour les cas décrits ici.

6.1 PRINCIPE & INTÉRÊTS DU PARTITIONNEMENT

- Faciliter la maintenance de gros volumes
  - `VACUUM` (`FULL`), réindexation, déplacements, sauvegarde logique…
- Performances
  - parcours complet sur de plus petites tables
  - statistiques par partition plus précises
  - purge par partitions entières
  - `pg_dump` parallélisable
  - tablespaces différents (données froides/chaudes)
- Attention à la maintenance sur le code

Maintenir de très grosses tables peut devenir fastidieux, voire impossible : `VACUUM FULL` trop long, espace disque insuffisant, autovacuum pas assez réactif, réindexation interminable… Il est aussi aberrant de conserver beaucoup de données d'archives dans des tables lourdement sollicitées pour les données récentes.

Le partitionnement consiste à séparer une même table en plusieurs sous-tables (partitions) manipulables en tant que tables à part entière.

Maintenance

La maintenance s'effectue sur les partitions plutôt que sur l'ensemble complet des données. En particulier, un `VACUUM FULL` ou une réindexation peuvent s'effectuer partition par partition, ce qui permet de limiter les interruptions en production, et lisser la charge. `pg_dump` ne sait pas paralléliser la sauvegarde d'une table volumineuse et non partitionnée, mais parallélise celle de différentes partitions d'une même table.

C'est aussi un moyen de déplacer une partie des données dans un autre *tablespace* pour des raisons de place, ou pour déporter les parties les moins utilisées de la table vers des disques plus lents et moins chers.

Parcours complet de partitions

Certaines requêtes (notamment décisionnelles) ramènent tant de lignes, ou ont des critères si complexes, qu'un parcours complet de la table est souvent privilégié par l'optimiseur.

Un partitionnement, souvent par date, permet de ne parcourir qu'une ou quelques partitions au lieu de l'ensemble des données. C'est le rôle de l'optimiseur de choisir la partition (*partition pruning*), par exemple celle de l'année en cours, ou des mois sélectionnés.

Suppression des partitions

La suppression de données parmi un gros volume peut poser des problèmes d'accès concurrents ou de performance, par exemple dans le cas de purges. En configurant judicieusement les partitions, on peut résoudre cette problématique en supprimant une partition :

```
DROP TABLE nompartition ;
```

ou en la *détachant* :

```
ALTER TABLE table_partitionnee DETACH PARTITION nompartition ;
```

pour l'archiver (et la réattacher au besoin) ou la supprimer ultérieurement.

D'autres optimisations sont décrites dans ce billet de blog d'Adrien Nayrat[1] : statistiques plus précises au niveau d'une partition, réduction plus simple de la fragmentation des index, jointure par rapprochement des partitions…

La principale difficulté d'un système de partitionnement consiste à partitionner avec un impact minimal sur la maintenance du code par rapport à une table classique.

[1] https://blog.anayrat.info/2021/09/01/cas-dusages-du-partitionnement-natif-dans-postgresql/

6.2 PARTITIONNEMENT 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.

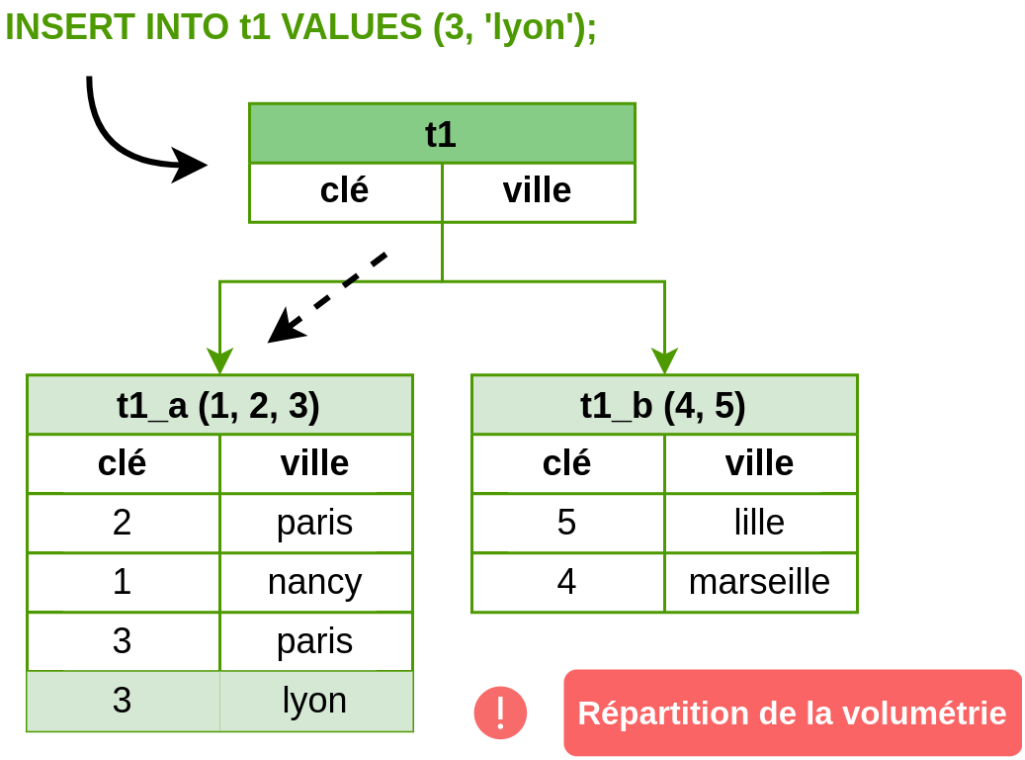
Les clés étrangères entre tables partitionnées ne sont pas un problème dans les versions récentes de PostgreSQL.

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



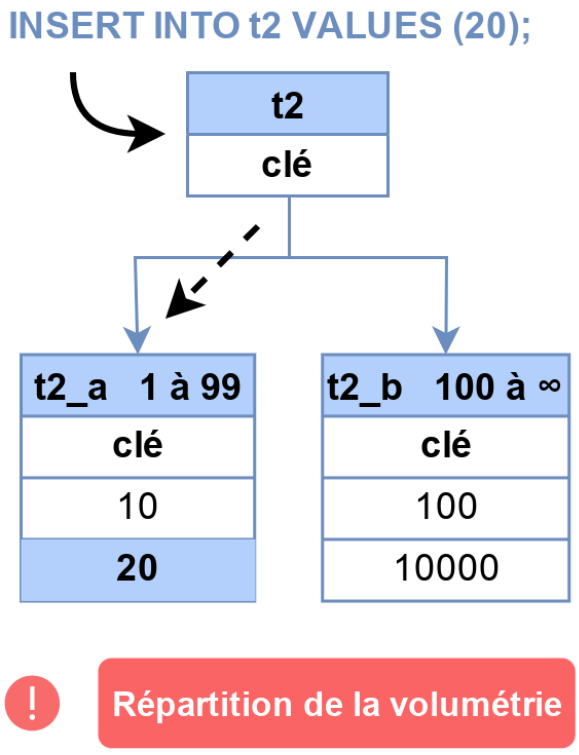
6.2.2 Partitionnement par liste : implémentation

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1) ;

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3) ;
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5) ;
…
```

Le partitionnement par liste définit les valeurs d'une colonne acceptables dans chaque partition.

Utilisations courantes : partitionnement par année, par statut, par code géographique…

6.2.3 Partitionnement par intervalle



6.2.4 Partitionnement par intervalle : implémentation

```
CREATE TABLE logs ( d timestamptz, contenu text) PARTITION BY RANGE (d) ;

CREATE TABLE logs_201901 PARTITION OF logs
                         FOR VALUES FROM ('2019-01-01') TO ('2019-02-01');
CREATE TABLE logs_201902 PARTITION OF logs
                         FOR VALUES FROM ('2019-02-01') TO ('2019-03-01');
…
CREATE TABLE logs_201912 PARTITION OF logs
                         FOR VALUES FROM ('2019-12-01') TO ('2020-01-01');
…
CREATE TABLE logs_autres PARTITION OF logs
                         DEFAULT ;              -- pour ne rien perdre
```

Utilisations courantes : partitionnement par date, par plages de valeurs continues, alphabétiques…

L'exemple ci-dessus utilise le partitionnement par mois. Chaque partition est définie par des plages de date. Noter que la borne supérieure ne fait *pas* partie des données de la partition. Elle doit donc être aussi la borne inférieure de la partie suivante. La description de la table partitionnée devient :

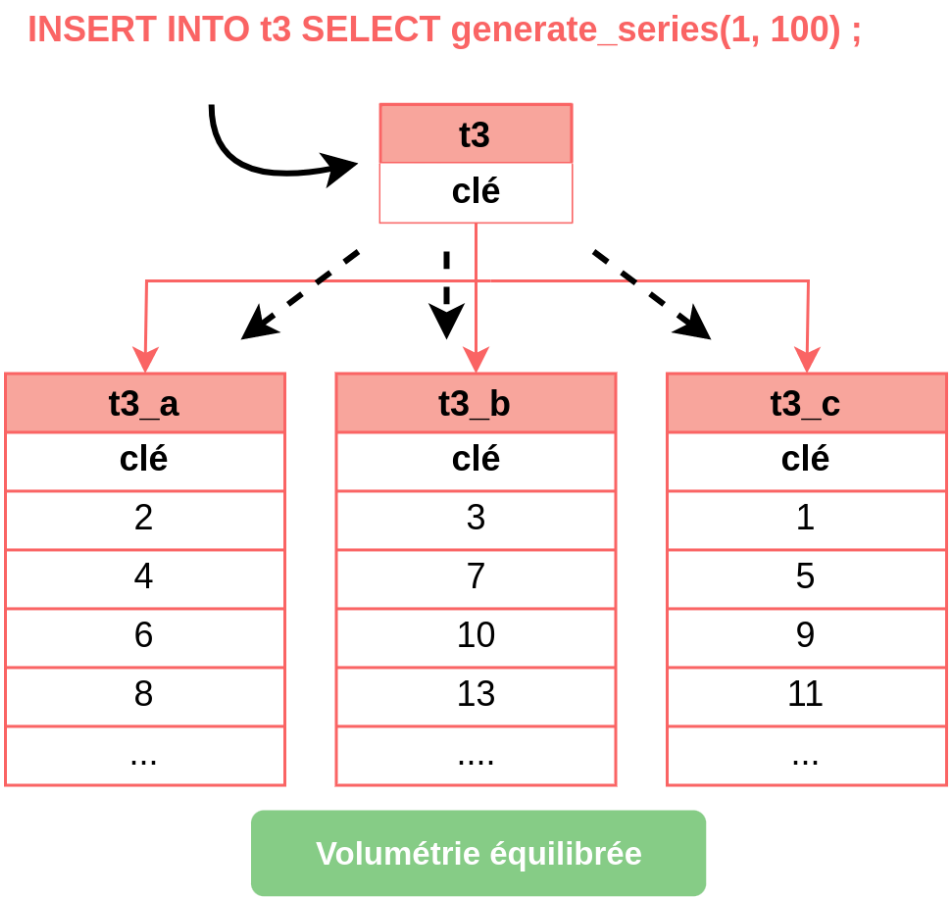
```
=# \d+ logs
                                Table partitionnée « public.logs »
 Colonne |           Type           | ... | Stockage | …
---------+--------------------------+-----+----------+ …
 d       | timestamp with time zone | ... | plain    | …
 contenu | text                     | ... | extended | …
Clé de partition : RANGE (d)
Partitions: logs_201901 FOR VALUES FROM ('2019-01-01 00:00:00+01') TO ('2019-02-01
 ↪  00:00:00+01'),
            logs_201902 FOR VALUES FROM ('2019-02-01 00:00:00+01') TO ('2019-03-01
 ↪  00:00:00+01'),
            …
            logs_201912 FOR VALUES FROM ('2019-12-01 00:00:00+01') TO ('2020-01-01
 ↪  00:00:00+01'),
            logs_autres DEFAULT
```

La partition par défaut reçoit toutes les données qui ne vont dans aucune autre partition : cela évite des erreurs d'insertion. Il vaut mieux que la partition par défaut reste très petite.

Il est possible de définir des plages sur plusieurs champs :

```
CREATE TABLE tt_a PARTITION OF tt
FOR VALUES FROM (1,'2020-08-10') TO (100, '2020-08-11') ;
```

6.2.5 Partitionnement par hachage



6.2.6 Partitionnement par hachage : implémentation

- Hachage des valeurs
- Répartition homogène
- Indiquer un modulo et un reste

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);

CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 0);
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 1);
CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 2);
```

Ce type de partitionnement vise à répartir la volumétrie dans plusieurs partitions de manière homogène, quand il n'y a pas de clé évidente. En général, il y aura plus que 3 partitions.

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
                                  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 ! Attention si > 100 partitions
- Création non automatique
- Pas d'héritage multiple, schéma fixe
- Limitations avant PostgreSQL 13/14

Certaines limitations du partitionnement sont liées à son principe. Les partitions ont forcément le même schéma de données que leur partition mère. Il n'y a pas de notion d'héritage multiple.

La création des partitions n'est pas automatique (par exemple dans un partitionnement par date). Il faudra prévoir de les créer par avance.

Une limitation sérieuse du partitionnement tient au temps de planification qui augmente très vite avec le nombre de partitions, même petites. En général, on considère qu'il faut éviter de dépasser 100 partitions.

Pour contourner cette limite, il reste possible de manipuler directement les partitions s'il est facile de trouver leur nom.

Avant PostgreSQL 13, de nombreuses limitations rendent l'utilisation moins pratique ou moins performante. Si le partitionnement vous intéresse, il est conseillé d'utiliser une version la plus récente possible.

6.4 CONCLUSION

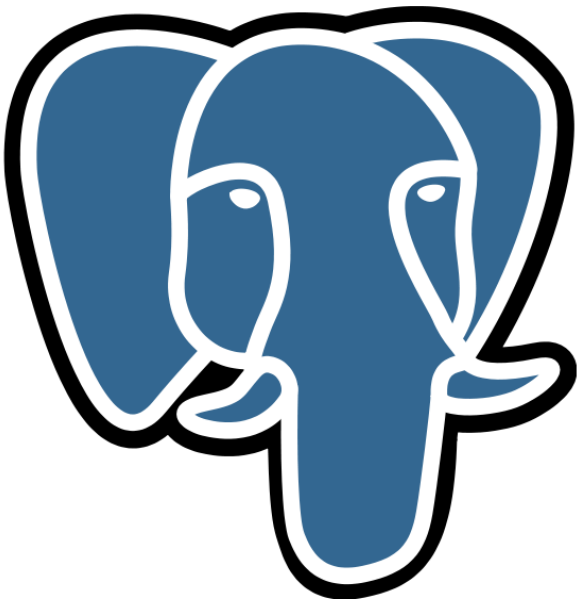
- Préférer une version récente de PostgreSQL
- Pour plus de détails sur le partitionnement
  - https://dali.bo/v1_html

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*).

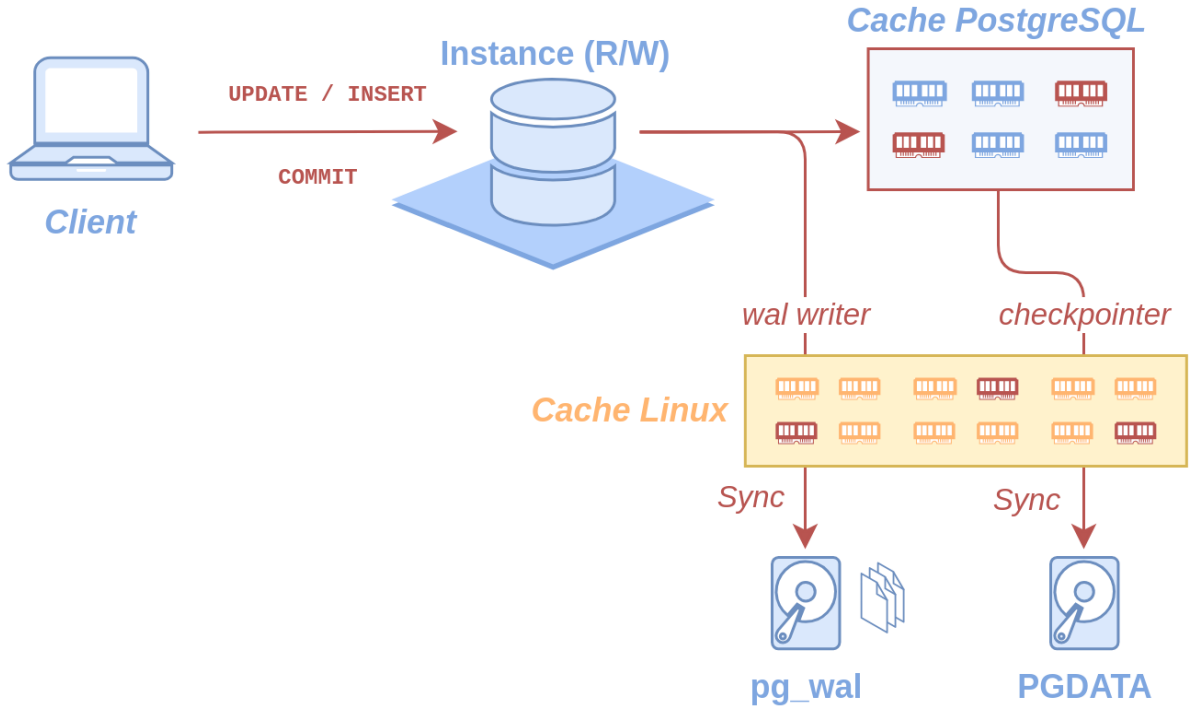
7.1.1 Au menu

- Rappel sur la journalisation
- Principe de la sauvegarde PITR
- Mise en place
  - sauvegarde : manuelle, ou avec `pg_basebackup`
  - archivage : manuel, ou avec `pg_receivewal`
- Restaurer une sauvegarde PITR
- Des outils

Ce module fait le tour de la sauvegarde PITR, de la mise en place de l'archivage (manuelle ou avec

l'outil `pg_receivewal`) à la sauvegarde des fichiers (en manuel, ou avec l'outil `pg_basebackup`). Il discute aussi de la restauration d'une telle sauvegarde. Nous évoquerons très rapidement quelques outils externes pour faciliter ces sauvegardes.

7.2 RAPPEL SUR LA JOURNALISATION



7.2.1 Journaux de transaction

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

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

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

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

Ainsi en cas de crash :

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

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

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

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

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

Pour plus d'informations :

- *PostgreSQL et ses journaux de transactions*[1], Guillaume Lelarge, GNU/Linux Magazine n°108 (septembre 2008), encore d'actualité pour l'essentiel.

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

7.3 PITR

- *Point In Time Recovery*
- À chaud
- En continu
- Cohérente

PITR est l'acronyme de *Point In Time Recovery*, autrement dit restauration à un point dans le temps.

C'est une sauvegarde à chaud et surtout en continu. Là où une sauvegarde logique du type `pg_dump` se fait au mieux une fois toutes les 24 h, la sauvegarde PITR se fait en continu grâce à l'archivage des journaux de transactions. De ce fait, ce type de sauvegarde diminue très fortement la fenêtre de perte de données.

Bien qu'elle se fasse à chaud, la sauvegarde est cohérente.

7.3.1 Principes du PITR

- Les journaux de transactions contiennent toutes les modifications
- Il faut les archiver
  - …et avoir une image des fichiers à un instant t (*base backup*)
- La restauration se fait en restaurant cette image
  - puis rejouant tous les journaux
  - dans l'ordre
  - entièrement
  - tous, jusqu'au moment voulu

Quand une transaction est validée, les données à écrire dans les fichiers de données sont d'abord écrites dans un journal de transactions. Ces journaux décrivent donc toutes les modifications survenant sur les fichiers de données, que ce soit les objets utilisateurs comme les objets systèmes. Pour reconstruire un système, il suffit donc d'avoir ces journaux et d'avoir un état des fichiers du répertoire des données à un instant t (*base backup*). Toutes les actions effectuées après cet instant t pourront être rejouées en demandant à PostgreSQL d'appliquer les actions contenues dans les journaux. Les opérations stockées dans les journaux correspondent à des modifications physiques de fichiers, il

faut donc partir d'une sauvegarde au niveau du système de fichier, un export avec `pg_dump` n'est pas utilisable.

Il est donc nécessaire de conserver ces journaux de transactions. Or PostgreSQL les recycle dès qu'il n'en a plus besoin. La solution est de demander au moteur de les archiver ailleurs avant ce recyclage. On doit aussi disposer de l'ensemble des fichiers qui composent le répertoire des données (incluant les tablespaces si ces derniers sont utilisés).

La restauration a besoin des journaux de transactions archivés. Il ne sera pas possible de restaurer et éventuellement revenir à un point donné avec la sauvegarde seule. En revanche, une fois la sauvegarde des fichiers restaurée et la configuration réalisée pour rejouer les journaux archivés, il sera possible de les rejouer, tous ou seulement une partie d'entre eux (en s'arrêtant à un certain moment).

Comme les journaux redéroulent toute l'activité depuis le début de la sauvegarde PITR, ils doivent impérativement être rejoués dans l'ordre de leur écriture (et donc de leur nom), et leur contenu entier est appliqué.

Le rejeu s'arrête quand les journaux à rejouer sont épuisés, ou si le DBA a demandé à s'arrêter à un moment précis.

Il est critique de ne perdre aucun journal. S'il en manque un, ou s'il est inutilisable, la restauration n'ira pas plus loin, les journaux suivants ne seront pas rejoués. La base sera cohérente et utilisable uniquement si l'on a pu réappliquer au moins les journaux générés *pendant* la sauvegarde (point de cohérence).

7.3.2 Avantages du PITR

- Sauvegarde à chaud
- Rejeu d'un grand nombre de journaux
- Moins de perte de données

Tout le travail est réalisé à chaud, que ce soit l'archivage des journaux ou la sauvegarde des fichiers de la base. En effet, il importe peu que les fichiers de données soient modifiés pendant la sauvegarde car les journaux de transactions archivés permettront de corriger toute incohérence par leur application.

Il est possible de rejouer un très grand nombre de journaux (une journée, une semaine, un mois, etc.). Évidemment, plus il y a de journaux à appliquer, plus cela prendra du temps. Mais il n'y a pas de limite au nombre de journaux à rejouer.

Dernier avantage, c'est le système de sauvegarde qui occasionnera le moins de perte de données (RPO). Avec l'archivage continu des journaux de transactions, la fenêtre de perte de données va être

fortement réduite. Plus l'activité est intense, plus la fenêtre de temps sera petite, car les fichiers des journaux sont de taille fixe, et ils ne sont archivés que complets. (À l'inverse, une sauvegarde logique avec `pg_dump` entraînera une perte de données bien plus importante. Si elle est lancée à 3 h et restaurée après un souci à 12 h, on perd 9 heures de données.)

Pour les systèmes n'ayant pas une grosse activité, il est aussi possible de forcer un changement de journal à intervalle régulier, ce qui a pour effet de forcer son archivage, et donc dans les faits de pouvoir s'assurer une perte correspondant au maximum à cet intervalle.

7.3.3 Inconvénients du PITR

- Sauvegarde/restauration de l'instance complète
- Impossible de changer d'architecture (même OS conseillé)
- Nécessite un grand espace de stockage (données + journaux)
- Interdiction de perdre un journal
- Risque d'accumulation des journaux
  - dans `pg_wal/` si échec d'archivage… et arrêt si plein !
  - dans le dépôt d'archivage si échec des sauvegardes
- Plus complexe

Le premier inconvénient vient directement du fait qu'on copie les fichiers : la sauvegarde et la restauration concernent l'instance complète. Il est impossible de ne restaurer qu'une seule base ou que quelques tables.

La restauration se fait impérativement sur la même architecture (x86/ARM, 32/64 bits, *little/big endian*). Il est même fortement conseillé de restaurer dans la même version du même système d'exploitation, sous peine de devoir réindexer l'instance ensuite (à cause d'une différence de définition des locales entre deux versions majeures d'une distribution).

Une sauvegarde PITR nécessite en plus un plus grand espace de stockage. Non seulement il faut sauvegarder les fichiers, y compris les index et la fragmentation, mais aussi tous les journaux de transactions sur une certaine période, ce qui peut être volumineux (en tout cas beaucoup plus que des `pg_dump`). La volumétrie des journaux dépend fortement de l'activité.

En cas de problème dans l'archivage et selon la méthode choisie, l'instance ne voudra pas effacer les journaux non archivés. Il y a donc un risque d'accumulation de ceux-ci. Il faudra donc surveiller la taille du `pg_wal`. En cas de saturation, PostgreSQL s'arrête !

Si un journal est perdu ou corrompu, la restauration ne pourra jamais aller au-delà de ce journal. Le risque augmente avec le nombre de journaux conservés.

Pour réduire le risque de perte d'un journal, et aussi pour accélérer le temps de rejeu, il est conseillé de procéder à des *base backups* (complet, différentiels… selon l'outil) assez fréquemment.

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

7.4 COPIE PHYSIQUE À CHAUD PONCTUELLE AVEC PG_BASEBACKUP

(Non PITR)

7.4.1 pg_basebackup

- Réalise les différentes étapes d'une sauvegarde
  - via 1 ou 2 connexions de réplication + slots de réplication
  - base backup + journaux nécessaires
- Copie intégrale
  - image de la base à la **fin** du backup
  - peut servir de base pour du PITR plus tard
- Pas de copie incrémentale avant v17
- Configuration : *streaming* (rôle, droits, slots)

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

Description :

`pg_basebackup` est un produit qui a beaucoup évolué dans les dernières versions de PostgreSQL. De plus, le paramétrage par défaut le rend immédiatement utilisable.

Il permet de réaliser toute la sauvegarde de l'instance, à distance, via deux connexions de réplication : une pour les données, une pour les journaux de transactions qui sont générés pendant la copie. Sa compression permet d'éviter une durée de transfert ou une place disque occupée trop importante. Cela a évidemment un coût, notamment au niveau CPU, sur le serveur ou sur le client suivant le besoin. Il est simple à mettre en place et à utiliser, et permet d'éviter de nombreuses étapes que nous verrons par la suite.

Par contre, il ne permet pas de réaliser une sauvegarde incrémentale avant PostgreSQL 17. À partir de cette version, des sauvegardes incrémentales faite avec `pg_basebackup` peuvent se combiner avec un outil nommé `pg_combinebackup`, mais les fonctionnalités sont encore un peu sommaire.

Il ne permet pas non plus de continuer à archiver les journaux, contrairement aux outils de PITR classiques. Cependant, ceux-ci peuvent l'utiliser pour réaliser la première copie des fichiers d'une instance (c'est le cas de barman, notamment).

Mise en place :

`pg_basebackup` nécessite des connexions de réplication. Il peut utiliser un slot de réplication, une technique qui fiabilise la sauvegarde ou la réplication en indiquant à l'instance quels journaux elle doit conserver. Par défaut, tout est en place pour une connexion en local :

```
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
```

Ensuite, il faut configurer le fichier `pg_hba.conf` pour accepter la connexion du serveur où est exécutée `pg_basebackup`. Dans notre cas, il s'agit du même serveur avec un utilisateur dédié :

```
host  replication  sauve  127.0.0.1/32  scram-sha-256
```

Enfin, il faut créer un utilisateur dédié à la réplication (ici `sauve`) qui sera le rôle créant la connexion et lui attribuer un mot de passe :

```
CREATE ROLE sauve LOGIN REPLICATION;
\password sauve
```

Dans un but d'automatisation, le mot de passe finira souvent dans un fichier `.pgpass` ou équivalent.

Il ne reste plus qu'à :

- lancer `pg_basebackup`, ici en lui demandant une archive au format `tar` ;
- archiver les journaux en utilisant une connexion de réplication par *streaming* ;
- forcer le *checkpoint*.

Cela donne la commande suivante, ici pour une sauvegarde en local :

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/

644320/644320 kB (100%), 1/1 tablespace
```

Le résultat est ici un ensemble des deux archives : les journaux sont à part et devront être dépaquetés dans le `pg_wal` à la restauration.

```
$ ls -l /var/lib/postgresql/backups/
total 4163772
-rw------- 1 postgres postgres 659785216 Oct  9 11:37 base.tar
-rw------- 1 postgres postgres  16780288 Oct  9 11:37 pg_wal.tar
```

La cible doit être vide. En cas d'arrêt avant la fin, il faudra tout recommencer de zéro, c'est une limite de l'outil.

Restauration :

Pour restaurer, il suffit de remplacer le PGDATA corrompu par le contenu de l'archive, ou de créer une nouvelle instance et de remplacer son PGDATA par cette sauvegarde. Au démarrage, l'instance

repérera qu'elle est une sauvegarde restaurée et réappliquera les journaux. L'instance contiendra les données telles qu'elles étaient à la **fin** du `pg_basebackup`.

Noter que les fichiers de configuration ne sont PAS inclus s'ils ne sont pas dans le PGDATA, notamment sur Debian et ses versions dérivées.

Différences entre les versions :

Un slot de réplication temporaire sera créé par défaut pour garantir que le serveur gardera les journaux jusqu'à leur copie intégrale.

La commande `pg_basebackup` crée un fichier manifeste contenant la liste des fichiers sauvegardés, leur taille et une somme de contrôle. Cela permet de vérifier la sauvegarde avec l'outil `pg_verifybackup` (ce dernier ne fonctionne hélas que sur une sauvegarde au format `plain`, ou décompressée).

Lisez bien la documentation de pg_basebackup[2] pour votre version précise de PostgreSQL, des options ont changé de nom au fil des versions.

Même avec un serveur un peu ancien, il est possible d'utiliser un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

L'outil est développé plus en détail dans notre module I4[3].

[2] https://docs.postgresql.fr/current/app-pgbasebackup.html
[3] https://dali.bo/i4_html

7.5 SAUVEGARDE PITR

7.5.1 Étapes d'une sauvegarde PITR

2 étapes :

- Archivage des journaux de transactions
  - archivage interne
  - ou outil `pg_receivewal`
- Sauvegarde des fichiers
  - `pg_basebackup`
  - ou manuellement (outils de copie classiques)

Même si la mise en place est plus complexe qu'un `pg_dump`, la sauvegarde PITR demande peu d'étapes. La première chose à faire est de mettre en place l'archivage des journaux de transactions. Un choix est à faire entre un archivage classique et l'utilisation de l'outil `pg_receivewal`.

Lorsque cette étape est réalisée (et fonctionnelle), il est possible de passer à la seconde : la sauvegarde des fichiers. Là-aussi, il y a différentes possibilités : soit manuellement, soit `pg_basebackup`, soit son propre script ou un outil extérieur.

7.5.2 Méthodes d'archivage

- Deux méthodes :
  - processus interne `archiver`
  - outil `pg_receivewal` (flux de réplication)

La méthode historique, et la plus utilisée toujours, utilise le processus `archiver`. Ce processus fonctionne sur l'instance sauvegardée et fait partie des processus du serveur PostgreSQL. Seule sa (bonne) configuration incombe au DBA, notamment le paramètre `archive_command`.

Une autre méthode existe : `pg_receivewal`. Cet outil livré aussi avec PostgreSQL se comporte comme un serveur secondaire, tournant sur un autre serveur. Il reconstitue les journaux de transactions à partir du flux de réplication.

Chaque solution a ses avantages et inconvénients.

7.5.3 Choix du répertoire d'archivage

- À faire quelle que soit la méthode d'archivage
- Attention aux droits d'écriture dans le répertoire
  - la commande configurée pour la copie doit pouvoir écrire dedans
  - et potentiellement y lire

Dans le cas de l'archivage historique, le serveur PostgreSQL va exécuter une commande, dont le rôle sera de copier les journaux à l'extérieur de son répertoire de travail :

- sur un disque différent du même serveur (à éviter) ;
- sur un disque d'un autre serveur (montage réseau, transfert par SSH, bucket S3…) ;
- sur des bandes…

L'exemple pris ici utilise le répertoire `/mnt/nfs1/archivage` comme répertoire de copie. Ce répertoire est en fait un montage NFS. Il faut donc commencer par créer ce répertoire et s'assurer que l'utilisateur système **postgres** peut écrire dedans :

```
# mkdir /mnt/nfs1/archivage
# chown postgres:postgres /mnt/nfs1/archivage
```

Dans le cas de l'archivage avec `pg_receivewal`, c'est cet outil qui va écrire les journaux dans un répertoire de travail. Cette écriture ne peut se faire qu'en local. Cependant, le répertoire peut se trouver dans un montage réseau (NFS…).

7.5.4 Processus archiver : configuration (1/4)

Préalables :

- Dans `postgresql.conf` :
  - `wal_level = replica`
  - `archive_mode = on` (ou `always`)

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

Niveau d'archivage :

La valeur par défaut de `wal_level` est adéquate :

```
wal_level = replica
```

Ce paramètre indique le niveau des informations écrites dans les journaux de transactions. Avec un niveau `minimal`, les journaux ne servent qu'à garantir la cohérence des fichiers de données en cas de crash. Dans le cas d'un archivage, il faut écrire plus d'informations, d'où la nécessité du niveau `replica` (qui est celui par défaut). Le niveau `logical`, nécessaire à la réplication logique[4], convient également.

Mode d'archivage :

Il s'active ainsi sur une instance seule ou primaire :

```
archive_mode = on
```

(La valeur `always` permet d'archiver depuis un secondaire. Avec `on`, l'instance n'archive les journaux que si elle est primaire.) Le changement nécessite un redémarrage.

7.5.5 Processus archiver : configuration (2/4)

La commande d'archivage :

- Dans `postgresql.conf` :
  - `archive_command = '… une commande …'`
  - ou : `archive_library = '… une bibliothèque …'` (v15+)

Enfin, une commande d'archivage doit être définie par le paramètre `archive_command`. `archive_command` sert à archiver un seul fichier à chaque appel.

PostgreSQL l'appelle une fois pour chaque fichier WAL, impérativement dans l'ordre des fichiers. En cas d'échec, elle est répétée indéfiniment jusqu'à réussite, avant de passer à l'archivage du fichier suivant.

(À noter qu'à partir de la version 15, il existe une alternative, avec l'utilisation du paramètre `archive_library`. Il est possible d'indiquer une bibliothèque partagée qui fera ce travail d'archivage. Une telle bibliothèque, écrite en C, devrait être plus puissante et performante. Un module basique est fourni avec PostgreSQL : basic_archive[5]. Notre blog présente un exemple fonctionnel de module d'archivage[6] utilisant une extension en C pour compresser les journaux de transactions. En

[4] https://dali.bo/w5_html
[5] https://docs.postgresql.fr/current/basic-archive.html
[6] https://blog.dalibo.com/2023/07/28/hackingpg2.html

production, il vaudra mieux utiliser une bibliothèque fournie par un outil PITR reconnu. Cependant, à notre connaissance (en décembre 2024), aucun outil n'utilise encore cette fonctionnalité, qui est sans doute plutôt utilisée par des opérateurs *cloud*. L'utilisation simultanée de `archive_command` et `archive_library` est déconseillée, et interdite depuis PostgreSQL 16.)

7.5.6 Processus archiver : configuration (3/4)

- Exemples d'`archive_command` :

```
archive_command='cp %p /mnt/nfs1/archivage/%f && sync /mnt/nfs1/'
archive_command='test ! -f /arch/%f && cp %p /arch/%f'
archive_command='/usr/bin/rsync -az %p postgres@10.9.8.7:/archives/%f'
archive_command='/opt/mon_script.sh %p %f'
archive_command='/usr/bin/pgbackrest --stanza=prod archive-push %p'
archive_command='/usr/bin/barman-wal-archive backup prod %p'
archive_command='/bin/true'  # désactivation
```

- Ne pas oublier de forcer l'écriture de l'archive sur disque
- Code retour de l'archivage entre 0 (ok) et 125

PostgreSQL laisse le soin à l'administrateur de définir la méthode d'archivage des journaux de transactions suivant son contexte. Si vous utilisez un outil de sauvegarde, la commande vous sera probablement fournie. Une simple commande de copie suffit dans la plupart des cas. La directive `archive_command` peut alors être positionnée comme suit :

```
archive_command = 'cp %p /mnt/nfs1/archivage/%f'
```

Le joker `%p` est remplacé par le chemin complet vers le journal de transactions à archiver, par exemple `pg_wal/000000010000000A900000065`. Le joker `%f` correspond au nom du journal de transactions une fois archivé, par exemple `000000010000000A900000065`. La commande réellement exécutée ressemblera donc à ceci :

```
cp pg_wal/000000010000000A900000065 /mnt/nfs1/archivage/000000010000000A900000065
```

En toute rigueur, une copie du fichier ne suffit pas. Par exemple, dans le cas de la commande `cp`, le nouveau fichier n'est pas immédiatement écrit sur disque. La copie est effectuée dans le cache disque du système d'exploitation. En cas de crash juste après la copie, il est tout à fait possible de perdre l'archive. Il est donc essentiel d'ajouter une étape de synchronisation du cache sur disque (ordre `sync`).

La commande d'archivage suivante est donnée dans la documentation officielle à titre d'exemple :

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p
↪  /mnt/server/archivedir/%f'
```

Cette commande a deux inconvénients. Elle ne garantit pas que les données seront synchronisées sur disque. Et si le fichier existe ou a été copié partiellement à cause d'une erreur précédente, la copie ne s'effectuera pas.

Cette protection est une bonne chose. Cependant, il faut être vigilant lorsque l'on rétablit le fonctionnement de l'*archiver* suite à un incident ayant provoqué des écritures partielles dans le répertoire d'archive, comme une saturation de l'espace disque.

Il est aussi possible de placer dans `archive_command` le nom d'un script bash, perl ou autre. L'intérêt est de pouvoir faire plus qu'une simple copie. On peut y ajouter la demande de synchronisation du cache sur disque, ou de la gestion d'erreur plus complexe. Il peut aussi être intéressant de tracer l'action de l'archivage, ou de compresser le journal avant archivage.

Dans vos commandes et scripts, il faut s'assurer d'une chose : la commande d'archivage doit retourner 0 en cas de réussite et surtout une valeur différente de 0 en cas d'échec.
Si le code retour de la commande est compris entre 1 et 125, PostgreSQL va tenter périodiquement d'archiver le fichier jusqu'à ce que la commande réussisse (autrement dit, renvoie 0).
Tant qu'un fichier journal n'est pas considéré comme archivé avec succès, PostgreSQL ne le supprimera ou recyclera pas ! Il ne cherchera pas non plus à archiver les fichiers suivants.

De plus si le code retour de la commande est supérieur à 125, le processus `archiver` redémarrera, et l'erreur ne sera pas comptabilisée dans la vue `pg_stat_archiver` !
Ce cas de figure inclut les erreurs de type `command not found` associées aux codes retours 126 et 127, ou le cas de `rsync`, qui renvoie un code retour 255 en cas d'erreur de syntaxe ou de configuration du ssh.

Il est donc important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur. Les causes d'échec sont nombreuses : problème réseau, montage inaccessible, erreur de paramétrage de l'outil, droits insuffisants ou expirés, génération de journaux trop rapide…

À titre d'exemple encore, les commandes fournies par pgBackRest ou barman ressemblent à ceci :

```
# pgBackRest
archive_command='/usr/bin/pgbackrest --stanza=prod archive-push %p'

# barman
archive_command='/usr/bin/barman-wal-archive backup prod %p'
```

Enfin, le paramétrage suivant archive « dans le vide ». Cette astuce est utilisée lors de certains dépannages, ou pour éviter le redémarrage que nécessiterait la désactivation de `archive_mode`.

```
archive_mode = on
archive_command = '/bin/true'
```

7.5.7 Processus archiver : configuration (4/4)

- Dans `postgresql.conf` (suite) :
  - période maximale entre deux archivages
  - `archive_timeout = '… min'`

Si l'activité en écriture est très réduite en volume, il peut se passer des heures entre deux archivages de journaux. Il est alors conseillé de forcer un archivage périodique, même si le journal n'a pas été rempli complètement, en indiquant un délai maximum entre deux archivages :

```
archive_timeout = '5min'
```

(La valeur par défaut, 0, désactive ce comportement.)

Ainsi, la perte de données maximale sera de cette durée.

Comme la taille d'un fichier journal, même incomplet, reste fixe (16 Mo par défaut), la consommation en terme d'espace disque sera plus importante (la compression par l'outil d'archivage peut compenser cela), et le temps de restauration plus long.

7.5.8 Processus archiver : lancement

- Redémarrage de PostgreSQL
  - si modification de `wal_level` et/ou `archive_mode`
- ou rechargement de la configuration

Il ne reste plus qu'à indiquer à PostgreSQL de recharger sa configuration pour que l'archivage soit en

place (avec `SELECT pg_reload_conf();` ou la commande `reload` adaptée au système). Dans le cas où il a fallu définir `wal_level = replica` ou `archive_mode = on`, il faudra relancer PostgreSQL.

7.5.9 Processus archiver : supervision

- Vue `pg_stat_archiver`
- `pg_wal/archive_status/`
  - fichiers `.ready` et `.done`
- Archivage dans l'ordre des fichiers
- Taille de `pg_wal`
  - si saturation : Arrêt !
- Traces

PostgreSQL archive les journaux impérativement dans l'ordre où ils ont été générés.

S'il y a un problème d'archivage d'un journal, les suivants ne seront pas archivés non plus, et vont s'accumuler dans `pg_wal` ! De plus, une saturation de la partition portant `pg_wal` mènera à l'arrêt de l'instance PostgreSQL !

La supervision se fait de quatre manières complémentaires.

Taille :

Si le répertoire `pg_wal` commence à grossir fortement, c'est que PostgreSQL n'arrive plus à recycler ses journaux de transactions : c'est un indicateur d'une commande d'archivage n'arrivant pas à faire son travail pour une raison ou une autre. Ce peut être temporaire si l'archivage est juste lent. Les causes classiques sont un réseau saturé, une compression des journaux trop lente, ou des écritures trop intenses. Si l'archivage est complètement bloqué (à cause d'un disque saturé par exemple), ce répertoire grossira indéfiniment.

Vue pg_stat_archiver :

La vue système `pg_stat_archiver` indique les derniers journaux archivés et les dernières erreurs. Dans l'exemple suivant, il y a eu un problème pendant quelques secondes, d'où 6 échecs, avant que l'archivage reprenne :

```
SELECT * FROM pg_stat_archiver \gx
```

```
-[ RECORD 1 ]------+------------------------------
archived_count     | 156
last_archived_wal  | 0000000200000001000000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count       | 6
last_failed_wal    | 0000000200000001000000D7
last_failed_time   | 2020-01-17 18:24:24.463038+00
stats_reset        | 2020-01-17 16:08:37.980214+00
```

Comme dit plus haut, pour que cette supervision soit fiable, la commande exécutée doit renvoyer un code retour inférieur ou égal à 125. Dans le cas contraire, le processus `archiver` redémarre et l'erreur n'apparaît pas dans la vue !

L'ordre `SELECT pg_switch_wal()` force un changement de journal, et donc l'archivage du journal en cours, à condition qu'il y ait eu une activité minimale. Cette commande est pratique pour tester.

Traces :

On trouvera la sortie et surtout les messages d'erreurs du script d'archivage dans les traces (qui dépendent bien sûr du script utilisé) :

```
2020-01-17 18:24:18.427 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:18.427 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/0000000200000001000000D7 /opt/pgsql/archives/0000000200000001000000D7
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/0000000200000001000000D7 /opt/pgsql/archives/0000000200000001000000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:20.463 UTC [15431] LOG:  archive command failed with exit code 3
```

C'est donc le premier endroit à regarder en cas de souci ou lors de la mise en place de l'archivage.

pg_wal/archive_status :

Enfin, on peut monitorer les fichiers présents dans `pg_wal/archive_status`. Les fichiers `.ready`, de taille nulle, indiquent en permanence quels sont les journaux prêts à être archivés. Théoriquement, leur nombre doit donc rester faible et retomber rapidement à 0 ou 1. Le service `ready_archives` de la sonde Nagios check_pgactivity[7] se base sur ce répertoire.

```
SELECT * FROM pg_ls_dir ('pg_wal/archive_status') ORDER BY 1;

           pg_ls_dir
-------------------------------
```

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

```
000000020000000100000DE.done
000000020000000100000DF.done
000000020000000100000E0.done
000000020000000100000E1.ready
000000020000000100000E2.ready
000000020000000100000E3.ready
000000020000000100000E4.ready
000000020000000100000E5.ready
000000020000000100000E6.ready
00000002.history.done
```

7.5.10 pg_receivewal

- Archivage via le protocole de réplication
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
  - toujours au plus près du primaire
- Slots de réplication obligatoires

`pg_receivewal` est un outil permettant de se faire passer pour un serveur secondaire utilisant la réplication en flux (*streaming replication*) dans le but d'archiver en continu les journaux de transactions. Il fonctionne habituellement sur un autre serveur, où seront archivés les journaux. C'est une alternative à l'`archiver`.

Comme il utilise le protocole de réplication, les journaux archivés ont un retard bien inférieur à celui induit par la configuration du paramètre `archive_command` ou du paramètre `archive_library`, les journaux de transactions étant écrits au fil de l'eau avant d'être complets. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le serveur primaire. On peut même utiliser une réplication synchrone (paramètres `synchronous_commit` et `synchronous_standby_names`) pour ne perdre aucune transaction, si l'on accepte un impact certain sur la latence des transactions.

Cet outil utilise les mêmes options de connexion que la plupart des outils PostgreSQL, avec en plus l'option `-D` pour spécifier le répertoire où sauvegarder les journaux de transactions. L'utilisateur spécifié doit bien évidemment avoir les attributs `LOGIN` et `REPLICATION`.

Comme il s'agit de conserver toutes les modifications effectuées par le serveur dans le cadre d'une sauvegarde permanente, il est nécessaire de s'assurer qu'on ne puisse pas perdre des journaux de transactions. Il n'y a qu'un seul moyen pour cela : utiliser la technologie des slots de réplication. En utilisant un slot de réplication, `pg_receivewal` s'assure que le serveur ne va pas recycler des journaux dont `pg_receivewal` n'aurait pas reçu les enregistrements. On retrouve donc le risque d'accumulation des journaux sur le serveur principal si `pg_receivewal` ne fonctionne pas.

Voici l'aide de cet outil en v15 :

```
$ pg_receivewal --help
pg_receivewal reçoit le flux des journaux de transactions PostgreSQL.

Usage :
  pg_receivewal [OPTION]...

Options :
  -D, --directory=RÉPERTOIRE     reçoit les journaux de transactions dans ce
                                 répertoire
  -E, --endpos=LSN               quitte après avoir reçu le LSN spécifié
      --if-not-exists            ne pas renvoyer une erreur si le slot existe
                                 déjà lors de sa création
  -n, --no-loop                  ne boucle pas en cas de perte de la connexion
      --no-sync                  n'attend pas que les modifications soient
                                 proprement écrites sur disque
  -s, --status-interval=SECS     durée entre l'envoi de paquets de statut au
                                 (par défaut 10)
  -S, --slot=NOMREP              slot de réplication à utiliser
      --synchronous              vide le journal de transactions immédiatement
                                 après son écriture
  -v, --verbose                  affiche des messages verbeux
  -V, --version                  affiche la version puis quitte
  -Z, --compress=METHOD[:DETAIL]
                                 compresse comme indiqué
  -?, --help                     affiche cette aide puis quitte

Options de connexion :
  -d, --dbname=CHAÎNE_CONNEX     chaîne de connexion
  -h, --host=HÔTE                hôte du serveur de bases de données ou
                                 répertoire des sockets
  -p, --port=PORT                numéro de port du serveur de bases de données
  -U, --username=UTILISATEUR     se connecte avec cet utilisateur
  -w, --no-password              ne demande jamais le mot de passe
  -W, --password                 force la demande du mot de passe (devrait
                                 survenir automatiquement)

Actions optionnelles :
      --create-slot              crée un nouveau slot de réplication
                                 (pour le nom du slot, voir --slot)
      --drop-slot                supprime un nouveau slot de réplication
                                 (pour le nom du slot, voir --slot)

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
Page d'accueil de PostgreSQL : <https://www.postgresql.org/>
```

`pg_receivewal` est utilisé par exemple par l'outil de sauvegarde PITR barman. Les auteurs de pgBackRest préfèrent utiliser `archive_command` car ils peuvent ainsi mieux paralléliser des débits élevés.

7.5.11 pg_receivewal - configuration serveur

- `postgresql.conf` :

```
# configuration  par défaut
max_wal_senders = 10
max_replication_slots = 10
```

- `pg_hba.conf` :

```
host  replication  repli_user  192.168.0.0/24  scram-sha-256
```

- Utilisateur de réplication :

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

Le paramètre `max_wal_senders` indique le nombre maximum de connexions de réplication sur le serveur. Logiquement, une valeur de 1 serait suffisante, mais il faut compter sur quelques soucis réseau qui pourraient faire perdre la connexion à `pg_receivewal` sans que le serveur primaire n'en soit mis au courant, et du fait que certains autres outils peuvent utiliser la réplication. `max_replication_slots` indique le nombre maximum de slots de réplication. Pour ces deux paramètres, le défaut est 10 et suffit dans la plupart des cas.

Si l'on modifie un de ces paramètres, il est nécessaire de redémarrer le serveur PostgreSQL.

Les connexions de réplication nécessitent une configuration particulière au niveau des accès. D'où la modification du fichier `pg_hba.conf`. Le sous-réseau (192.168.0.0/24) est à modifier suivant l'adressage utilisé. Il est d'ailleurs préférable de n'indiquer que le serveur où est installé `pg_receivewal` (plutôt que l'intégralité d'un sous-réseau).

L'utilisation d'un utilisateur de réplication n'est pas obligatoire mais fortement conseillée pour des raisons de sécurité.

7.5.12 pg_receivewal - redémarrage du serveur

- Redémarrage de PostgreSQL
- Slot de réplication

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

Enfin, nous devons créer le slot de réplication qui sera utilisé par `pg_receivewal`. La fonction `pg_create_physical_replication_slot()` est là pour ça. Il est à noter que la liste des slots est disponible dans le catalogue système `pg_replication_slots`.

7.5.13 pg_receivewal - lancement de l'outil

- Exemple de lancement

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

- Journaux créés en temps réel dans le répertoire de stockage
- Mise en place d'un script de démarrage
- S'il n'arrive pas à joindre le serveur primaire
  - Au démarrage de l'outil : `pg_receivewal` s'arrête
  - En cours d'exécution : `pg_receivewal` tente de se reconnecter
- Nombreuses options

On peut alors lancer `pg_receivewal` :

```
pg_receivewal -h 192.168.0.1 -U repli_user -D /data/archives -S archivage
```

Les journaux de transactions sont alors créés en temps réel dans le répertoire indiqué (ici, `/data/archives`) :

```
-rwx------  1 postgres postgres  16MB juil. 27 00000001000000000000000E*
-rwx------  1 postgres postgres  16MB juil. 27 00000001000000000000000F*
-rwx------  1 postgres postgres  16MB juil. 27 00000001000000000000000.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.

7.5.14 Avantages et inconvénients

- Méthode archiver
  - simple à mettre en place
  - perte au maximum d'un journal de transactions
- Méthode `pg_receivewal`
  - mise en place plus complexe
  - perte minimale voire nulle

La méthode archiver est la méthode la plus simple à mettre en place. Elle est gérée intégralement par PostgreSQL, donc il n'est pas nécessaire de créer et installer un script de démarrage. Cependant, un journal de transactions n'est archivé que quand PostgreSQL l'ordonne, soit parce qu'il a rempli le journal en question, soit parce qu'un utilisateur a forcé un changement de journal (avec la fonction `pg_switch_wal` ou suite à un `pg_backup_stop`), soit parce que le délai maximum entre deux archivages a été dépassé (paramètre `archive_timeout`). Il est donc possible de perdre un grand nombre de transactions (même si cela reste bien inférieur à la perte qu'une restauration d'une sauvegarde logique occasionnerait).

La méthode `pg_receivewal` est plus complexe à mettre en place. Il faut exécuter ce démon, généralement sur un autre serveur. Un script de démarrage doit donc être configuré. Par contre, elle a le gros avantage de ne perdre pratiquement aucune transaction, surtout en mode synchrone. Les enregistrements de transactions sont envoyés en temps réel à `pg_receivewal`. Ce dernier les place dans un fichier de suffixe `.partial`, qui est ensuite renommé pour devenir un journal de transactions complet.

7.6 SAUVEGARDE PITR MANUELLE

7.7 ÉTAPES D'UNE SAUVEGARDE PITR MANUELLE

- 3 étapes :
  - fonction de démarrage
  - copie des fichiers par outil externe
  - fonction d'arrêt
- Exclusive : simple… & obsolète ! (< v15)
- Concurrente : plus complexe à scripter
- Aucun impact pour les utilisateurs ; pas de verrou
- Préférer des outils dédiés qu'un script maison

Une fois l'archivage en place, une sauvegarde à chaud a lieu en trois temps :

- l'appel à la fonction de démarrage ;
- la copie elle-même par divers outils externes (PostgreSQL ne s'en occupe pas) ;
- l'appel à la fonction d'arrêt.

La fonction de démarrage s'appelle `pg_backup_start()` à partir de la version 15 mais avait pour nom `pg_start_backup()` auparavant. De la même façon, la fonction d'arrêt s'appelle `pg_backup_stop()` à partir de la version 15, mais `pg_stop_backup()` avant.

La sauvegarde exclusive était une méthode simple, et cela en faisait le choix par défaut. Il suffisait d'appeler les fonctions concernées avant et après la copie des fichiers. Il ne pouvait y en avoir qu'une à la fois. Elle ne fonctionnait que depuis un primaire.

À cause de ces limites et de différents problèmes, , la sauvegarde exclusive est déclarée obsolète depuis la 9.6, et n'est plus disponible depuis la version 15. Même sur les versions antérieures, il est conseillé d'utiliser dès maintenant des scripts utilisant les sauvegardes concurrentes.

Tout ce qui suit ne concerne plus que la sauvegarde concurrente.

La sauvegarde concurrente peut être lancée plusieurs fois en parallèle. C'est utile pour créer des secondaires alors qu'une sauvegarde physique tourne, par exemple. Elle est nettement plus complexe à gérer par script. Elle peut être exécutée depuis un serveur secondaire, ce qui allège la charge sur le primaire.

Pendant la sauvegarde, l'utilisateur ne verra aucune différence (à part peut-être la conséquence d'I/O saturées pendant la copie). Aucun verrou n'est posé. Lectures, écritures, suppression et création de tables, archivage de journaux et autres opérations continuent comme si de rien n'était.

La description du mécanisme qui suit est essentiellement destinée à la compréhension et l'expérimentation. En production, un script maison reste une possibilité, mais préférez des outils dédiés et fiables : pg_basebackup, pgBackRest…

Les sauvegardes manuelles servent cependant encore quand on veut utiliser une sauvegarde par snapshot de partition ou de baie, ou avec `rsync` (car pg_basebackup ne sait pas synchroniser vers une sauvegarde interrompue ou ancienne), et quand les outils conseillés ne sont pas utilisables ou disponibles sur le système.

7.7.1 Sauvegarde manuelle - 1/3 : pg_backup_start

`SELECT pg_backup_start (`

- `un_label` : texte
- `fast` : forcer un checkpoint ?

`)`

L'exécution de `pg_backup_start()` peut se faire depuis n'importe quelle base de données de l'instance.

(Rappelons que pour les versions avant la 15, la fonction s'appelle `pg_start_backup()`. Pour effectuer une sauvegarde non-exclusive avec ces versions, il faudra positionner un troisième paramètre[8] à `false`.)

Le label (le texte en premier argument) n'a aucune importance pour PostgreSQL (il ne sert qu'à l'administrateur, pour reconnaître le backup).

Le deuxième argument est un booléen qui permet de demander un *checkpoint* immédiat, si l'on est pressé et si un pic d'I/O n'est pas gênant. Sinon il faudra attendre souvent plusieurs minutes (selon la configuration du déclenchement du prochain checkpoint, dépendant des paramètres `checkpoint_timeout` et `max_wal_size` et de la rapidité d'écriture imposée par `checkpoint_completion_target`).

La session qui exécute la commande `pg_backup_start()` doit être la même que celle qui exécutera plus tard `pg_backup_stop()`. Nous verrons que cette dernière fonction fournira de quoi créer deux fichiers, qui devront être nommés `backup_label` et `tablespace_map`. Si la connexion est interrompue avant `pg_backup_stop()`, alors la sauvegarde doit être considérée comme invalide.

[8] https://docs.postgresql.fr/14/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-EXCLUSIVE

En plus de rester connectés à la base, les scripts qui gèrent la sauvegarde concurrente doivent donc récupérer et conserver les informations renvoyées par la commande de fin de sauvegarde.

La sauvegarde PITR est donc devenue plus complexe au fil des versions, et il est donc recommandé d'utiliser plutôt pg_basebackup ou des outils la supportant (barman, pgBackRest…).

7.7.2 Sauvegarde manuelle - 2/3 : copie des fichiers

- Cas courant : snapshot
  - cohérence ? redondance ?
- Sauvegarde des fichiers **à chaud**
  - répertoire principal des données
  - tablespaces
- Copie forcément incohérente (la restauration des journaux corrigera)
- `rsync` et autres outils
- Ignorer :
  - `postmaster.pid`, `log`, `pg_wal`, `pg_replslot` et quelques autres
- Ne pas oublier : configuration !

La deuxième étape correspond à la sauvegarde des fichiers. Le choix de l'outil dépend de l'administrateur. Cela n'a aucune incidence au niveau de PostgreSQL.

La sauvegarde doit comprendre aussi les tablespaces si l'instance en dispose.

Snapshot :

Il est possible d'effectuer cette étape de copie des fichiers par snapshot au niveau de la baie, de l'hyperviseur, ou encore de l'OS (LVM, ZFS…).

Un snaphost cohérent, y compris entre les tablespaces, permet théoriquement de réaliser une sauvegarde en se passant des étapes `pg_backup_start()` et `pg_backup_stop()`. La restauration de ce snapshot équivaudra pour PostgreSQL à un redémarrage brutal.
Pour une sauvegarde PITR, il faudra cependant toujours encadrer le snapshot des appels aux fonctions de démarrage et d'arrêt ci-dessus, et c'est généralement ce que font les outils comme Veeam ou Tina. L'utilisation d'un tel outil implique de vérifier qu'il sait gérer les sauvegardes non exclusives pour utiliser PostgreSQL 15 et supérieurs.

Le point noir de la sauvegarde par snapshot est d'être liée au même système matériel que l'instance PostgreSQL (disque, hyperviseur, datacenter…) Une défaillance grave du matériel, ou un bug de la baie, peut donc emporter, corrompre ou bloquer la sauvegarde en même temps que les données originales. La sécurité de l'instance est donc reportée sur celle de l'infrastructure sous-jacente : il vaut mieux que celle-ci soit répliquée sur plusieurs sites. Une copie parallèle, hors infastructure, des données de manière plus classique reste conseillée pour éviter un désastre total, et pour parer à la malveillance.

Copie manuelle :

La sauvegarde se fait à chaud : il est donc possible que pendant ce temps des fichiers changent, disparaissent avant d'être copiés ou apparaissent sans être copiés. Cela n'a pas d'importance en soi car les journaux de transactions corrigeront cela (leur archivage doit donc commencer **avant** le début de la sauvegarde et se poursuivre sans interruption jusqu'à la fin).
Il **faut** s'assurer que l'outil de sauvegarde supporte cela, c'est-à-dire qu'il soit capable de différencier les codes d'erreurs dus à « des fichiers ont bougé ou disparu lors de la sauvegarde » des autres erreurs techniques. `tar` par exemple convient : il retourne 1 pour le premier cas d'erreur, et 2 quand l'erreur est critique. `rsync` est très courant également.

Sur les plateformes Microsoft Windows, peu d'outils sont capables de copier des fichiers en cours de modification. Assurez-vous d'en utiliser un possédant cette fonctionnalité (il existe différents émulateurs des outils GNU sous Windows). Le plus sûr et simple est sans doute de renoncer à une copie manuelle des fichiers et d'utiliser `pg_basebackup`.

Exclusions :

Des fichiers et répertoires sont à ignorer, pour gagner du temps ou faciliter la restauration. Voici la liste exhaustive (disponible aussi dans la documentation officielle[9]) :

- `postmaster.pid`, `postmaster.opts`, `pg_internal.init` ;
- les fichiers de données des tables non journalisées (*unlogged*) ;
- `pg_wal`, ainsi que les sous-répertoires (mais à archiver séparément !) ;
- `pg_replslot` : les slots de réplication seront au mieux périmés, au pire gênants sur l'instance restaurée ;
- `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans` ne doivent pas être copiés (ils contiennent des informations propres à l'instance, ou qui ne survivent pas à un redémarrage) ;
- les fichiers et répertoires commençant par `pgsql_tmp` (fichiers temporaires) ;
- les fichiers autres que les fichiers et les répertoires standards (donc pas les liens symboliques).

[9] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

On n'oubliera pas les fichiers de configuration s'ils ne sont pas dans le PGDATA.

7.7.3 Sauvegarde manuelle - 3/3 : pg_backup_stop

Ne pas oublier !!
`SELECT * FROM pg_backup_stop (`

- `true` : attente de l'archivage

`)`

La dernière étape correspond à l'exécution de la procédure stockée `SELECT * FROM pg_backup_stop()`.

N'oubliez pas d'exécuter `pg_backup_stop()`, de vérifier qu'il finit avec succès et de récupérer les informations qu'il renvoie !
Cet oubli trop courant rend vos sauvegardes inutilisables !

PostgreSQL va alors :

- marquer cette fin de backup dans le journal des transactions (étape capitale pour la restauration) ;
- forcer la finalisation du journal de transactions courant et donc son archivage, afin que la sauvegarde (fichiers + archives) soit utilisable même en cas de crash juste l'appel à la fonction : `pg_backup_stop()` ne rendra pas la main (par défaut) tant que ce dernier journal n'aura pas été archivé avec succès.

La fonction renvoie :

- le *lsn* de fin de backup ;
- un champ destiné au fichier `backup_label` ;
- un champ destiné au fichier `tablespace_map`.

```
SELECT * FROM pg_stop_backup() \gx

NOTICE:  all required WAL segments have been archived
-[ RECORD 1 ]----------------------------------------------------------------
lsn        | 22/2FE5C788
labelfile  | START WAL LOCATION: 22/2B000028 (file 000000010000000220000002B)+
           | CHECKPOINT LOCATION: 22/2B000060                                +
           | BACKUP METHOD: streamed                                         +
           | BACKUP FROM: master                                             +
           | START TIME: 2019-12-16 13:53:41 CET                            +
```

```
           | LABEL: rr                                                    +
           | START TIMELINE: 1                                           +
           |
spcmapfile | 134141 /tbl/froid                                          +
           | 134152 /tbl/quota                                          +
           |
```

Ces informations se retrouvent aussi dans un fichier `.backup` mêlé aux journaux :

```
# cat /var/lib/postgresql/12/main/pg_wal/000000010000002200000002B.00000028.backup

START WAL LOCATION: 22/2B000028 (file 000000010000002200000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 000000010000002200000002F)
CHECKPOINT LOCATION: 22/2B000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2019-12-16 13:53:41 CET
LABEL: rr
START TIMELINE: 1
STOP TIME: 2019-12-16 13:54:04 CET
STOP TIMELINE: 1
```

Il faudra créer le fichier `tablespace_map` avec le contenu du champ `spcmapfile` :

```
134141 /tbl/froid
134152 /tbl/quota
```

… ce qui n'est pas trivial à scripter.

Ces deux fichiers devront être placés dans la sauvegarde, pour être présent d'entrée dans le PGDATA du serveur restauré.

À partir du moment où `pg_backup_stop()` rend la main, il est possible de restaurer la sauvegarde obtenue puis de rejouer les journaux de transactions suivants en cas de besoin, sur un autre serveur ou sur ce même serveur.

Tous les journaux archivés avant celui précisé par le champ `START WAL LOCATION` dans le fichier `backup_label` ne sont plus nécessaires pour la récupération de la sauvegarde du système de fichiers et peuvent donc être supprimés. Attention, il y a plusieurs compteurs hexadécimaux différents dans le nom du fichier journal, qui ne sont pas incrémentés de gauche à droite.

7.7.4 Sauvegarde de base à chaud : pg_basebackup

Outil de sauvegarde pouvant aussi servir au sauvegarde basique

- Backup de base ici **sans** les journaux :

```
$ pg_basebackup --format=tar --wal-method=none \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

`pg_basebackup` a été décrit plus haut. Il a l'avantage d'être simple à utiliser, de savoir quels fichiers ne pas copier, de fiabiliser la sauvegarde par un slot de réplication. Il ne réclame en général pas de configuration supplémentaire.

Si l'archivage est déjà en place, copier les journaux est inutile (`--wal-method=none`). Nous verrons plus tard comment lui indiquer où les chercher.

L'inconvénient principal de `pg_basebackup` reste son incapacité à reprendre une sauvegarde interrompue ou à opérer une sauvegarde différentielle ou incrémentale, du moins avant PostgreSQL 17.

7.7.5 Fréquence de la sauvegarde de base

- Dépend des besoins
- De tous les jours à tous les mois
- Plus elles sont espacées, plus la restauration est longue
  - et plus le risque d'un journal corrompu ou absent est important

La fréquence dépend des besoins. Une sauvegarde par jour est le plus commun, mais il est possible d'espacer encore plus la fréquence.

Cependant, il faut conserver à l'esprit que plus la sauvegarde est ancienne, plus la restauration sera longue, car un plus grand nombre de journaux seront à rejouer.

7.7.6 Suivi de la sauvegarde de base

- Vue `pg_stat_progress_basebackup`

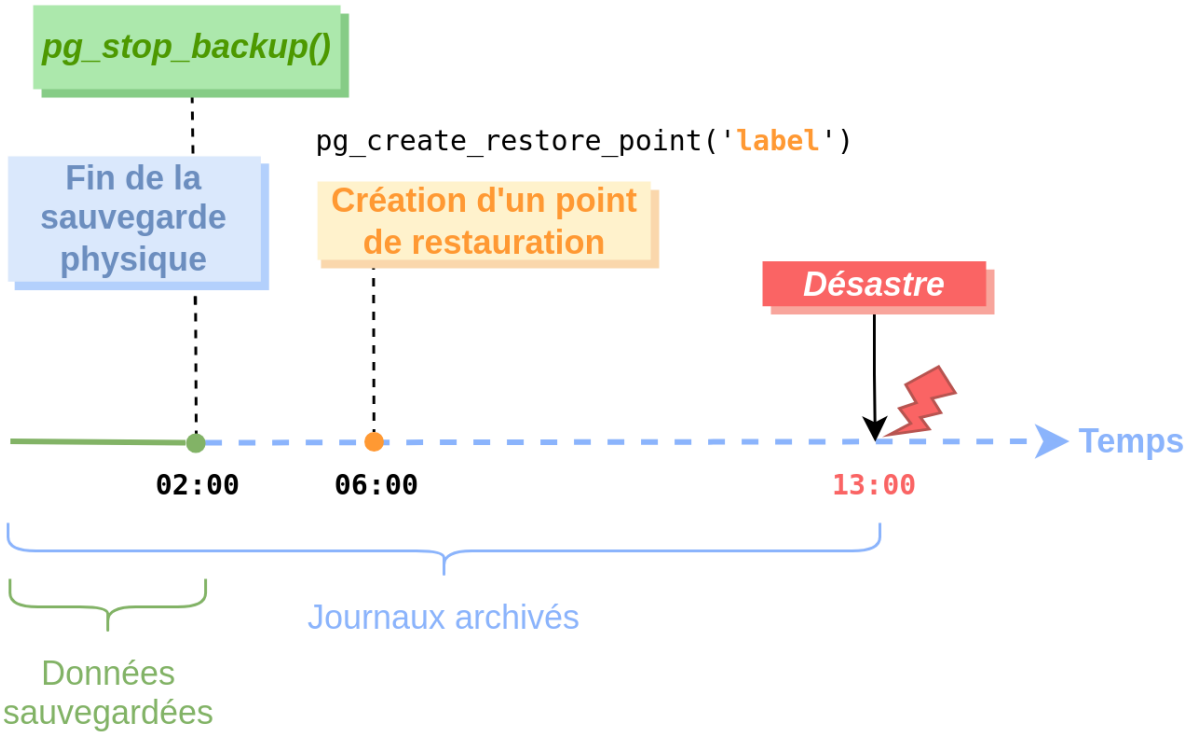
```
SELECT *, pg_size_pretty (backup_total) AS total,
round(100.0*backup_streamed/backup_total::numeric,2) AS "%"
FROM  pg_stat_progress_basebackup  \gx

-[ RECORD 1 ]--------+-------------------------
pid                  | 3608155
phase                | streaming database files
backup_total         | 925114368
backup_streamed      | 197094400
tablespaces_total    | 1
tablespaces_streamed | 0
total                | 882 MB
%                    | 21.30
```

La vue `pg_stat_progress_basebackup` permet de suivre la progression de la sauvegarde de base, quelque soit l'outil utilisé, à condition qu'il passe par le protocole de réplication. Cela permet ainsi de savoir à quelle phase la sauvegarde se trouve, quelle volumétrie a été envoyée, celle à envoyer, etc.

7.8 RESTAURER UNE SAUVEGARDE PITR

Simple, mais à appliquer rigoureusement

7.8.1 Exemple de scénario : sauvegarde



Dans cet exemple, la sauvegarde a fini à 02 h du matin (le moment où une fonction `pg_backup_stop()` est appelée par un outil ou un script). La sauvegarde des fichiers de données s'est effectuée en parallèle de l'archivage des journaux, qui continue indéfiniment ensuite.

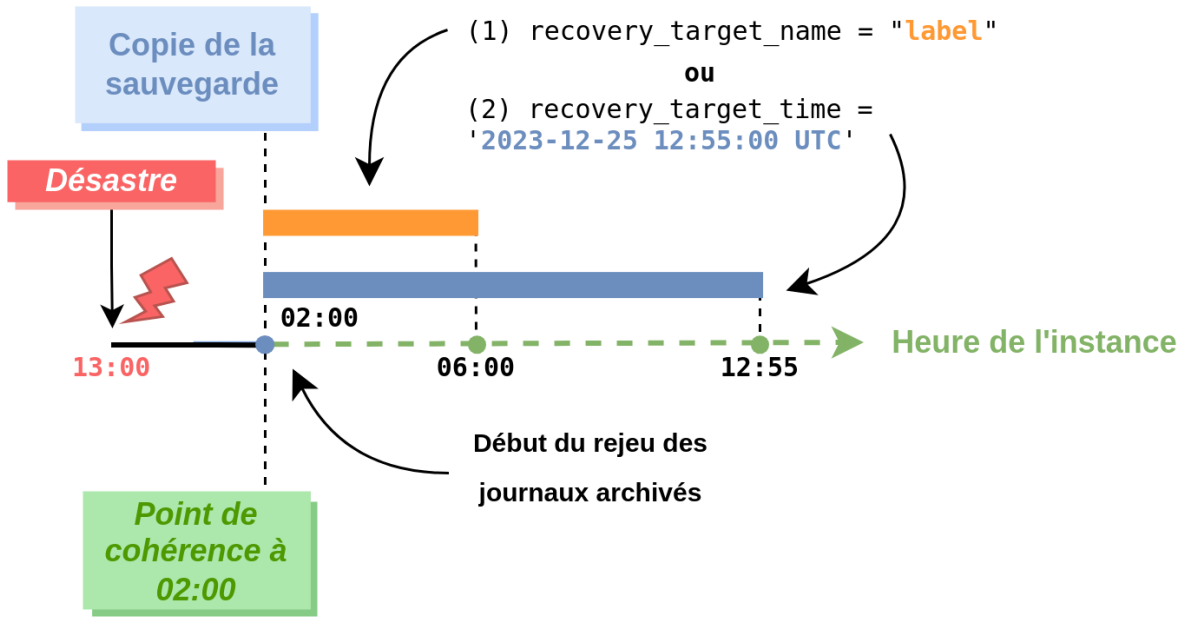
À 06 h, le DBA a créé un « point de restauration », ainsi (le nom est arbitraire) :

```
SELECT pg_create_restore_point ('label');

 pg_create_restore_point
-------------------------
 26/9B000090
```

Ces points de restauration sont totalement optionnels, et peuvent être créés avant certaines opérations (par exemple un batch ou une mise en production), ou périodiquement.

Une catastrophe quelconque frappe à 13 h et il faut restaurer.

7.8.2 Exemple de scénario : restauration



La ligne de temps de ce schéma correspond aux heures des transactions originales.

Pour restaurer, le DBA copie la sauvegarde de base, modifie la configuration et démarre l'instance qui commence à rejouer les journaux. Elle atteint le point de cohérence (correspondant à la fin de la sauvegarde), et est donc dans l'état correspondant à la fin de la sauvegarde, donc comme à 02 h.

Deux possibilités sont montrées ici :

1. le DBA demande à redérouler les journaux jusqu'au point de restauration `label`, pour avoir une image de la base à 06 h ;
2. le DBA demande à redérouler les journaux jusque 12 h 55, juste avant la catastrophe.

Nous verrons qu'il lui suffira de choisir les bons paramètres (ici `recovery_target_name` ou `recovery_target_time`).

7.8.3 Restaurer une sauvegarde PITR (1/5)

- S'il s'agit du même serveur
  - arrêter PostgreSQL
- Nettoyer les répertoires des données
  - y compris les tablespaces
  - sauf outil travaillant en mode delta

La restauration se déroule en trois, voire quatre étapes suivant qu'elle est effectuée sur le même serveur ou sur un autre serveur. Dans le cas où la restauration a lieu sur le même serveur, les étapes préliminaires suivantes sont à effectuer.

Il faut arrêter PostgreSQL s'il n'est pas arrêté. Cela arrive quand la restauration a pour but, par exemple, de récupérer des données qui ont été supprimées par erreur.

Ensuite, il faut supprimer (ou archiver) l'ancien répertoire des données pour pouvoir y placer la sauvegarde des fichiers. Écraser l'ancien répertoire n'est pas suffisant, il faut le supprimer, ainsi que les répertoires des tablespaces au cas où l'instance en possède. (L'exception est l'utilisation d'outils capable de trouver les différence entre les fichiers à restaurer et ceux présents, pour gagner du temps, comme `rsync` ou `pgbackrest restore --delta`.) Cela est valable aussi pour chaque tablespace. Une exception : on peut vouloir mettre de côté le dernier WAL (incomplet) d'une instance que l'on restaure pour ne perdre aucune transaction (voir plus bas).

7.8.4 Restaurer une sauvegarde PITR (2/5)

- Restaurer les fichiers de la sauvegarde
- Peut-être besoin de nettoyer les fichiers restaurés
  - ex : `pg_wal`, `postmaster.pid`, `log/`
  - un bon outil ne les a pas copiés
- Si restauration après crash :
  - récupérer le dernier journal de transactions connu (si disponible)

La sauvegarde des fichiers peut enfin être restaurée. Il faut bien porter attention à ce que les fichiers

soient restaurés au même emplacement, tablespaces compris.

Une fois cette étape effectuée, il peut être intéressant de faire un peu de ménage. Des outils comme pgBackRest ou Barman, ou un bon script, rendent cette étape inutile, car ils n'auront pas copié les fichiers inutiles.

Par exemple, le fichier `postmaster.pid` peut poser un problème au démarrage. On peut supprimer les traces, si elles sont dans `$PGDATA/log/`, pour éviter toute confusion entre l'ancienne et la nouvelle incarnation de l'instance, surtout si on restaure sur une nouvelle machine.

Si des journaux de transactions sont compris dans la sauvegarde (`$PGDATA/pg_wal/`), il est préférable de les supprimer. De toute façon, PostgreSQL les ignorera s'ils sont dans les archives. La commande sera similaire à celle-ci :

```
$ rm postmaster.pid log/* pg_wal/[0-9A-F]*
```

Enfin, si on restaure après un crash, on peut chercher à récupérer le dernier journal en cours lors de l'arrêt, incomplet et qui n'a pu être archivé. S'il n'a pas disparu avec le disque par exemple, le récupérer peut permettre de sauver les transactions qui y figurent. Ce dernier journal sera pris en compte après le rejeu des journaux archivés, après l'échec de la `restore_command` sur ce journal.

7.8.5 Restaurer une sauvegarde PITR (3/5)

- Indiquer qu'on est en restauration
  - fichier vide `recovery.signal`
- Commande de restauration
  - `restore_command = '… une commande …'`
  - directement dans `pg_wal/`
  - dans `postgresql.[auto.]conf`

Quand PostgreSQL démarre après avoir subi un arrêt brutal, il ne restaure que les journaux en place dans `pg_wal/`, puis il s'ouvre en écriture. Pour une restauration, il faut lui indiquer qu'il doit plutôt demander les journaux quelque part, et les rejouer tous jusqu'à épuisement, avant de s'ouvrir. Pour cela, il suffit de créer un fichier vide `recovery.signal` dans le répertoire des données.

Pour la récupération des journaux, le paramètre essentiel est `restore_command`. Il contient une commande symétrique des paramètres `archive_command` (ou `archive_library`) pour l'archivage. Il s'agit d'une commande copiant un journal dans le répertoire des journaux `pg_wal/`. Cette commande est souvent fournie par l'outil de sauvegarde PITR s'il y en a un. Si nous poursuivons notre exemple, ce paramètre pourrait être :

```
restore_command = 'cp /mnt/nfs1/archivage/%f %p'
```

Cette commande est appelée après la restauration de chaque journal pour récupérer le suivant, qui est restauré, et ainsi de suite.

Techniquement, la commande est lancée depuis le PGDATA, en remplaçant `%f` par le nom du journal attendu (par exemple `000000010000009800000E0`) et `%p` par `pg_wal/RECOVERYXLOG`. Ce dernier fichier sera ensuite renommé avec le nom du journal. On peut également constater la recherche d'un fichier d'historique des *timelines*, par exemple `00000002.history`, sauvé temporairement sous le nom `pg_wal/RECOVERYHISTORY`. PostgreSQL cherche ces fichiers d'historique pour savoir quelle chaîne de journaux suivre quand il y a eu des restaurations ou bascules sur un secondaire (voir plus loin).

Il n'y a aucune parallélisation prévue, mais des outils de sauvegarde PITR peuvent en faire en arrière-plan pendant l'exécution de la commande (par exemple pgBackRest en mode asynchrone).

Si le but est de restaurer tous les journaux archivés, il n'est pas nécessaire d'aller plus loin dans la configuration. La restauration se poursuivra jusqu'à ce que `restore_command` tombe en erreur, ce qui signifie l'épuisement de tous les journaux disponibles, et la fin de la restauration.

Au cas où vous rencontreriez une instance en version 11 ou antérieure : il faut savoir que la restauration se paramétrait dans un fichier texte nommé `recovery.conf`, dans le PGDATA, contenant `recovery_command` et éventuellement les options de restauration.

7.8.6 Restaurer une sauvegarde PITR (4/5)

- Jusqu'où restaurer :
  - `recovery_target_name`, `recovery_target_time`
  - `recovery_target_xid`, `recovery_target_lsn`
  - `recovery_target_inclusive`
- Le backup de base doit être antérieur !
- Suivi de timeline :
  - `recovery_target_timeline` : `latest` (en général)
- Et on fait quoi ?
  - `recovery_target_action` : `pause`
  - `pg_wal_replay_resume` pour ouvrir immédiatement
  - ou modifier & redémarrer

Si l'on ne veut pas simplement restaurer tous les journaux, par exemple pour s'arrêter avant une fausse manipulation désastreuse, plusieurs paramètres permettent de préciser le point d'arrêt :

- jusqu'à un certain nom, grâce au paramètre `recovery_target_name` (le nom correspond à un label enregistré précédemment dans les journaux de transactions grâce à la fonction `pg_create_restore_point()`) ;
- jusqu'à une certaine heure, grâce au paramètre `recovery_target_time` ;
- jusqu'à un certain identifiant de transaction, grâce au paramètre `recovery_target_xid`, numéro de transaction qu'il est possible de chercher dans les journaux eux-mêmes grâce à l'utilitaire `pg_waldump` (voir cet article[10]) ;
- jusqu'à un certain LSN (*Log Sequence Number*[11]), grâce au paramètre `recovery_target_lsn`, que là aussi on doit aller chercher dans les journaux eux-mêmes.

Évidemment, il ne faudra choisir qu'un paramètre parmi ceux-là.

Avec le paramètre `recovery_target_inclusive` (par défaut à `true`), il est possible de préciser si la restauration se fait en incluant les transactions au nom, à l'heure ou à l'identifiant de transaction demandé, ou en les excluant.

Dans les cas complexes, nous verrons plus tard que choisir la *timeline* peut être utile (avec `recovery_target_timeline`, en général à `latest`).

Exemples de paramétrage :

```
recovery_target_name = 'label';
```

```
recovery_target_time = '2022-12-31 12:45:00 UTC'
```

```
recovery_target_lsn = '0/2000060'
```

```
recovery_target_xid = '1100842'
```

Ces restaurations à un moment précis ne sont possibles que si elles correspondent à un état cohérent d'**après** la fin du *base backup*, soit après le moment du `pg_stop_backup`. Si l'on a un historique de plusieurs sauvegardes, il faudra en choisir une antérieure au point de restauration voulu. Ce n'est pas forcément la dernière. Les outils ne sont pas forcément capables de deviner la bonne sauvegarde à restaurer.

Il est possible de demander à la restauration de s'arrêter une fois arrivée au stade voulu avec :

```
recovery_target_action = pause
```

C'est même l'action par défaut si une des options d'arrêt ci-dessus a été choisie : cela permet à l'utilisateur de vérifier que le serveur est bien arrivé au point qu'il désirait. Les alternatives sont `promote` (ouverture en écriture après le rejeu) et `shutdown`.

[10] https://blog.hagander.net/locating-the-recovery-point-just-before-a-dropped-table-230/
[11] https://docs.postgresql.fr/current/datatype-pg-lsn.html

Si la cible est atteinte mais que l'on décide de continuer la restauration jusqu'à un autre point (évidemment postérieur), il faut modifier la cible de restauration dans le fichier de configuration, et **redémarrer** PostgreSQL. C'est le seul moyen de rejouer d'autres journaux sans ouvrir l'instance en écriture.

Si l'on est arrivé au point de restauration voulu, un message de ce genre apparaît :

```
LOG:  recovery stopping before commit of transaction 8693270, time 2021-09-02
↪  11:46:35.394345+02
LOG:  pausing at the end of recovery
HINT:  Execute pg_wal_replay_resume() to promote.
```

(Le terme *promote* pour une restauration est un peu abusif.) `pg_wal_replay_resume()` — malgré ce que pourrait laisser croire son nom ! — provoque ici l'arrêt immédiat de la restauration, donc ignore les opérations contenues dans les WALs que l'on n'a pas souhaités restaurer, puis le serveur s'ouvre en écriture sur une nouvelle timeline.

Attention : jusque PostgreSQL 12 inclus, si un `recovery_target` était spécifié mais n'était toujours *pas* atteint à la fin du rejeu des archives, alors le mode *recovery* se terminait et le serveur était promu sans erreur, et ce, même si `recovery_target_action` avait la valeur `pause` ! (À condition, bien sûr, que le point de cohérence ait tout de même été dépassé.) Il faut donc être vigilant quant aux messages dans le fichier de trace de PostgreSQL !
À partir de PostgreSQL 13, l'instance détecte le problème et s'arrête avec un message `FATAL` : la restauration ne s'est pas déroulée comme attendu. S'il manque juste certains journaux de transactions, cela permet de relancer PostgreSQL après correction de l'oubli.

La documentation officielle complète sur le sujet est sur le site du projet[12].

7.8.7 Restaurer une sauvegarde PITR (5/5)

- Démarrer PostgreSQL
- Rejeu des journaux
- Vérifier que le point de cohérence est atteint !
- Ne jamais effacer `recovery.signal` volontairement

La dernière étape est particulièrement simple. Il suffit de démarrer PostgreSQL. PostgreSQL va comprendre qu'il doit rejouer les journaux de transactions.

[12] https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET

Les éventuels journaux présents sont rejoués, puis `restore_command` est appelé pour fournir d'autres journaux, jusqu'à ce que la commande ne trouve plus rien dans les archives.

Les journaux doivent se dérouler au moins jusqu'à rencontrer le « point de cohérence », c'est-à-dire la mention insérée par `pg_backup_stop()`. Avant ce point, il n'est pas possible de savoir si les fichiers issus du *base backup* sont à jour ou pas, et il est impossible de démarrer l'instance. Le message apparaît dans les traces et, dans le doute, on doit vérifier sa présence :

```
2020-01-17 16:08:37.285 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000031"…
2020-01-17 16:08:37.789 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000032"…
2020-01-17 16:08:37.949 UTC [15221] LOG: consistent recovery state reached
                                         at 1/32BFDD88
2020-01-17 16:08:37.949 UTC [15217] LOG: database system is ready to accept
                                         read only connections
2020-01-17 16:08:38.009 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000033"…
```

Si le message apparaît, le rejeu n'est pas terminé, mais on a au moins Au moment où ce message apparaît, le rejeu n'est pas terminé, mais il a atteint un stade où l'instance est cohérente et utilisable.

PostgreSQL continue ensuite jusqu'à arriver à la limite fixée, jusqu'à ce qu'il ne trouve plus de journal à rejouer (`restore_command` tombe en erreur), ou que le bloc de journal lu soit incohérent (ce qui indique qu'on est arrivé à la fin d'un journal qui n'a pas été terminé, le journal courant au moment du crash par exemple). PostgreSQL vérifie qu'il n'existe pas une *timeline* supérieure sur laquelle basculer (par exemple s'il s'agit de la deuxième restauration depuis la sauvegarde du PGDATA).

Puis il va s'ouvrir en écriture (sauf si vous avez demandé `recovery_target_action = pause`).

```
2020-01-17 16:08:45.938 UTC [15221] LOG: restored log file "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"…
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
```

Le fichier `recovery.signal` est effacé pour ne pas poser problème en cas de crash immédiat. (Ne l'effacez jamais manuellement !)

Le fichier `backup_label` d'une sauvegarde exclusive est renommé en `backup_label.old`.

7.8.8 Restauration PITR : durée

- Durée dépendante du nombre de journaux
  - rejeu séquentiel des WAL
- Accéléré en version 15 (*prefetch*)

La durée de la restauration est fortement dépendante du nombre de journaux. Ils sont rejoués séquentiellement. Mais avant cela, un fichier journal peut devoir être récupéré, décompressé, et restauré dans `pg_wal`.

Il est donc préférable qu'il n'y ait pas trop de journaux à rejouer, et donc qu'il n'y ait pas trop d'espaces entre sauvegardes complètes successives.

La version 15 a optimisé le rejeu en permettant l'activation du *prefetch* des blocs de données lors du rejeu des journaux.

Un outil comme pgBackRest en mode asynchrone permet de paralléliser la récupération des journaux, ce qui permet de les récupérer via le réseau et de les décompresser par avance pendant que PostgreSQL traite les journaux précédents.

7.8.9 Restauration PITR : différentes timelines

- Fin de *recovery* => changement de *timeline* :
  - l'historique des données prend une autre voie
  - le nom des WAL change (ex : `00000002000000000000000C`)
  - fichiers `.history`
- Permet plusieurs restaurations PITR à partir du même *basebackup*
- Choix : `recovery_target_timeline`
  - défaut : `latest`

Lorsque le mode *recovery* s'arrête, au point dans le temps demandé ou faute d'archives disponibles, l'instance accepte les écritures. De nouvelles transactions se produisent alors sur les différentes bases

de données de l'instance. Dans ce cas, l'historique des données prend un chemin différent par rapport aux archives de journaux de transactions produites avant la restauration. Par exemple, dans ce nouvel historique, il n'y a pas le `DROP TABLE` malencontreux qui a imposé de restaurer les données. Cependant, cette transaction existe bien dans les archives des journaux de transactions.

On a alors plusieurs historiques des transactions, avec des « bifurcations » aux moments où on a réalisé des restaurations. PostgreSQL permet de garder ces historiques grâce à la notion de *timeline*. Une *timeline* est donc l'un de ces historiques. Elle est identifiée par un numéro et se matérialise par un ensemble de journaux de transactions. Le numéro de la *timeline* est le premier nombre hexadécimal du nom des segments de journaux de transactions, en 8e position (le second est le numéro du journal, et le troisième, à la fin, le numéro du segment). Ainsi, lorsqu'une instance s'ouvre après une restauration PITR, elle peut archiver immédiatement ses journaux de transactions au même endroit, les fichiers ne seront pas écrasés vu qu'ils seront nommés différemment. Par exemple, après une restauration PITR s'arrêtant à un point situé dans le segment `000000010000000000000009` :

```
$ ls -1 /backup/postgresql/archived_wal/
000000010000000000000007
000000010000000000000008
000000010000000000000009
00000001000000000000000A
00000001000000000000000B
00000001000000000000000C
00000001000000000000000D
00000001000000000000000E
00000001000000000000000F
000000010000000000000010
000000010000000000000011
000000020000000000000009
00000002000000000000000A
00000002000000000000000B
00000002000000000000000C
00000002.history
```

Noter les timelines `1` et `2` en 8e position des noms des fichiers. Il y a deux fichiers finissant par `09` : le premier `000000010000000000000009` contient des informations communes aux deux *timelines* mais sa fin ne figure pas dans la *timeline* 2. Les fichiers `00000001000000000000000A` à `000000010000000000000011` contiennent des informations qui ne figurent que dans la *timeline* 1. Les fichiers `00000002000000000000000A` jusque `00000002000000000000000C` sont uniquement dans la *timeline* 2. Le fichier `00000002.history` contient l'information sur la transition entre les deux *timelines*.

Ce fichier sert pendant le *recovery*, quand l'instance doit choisir les *timelines* à suivre et les fichiers à restaurer. Les *timelines* connues avec leur point de départ sont suivies grâce aux fichiers d'historique, nommés d'après le numéro hexadécimal sur huit caractères de la *timeline* et le suffixe `.history`, et archivés avec les journaux. En partant de la *timeline* qu'elle quitte, l'instance restaure les fichiers historiques des *timelines* suivantes pour choisir la première disponible. Une fois la restauration finie, avant de s'ouvrir en écriture, l'instance archive un nouveau fichier `.history` pour la nouvelle *timeline* sélectionnée. Il contient l'adresse du point de départ dans la *timeline* qu'elle quitte, c'est-à-dire le point de bifurcation entre la 1 et la 2 :

```
$ cat 00000002.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
```

Puis l'instance continue normalement, et archive ses journaux commençant par `00000002`.

Après une seconde restauration, repartant de la *timeline* 2, l'instance choisit la *timeline* 3 et écrit un nouveau fichier :

```
$ cat 00000003.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
2   0/105AF7D0  before 2015-10-22 10:25:56.614316+02
```

Ce fichier reprend les *timelines* précédemment suivies par l'instance. En effet, l'enchaînement peut être complexe s'il y a eu plusieurs retours en arrière ou restauration.

À la restauration, on peut choisir la *timeline* cible en configurant le paramètre `recovery_target_timeline`. Il vaut par défaut `latest`, et la restauration suit donc les changements de *timeline* depuis le moment de la sauvegarde.

Pour choisir une autre *timeline* que la dernière, il faut donner le numéro de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. Les timelines permettent ainsi d'effectuer plusieurs restaurations successives à partir du même *base backup*, avec des retours en arrière, et d'archiver vers le même dépôt sans mélanger les journaux.

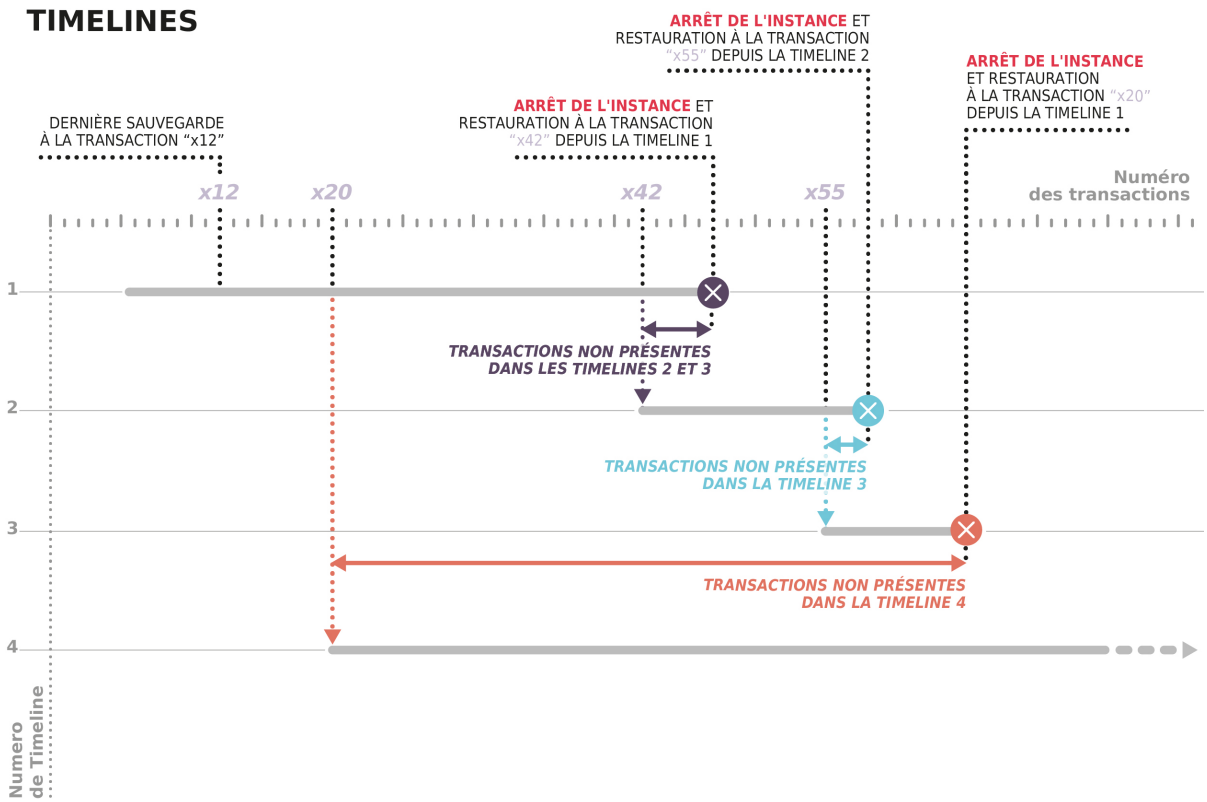
Bien sûr, pour restaurer dans une *timeline* précise, il faut que le fichier `.history` correspondant soit encore présent dans les archives, sous peine d'erreur.

Un changement de *timeline* ne se produit que lors d'une restauration explicite, et pas en cas de *recovery* après crash, notamment. (Cela arrive aussi quand un serveur secondaire est promu : il crée une nouvelle *timeline*.)

Il y a quelques pièges :

- Le numéro de timeline dans les traces, ou affiché par `pg_controldata`, est en décimal. Mais les fichiers `.history` portent un numéro en hexadécimal (par exemple `00000014.history` pour la timeline 20). On peut fournir les deux à `recovery_target_timeline` (`20` ou `'0x14'`). Attention, il n'y a pas de contrôle !
- **Attention sur les anciennes versions** : jusque PostgreSQL 11 compris, la valeur par défaut de `recovery_target_timeline` était `current` : la restauration se faisait donc dans la même *timeline* que le *base backup*. Si entre-temps il y avait eu une bascule ou une précédente restauration, la nouvelle timeline n'était pas automatiquement suivie !

7.8.10 Restauration PITR : illustration des timelines



Ce schéma illustre ce processus de plusieurs restaurations successives, et la création de différentes *timelines* qui en résulte.

On observe ici les éléments suivants avant la première restauration :

- la fin de la dernière sauvegarde se situe en haut à gauche sur l'axe des transactions, à la transaction `x12` ;
- cette sauvegarde a été effectuée alors que l'instance était en activité sur la *timeline* 1.

On décide d'arrêter l'instance alors qu'elle est arrivée à la transaction `x47`, par exemple parce qu'une nouvelle livraison de l'application a introduit un bug qui provoque des pertes de données. L'objectif est de restaurer l'instance avant l'apparition du problème afin de récupérer les données dans un état cohérent, et de relancer la production à partir de cet état. Pour cela, on restaure les fichiers de l'instance à partir de la dernière sauvegarde, puis on modifie le fichier de configuration pour que l'instance, lors de sa phase de *recovery* :

- restaure les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaure les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x42`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la

timeline 2, la bifurcation s'effectuant à la transaction `x42`. L'instance étant de nouveau ouverte en écriture, elle va générer de nouveaux WAL, qui seront associés à la nouvelle *timeline* : ils n'écrasent pas les fichiers WAL archivés de la *timeline* 1, ce qui permet de les réutiliser pour une autre restauration en cas de besoin (par exemple si la transaction `x42` utilisée comme point d'arrêt était trop loin dans le passé, et que l'on désire restaurer de nouveau jusqu'à un point plus récent).

Un peu plus tard, on a de nouveau besoin d'effectuer une restauration dans le passé - par exemple, une nouvelle livraison applicative a été effectuée, mais le bug rencontré précédemment n'était toujours pas corrigé. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, puis on configure PostgreSQL pour suivre la *timeline* 2 (paramètre `recovery_target_timeline = 2`) jusqu'à la transaction `x55`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point de la bifurcation (transaction `x42`) ;
- suivre la *timeline* indiquée (2) et rejouer les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x55`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 3, la bifurcation s'effectuant cette fois à la transaction `x55`.

Enfin, on se rend compte qu'un problème bien plus ancien et subtil a été introduit précédemment aux deux restaurations effectuées. On décide alors de restaurer l'instance jusqu'à un point dans le temps situé bien avant, jusqu'à la transaction `x20`. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, et on configure le serveur pour restaurer jusqu'à la transaction `x20`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point précédant immédiatement l' apparition du bug applicatif (transaction `x20`).

Comme la création des deux *timelines* précédentes est archivée dans les fichiers *history*, l'ouverture de l'instance en écriture va basculer sur une nouvelle *timeline* (4). Suite à cette restauration, toutes les modifications de données provoquées par des transactions effectuées sur la *timeline* 1 après la transaction `x20`, ainsi que celles effectuées sur les *timelines* 2 et 3, ne sont donc pas présentes dans l'instance.

7.8.11 Après la restauration

- Bien vérifier que l'archivage a repris
  - et que les archives des journaux sont complètes
- Ne pas hésiter à reprendre une sauvegarde complète
- Bien vérifier que les secondaires ont suivi

Une fois le nouveau primaire en place, la production peut reprendre, mais il faut vérifier que la sauvegarde PITR est elle aussi fonctionnelle.

Ce nouveau primaire a généralement commencé à archiver ses journaux à partir du dernier journal récupéré de l'ancien primaire, renommé avec l'extension `.partial`, juste avant la bascule sur la nouvelle *timeline*. Il faut bien sûr vérifier que l'archivage des nouveaux journaux fonctionne.

Sur l'ancien primaire, les derniers journaux générés juste avant l'incident n'ont pas forcément été archivés. Ceux-ci possèdent un fichier témoin `.ready` dans `pg_wal/archive_status`. Même s'ils ont été copiés manuellement vers le nouveau primaire avant sa promotion, celui-ci ne les a pas archivés.

Rappelons qu'un « trou » dans le flux des journaux dans le dépôt des archives empêchera la restauration d'aller au-delà de ce point !

Il est possible de forcer l'archivage des fichiers `.ready` depuis l'ancien primaire, avant la bascule, en exécutant à la main les `archive_command` que PostgreSQL aurait générées, mais la facilité pour le faire dépend de l'outil. La copie de journaux à la main est donc risquée.

De plus, s'il y a eu plusieurs restaurations successives, qui ont provoqué quelques archivages et des apparitions de *timelines* dans le même dépôt d'archives, avant d'être abandonnées, il faut faire attention au paramètre `recovery_target_timeline` (`latest` ne convient plus), ce qui complique une future restauration.

Pour faciliter des restaurations ultérieures, il est recommandé de procéder au plus tôt à une sauvegarde complète du nouveau primaire.

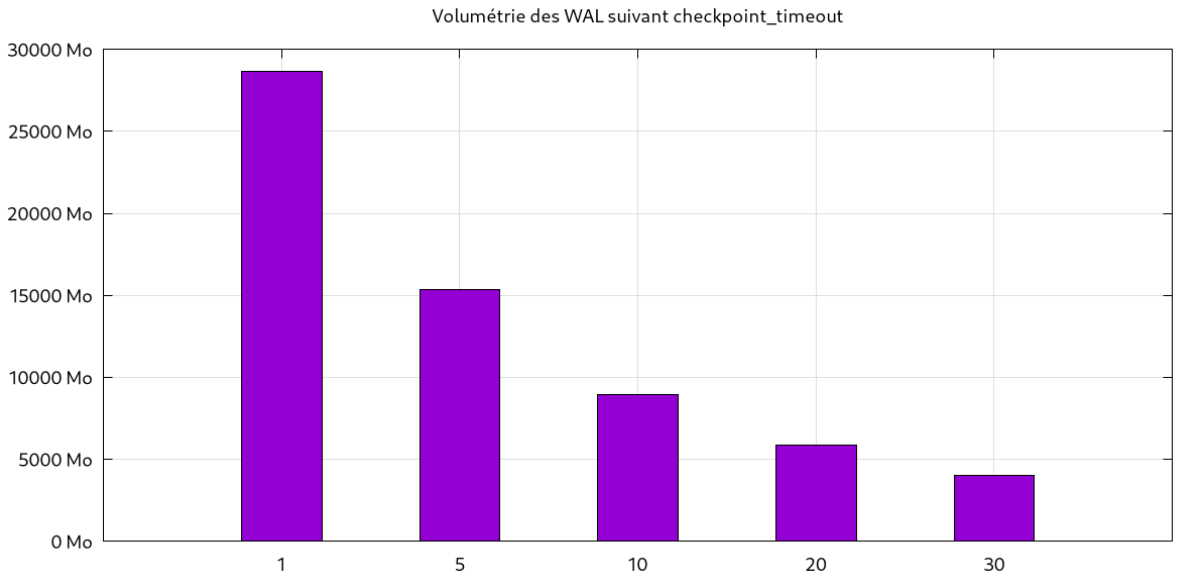
Quant aux éventuelles instances secondaires, il est vraisemblable qu'elles doivent être reconstruites suite à la restauration de l'instance primaire. (Si elles ont appliqué des journaux qui ont été perdus et

n'ont pas été repris par le primaire restauré, ces secondaires ne pourront se raccrocher. Consulter les traces.)

7.9 POUR ALLER PLUS LOIN

- Limiter la volumétrie des journaux sauvegardés
- Quels sont les outils PITR ?

7.9.1 Réduire le nombre de journaux sauvegardés



- Monter
  - `checkpoint_timeout`
  - `max_wal_size`

L'un des problèmes de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Si un journal de 16 Mo (par défaut) est généré toutes les minutes, le total est de 23 Go de journaux par jour, et parfois beaucoup plus. Il n'est pas forcément possible de conserver autant de journaux.

Un premier moyen est de reduire la volumétrie à la source en espaçant les checkpoints. Le graphique ci-dessus représente la volumétrie générée par un simple test avec `pgbench` (OLTP classique donc) avec `checkpoint_timeout` variant entre 1 et 30 minutes : les écarts sont énormes.

La raison est que, pour des raisons de fiabilité, un bloc modifié est intégralement écrit (8 ko) dans les journaux à sa première modification après un checkpoint. Par la suite, seules les modifications de ce bloc, souvent beaucoup plus petites, sont journalisées. (Ce comportement dépend du paramètre `full_page_writes`[13], activé par défaut et qu'il est impératif de laisser tel quel, sauf peut-être sur ZFS.)

Espacer les checkpoints permet d'économiser des écritures de blocs complets, si l'activité s'y prête (en OLTP surtout). Il y a un intérêt en performances, mais surtout en place disque économisée quand les journaux sont archivés, aussi accessoirement en CPU s'ils sont compressés, et en trafic réseau s'ils sont répliqués. Un exemple figure dans ce billet du blog Dalibo[14].

Par cohérence, si l'on monte `checkpoint_timeout`, il faut penser à augmenter aussi `max_wal_size`, et vice-versa. Des valeurs courantes sont respectivement 15 minutes, parfois plus, et plusieurs gigaoctets.

Il y a cependant un inconvénient : un écart plus grand entre checkpoints peut allonger la restauration après un arrêt brutal, car il y aura plus de journaux à rejouer, parfois des centaines ou des milliers.

7.9.2 Compresser les journaux de transactions

- `wal_compression = on`
  - moins de journaux
  - un peu plus de CPU
  - à activer : `pglz` (`on`), `lz4`, `zstd` (v15+)
- Outils de compression standards : `gzip`, `bzip2`, `lzma`…
  - attention à ne pas ralentir l'archivage

PostgreSQL peut compresser les journaux à la source, si le paramètre `wal_compression` (désactivé par défaut) est passé à `on`. La compression est opérée par PostgreSQL au niveau de la page, avec un gros gain en volumétrie (souvent plus de 50 % !). Les journaux font toujours 16 Mo (par défaut), mais comme ils sont moins nombreux, leur rejeu est plus rapide, ce qui accélère la réplication et la reprise après un crash. Cette compression des journaux est totalement transparente pour l'archivage ou la restauration. Le prix est une augmentation de la consommation en CPU, souvent négligeable.

Depuis PostgreSQL 15, on peut même choisir l'algorithme de compression : `pglz`, `lz4` ou `zstd`. `on` est le synonyme de `pglz`… qui est sans doute le moins bon des trois (voir ce petit test[15]), surtout en terme de consommation CPU.

[13] https://wiki.postgresql.org/wiki/Full_page_writes
[14] https://blog.dalibo.com/2024/01/05/cambouis.html
[15] https://www.postgresql.org/message-id/YMmlvyVyAFlxZ%2B/H%40paquier.xyz

Une autre solution est la compression à la volée des journaux archivés dans l'`archive_command`. Les outils classiques comme `gzip`, `bzip2`, `lzma`, `xz`, etc. conviennent. Tous les outils PITR incluent plusieurs de ces algorithmes. Un fichier de 16 Mo aura généralement une taille compressée comprise entre 3 et 6 Mo.

Cependant, attention au temps de compression des journaux : en cas d'écritures lourdes, une compression élevée mais lente peut mener à un retard conséquent de l'archivage par rapport à l'écriture des journaux, jusque saturation de `pg_wal`, et arrêt de l'instance. Il est courant de se contenter de `gzip -1` ou `lz4 -1` pour les journaux, et de ne compresser agressivement que les sauvegardes des fichiers de la base.

7.9.3 Outils de sauvegarde PITR dédiés

- Se faciliter la vie avec différents outils
  - pgBackRest
  - barman
- Fournissent :
  - un outil pour les backups, les purges…
  - une commande pour l'archivage

Il n'est pas conseillé de réinventer la roue et d'écrire soi-même des scripts de sauvegarde, qui doivent prévoir de nombreux cas et bien gérer les erreurs. La sauvegarde concurrente est également difficile à manier. Des outils reconnus existent, dont nous évoquerons brièvement les plus connus. Il en existe d'autres. Ils ne font pas partie du projet PostgreSQL à proprement parler et doivent être installés séparément.

Les outils décrits succinctement plus bas fournissent :

- un outil pour procéder aux sauvegardes, gérer la péremption des archives… ;
- un outil qui sera appelé par `archive_command`.

Leur philosophie peut différer, notamment en terme de centralisation ou de compromis entre simplicité et fonctionnalités. Ces dernières s'enrichissent d'ailleurs au fil du temps.

Voir https://dali.bo/i4_html pour une description plus complète.

7.9.4 pgBackRest

- Gère la sauvegarde et la restauration
  - *pull* ou *push*, multidépôts
  - mono- ou multiserveur
- Indépendant des commandes système
  - protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multithread, sauvegarde depuis un secondaire, archivage asynchrone…
- Projet mature

pgBackRest[16] est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multithread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de `pg_wal` ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser, ou avoir plusieurs niveaux d'archives) ;
- le support intégré de dépôts S3 ou Azure ;
- le support d'un accès TLS géré par pgBackRest en alternative à SSH ;
- la sauvegarde depuis un serveur secondaire ;
- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé ;
- la reprise d'une sauvegarde échouée.

pgBackRest n'utilise pas `pg_receivewal` pour garantir la sauvegarde du dernier journal (non terminé) avant un sinistre. Les auteurs considèrent que dans ce cas un secondaire synchrone est plus adapté et plus fiable.

Le projet est très actif et considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

[16] https://pgbackrest.org/

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest[17].

7.9.5 barman

- Gère la sauvegarde et la restauration
  - mode *pull*
  - multiserveurs
- Une seule commande (`barman`)
- Et de nombreuses actions
  - `list-server`, `backup`, `list-backup`, `recover`...
- Spécificité : gestion de `pg_receivewal`

barman est un outil créé par 2ndQuadrant (racheté depuis par EDB). Il a pour but de faciliter la mise en place de sauvegardes PITR. Il gère à la fois la sauvegarde et la restauration.

La commande `barman` dispose de plusieurs actions :

- `list-server`, pour connaître la liste des serveurs configurés ;
- `backup`, pour lancer une sauvegarde de base ;
- `list-backup`, pour connaître la liste des sauvegardes de base ;
- `show-backup`, pour afficher des informations sur une sauvegarde ;
- `delete`, pour supprimer une sauvegarde ;
- `recover`, pour restaurer une sauvegarde (la restauration peut se faire à distance).

Contrairement aux autre outils présentés ici, barman permet d'utiliser `pg_receivewal`.

Il supporte aussi les dépôts S3 ou blob Azure.

Site web de barman[18]

[17] https://github.com/pgstef/check_pgbackrest/
[18] https://www.pgbarman.org/

7.10 CONCLUSION

- Une sauvegarde
  - fiable
  - éprouvée
  - rapide
  - continue
- Mais
  - plus complexe à mettre en place que `pg_dump`
  - qui restaure toute l'instance

Cette méthode de sauvegarde est la seule utilisable dès que les besoins de performance de sauvegarde et de restauration augmentent (*Recovery Time Objective* ou RTO), ou que le volume de perte de données doit être drastiquement réduit (*Recovery Point Objective* ou RPO).

7.10.1 Questions

N'hésitez pas, c'est le moment !

7.11 QUIZ

https://dali.bo/i2_quiz

7.12 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/i2_solutions.

7.12.1 pg_basebackup : sauvegarde ponctuelle & restauration

But : Créer une sauvegarde physique à chaud à un moment précis de la base avec `pg_basebackup`, et la restaurer.

Configurer la réplication dans `postgresql.conf` et `pg_hba.conf` :

- désactiver l'archivage s'il est actif
- autoriser des connexions de réplication en streaming en local.

Pour insérer des données :

- générer de l'activité avec `pgbench` en tant qu'utilisateur **postgres** :

```
$ createdb bench
$ /usr/pgsql-17/bin/pgbench -i -s 100 bench
$ /usr/pgsql-17/bin/pgbench bench -n -P 3 -R 50 -T 1800
```

- laisser tourner en arrière-plan
- surveiller l'évolution de l'activité sur la table `pgbench_history`, par exemple ainsi :

```
$ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
```

En parallèle, sauvegarder l'instance avec :

- `pg_basebackup` au format tar, compressé avec gzip ;
- sans oublier les journaux ;
- avec l'option `--max-rate=30M` pour ralentir la sauvegarde ;
- le répertoire de sauvegarde sera `/var/lib/pgsql/17/backups/basebackup` ;
- surveillez la progression dans une autre session avec la vue système adéquate.

Une fois la sauvegarde terminée :

- arrêter la session `pgbench` ;
- regarder les fichiers générés ; Afficher la date de dernière modification dans `pgbench_history`.

- Arrêter l'instance.
- Faire une copie à froid des données (par exemple avec `cp -rfp`) vers `/var/lib/pgsql/17/data.old` (cette copie resservira plus tard).

- Vider le répertoire des données.
- Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.
- Redémarrer l'instance.

Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

Quelle est la dernière donnée restaurée ?

Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction. Que se passe-t-il ?

7.12.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants

But : Coupler une sauvegarde à chaud avec `pg_basebackup` et l'archivage

Si le TP précédent a été déroulé et que l'instance n'est pas fonctionnelle, remettre en place la copie à froid de l'instance depuis `/var/lib/pgsql/17/data.old`.

Configurer l'archivage vers un répertoire `/archives`, par exemple avec `rsync`. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

Générer à nouveau de l'activité avec `pgbench` : en tant qu'utilisateur **postgres** :

```
$ createdb bench                # si pas déjà fait précédemment
$ /usr/pgsql-17/bin/pgbench -i -s 100 bench             # idem
$ /usr/pgsql-17/bin/pgbench bench -n -P 3 -R 50 -T 1800
```

Vérifier que l'archivage fonctionne dans le répertoire `/archives`, dans les traces et dans la vue `pg_stat_archiver`.

En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup` au format plain.

Utiliser `pg_verify_backup` pour contrôler l'intégrité de la sauvegarde.

À quoi correspond le fichier finissant par `.backup` dans les archives ?

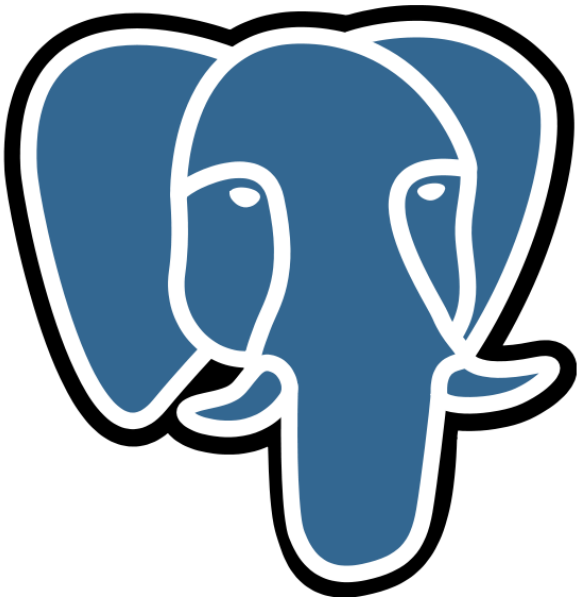
Arrêter `pgbench` et noter la date des dernières données insérées.

Effacer le PGDATA. Restaurer la sauvegarde précédente *sans* les journaux. Configurer la `restore_command`. Créer le fichier `recovery.signal`. Démarrer PostgreSQL.

Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

Vérifier quelles données ont été restaurées.

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[1]

8.1.1 Au menu

- Anticiper les désastres
- Réagir aux désastres
- Rechercher l'origine du problème
- Outils utiles
- Cas type de désastres

[1] http://thebuild.com/presentations/worst-day-fosdem-2014.pdf

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.

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 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
  - Patroni, PAF (Pacemaker)
- Automatiser une bascule est complexe
  - cela peut mener à davantage d'incidents
  - voire à des désastres (*split brain*)

Si on poursuit jusqu'au bout le raisonnement précédent sur le risque à faire effectuer de nombreuses opérations manuelles lors d'un incident, la conclusion logique est que la solution idéale serait de les éliminer complètement, et d'automatiser complètement le déclenchement et l'exécution de la procédure.

Un problème est que toute solution visant à automatiser une tâche se base sur un nombre limité de paramètres et sur une vision restreinte de l'architecture.

De plus, il est difficile à un outil de bascule automatique de diagnostiquer correctement certains types d'incident, par exemple une partition réseau. L'outil peut donc détecter à tort à un incident, surtout s'il est réglé de façon à être assez sensible, et ainsi provoquer lui-même une coupure de service inutile.

Dans le pire des cas, l'outil peut être amené à prendre une mauvaise décision amenant à une situation de désastre, comme un *split brain* (deux instances PostgreSQL se retrouvent ouvertes en écriture en même temps sur les mêmes données).

Il est donc fortement préférable de laisser un administrateur prendre les décisions potentiellement dangereuses, comme une bascule ou une restauration.

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[2].

[2] 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[3].

Notamment (liste non exhaustive) :

- indiquer la version précise de PostgreSQL installée, et la méthode d'installation utilisée ;
- préciser la plate-forme utilisée, notamment la version du système d'exploitation utilisé et la configuration des ressources du serveur ;
- signaler uniquement les faits observés, éviter les spéculations sur l'origine du problème ;
- joindre le détail des messages d'erreurs observés (augmenter la verbosité des erreurs avec le paramètre `log_error_verbosity`) ;
- joindre un cas complet permettant de reproduire le problème de façon aussi simple que possible.

Pour les problèmes relevant du domaine de la sécurité (découverte d'une faille), la liste adéquate est security@postgresql.org.

[3] https://www.postgresql.org/docs/current/static/bug-reporting.html

8.3.9 Bons réflexes 8

- Après correction
- Tester complètement l'intégrité des données
  - pour détecter tous les problèmes
- Validation avec export logique complet

```
pg_dumpall > /dev/null
```

- ou sauvegarde physique
  - `pg_basebackup`…
- Reconstruction dans une autre instance (vérification de cohérence)

```
pg_dumpall | psql -h autre serveur
```

Une fois les actions correctives réalisées (restauration, recréation d'objets, mise à jour des données…), il faut tester intensivement pour s'assurer que le problème est bien complètement résolu.

Il est donc extrêmement important d'avoir préparé des cas de tests permettant de reproduire le problème de façon certaine, afin de valider la solution appliquée.

En cas de suspicion de corruption de données, il est également important de tenter de procéder à la lecture de la totalité des données depuis PostgreSQL.

Un premier outil pour cela est une sauvegarde physique avec `pg_basebackup` ou un autre outil (voir plus loin).

Alternativement, la commande suivante, exécutée avec l'utilisateur système propriétaire de l'instance (généralement `postgres`) effectue une lecture complète de toutes les tables (et uniquement les tables), sans nécessiter de place sur disque supplémentaire :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell, la commande est :

```
PS C:\ pg_dumpall > $null
```

Cette commande ne devrait renvoyer aucune erreur. En cas de problème, notamment une somme de contrôle qui échoue, une erreur apparaîtra :

```
pg_dump: WARNING:  page verification failed, calculated checksum 20565 but expected
 ↪  17796
pg_dump: erreur : Sauvegarde du contenu de la table « corrompue » échouée :
                  échec de PQgetResult().
pg_dump: erreur : Message d'erreur du serveur :
```

```
        ERROR:  invalid page in block 0 of relation base/104818/104828
pg_dump: erreur : La commande était : COPY public.corrompue (i) TO stdout;
pg_dumpall: erreur : échec de pg_dump sur la base de données « corruption », quitte
```

Attention, les fichiers associés aux index ou aux vues matérialisées ne sont pas parcourus lors de la lecture des données par `pg_dumpall` ou `pg_dump`. Une corruption physique d'index ne sera donc pas remontée. Une vérification exhaustive implique d'autres outils comme `pg_checksums` ou `pg_basebackup`.
De plus, l'absence d'erreur physique ne garantit en aucun cas l'intégrité fonctionnelle des données : les corruptions peuvent très bien être logiques.

Dans les situations les plus extrêmes (problème de stockage, fichiers corrompus), il est important de tester la validité des données dans une nouvelle instance en effectuant un export/import complet des données.

Par exemple, initialiser une nouvelle instance avec `initdb`, sur un autre système de stockage, voire sur un autre serveur, puis lancer la commande suivante (l'application doit être coupée, ce qui est normalement le cas depuis la détection de l'incident si les conseils précédents ont été suivis) pour exporter et importer à la volée :

```
$ pg_dumpall -h <serveur_corrompu> -U postgres | psql -h <nouveau_serveur> \
                                                  -U postgres postgres
$ vacuumdb --analyze -h <nouveau_serveur> -U postgres postgres
```

D'éventuels problèmes peuvent être détectés lors de l'import des données, par exemple si des corruptions entraînent l'échec de la reconstruction de clés étrangères. Il faut alors procéder au cas par cas.

Enfin, même si cette étape s'est déroulée sans erreur, tout risque n'est pas écarté, il reste la possibilité de corruption de données silencieuses. Sauf si la fonctionnalité de checksum de PostgreSQL a été activée sur l'instance (ce n'est pas activé par défaut !), le seul moyen de détecter ce type de problème est de valider les données fonctionnellement.

Dans tous les cas, en cas de suspicion de corruption de données en profondeur, il est fortement préférable d'accepter une perte de données et de restaurer une sauvegarde d'avant le début de l'incident, plutôt que de continuer à travailler avec des données dont l'intégrité n'est pas assurée.

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, par exemple :
  - `pg_resetwal`
  - l'extension `pg_surgery`
  - DANGER, dernier espoir

Quelle que soit la criticité du problème rencontré, la panique peut en faire quelque chose de pire.

Il faut impérativement garder son calme, et résister au mieux au stress et aux pressions qu'une situation de désastre ne manque pas de provoquer.

Il est également préférable d'éviter de sauter immédiatement à la conclusion la plus évidente. Il ne faut pas hésiter à retirer les mains du clavier pour prendre de la distance par rapport aux conséquences du problème, réfléchir aux causes possibles, prendre le temps d'aller chercher de l'information pour réévaluer l'ampleur réelle du problème.

La plus mauvaise décision que l'on peut être amenée à prendre lors de la gestion d'un incident est celle que l'on prend dans la précipitation, sans avoir bien réfléchi et mesuré son impact. Cela peut provoquer des dégâts irrécupérables, et transformer une situation d'incident en situation de crise majeure.

Un exemple classique de ce type de comportement est le cas où PostgreSQL est arrêté suite au remplissage du système de fichiers contenant les fichiers WAL, `pg_wal`.

Le réflexe immédiat d'un administrateur non averti pourrait être de supprimer les plus vieux fichiers dans ce répertoire, ce qui répond bien aux symptômes observés mais reste une erreur dramatique qui va rendre le démarrage de l'instance impossible.

Quoi qu'il arrive, ne jamais exécuter une commande sans être certain qu'elle correspond bien à la situation rencontrée, et sans en maîtriser complètement les impacts. Même si cette commande provient d'un document mentionnant les mêmes messages d'erreur que ceux rencontrés (et tout particulièrement si le document a été trouvé via une recherche hâtive sur Internet) !

Là encore, nous disposons comme exemple d'une erreur malheureusement fréquente, l'exécution de la commande `pg_resetwal` sur une instance rencontrant un problème. Comme l'indique la documentation, « cette commande ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption_ » et « *il ne faut pas perdre de vue que la base de données peut contenir des données incohérentes du fait de transactions partiellement validées* » (documentation[4]). Nous

[4] https://docs.postgresql.fr/current/app-pgresetwal.html

reviendrons ultérieurement sur les (rares) cas d'usage réels de cette commande, mais dans l'immense majorité des cas, l'utiliser va aggraver le problème, en ajoutant des problématiques de corruption logique des données !

Il convient donc de bien s'assurer de comprendre les conséquences de l'exécution de chaque action effectuée.

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 (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 surallocation de la mémoire, empêchant ainsi le déclenchement de ce phénomène.

Voir pour cela les paramètres kernel `vm.overcommit_memory` et `vm.overcommit_ratio` (référence : https://kb.dalibo.com/overcommit_memory).

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 : écriture des fichiers

- La désactivation de certains paramètres est dangereuse
  - `fsync`
  - `full_page_write`

Si des corruptions de données sont relevées suite à un crash de l'instance, il convient particulièrement de vérifier la valeur du paramètre `fsync`.

En effet, si celui-ci est désactivé, les écritures dans les journaux de transactions ne sont pas effectuées de façon synchrone, ce qui implique que l'ordre des écritures ne sera pas conservé en cas de crash. Le processus de *recovery* de PostgreSQL risque alors de provoquer des corruptions si l'instance est malgré tout redémarrée.

Ce paramètre ne devrait jamais être positionné à une autre valeur que `on`, sauf dans des cas extrêmement particuliers (en bref, si l'on peut se permettre de restaurer intégralement les données en cas de crash, par exemple dans un chargement de données initial).

Le paramètre `full_page_write` indique à PostgreSQL d'effectuer une écriture complète d'une page chaque fois qu'elle reçoit une nouvelle écriture après un checkpoint, pour éviter un éventuel mélange entre des anciennes et nouvelles données en cas d'écriture partielle.

La désactivation de `full_page_write` peut avoir le même type de conséquences catastrophiques que celle de `fsync` !

8.4.10 Paramétrage de PostgreSQL : les sommes de contrôle

- Activez les checksums !
  - pas (encore ?) par défaut
  - `initdb --data-checksums` (création)
  - `pg_checksums --enable` (à posteriori)
- Détecte les corruptions silencieuses
- Impact faible sur les performances
- Vérification :
  - `pg_checksums --check`
  - outils de sauvegarde et lecture

PostgreSQL ne verrouille pas tous les fichiers dès son ouverture. Sans mécanisme de sécurité, il est donc possible de modifier un fichier sans que PostgreSQL s'en rende compte, ce qui aboutit à une corruption silencieuse.

Les sommes de contrôles (*checksums*) permettent de se prémunir contre des corruptions silencieuses de données. Hélas, elles ne sont pas actives par défaut (cela pourrait bientôt changer). Leur mise en place est donc fortement recommandée sur une nouvelle instance. L'activation se fait différemment suivant les environnements (voir les procédures pour les paquets RPM et Debian du PGDG[5] mais revient toujours à transmettre le paramètre `--data-checksums` à `initdb`.

Il est possible de mettre en place les *checksums* sur une instance existante avec l'utilitaire `pg_checksums`[6]. Les gros inconvénients d'une activation tardive sont :

- tous les fichiers de données et d'index seront réécrits (puisqu'il faudra calculer le *checksum* de chaque bloc, écrit dans le bloc même) ;
- l'instance doit être arrêtée pendant toute l'opération.

À titre d'exemple, créons une instance sans utiliser les *checksums*, et une autre qui les utilisera :

```
$ initdb -D /tmp/sans_checksums/
$ initdb -D /tmp/avec_checksums/ --data-checksums
```

Insérons une valeur de test, sur chacun des deux clusters :

```
CREATE TABLE test (name text);
INSERT INTO test (name) VALUES ('toto');
```

On récupère le chemin du fichier de la table pour aller le corrompre à la main (seul celui sans *checksums* est montré en exemple).

[5] https://dali.bo/b_html#installation-de-postgresql-depuis-les-paquets-communautaires
[6] https://docs.postgresql.fr/current/app-pgchecksums.html

```
SELECT pg_relation_filepath('test');

 pg_relation_filepath
----------------------
 base/12036/16317
```

Instance arrêtée (pour ne pas être gêné par le cache), on va s'attacher à corrompre ce fichier, en remplaçant la valeur « toto » par « goto » avec un éditeur hexadécimal :

```
$ hexedit /tmp/sans_checksums/base/12036/16317
$ hexedit /tmp/avec_checksums/base/12036/16399
```

(On remarquera au passage que la ligne est à la fin du bloc de 8 ko.) Enfin, on peut ensuite exécuter des requêtes sur ces deux clusters.

Sans *checksums* :

```
TABLE test;

 name
------
 qoto
```

Avec *checksums* :

```
TABLE test;

WARNING:  page verification failed, calculated checksum 16321
          but expected 21348
ERROR:  invalid page in block 0 of relation base/12036/16387
```

Les sommes de contrôle sont vérifiées lors des lectures par les requêtes, mais aussi lors des sauvegardes, notamment lors d'un appel à `pg_dump` (données de la base concernée uniquement), `pg_basebackup`, `pgbackrest`… En cas de corruption des données, l'opération sera interrompue. `pg_checksums --check` peut vérifier complètement une instance arrêtée.

En pratique, avec PostgreSQL 9.5 au moins et des processeurs supportant les instructions SSE 4.2 (voir dans `/proc/cpuinfo`, mais cela concerne au moins tous les processeurs Intel depuis 15 ans), il n'y aura pas d'impact notable en performances. Par contre, il y a aura un peu plus de journaux générés, pour des raisons techniques.

L'activation ou non des sommes de contrôle peut se faire indépendamment sur un serveur primaire et son secondaire, mais il est fortement conseillé de les activer simultanément des deux côtés pour éviter de gros problèmes dans certains scénarios de restauration.

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` est livré avec PostgreSQL. Il lit les informations du fichier de contrôle d'une instance PostgreSQL. Cet outil ne se connecte pas à l'instance, il a juste besoin d'avoir un accès en lecture sur le répertoire `PGDATA` de l'instance.

Les informations qu'il récupère ne sont donc pas du temps réel, il s'agit d'une vision de l'instance telle qu'elle était la dernière fois que le fichier de contrôle a été mis à jour. L'avantage est qu'elle peut être utilisée même si l'instance est arrêtée.

`pg_controldata` affiche notamment les informations initialisées lors d'`initdb`, telles que la version du catalogue, ou la taille des blocs, qui peuvent être cruciales si l'on veut restaurer une instance sur un nouveau serveur à partir d'une copie des fichiers.

Il affiche également de nombreuses informations utiles sur le traitement des journaux de transactions et des checkpoints, par exemple :

- positions de l'avant-dernier checkpoint et du dernier checkpoint dans les WAL ;
- nom du WAL correspondant au dernier WAL ;
- timeline sur laquelle se situe le dernier checkpoint ;
- instant précis du dernier checkpoint.

Quelques informations de paramétrage sont également renvoyées, comme la configuration du niveau de WAL, ou le nombre maximal de connexions autorisées.

En complément, le dernier état connu de l'instance est également affiché. Les états potentiels sont :

- `in production` : l'instance est démarrée et est ouverte en écriture ;
- `shut down` : l'instance est arrêtée ;
- `in archive recovery` : l'instance est démarrée et est en mode `recovery` (restauration, `Warm` ou `Hot Standby`) ;
- `shut down in recovery` : l'instance s'est arrêtée alors qu'elle était en mode `recovery` ;
- `shutting down` : état transitoire, l'instance est en cours d'arrêt ;
- `in crash recovery` : état transitoire, l'instance est en cours de démarrage suite à un crash ;
- `starting up` : état transitoire, concrètement jamais utilisé.

Bien entendu, comme ces informations ne sont pas mises à jour en temps réel, elles peuvent être erronées.

Cet asynchronisme est intéressant pour diagnostiquer un problème, par exemple si `pg_controldata` renvoie l'état `in production` mais que l'instance est arrêtée, cela signifie que l'arrêt n'a pas été effectué proprement (*crash* de l'instance, qui sera donc suivi d'un `recovery` au démarrage).

Exemple de sortie de la commande :

```
$ /usr/pgsql-10/bin/pg_controldata /var/lib/pgsql/10/data

pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6451139765284827825
Database cluster state:               in production
pg_control last modified:             Mon 28 Aug 2017 03:40:30 PM CEST
Latest checkpoint location:           1/2B04EC0
Prior checkpoint location:            1/2B04DE8
Latest checkpoint's REDO location:    1/2B04E88
Latest checkpoint's REDO WAL file:    000000010000000100000002
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:1023
Latest checkpoint's NextOID:          41064
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        548
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  1022
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Mon 28 Aug 2017 03:40:30 PM CEST
Fake LSN counter for unlogged rels:   0/1
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
```

```
wal_level setting:                    replica
wal_log_hints setting:                off
max_connections setting:              100
max_worker_processes setting:         8
max_prepared_xacts setting:           0
max_locks_per_xact setting:           64
track_commit_timestamp setting:       off
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Data page checksum version:           0
Mock authentication nonce:            7fb23aca2465c69b2c0f54ccf03e0ece
                                      3c0933c5f0e5f2c096516099c9688173
```

8.5.2 Outils pour l'export/import de données

- `pg_dump`
- `pg_dumpall`
- `COPY`
- `psql` / `pg_restore`
  - `--section=pre-data` / `data` / `post-data`

Les outils `pg_dump` et `pg_dumpall` permettent d'exporter des données à partir d'une instance démarrée. Dans le cadre d'un incident grave, il est possible de les utiliser pour :

- extraire le contenu de l'instance suspecte ;
- extraire le contenu des bases de données ;
- tester si les données sont lisibles dans un format compréhensible par PostgreSQL.

Par exemple, un moyen rapide de s'assurer que tous les fichiers des tables de l'instance sont lisibles est de forcer leur lecture complète, notamment grâce à la commande suivante :

```
$ pg_dumpall > /dev/null
```

Sous Windows Powershell :

```
pg_dumpall > $null
```

Rappelons que les fichiers associés aux index et vues matérialisées ne sont pas parcourus pendant cette opération.

Si `pg_dumpall` ou `pg_dump` renvoient des messages d'erreur et ne parviennent pas à exporter certaines tables, il est possible de contourner le problème à l'aide de la commande `COPY`, en sélectionnant exclusivement les données lisibles autour du bloc corrompu. Exemples :

```
-- si accès par index possible
COPY (SELECT * FROM pgbench_accounts WHERE aid BETWEEN 50000 AND 60000)
TO 'backuppartiel.dmp';
-- accès par références de lignes
COPY (SELECT * FROM pgbench_accounts WHERE ctid BETWEEN '(0,1)' AND '(1639,20)')
TO 'backuppartiel.dmp';
```

Il convient ensuite d'utiliser `psql` ou `pg_restore` pour importer les données dans une nouvelle instance, probablement sur un nouveau serveur, dans un environnement non affecté par le problème. Pour parer au cas où le réimport échoue à cause de contraintes non respectées, il est souvent préférable de faire le réimport par étapes :

```
$ pg_restore -1 --section=pre-data --verbose -d cible base.dump
$ pg_restore -1 --section=data --verbose -d cible base.dump
$ pg_restore -1 --section=post-data --exit-on-error --verbose -d cible base.dump
```

En cas de problème, on verra les contraintes posant problème.

Il peut être utile de générer les scripts en pur SQL avant de les appliquer, éventuellement par étape :

```
$ pg_restore --section=post-data -f postdata.sql base.dump
```

Pour rappel, même après un export/import de données réalisé avec succès, des corruptions logiques peuvent encore être présentes. Il faut donc être particulièrement vigilant et prendre le temps de valider l'intégrité fonctionnelle des données.

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      | \x010000000100000001000000001000000
-[ 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      | \x020000000100000001000000002000000
```

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) ;
- ensuite de procéder à une validation méthodique des données.

Il est probable que certaines données incohérentes puissent être identifiées à l'import, lors de la phase de recréation des contraintes : celles-ci échoueront si les données ne les respectent pas, ce qui permettra de les identifier.

En ce qui concerne les incohérences qui passeront au travers de ces tests, il faudra les trouver et les corriger manuellement, en procédant à une validation fonctionnelle des données.

Il faut donc bien retenir les points suivants :

- `pg_resetwal` n'est pas magique ;
- `pg_resetwal` rend les données incohérentes (ce qui est souvent pire qu'une simple perte d'une partie des données, comme on aurait eu en restaurant une sauvegarde) ;
- `pg_resetwal` n'est à utiliser qu'en dernier recours s'il n'y a aucun autre moyen de faire autrement pour récupérer les données ;
- il ne faut pas l'utiliser sur l'instance ayant subi le problème, mais sur une copie complète effectuée à froid ;
- il permet au mieux d'exporter les données pour les importer dans une nouvelle instance ;
- la validation soigneuse des données de la nouvelle instance est capitale.

8.5.5 Outils : pg_surgery

- Extension (v14+)
- Collection de fonctions permettant de modifier le statut des tuples d'une relation
- **Extrêmement dangereuse**

À partir de PostgreSQL 14, cette extension regroupe des fonctions qui permettent de modifier le statut d'un tuple dans une relation. Il est par exemple possible de rendre une ligne morte ou de rendre visible des tuples qui sont invisibles à cause des informations de visibilité.

Ces fonctions sont dangereuses et peuvent provoquer ou aggraver des corruptions. Elles peuvent par exemple rendre une table incohérente par rapport à ses index, ou provoquer une violation de contrainte d'unicité ou de clé étrangère. Il ne faut donc les utiliser qu'en dernier recours, sur une copie de votre instance.

Référence : Documentation officielle[7]

8.5.6 Outils pour vérifier les sommes de contrôle

- Base arrêtée :
  - `pg_checksums --check`
- À la lecture
  - dont `pg_dump`, `pg_dumpall`
- Lors d'une sauvegarde physique :
  - `pg_basebackup`, voire `pg_basebackup --target=blackhole`
  - pgBackRest, barman (selon configuration)

Une fois les sommes de contrôle en place, elles sont revérifiées à chaque lecture du bloc, mais il n'y a pas d'automatisme pour une vérification globale.

pg_checksums :

`pg_checksums --check` est la commande dédiée pour vérifier les sommes de contrôles existantes sur les bases de données **à froid** : l'instance doit être arrêtée proprement auparavant.

Par exemple, suite à une modification de deux blocs dans une table avec l'outil `hexedit`, on peut rencontrer ceci :

```
$ /usr/pgsql-12/bin/pg_checksums -D /var/lib/pgsql/12/data -c

pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 0:
  calculated checksum 5BF9 but block contains C55D
pg_checksums: error: checksum verification failed in file
  "/var/lib/pgsql/12/data/base/14187/16389", block 4438:
  calculated checksum A3 but block contains B8AE
Checksum operation completed
```

[7] https://docs.postgresql.fr/current/pgsurgery.html

```
Files scanned:  1282
Blocks scanned: 28484
Bad checksums:  2
Data checksum version: 1
```

Rappelons que `pg_checksums` sait aussi ajouter ou supprimer les sommes de contrôle sur une instance existante arrêtée (donc après le `initdb`).

Sauvegarde logique :

`pg_dumpall > /dev/null` vérifie certaines sommes de contrôles car il lit tous les fichiers de données, mais pas les index. Si une sauvegarde logique fonctionne (et se restaure sans souci…), on a au moins sauvé l'essentiel.

pg_basebackup :

La meilleure alternative est d'effectuer une sauvegarde physique avec `pg_basebackup`. Par défaut, toutes les sommes de contrôle sont alors vérifiées. Cela n'oblige pas à arrêter la base. De plus, si l'outil est utilisé pour faire des sauvegardes, on obtient une vérification gratuite. S'il s'agit juste de forcer la lecture intégrale, il existe même une option dédiée `--target=blackhole` qui n'écrit pas la sauvegarde (à partir de PostgreSQL 15) :

```
$ pg_basebackup -h /var/run/postgresql -X none --target=blackhole
WARNING:  checksum verification failed in file "./base/5/1824979", block 1:
 ↪  calculated B15 but expected B41E
WARNING:  checksum verification failed in file "./base/5/1824979", block 3:
 ↪  calculated 601A but expected 4D7
WARNING:  checksum verification failed in file "./base/5/1824979", block 35:
 ↪  calculated 22A2 but expected 4DC
WARNING:  checksum verification failed in file "./base/5/1824979", block 67:
 ↪  calculated 633C but expected 4E5
WARNING:  checksum verification failed in file "./base/5/1824979", block 99:
 ↪  calculated 4CB9 but expected 4E2
WARNING:  further checksum verification failures in file "./base/5/1824979" will not
 ↪  be reported
WARNING:  file "./base/5/1824979" has a total of 142 checksum verification failures
WARNING:  could not verify checksum in file "./base/5/1824645", block 0: read buffer
 ↪  size 2 and page size 8192 differ
NOTICE:  all required WAL segments have been archived
WARNING:  142 total checksum verification failures
pg_basebackup: erreur : erreur de somme de contrôle
```

`pg_basebackup` peut désactiver la vérification des sommes de contrôle avec l'option `--no-verify-checksums` afin d'obtenir une copie, aussi corrompue que l'original, mais pouvant servir de base de travail.

Une sauvegarde physique périodique est un bon moyen de vérifier les sommes de contrôle régulièrement, tout en permettant de parer aux soucis qui pourraient survenir. L'outil barman, selon sa configuration, peut utiliser `pg_basebackup`. pgBackRest vérifie aussi les sommes de contrôle lors de ses sauvegardes.

8.5.7 Outils : amcheck & pg_amcheck

Pour les index :

- `amcheck` : vérification uniquement
  - fonction pour l'intégrité des tables
  - vérification de la cohérence index/table (probabiliste)
  - et de l'unicité
- `pg_amcheck` (v14+)
  - version ligne de commande
  - en masse
- Si problème : `REINDEX`

amcheck :

Plus finement, `amcheck` permet de vérifier la cohérence des tables et index et de leur structure interne, et ainsi détecter des corruptions. Il définit plusieurs fonctions :

- `verify_heapam` (PostgreSQL 14 et suivants) vérifie que chaque bloc est bien lisible, correctement construit ; cette fonction peut couvrir les TOAST et peut se limiter à certains blocs ;
- `bt_index_check` est destinée aux vérifications de routine des index, et ne pose qu'un verrou *AccessShareLock* peu gênant ;
- `bt_index_parent_check` est plus minutieuse, mais son exécution gêne les modifications dans la table (verrou *ShareLock* sur la table et l'index) et elle ne peut pas être exécutée sur un serveur secondaire.

Avec le paramètre `heapallindex` à `true`, les deux dernières fonctions effectuent une vérification supplémentaire en recréant temporairement une structure d'index et en la comparant avec l'index original. `bt_index_check` vérifie alors que chaque entrée de la table possède une entrée dans l'index. `bt_index_parent_check` vérifie en plus qu'à chaque entrée de l'index correspond une entrée dans la table.

Les verrous posés par les fonctions ne changent pas. Néanmoins, l'utilisation de ce mode a un impact sur la durée d'exécution des vérifications. Pour limiter l'impact, l'opération n'a lieu qu'en mémoire, et dans la limite du paramètre `maintenance_work_mem` (soit entre 256 Mo et 1 Go, parfois plus, sur les serveurs récents). C'est cette restriction mémoire qui implique que la détection de problèmes est probabiliste pour les plus grosses tables : selon la documentation, la probabilité de rater une incohérence est de 2 % si l'on peut consacrer 2 octets de mémoire à chaque ligne. Mais rien n'empêche de relancer les vérifications régulièrement, diminuant ainsi les chances de rater une erreur.

À partir de PostgreSQL 17, le paramètre `checkunique` à `true` permet de détecter des violations de contrainte d'unicité.

`amcheck` ne fournit aucun moyen de corriger une erreur, puisqu'il détecte des choses qui ne devraient jamais arriver. `REINDEX` sera souvent la solution la plus sûre, simple et facile, mais tout dépend de la cause du problème.

Exemple :

Soit `unetable_pkey`, un index de 10 Go sur un entier :

```
CREATE EXTENSION amcheck ;

SELECT bt_index_check('unetable_pkey');
Durée : 63753,257 ms (01:03,753)

SELECT bt_index_check('unetable_pkey', true);
Durée : 234200,678 ms (03:54,201)
```

Ici, la vérification exhaustive multiplie le temps de vérification par un facteur 4.

Voir la documentation d'amcheck[8] pour tous les détails et les options disponibles.

pg_amcheck :

À partir de la version 14, PostgreSQL dispose d'un nouvel outil en ligne de commande appelé `pg_amcheck`, basé sur `amcheck`.

Ses options de ligne de commande reprennent celles de l'extension, tout en permettant de lancer les vérifications en masse (choix des bases, des tables, parallélisation).

Voir sa documentation[9].

Exemple de recherche de corruption d'unicité :

Cet exemple, dérivé d'un autre de la liste pgsql-hackers, modifie les tables systèmes pour provoquer la corrution d'un index unique (à ne jamais faire en production, bien sûr !) :

```
CREATE TABLE junk (t text UNIQUE);
-- données
INSERT INTO junk (t) VALUES ('ba'), ('be'), ('bo'), ('by');
UPDATE pg_catalog.pg_index
   SET indisunique = false
 WHERE indrelid = 'junk'::regclass;
-- nouvelles données avec doublons
INSERT INTO junk (t) VALUES ('ba'), ('bi'), ('bu'), ('by');
-- remise en place de l'unicité
UPDATE pg_catalog.pg_index
   SET indisunique = true
 WHERE indrelid = 'junk'::regclass;
SELECT * FROM junk ORDER BY 1;
```

Tout semble fonctionner ensuite, si ce n'est que que l'unicité des données déjà présentes n'est pas là.

- Détection avec l'extension `amcheck` :

[8] https://docs.postgresql.fr/current/amcheck.html
[9] https://docs.postgresql.fr/current/app-pgamcheck.html

```
CREATE EXTENSION amcheck ;
SELECT bt_index_check('junk_t_key', false, true);

ERROR:  index uniqueness is violated for index "junk_t_key"
DÉTAIL : Index tid=(1,1) and tid=(1,2) (point to heap tid=(0,1) and tid=(0,5)) page
 ↪  lsn=65/EB31C810.
```

- Détection avec `pg_amcheck` (l'extension `amcheck` doit être installée) :

```
pg_amcheck --database=postgres --relation=junk --checkunique

vérification de l'index btree« postgres public.junk_t_key » :
    ERROR:  index uniqueness is violated for index "junk_t_key"
    DÉTAIL : Index tid=(1,1) and tid=(1,2) (point to heap tid=(0,1) and tid=(0,5))
 ↪  page lsn=65/EB31C810.
```

Les données responsables peuvent être trouvées dans la table (*heap*) grâce aux `tid` indiqués :

```
SELECT * FROM junk WHERE ctid='(0,1)' or ctid='(0,5)';

 t
----
 ba
 ba
```

Malheureusement, l'outil ne fournit que la première erreur rencontrée. Il faudra faire une requête pour les trouver :

```
SELECT t, count(*) FROM junk
GROUP BY 1  HAVING count(*) > 1  ORDER BY t ;
```

Sachant que cette requête risque de se baser sur l'index problématique, il faudra se méfier, ou plutôt agréger avec `t||''`.

8.6 CAS TYPE DE DÉSASTRES

8.7 CAS TYPE DE DÉSASTRES

- Les cas suivants sont assez rares
- Ils nécessitent généralement une restauration
- Certaines manipulations à haut risque sont possibles
    - mais complètement déconseillées !
- Généralement, il faudra restaurer une sauvegarde

Cette section décrit quelques-unes des pires situations de corruptions que l'on peut être amené à observer.

Dans la quasi-totalité des cas, la seule bonne réponse est la restauration de l'instance à partir d'une sauvegarde fiable.

8.7.1 Avertissement

- Privilégier une solution fiable (restauration, bascule)
- Les actions listées ici sont parfois destructrices
- La plupart peuvent (et vont) provoquer des incohérences
- Travailler sur une copie

La plupart des manipulations mentionnées dans cette partie sont destructives, et peuvent (et vont) provoquer des incohérences dans les données.

Tous les experts s'accordent pour dire que l'utilisation de telles méthodes pour récupérer une instance tend à aggraver le problème existant ou à en provoquer de nouveaux, plus graves. S'il est possible de l'éviter, ne pas les tenter (*ie* : préférer la restauration d'une sauvegarde) !

S'il n'est pas possible de faire autrement (ie : pas de sauvegarde utilisable, données vitales à extraire…), alors TRAVAILLER SUR UNE COPIE.

Il ne faut pas non plus oublier que chaque situation est unique, il faut prendre le temps de bien cerner l'origine du problème, documenter chaque action prise, s'assurer qu'un retour arrière est toujours possible.

8.7.2 Corruption de blocs dans des index

- Messages d'erreur lors des accès par l'index
- Requêtes incohérentes
- Données différentes entre un indexscan et un seqscan
- Supprimer et recréer l'index : `REINDEX`
- Est-ce juste cet index ?

Les index sont des objets de structure complexe, ils sont donc particulièrement vulnérables aux corruptions.

Lorsqu'un index est corrompu, on aura généralement des messages d'erreur de ce type :

```
ERROR: invalid page header in block 5869177 of relation base/17291/17420
```

Il peut arriver qu'un bloc corrompu ne renvoie pas de message d'erreur à l'accès, mais que les données elles-mêmes soient altérées, ou que des filtres ne renvoient pas les données attendues. Ce cas est néanmoins très rare dans un bloc d'index.

Dans la plupart des cas, si les données de la table sous-jacente ne sont pas affectées, il est possible de réparer l'index en le reconstruisant intégralement grâce à la commande `REINDEX`. Une corruption d'index se corrige donc facilement, mais le plus inquiétant est la présence même d'une corruption. Le contrôle complet de la base est fortement conseillé.

8.7.3 Corruption de blocs dans des tables 1

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

- Cas plus problématique
- Restauration probablement nécessaire

Les corruptions de blocs vont généralement déclencher des erreurs du type suivant :

```
ERROR: invalid page header in block 32570 of relation base/16390/2663
ERROR: could not read block 32570 of relation base/16390/2663:
       read only 0 of 8192 bytes
```

Si la relation concernée est une table, tout ou partie des données contenues dans ces blocs est perdu.

L'apparition de ce type d'erreur est un signal fort qu'une restauration est certainement nécessaire.

8.7.4 Corruption de blocs dans des tables 2

```
SET zero_damaged_pages = true ;
VACUUM FULL tablecorrompue ;
```

- Des données vont certainement être perdues !

S'il est nécessaire de lire le maximum de données possibles de la table, il est possible d'utiliser l'option de PostgreSQL `zero_damaged_pages` pour demander au moteur de réinitialiser les blocs invalides à zéro lorsqu'ils sont lus au lieu de tomber en erreur. Il s'agit d'un des très rares paramètres absents de `postgresql.conf`.

Par exemple :

```
SET zero_damaged_pages = true ;
VACUUM FULL tablecorrompue ;

WARNING:  invalid page header in block 32570 of relation base/16390/2663; zeroing
 ↪  out page
```

Si l'opération continue et se termine sans erreur, les blocs invalides ont été réinitialisés.

Les données qu'ils contenaient sont évidemment perdues, mais la table peut désormais être accédée dans son intégralité en lecture, permettant ainsi par exemple de réaliser un export des données pour récupérer ce qui peut l'être.

Attention, du fait des données perdues, le résultat peut être incohérent (contraintes non respectées…).

Par ailleurs, sans les *checksums*, PostgreSQL ne détecte pas les corruptions logiques, c'est-à-dire n'affectant pas la structure des données mais corrompant le contenu. Il ne faut donc pas penser que la procédure d'export complet de données suivie d'un import sans erreur garantit l'absence de corruption.

8.7.5 Corruption de blocs dans des tables 3

- Si la corruption est importante, l'accès au bloc peut faire crasher l'instance
- Il est tout de même possible de réinitialiser le bloc
  - identifier le fichier à l'aide de `pg_relation_filepath()`
  - trouver le bloc avec `ctid` / `pageinspect`
  - réinitialiser le bloc avec `dd`
  - il faut vraiment ne pas avoir d'autre choix

Dans certains cas, il arrive que la corruption soit suffisamment importante pour que le simple accès au bloc fasse crasher l'instance. Dans ce cas, le seul moyen de réinitialiser le bloc est de le faire manuellement au niveau du fichier, **instance arrêtée**, par exemple avec la commande `dd`.

Pour identifier le fichier associé à la table corrompue, utiliser la fonction `pg_relation_filepath()` :

```
SELECT pg_relation_filepath('test_corruptindex') ;

 pg_relation_filepath
----------------------
 base/16390/40995
```

Le résultat donne le chemin vers le fichier principal de la table, relatif au `PGDATA` de l'instance.

Attention, une table peut contenir plusieurs fichiers. Par défaut une instance PostgreSQL sépare les fichiers en segments de 1 Go. Une table dépassant cette taille aura donc des fichiers supplémentaires (`base/16390/40995.1`, `base/16390/40995.2` …).

Pour trouver le fichier contenant le bloc corrompu, il faudra donc prendre en compte le numéro du bloc trouvé dans le champ `ctid`, multiplier ce numéro par la taille du bloc (paramètre `block_size`, 8 ko par défaut), et diviser le tout par la taille du segment.

Cette manipulation est évidemment extrêmement risquée, la moindre erreur pouvant rendre irrécupérables de grandes portions de données. Il est donc fortement déconseillé de se lancer dans ce genre de manipulations à moins d'être absolument certain que c'est indispensable.

Encore une fois, ne pas oublier de travailler sur une copie, et pas directement sur l'instance de production.

8.7.6 Corruption des WAL 1

- Situés dans le répertoire `pg_wal`
- Les WAL sont nécessaires au *recovery*
- Démarrage impossible s'ils sont corrompus ou manquants
- Si les fichiers WAL ont été archivés, les récupérer
- Sinon, la restauration est la seule solution viable

Les fichiers WAL sont les journaux de transactions de PostgreSQL. Leur fonction est d'assurer que les transactions qui ont été effectuées depuis le dernier checkpoint ne seront pas perdues en cas de crash de l'instance.

Si certains sont corrompus ou manquants, alors PostgreSQL ne pourra pas redémarrer.

Il ne faut **jamais** supprimer les fichiers WAL, même si le système de fichiers est plein ! (Dans ce dernier cas, chercher la source : il s'agit souvent d'un archivage qui traîne ou d'un slot de réplication bloqué.)

Si l'archivage était activé et que les fichiers WAL perdus ont été archivés, alors il est possible de les restaurer avant de tenter un nouveau démarrage (copie manuelle, utilisation de `restore_command`…).

Si ce n'est pas possible, ou si les archives ont également été corrompues ou supprimées, l'instance ne pourra pas redémarrer.

Dans cette situation, comme dans la plupart des autres évoquées ici, la seule solution permettant de s'assurer que les données ne seront pas corrompues est de procéder à une restauration de l'instance depuis la dernière sauvegarde complète disponible.

8.7.7 Corruption des WAL 2

- `pg_resetwal` permet de forcer le démarrage
- ATTENTION !!!
  - cela va provoquer des pertes de données
  - des corruptions de données sont également probables
  - ce n'est pas une action corrective !

L'utilitaire `pg_resetwal` a comme fonction principale de supprimer les fichiers WAL courants et d'en créer un nouveau, avant de mettre à jour le fichier de contrôle pour permettre le redémarrage.

Au minimum, cette action va provoquer la perte de toutes les transactions validées effectuées depuis le dernier checkpoint. Il est également probable que des incohérences vont apparaître, certaines relativement simples à détecter via un export/import (incohérences dans les clés étrangères par exemple), certaines complètement invisibles.

L'utilisation de cet utilitaire est extrêmement dangereuse, n'est pas recommandée, et ne peut jamais être considérée comme une action corrective. Il faut toujours privilégier la restauration d'une sauvegarde plutôt que son exécution.

Si l'utilisation de `pg_resetwal` est néanmoins nécessaire (par exemple pour récupérer des données absentes de la sauvegarde), alors il faut travailler sur une copie des fichiers de l'instance, récupérer ce qui peut l'être à l'aide d'un export de données, et les importer dans une autre instance.

Les données récupérées de cette manière devraient également être soigneusement validées avant d'être importée de façon à s'assurer qu'il n'y a pas de corruption silencieuse.

Il ne faut en aucun cas remettre une instance en production après une réinitialisation des WAL.

8.7.8 Corruption du fichier de contrôle

- Fichier `global/pg_control`
- Contient les informations liées au dernier checkpoint
- Sans lui, l'instance ne peut pas démarrer
- Recréation avec `pg_resetwal`… parfois
- Restauration nécessaire

Le fichier de contrôle de l'instance contient de nombreuses informations liées à l'activité et au statut de l'instance, notamment l'instant du dernier checkpoint, la position correspondante dans les WAL, le numéro de transaction courant et le prochain à venir…

Ce fichier est le premier lu par l'instance. S'il est corrompu ou supprimé, l'instance ne pourra pas démarrer.

Il est possible de forcer la réinitialisation de ce fichier à l'aide de la commande `pg_resetwal`, qui va se baser par défaut sur les informations contenues dans les fichiers WAL présents pour tenter de « deviner » le contenu du fichier de contrôle.

Ces informations seront très certainement erronées, potentiellement à tel point que même l'accès aux bases de données par leur nom ne sera pas possible :

```
$ pg_isready
/var/run/postgresql:5432 - accepting connections

$ psql postgres
psql: FATAL:  database "postgres" does not exist
```

Encore une fois, utiliser `pg_resetwal` n'est en aucun cas une solution, mais doit uniquement être considéré comme un contournement temporaire à une situation désastreuse.

Une instance altérée par cet outil ne doit pas être considérée comme saine.

8.7.9 Corruption du CLOG

- Fichiers dans `pg_xact`
- Statut des différentes transactions
- Son altération risque de causer des incohérences majeures
- Préférer la restauration

Le fichier CLOG (*Commit Log*) dans `PGDATA/pg_xact/` contient le statut des différentes transactions, notamment si celles-ci sont en cours, validées ou annulées.

S'il est altéré ou supprimé, il est possible que des transactions qui avaient été marquées comme annulées soient désormais considérées comme valides, et donc que les modifications de données correspondantes deviennent visibles aux autres transactions.

C'est évidemment un problème d'incohérence majeur, tout problème avec ce fichier devrait donc être soigneusement analysé.

Il est préférable dans le doute de procéder à une restauration et d'accepter une perte de données plutôt que de risquer de maintenir des données incohérentes dans la base.

8.7.10 Corruption du catalogue système

- Le catalogue contient la définition du schéma
- Sans lui, les données sont inaccessibles
- Situation très délicate…

Le catalogue système contient la définition de toutes les relations, les méthodes d'accès, la correspondance entre un objet et un fichier sur disque, les types de données existantes… S'il est incomplet, corrompu ou inaccessible, l'accès aux données en SQL risque de ne pas être possible du tout.

Cette situation est très délicate, et appelle là encore une restauration.

Si le catalogue était complètement inaccessible, sans sauvegarde la seule solution restante serait de tenter d'extraire les données directement des fichiers data de l'instance, en oubliant toute notion de cohérence, de type de données, de relation… Personne ne veut faire ça.

8.8 CONCLUSION

- Les désastres peuvent arriver
- Il faut s'y être préparé
- Faites des sauvegardes !
    - et testez-les

8.9 QUIZ

https://dali.bo/i5_quiz

8.10 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/i5_solutions.

8.10.1 Corruption d'un bloc de données

But : Corrompre un bloc et voir certains impacts possibles.

Vérifier que l'instance utilise bien les checksums. Au besoin les ajouter avec `pg_checksums`.

Créer une base **pgbench** et la remplir avec l'outil de même, avec un facteur d'échelle 10 et **avec les clés étrangères entre tables** ainsi :

```
/usr/pgsql-17/bin/pgbench -i -s 10 -d pgbench --foreign-keys
```

Voir la taille de `pgbench_accounts`, les valeurs que prend sa clé primaire.

Retrouver le fichier associé à la table `pgbench_accounts` (par exemple avec `pg_file_relationpath`).

Arrêter PostgreSQL.

Avec un outil `hexedit` (à installer au besoin, l'aide s'obtient par **F1**), modifier une ligne dans le PREMIER bloc de la table.

Redémarrer PostgreSQL et lire le contenu de `pgbench_accounts`.

Tenter un `pg_dumpall > /dev/null`.

- Arrêter PostgreSQL.
- Voir ce que donne `pg_checksums` (`pg_verify_checksums` en v11).

- Faire une copie de travail à froid du PGDATA.
- Protéger en écriture le PGDATA original.
- Dans la copie, supprimer la possibilité d'accès depuis l'extérieur.

Avant de redémarrer PostgreSQL, supprimer les sommes de contrôle dans la copie (en désespoir de cause).

Démarrer le cluster sur la copie avec `pg_ctl`.

Que renvoie ceci ?

```
SELECT * FROM pgbench_accounts LIMIT 100 ;
```

Tenter une récupération avec `SET zero_damaged_pages`. Quelles données ont pu être perdues ?

8.10.2 Corruption d'un bloc de données et incohérences

But : Corrompre une table portant une clé étrangère.

Nous continuons sur la copie de la base de travail, où les sommes de contrôle ont été désactivées.

Consulter le format et le contenu de la table `pgbench_branches`.

Retrouver les fichiers des tables `pgbench_branches` (par exemple avec `pg_file_relationpath`).

Pour corrompre la table :

- Arrêter PostgreSQL.
- Avec hexedit, dans le premier bloc en tête de fichier, remplacer les derniers caractères non nuls (`C0 9E 40`) par `FF FF FF`.
- En toute fin de fichier, remplacer le dernier `01` par un `FF`.
- Redémarrer PostgreSQL.

- Compter le nombre de lignes dans `pgbench_branches`.
- Recompter après `SET enable_seqscan TO off ;`.
- Quelle est la bonne réponse ? Vérifier le contenu de la table.

Qu'affiche `pageinspect` pour cette table ?

Avec l'extension `amcheck`, essayer de voir si le problème peut être détecté. Si non, pourquoi ?

Pour voir ce que donnerait une restauration :

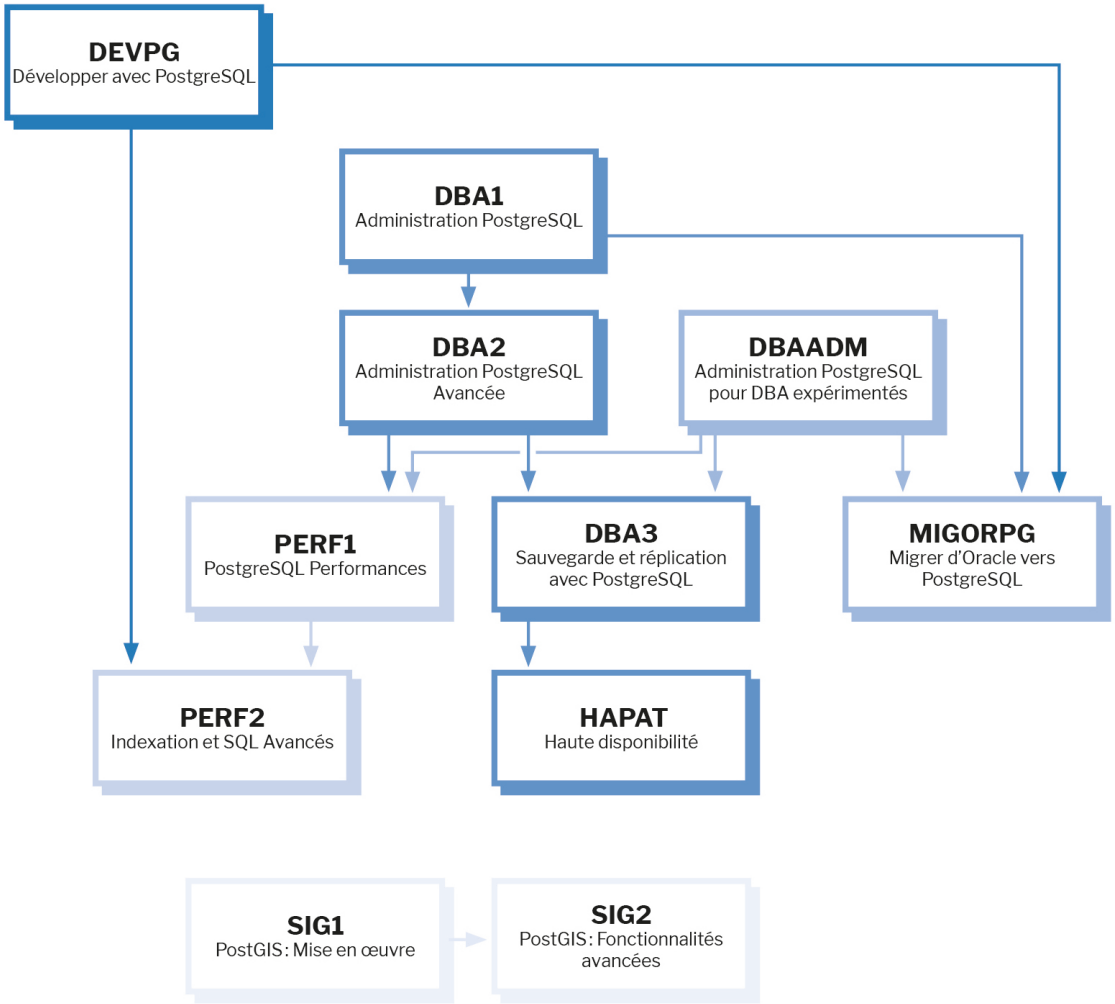
- Exporter `pgbench_accounts`, définition des index comprise.
- Supprimer la table (il faudra supprimer `pgbench_history` aussi).
- Tenter de la réimporter.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.