Module M2

Configuration de PostgreSQL

25.09

Table des matières

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

1/ Configuration de PostgreSQL **5**
1.1 Au menu . . . 6
1.2 Paramètres en lecture seule . . . 7
1.3 Fichiers de configuration . . . 8
1.4 postgresql.conf . . . 9
1.4.1 Surcharge des paramètres de postgresql.conf . . . 10
1.4.2 Précédence des paramètres . . . 13
1.4.3 Survol de postgresql.conf . . . 13
1.5 pg_hba.conf et pg_ident.conf . . . 16
1.6 Tablespaces . . . 17
1.6.1 Tablespaces : mise en place . . . 19
1.6.2 Tablespaces : configuration . . . 20
1.6.3 Tablespaces : performance . . . 22
1.7 Gestion des connexions . . . 24
1.7.1 TCP . . . 24
1.7.2 SSL . . . 25
1.8 Statistiques sur l'activité . . . 26
1.8.1 Statistiques d'activité collectées . . . 27
1.8.2 Vues système . . . 28
1.9 Statistiques sur les données . . . 36
1.10 Optimiseur . . . 39
1.10.1 Optimisation par les coûts . . . 40
1.10.2 Nombre de tables considérées par le planificateur . . . 43
1.10.3 Paramètres supplémentaires de l'optimiseur . . . 46
1.10.4 Débogage de l'optimiseur . . . 47
1.11 Conclusion . . . 50
1.11.1 Questions . . . 50
1.12 Quiz . . . 51
1.13 Travaux pratiques . . . 52
1.13.1 Tablespace . . . 52
1.13.2 Statistiques d'activités, tables et vues système . . . 52
1.13.3 Statistiques sur les données . . . 53

1.14 Travaux pratiques (solutions) 55
1.14.1 Tablespace 55
1.14.2 Statistiques d'activités, tables et vues système 56
1.14.3 Statistiques sur les données 59

Les formations Dalibo **63**
Cursus des formations 63
Les livres blancs 64
Téléchargement gratuit 64

Sur ce document

Formation	Module M2
Titre	Configuration de PostgreSQL
Révision	25.09
PDF	https://dali.bo/m2_pdf
EPUB	https://dali.bo/m2_epub
HTML	https://dali.bo/m2_html
Slides	https://dali.bo/m2_slides
TP	https://dali.bo/m2_tp
TP (solutions)	https://dali.bo/m2_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

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, Nizar Hamadi, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Cédric Martin, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

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

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

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

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

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

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

Cette licence interdit la réutilisation pour l'apprentissage d'une IA. Elle couvre 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

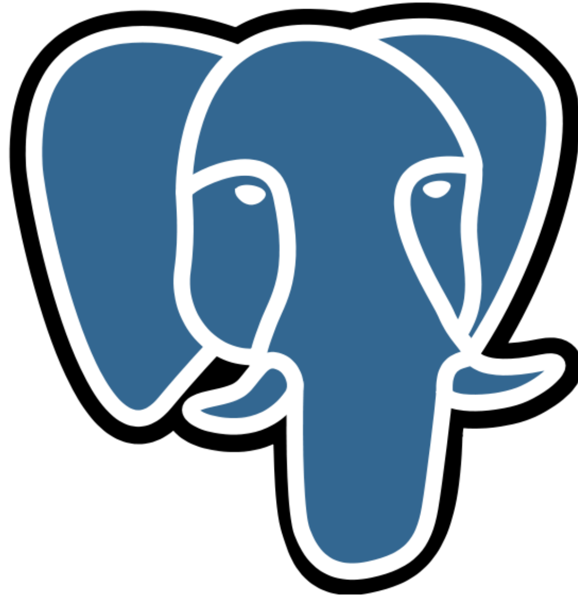
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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/ Configuration de PostgreSQL



1.1 AU MENU

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

1.2 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.

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.)

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

1.3 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`.

1.4 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.

1.4.1 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 :

```sql
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 :

```sql
SET parametre = valeur ;
```

ou même juste au sein d'une transaction :

```sql
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 :

```sql
SET client_min_messages = debug2;
```

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

```sql
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 :

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

SELECT pg_reload_conf() ;
```

```
 pg_reload_conf
----------------
 t
```

```sql
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      |
```

1.4.2 Précédence des paramètres

FIGURE 1/ .1 – Ordre de 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`.

1.4.3 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.

1.5 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

1.6 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*.

1.6.0.1 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 :

```sql
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.

1.6.0.2 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

1.6.1 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   |
```

1.6.2 Tablespaces : configuration

```
CREATE TABLESPACE  ssd      LOCATION '/mnt/data_ssd/' ;
CREATE TABLESPACE  ssd_tmp1 LOCATION '/mnt/temp1' ;
CREATE TABLESPACE  ssd_tmp2 LOCATION '/mnt/temp2' ;
GRANT CREATE ON TABLESPACE  ssd  TO dupont ;
GRANT CREATE ON TABLESPACE  ssd_tmp1,ssd_tmp2  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_tmp1,ssd_tmp2;
  ```

1.6.2.1 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
```

1.6.2.2 Tablespaces temporaires

Les fichiers temporaires générés par certains nœuds de requête (tris, *hash joins*, CTE matérialisées…) ainsi que les tables temporaires peuvent être déplacés vers un ou plusieurs tablespaces dédiés grâce au paramètre `temp_tablespaces`. Le premier intérêt est de leur dédier 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 temporaires car la mémoire de la machine ne doit servir qu'aux applications et aux outils, au cache de l'OS, et aux opérations de PostgreSQL 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_tmp1  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 temporaire peut s'étaler sur plusieurs tablespaces temporaires, 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.

1.6.3 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) ;
- `maintenance_io_concurrency` (idem, pour une opération de maintenance, défaut 10).

effective_io_concurrency :

`effective_io_concurrency` a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Il n'a d'intérêt que si un nœud *Bitmap Scan* a été choisi. Cela n'arrive qu'avec un certain nombre de lignes à récupérer, et est favorisé par une valeur importante de `effective_cache_size` et un peu de corrélation physique dans la table.

La valeur d'`effective_io_concurrency` n'influe pas sur le choix du plan, mais sa valeur peut notablement accélérer l'exécution du *Bitmap Heap Scan*. Le temps de lecture peut fréquemment être divisé par 3 ou plus.

Les valeurs possibles d'`effective_io_concurrency` vont de 0 à 1000. En principe, sur un disque magnétique seul, la valeur 1 ou 0 peut convenir. Avec du SSD, et encore plus du NVMe, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. Trouver la bonne valeur dépend de divers paramètres liés aux caractérisques exactes des disques et de leur paramétrage noyau. Le *read ahead* du noyau intervient également. Le comportement de PostgreSQL sur ce point change aussi avec les versions. De plus, à partir d'un certain nombre de blocs, les I/O peuvent simplement saturer.

La valeur par défaut de 1 (jusque PostgreSQL 17) a été choisie de manière très conservatrice. Les développeurs ont décidé[6] que la valeur 16 est plus intéressante, même sur de vieux disques, et que ce

[6] https://www.postgresql.org/message-id/flat/CAAKRu_Z%2BJa-mwXebOoOERMMUMvJeRhzTjad4dSThxG0JLXESxw%40mail.gmail.com#c028f5b93307aafb9fe6d09d0cf3eeae

sera le défaut à partir de PostgreSQL 18[7], qui inclut d'autres améliorations.

À l'inverse, la mauvaise latence de certains systèmes aux disques en principe performants (baies surchargées, certains stockages cloud…) peut parfois être compensée par un `effective_io_concurrency` plus élevé.

Il faut tester avec vos requêtes qui utilisent des *Bitmap Heap Scan*, et tenir compte du nombre de requêtes simultanées, mais `effective_io_concurrency = 16` semble un bon point de départ, même sur une configuration modeste. Montez si vous avez de bons disques.

Une valeur excessive de `effective_io_concurrency` mène à un gaspillage de CPU : ne pas monter trop haut sans preuve de l'utilité, surtout sur un système très chargé.

Pour les systèmes RAID matériels, selon la documentation[8], configurer ce paramètre en fonction du nombre n de disques utiles dans le RAID (n/2 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).

(Avant la version 13, le principe d' `effective_io_concurrency` 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[9].)

maintenance_io_concurrency :

`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`.

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 voire un disque NVMe les surchargera ainsi :

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

[7] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=ff79b5b2aba02d720f9b7fff644dd50ce07b8c6e
[8] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY
[9] https://docs.postgresql.fr/13/release.html

1.7 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.

1.7.1 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.

1.7.2 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[10].

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

1.8 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 d'activer `track_io_timing` sur une machine, vérifiez l'impact avec l'outil `pg_test_timing`.

pg_test_timing[11] est livré avec PostgreSQL et mesure les performances de l'horloge système. Si le temps de mesure renvoyé sur la deuxième ligne n'est que de quelques dizaines de nanosecondes, la machine est suffisamment rapide pour que `track_io_timing` renvoie des résultats précis et sans ralentir la requête. C'est le cas sur presque toutes les machines et systèmes d'exploitation actuels, mais il y a parfois des surprises, par exemple dans certaines machines virtuelles ou selon la source de l'horloge système. Sinon, éviter d'activer `track_io_timing` sur un serveur de production. Sur une machine de test ou de formation, ce n'est pas un problème.

`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.

1.8.1 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

[11] https://docs.postgresql.fr/current/pgtesttiming.html

1.8.2 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[12]. 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 :

```
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
```

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

```
-[ RECORD 1 ]----+---------------------------------------------------------------
datname          | pgbench_20000_hdd
application_name | pgbench
pid              | 786146
wait_event_type  | LWLock
wait_event       | WALWrite
query            | UPDATE pgbench_accounts SET abalance = abalance + 4055 WHERE…
description      | Waiting for WAL buffers to be written to disk
-[ 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.

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  | 0000000100000007000000E3
last_archived_time | 2024-11-14 16:00:00.418887+01
failed_count       | 13254
last_failed_wal    | 0000000100000007000000E4
```

```
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) :

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

1.9 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

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

  - Statistiques multicolonnes sur demande

```sql
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`).

1.10 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*).

1.10.1 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*). Il n'a d'intérêt que si un nœud *Bitmap Scan* a été choisi. Cela n'arrive qu'avec un certain nombre de lignes à récupérer, et est favorisé par une valeur importante de `effective_cache_size` et un peu de corrélation physique dans la table.

La valeur d'`effective_io_concurrency` n'influe pas sur le choix du plan, mais sa valeur peut notablement accélérer l'exécution du *Bitmap Heap Scan*. Le temps de lecture peut fréquemment être divisé par 3 ou plus.

Les valeurs possibles d'`effective_io_concurrency` vont de 0 à 1000. En principe, sur un disque magnétique seul, la valeur 1 ou 0 peut convenir. Avec du SSD, et encore plus du NVMe, il est possible de monter à plusieurs centaines, étant donné la rapidité de ce type de disque. Trouver la bonne valeur dépend de divers paramètres liés aux caractérisques exactes des disques et de leur paramétrage noyau. Le *read ahead* du noyau intervient également. Le comportement de PostgreSQL sur ce point change aussi avec les versions. De plus, à partir d'un certain nombre de blocs, les I/O peuvent simplement saturer.

La valeur par défaut de 1 (jusque PostgreSQL 17) a été choisie de manière très conservatrice. Les développeurs ont décidé[13] que la valeur 16 est plus intéressante, même sur de vieux disques, et que ce sera le défaut à partir de PostgreSQL 18[14], qui inclut d'autres améliorations.

À l'inverse, la mauvaise latence de certains systèmes aux disques en principe performants (baies surchargées, certains stockages cloud…) peut parfois être compensée par un `effective_io_concurrency` plus élevé.

Il faut tester avec vos requêtes qui utilisent des *Bitmap Heap Scan*, et tenir compte du nombre de requêtes simultanées, mais `effective_io_concurrency = 16` semble un bon point de départ, même sur une configuration modeste. Montez si vous avez de bons disques.

Une valeur excessive de `effective_io_concurrency` mène à un gaspillage de CPU : ne pas monter trop haut sans preuve de l'utilité, surtout sur un système très chargé.

Pour les systèmes RAID matériels, selon la documentation[15], configurer ce paramètre en fonction du

[13] https://www.postgresql.org/message-id/flat/CAAKRu_Z%2BJa-mwXebOoOERMMUMvJeRhzTjad4dSThxG0JLXESxw%40mail.gmail.com#c028f5b93307aafb9fe6d09d0cf3eeae
[14] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=ff79b5b2aba02d720f9b7fff644dd50ce07b8c6e
[15] https://docs.postgresql.fr/current/runtime-config-resource.html#GUC-EFFECTIVE-IO-CONCURRENCY

nombre n de disques utiles dans le RAID (n/2 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).

(Avant la version 13, le principe d'`effective_io_concurrency` 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[16].)

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

Pour plus de détails, voir l'article sur les coûts de planification[17] issu de notre base de connaissance.

[16] https://docs.postgresql.fr/13/release.html
[17] https://support.dalibo.com/kb/cout_planification

1.10.2 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` !
- `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 [18]. Monter `join_collapse_limit` de 8 à 9 bascule vers un plan bien meilleur [19]. Une autre solution aurait été de remonter la table `INNER JOIN contacts`, qui porte le critère, plus haut dans le `FROM`, pour que le critère soit pris en compte dès le début. Mais il n'est pas toujours possible de modifier le code, qui d'ailleurs peut être dynamique. (Pour l'exemple complet, voir https:/dali.bo/j2_html#requête-avec-beaucoup-de-tables-1)

[18] https://explain.dalibo.com/plan/D0U
[19] https://explain.dalibo.com/plan/EQN

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[20], 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[21], résumé aussi dans ce schéma :

[20] https://docs.postgresql.fr/current/explicit-joins.html
[21] https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=f5bc74192d2ffb32952a06c62b3458d28ff7f98f

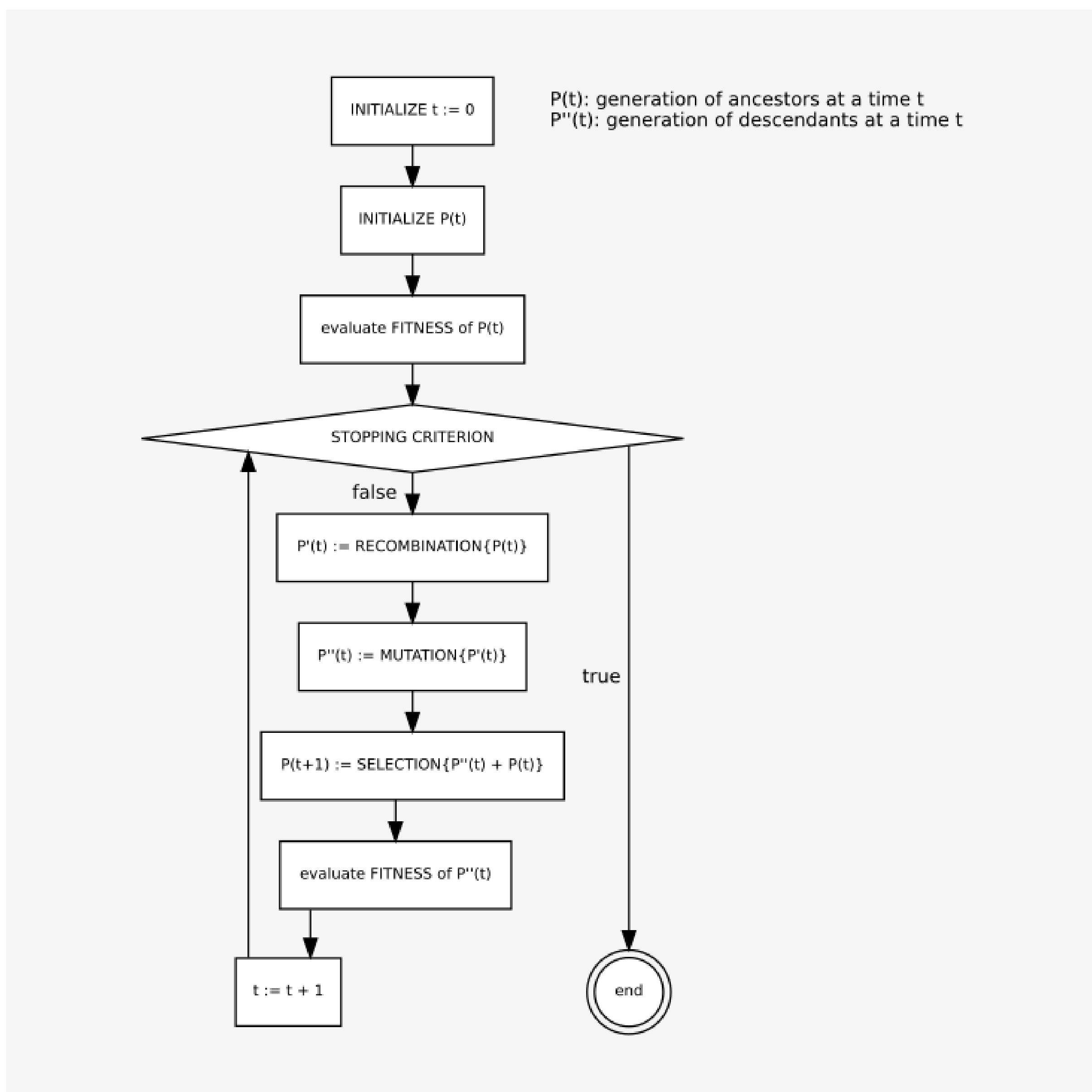


FIGURE 1/ .2 – Principe d'un algorithme génétique (schéma de la documentation officielle, licence PostgreSQL)

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 recherche d'autres plans.

À `geqo_seed` identique, les plans générés seront toujours les mêmes[22] (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.

1.10.3 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.

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

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

`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`.

1.10.4 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 :

```sql
-- 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`.

1.11 CONCLUSION

- Nombreuses fonctionnalités
  - donc nombreux paramètres

1.11.1 Questions

N’hésitez pas, c’est le moment !

1.12 QUIZ

https://dali.bo/m2_quiz

1.13 TRAVAUX PRATIQUES

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

1.13.1 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 ?

1.13.2 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 ?

1.13.3 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.

1.14 TRAVAUX PRATIQUES (SOLUTIONS)

1.14.1 Tablespace

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

En tant qu'utilisateur **root** :

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

En tant qu'utilisateur **postgres** :

```
$ psql

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

postgres=# \db

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

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

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

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

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

                               chemin
--------------------------------------------------------------------
  /var/lib/pgsql/15/data/pg_tblspc/16394/PG_15_202107181/16393/16395
```

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

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

```
$ cd /opt/ts1/PG_15_202107181/
$ ls -lR
.:
total 0
drwx------ 2 postgres postgres 18 Apr 16 16:26 16393
./16393:
total 0
-rw------- 1 postgres postgres 0 Apr 16 16:26 16395
```

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

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

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

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

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

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

1.14.2 Statistiques d'activités, tables et vues système

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

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

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

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

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

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

-[ RECORD 1 ]-----+-------
relid             | 24594
schemaname        | public
relname           | t3
seq_scan          | 0
seq_tup_read      | 0
```

```
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 1000
n_tup_upd           | 0
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000
n_dead_tup          | 0
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    |
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
```

Les statistiques indiquent bien que 1000 lignes ont été insérées.

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

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

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

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

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

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

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

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

d'où par exemple :

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

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

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

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

```
$ psql -E
```

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

```
\l
********* QUERY **********
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges"
FROM pg_catalog.pg_database d
ORDER BY 1;
**************************
```

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

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

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

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

```
# terminal 1
$ psql b0

# terminal 2
$ psql b1
```

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

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

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

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

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

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

Ce qui donnerait par exemple :

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

1.14.3 Statistiques sur les données

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Toutes les lignes ont donc à présent la même valeur.

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

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

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

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

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

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

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

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

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

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

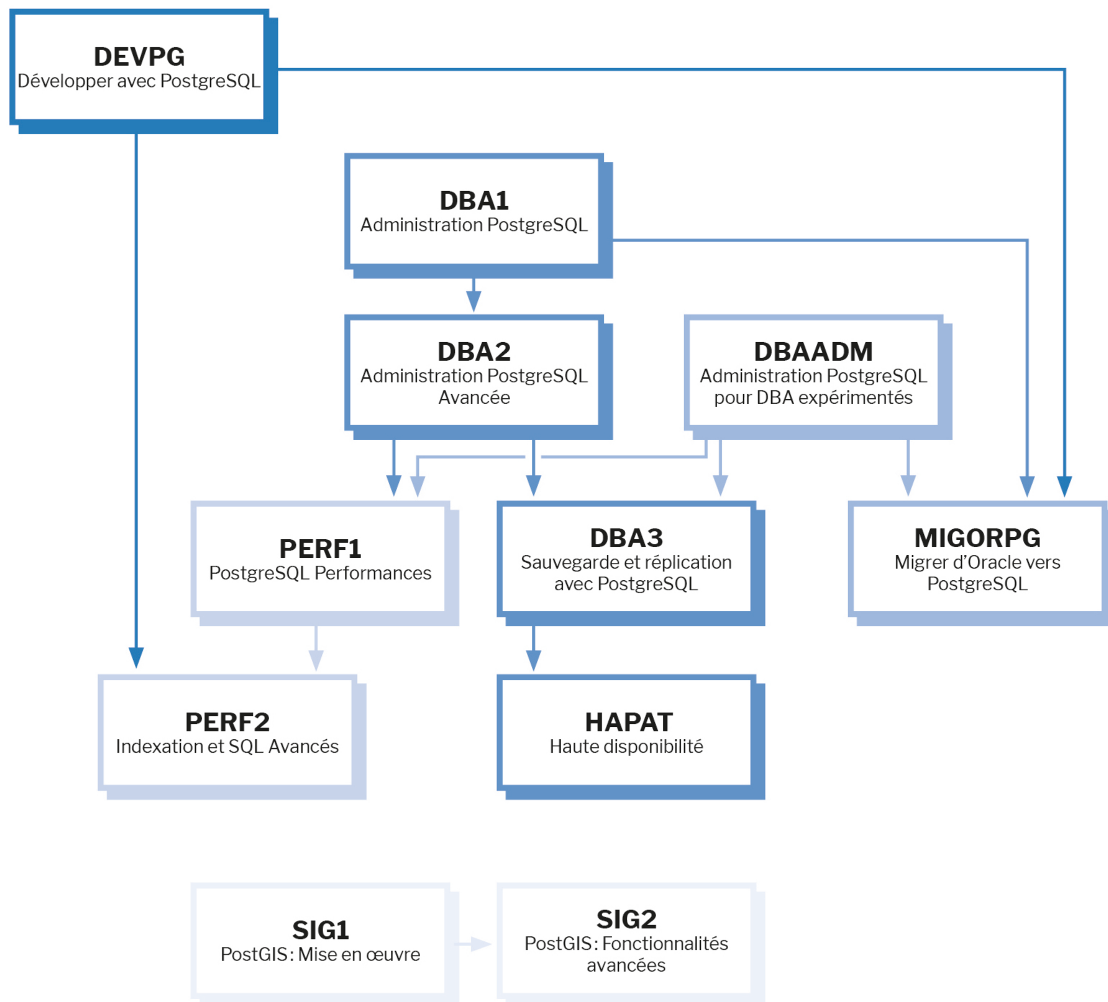
Si l'autovacuum avait été activé, les modifications massives dans la table auraient provoqué assez rapidement la mise à jour des statistiques.

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
— DEV1 : Introduction à SQL
https://dali.bo/dev1
— 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.