Module M2

# Configuration de PostgreSQL

DALIBO
L'expertise PostgreSQL

22.09

Dalibo SCOP

https://dalibo.com/formations

# Configuration de PostgreSQL

Module M2

TITRE : Configuration de PostgreSQL
SOUS-TITRE : Module M2

REVISION: 22.09
DATE: 02 septembre 2022


---

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

---

# LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

*Vous êtes autorisé à* :

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

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

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

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

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

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

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

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

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

[1] mailto:contact@dalibo.com

**Chers lectrices & lecteurs,**

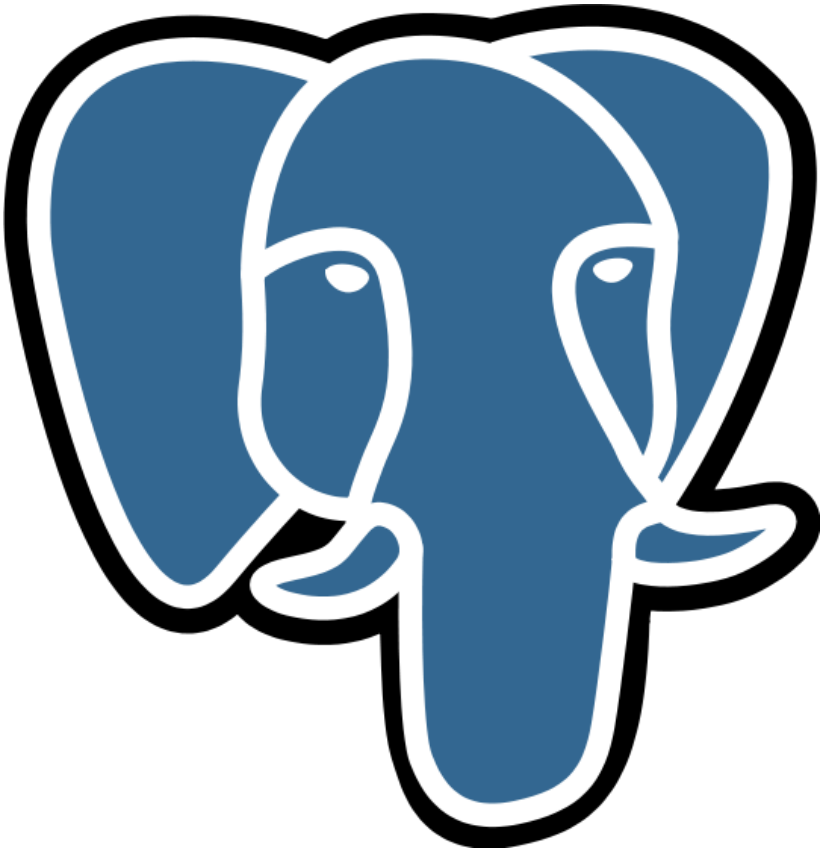
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

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

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

# Table des Matières

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

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

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

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

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

---

## 1.3 FICHIERS DE CONFIGURATION

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

Les fichiers de configuration sont habituellement les 4 suivants :

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

---

## 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/14/data/postgresql.conf` par exemple). Debian le place dans `/etc` (`/etc/postgresql/14/main/postgresql.conf` pour l'instance par défaut).

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

```
# SHOW config_file;

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

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

```
clé = valeur
```

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

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

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

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

---

### 1.4.1 SURCHARGE DES PARAMÈTRES DE POSTGRESQL.CONF

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
SET parametre = valeur ;
```

ou une transaction :

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

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

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

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

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

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

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

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

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

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

SELECT pg_reload_conf() ;

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

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

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

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

### 1.4.2 SURVOL DE POSTGRESQL.CONF

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

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

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

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

Les principales sections sont :

**Connections and authentication**

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

**Resource usage (except WAL)**

Cette partie fixe des limites à certaines consommations de ressources.

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

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

**Write-Ahead Log**

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

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

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

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

**Replication**

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

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

**Query tuning**

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

**Reporting and logging**

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

**Autovacuum**

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

**Client connection defaults**

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

**Lock management**

Les paramètres de cette section sont rarement modifiés.

---

## 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] .

---

## 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)
- Utilisation selon des droits

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

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

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

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

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

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

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

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

```
# \db

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

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

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

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

Les cas d'utilisation des tablespaces dans PostgreSQL sont :

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

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

---

### 1.6.1 TABLESPACES : MISE EN PLACE

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

CREATE DATABASE nom TABLESPACE 'chaud';

ALTER DATABASE nom SET default_tablespace TO 'chaud';

GRANT CREATE ON TABLESPACE chaud TO un_utilisateur ;

CREATE TABLE une_table (…) TABLESPACE chaud ;

ALTER TABLE une_table SET TABLESPACE chaud ;  -- verrou !

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

Les ordres ci-dessus permettent de :

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

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

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

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

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

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

---

### 1.6.2 TABLESPACES : PARAMÈTRES LIÉS

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

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

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

On peut définir un ou plusieurs tablespaces pour les opérations de tri, dans le paramètre `temp_tablespaces`. Il faudra donner des droits dessus aux utilisateurs avec `GRANT CREATE ON TABLESPACE ... TO ...` pour qu'ils soient utilisables. Si plusieurs tablespaces de tri sont paramétrés, ils seront utilisés de façon aléatoire à chaque création d'objet temporaire,

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

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

Par exemple, pour un tablespace sur un SSD :

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

---

## 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`

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

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

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

Le mécanisme de keepalive a deux intérêts :

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

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

### 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[4] .

## 1.8 STATISTIQUES SUR L'ACTIVITÉ

- Collectées par chaque session durant son travail
- (Ne pas confondre avec statistiques sur les données !)
- Remontées au *stats collector*
- Stockées régulièrement dans un fichier, consultable par des vues systèmes
- Paramètres :
  - `track_activities`, `track_activity_query_size`, `track_counts`, `track_io_timing` et `track_functions`
  - `update_process_title` et `stats_temp_directory`

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

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

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

Chaque session collecte des statistiques, dès qu'elle effectue une opération. Ces informations, si elles sont transitoires, comme la requête en cours, sont directement stockées

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

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

Voici les paramètres concernés :

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

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

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

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

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

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

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

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

### 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

---

### 1.8.2 VUES SYSTÈME

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

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

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

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

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

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

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

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

`pg_stat_activity` est une des vues les plus utilisées et est souvent le point de départ d'une recherche : elle donne des informations sur les processus en cours sur l'instance, que ce soit des processus en tâche de fond ou des processus backends associés aux clients : numéro de processus, adresse et port, date de début d'ordre, de transaction, de session, requête en cours, état, ordre SQL et nom de l'application si elle l'a renseigné. (Noter qu'avant la version 10, cette vue n'affichait que les processus backend ; à partir de la version 10 apparaissent des workers, le checkpointer, le walwriter… ; à partir de la version 14 apparaît le processus d'archivage).

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

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

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

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

Cette vue fournit aussi des informations sur ce que chaque session attend. Pour les détails sur `wait_event_type` (type d'événement en attente) et `wait_event` (nom de l'événement en attente), voir le tableau des événements d'attente[5] .

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

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

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

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

Des vues plus spécialisées existent :

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

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

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

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

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

---

## 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
- `ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300;`
- Statistiques multicolonnes sur demande
  - `CREATE STATISTICS`

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

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

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

Les statistiques sont collectées sur :

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

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

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

La vue `pg_stats` affiche les statistiques collectées :

```
\d pg_stats

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

- `inherited` : la statistique concerne-t-elle un objet utilisant l'héritage (table parente, dont héritent plusieurs tables) ;
- `null_frac` : fraction d'enregistrements dont la colonne vaut NULL ;
- `avg_width` : taille moyenne de cet attribut dans l'échantillon collecté ;
- `n_distinct` : si positif, nombre de valeurs distinctes, si négatif, fraction de valeurs distinctes pour cette colonne dans la table. Il est possible de forcer le nombre de valeurs distinctes, s'il est constaté que la collecte des statistiques n'y arrive pas :

`ALTER TABLE xxx ALTER COLUMN yyy SET (n_distinct = -0.5) ; ANALYZE xxx;` par exemple indique à l'optimiseur que chaque valeur apparaît statistiquement deux fois ;

- `most_common_vals` et `most_common_freqs` : les valeurs les plus fréquentes de la table, et leur fréquence. Le nombre de valeurs collecté est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target`. Le défaut de 100 échantillons sur 30 000 lignes peut être modifié par `ALTER TABLE matable ALTER COLUMN macolonne SET statistics 300 ;` (avec une évolution proportionnelle du nombre de lignes consultées) sachant que le temps de planification augmente exponentiellement et qu'il vaut mieux ne pas dépasser la valeur 1000 ;
- `histogram_bounds` : les limites d'histogramme sur la colonne. Les histogrammes permettent d'évaluer la sélectivité d'un filtre par rapport à sa valeur précise. Ils permettent par exemple à l'optimiseur de déterminer que 4,3 % des enregistrements d'une colonne `noms` commencent par un A, ou 0,2 % par AL. Le principe est de regrouper les enregistrements triés dans des groupes de tailles approximativement identiques, et de stocker les limites de ces groupes (on ignore les `most_common_vals`, pour lesquelles il y a déjà une mesure plus précise). Le nombre d'`histogram_bounds` est calculé de la même façon que les `most_common_vals` ;
- `correlation` : le facteur de corrélation statistique entre l'ordre physique et l'ordre logique des enregistrements de la colonne. Il vaudra par exemple `1` si les enregistrements sont physiquement stockés dans l'ordre croissant, `-1` si ils sont dans l'ordre décroissant, ou `0` si ils sont totalement aléatoirement répartis. Ceci sert à affiner le coût d'accès aux enregistrements ;
- `most_common_elems` et `most_common_elems_freqs` : les valeurs les plus fréquentes si la colonne est un tableau (NULL dans les autres cas), et leur fréquence. Le nombre de valeurs collecté est au maximum celui indiqué par le paramètre `STATISTICS` de la colonne, ou à défaut par `default_statistics_target` ;
- `elem_count_histogram` : les limites d'histogramme sur la colonne si elle est de type tableau.

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

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

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

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

Quant à `effective_io_concurrency`, il a pour but d'indiquer le nombre d'opérations disques possibles en même temps pour un client (*prefetch*). Le défaut vaut 1. Dans le cas d'un système disque utilisant un RAID matériel, il faut le configurer en fonction du nombre de disques utiles dans le RAID (n s'il s'agit d'un RAID 1 ou RAID 10, n-1 s'il s'agit d'un RAID 5). Avec du SSD, il est possible de monter encore bien au-delà de cette valeur, étant donné la rapidité de ce type de disque. La valeur maximale est de 1000. Cependant, seuls les parcours *Bitmap Scan* sont impactés par la configuration de ce paramètre. (Attention, à partir de la version 13, le principe reste le même, mais la valeur exacte de ce paramètre doit être 2 à 5 fois plus élevée qu'auparavant, selon la formule des notes de version[6] ).

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

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

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

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

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

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

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

Le parcours de l'espace des solutions est un parcours exhaustif. Sa complexité est principalement liée au nombre de jointures de la requête et est de type exponentiel. Par exemple, planifier de façon exhaustive une requête à une jointure dure 200 microsecondes environ, contre 7 secondes pour 12 jointures. Une autre stratégie, l'optimiseur génétique, est donc utilisée pour éviter le parcours exhaustif quand le nombre de jointure devient trop élevé.

Pour plus de détails, voir l'article sur les coûts de planification[7] issu de la base de connaissance Dalibo.

---

### 1.10.2 PARAMÈTRES SUPPLÉMENTAIRES DE L'OPTIMISEUR (1)

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

Tous les paramètres suivants peuvent être modifiés par session.

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

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

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

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

Pour limiter la complexité des plans d'exécution à étudier, il est possible de limiter la quantité de réécriture autorisée par l'optimiseur via les paramètres `from_collapse_limit` et `join_collapse_limit`. Le premier interdit que plus de 8 (par défaut) tables provenant d'une sous-requête ne soient déplacées dans la requête principale. Le second interdit que plus de 8 (par défaut) tables provenant de clauses `JOIN` ne soient déplacées vers la clause `FROM`. Ceci réduit la qualité du plan d'exécution généré, mais permet qu'il soit généré dans un temps raisonnable. Il est fréquent de monter les valeurs à 10 ou un peu au-delà si de longues requêtes impliquent beaucoup de tables.

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

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

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

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

toujours possible de désactiver ce mécanisme en passant `synchronize_seqscans` à `off`.

---

### 1.10.3 PARAMÈTRES SUPPLÉMENTAIRES DE L'OPTIMISEUR (2)

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

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

Le code source de PostgreSQL décrit le principe[8] , résumé aussi dans ce schéma :

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

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

Malgré l'introduction de ces mutations aléatoires, le moteur arrive tout de même à conserver un fonctionnement déterministe. Tant que le paramètre `geqo_seed` ainsi que les autres paramètres contrôlant GEQO restent inchangés, le plan obtenu pour une requête donnée restera inchangé. Il est donc possible de faire varier la valeur de `geqo_seed` pour chercher d'autres plans (voir la documentation officielle[9] ).

---

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

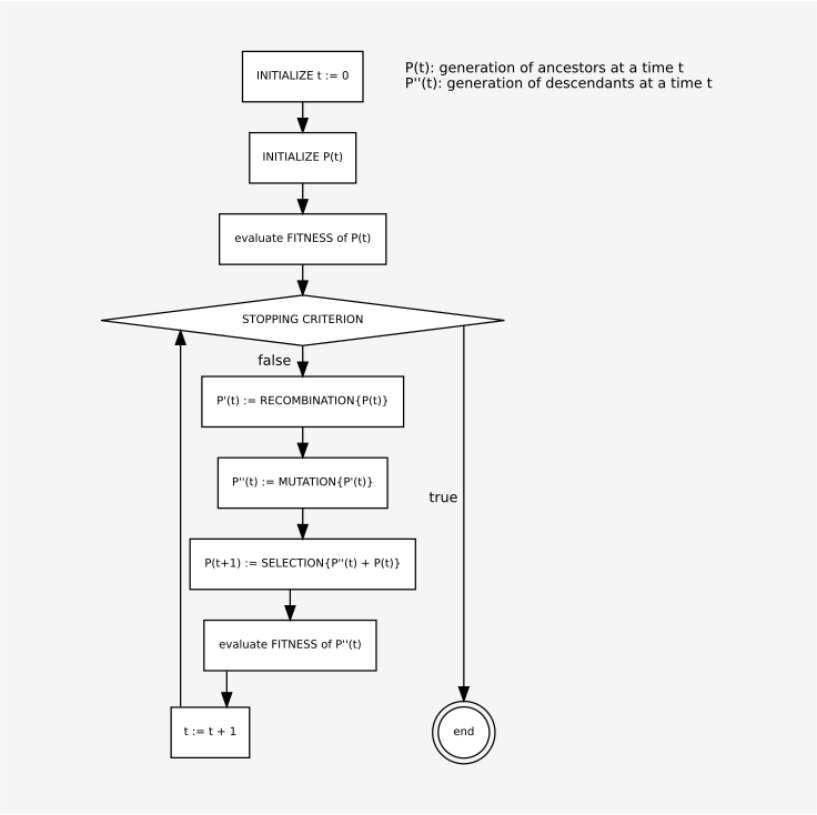



Figure 1: Principe d'un algorithme génétique (schéma de la documentation officielle, licence PostgreSQL)

### 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 :

```
-- 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

### 1.13.1 TABLESPACE

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

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

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

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

### 1.13.2 STATISTIQUES D'ACTIVITÉS, TABLES ET VUES SYSTÈME

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

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

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

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

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

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

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

### 1.13.3 STATISTIQUES SUR LES DONNÉES

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

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

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

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

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

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

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

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

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

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

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

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

## 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/14/data/pg_tblspc/16394/PG_14_202107181/16393/16395
```

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

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

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

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

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

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

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

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

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

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

### 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
```

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

```
# terminal 1
$ psql b0

# terminal 2
$ psql b1
```

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

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

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

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

processus système.

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

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

Ce qui donnerait par exemple :

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

### 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.

## NOTES

## NOTES

## NOTES

# NOS AUTRES PUBLICATIONS

## FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

## LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

## TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

## DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.