Ou comment maintenir ses instances en conditions opérationnelles

# Bonnes pratiques en PostgreSQL

21.02

Dalibo & Contributeurs

https://dalibo.com/formations

# Bonnes pratiques en PostgreSQL

Ou comment maintenir ses instances en conditions opérationnelles

TITRE : Bonnes pratiques en PostgreSQL
SOUS-TITRE : Ou comment maintenir ses instances en conditions opérationnelles

REVISION: 21.02
LICENCE: PostgreSQL

# POSTGRESQL

Figure 1: PostgreSQL

Photographie de Andrew Shiva, licence CC BY 4.0[1] , obtenue sur wikimedia.org[2] .

---

[1] https://creativecommons.org/licenses/by-sa/4.0/
[2] https://commons.wikimedia.org/wiki/File:Elephant_(Loxodonta_Africana)_01.jpg

## L'INTÉRIEUR DE POSTGRESQL

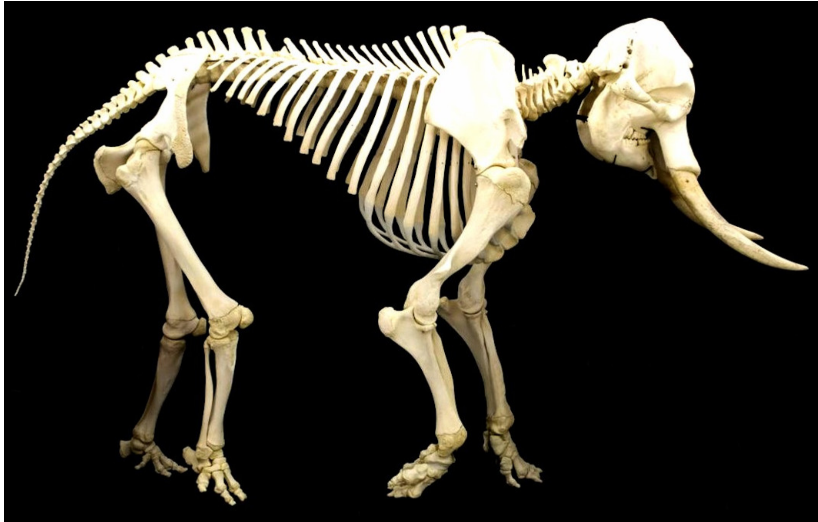


Figure 2: PostgreSQL

Photographie de Sklmsta, domaine publique, obtenue sur wikimedia.org[3] .

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. La supervision
2. L'audit
3. La mise à jour applicative
4. Le contrôle de la fragmentation
5. La sauvegarde
6. L'optimisation SQL

**Bonnes pratiques en PostgreSQL ou comment maintenir ses instances en conditions opérationnelles**

Depuis un ressenti de lenteur par l'utilisateur jusqu'à une corruption des données, un certain nombre de problèmes peuvent se produire sur vos instances PostgreSQL.

[3] https://commons.wikimedia.org/wiki/File:Elephant_skeleton.jpg

Partant d'incidents vécus, ce webinaire présentera un ensemble de méthodes, depuis la supervision jusqu'aux opérations de maintenances en passant par les outils d'audit, vous permettant de garantir un service PostgreSQL disponible et exploitable.

---

## MON ENTREPRISE : DALIBO

- Le spécialiste français de PostgreSQL crée en 2005
- Coopérative proposant des services :
  - de conseil
  - de support
  - de formation
- métier principal : DBA polyvalent
- Site : http://dalibo.com

Équipe : http://dalibo.com/equipe

Projets Open Source : https://github.com/dalibo/

---

## QUI SUIS-JE ?

- Thibaut MADELAINE
- DBA PostgreSQL à Dalibo depuis 4 ans
- Mainteneur de `pitrery` et `sqlserver2pgsql`

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. **La supervision**
2. L'audit
3. La mise à jour applicative
4. Le contrôle de la fragmentation
5. La sauvegarde
6. L'optimisation SQL

---

# LA SUPERVISION

**| sy.pɛʁ.vi.ze |**

« *Se placer au-dessus pour voir, remarquer, prendre des mesures* »

La supervision est la *surveillance du bon fonctionnement d'un système ou d'une activité*.

Elle permet de surveiller, rapporter et alerter les fonctionnements normaux et anormaux des systèmes informatiques.

Elle répond aux préoccupations suivantes :

- technique : surveillance du réseau, de l'infrastructure et des machines ;
- applicative : surveillance des applications et des processus métiers ;
- contrat de service : surveillance du respect des indicateurs contractuels ;
- métier : surveillance des processus métiers de l'entreprise.

---

## OBJECTIFS DE LA SUPERVISION

- Améliorer / mesurer les performances
- Améliorer l'applicatif
- Anticiper / prévenir les incidents
- Réagir vite en cas de crash

Généralement, les administrateurs mettant en place la supervision veulent pouvoir anticiper les problèmes qu'ils soient matériels, de performance, de qualité de service, etc.

Améliorer les performances du SGBD sans connaître les performances globales du système est très difficile. Si un utilisateur se plaint d'une perte de performance, pouvoir corroborer ses dires avec des informations provenant du système de supervision aide à s'assurer qu'il y a bien un problème de performances et peut fréquemment aider à résoudre le problème. De plus, il est important de pouvoir mesurer les gains obtenus après une modification matérielle ou logicielle.

Une supervision des traces de PostgreSQL permet aussi d'améliorer les applications qui utilisent une base de données. Toute requête en erreur est tracée dans les journaux applicatifs, ce qui permet de trouver rapidement les problèmes que les utilisateurs rencontrent.

Un suivi régulier de la volumétrie ou du nombre de connexions permet de prévoir les évolutions nécessaires du matériel ou de la configuration : achat de matériel, création d'index, amélioration de la configuration.

Prévenir les incidents peut se faire en ayant une sonde de supervision des erreurs disques par exemple. La supervision permet aussi d'anticiper les problèmes de configuration. Par

exemple, surveiller le nombre de sessions ouvertes sur PostgreSQL permet de s'assurer que ce nombre n'approche pas trop du nombre maximum de sessions configuré avec le paramètre `max_connections` dans le fichier `postgresql.conf`.

Enfin, une bonne configuration de la supervision implique d'avoir configuré finement la gestion des traces de PostgreSQL. Avoir un bon niveau de trace (autrement dit ni trop ni pas assez) permet de réagir rapidement après un crash.

---

## ACTEURS CONCERNÉS

- Administrateur de bases de données
  - surveillance, performance
  - mise à jour
- Administrateur système
  - surveillance, qualité de service
- Développeur
  - correction et optimisation de requêtes

Il y a trois types d'acteurs concernés par la supervision.

L'administrateur de bases de données a besoin de surveiller les bases pour s'assurer de la qualité de service, pour garantir les performances et pour réagir rapidement en cas de problème. Il doit aussi faire les mises à jour mineures dès qu'elles sont disponibles.

L'administrateur système doit s'assurer de la présence du service. Il doit aussi s'assurer que le service dispose des ressources nécessaires, en termes de processeur (donc de puissance de calcul), de mémoire et de disque (notamment pour la place disponible).

Enfin, le développeur doit pouvoir visualiser l'activité de la base de données. Il peut ainsi comprendre l'impact du code applicatif sur la base. De plus, le développeur est intéressé par la qualité des requêtes que son code exécute. Donc des traces qui ramènent les requêtes en erreur et celles qui ne sont pas performantes sont essentielles pour ce profil.

---

## INDICATEURS CÔTÉ SYSTÈME D'EXPLOITATION

- Charge CPU
- Entrées/sorties disque
- Espace disque
- Sur-activité et inactivité du serveur
- Temps de réponse

Voici quelques exemples d'indicateurs intéressants à superviser pour la partie du système d'exploitation.

La charge CPU (processeur) est importante. Elle peut expliquer pourquoi des requêtes, auparavant rapides, deviennent lentes. Cependant, la suractivité comme la non-activité sont un problème. En fait, si le service est tombé, le serveur sera en sous-activité, ce qui est un excellent indice.

Les entrées/sorties disque permettent de montrer un souci au niveau du système disque : soit PostgreSQL écrit trop à cause d'une mauvaise configuration des journaux de transactions, soit les requêtes exécutées utilisent des fichiers temporaires pour trier les données, ou pour une toute autre raison.

L'espace disque est essentiel à surveiller. PostgreSQL ne propose rien pour cela, il faut donc le faire au niveau système. L'espace disque peut poser problème s'il manque, surtout si cela concerne la partition des journaux de transactions.

Il est possible aussi d'utiliser une requête étalon dont la durée d'exécution sera testée de temps à autre pour détecter les moments problématiques sur le serveur.

---

## INDICATEURS CÔTÉ BASE DE DONNÉES

- Nombre de connexions
- Volumétries
- Requêtes lentes et/ou fréquentes
- Nombre de transactions par seconde
- Ratio d'utilisation du cache
- Retard de réplication

Il existe de nombreux indicateurs intéressants sur les bases :

- nombre de connexions : en faisant par exemple la différence entre connexions inactives, actives, en attente de verrous,
- nombre de requêtes lentes et / ou fréquentes,
- nombre de transactions par seconde

- volumétrie : en taille, en nombre de lignes,
- ratio de lecture du cache (souvent appelé *hit ratio*)
- retard de réplication
- nombre de parcours séquentiels et de parcours d'index
- etc.

---

## CHECK_PGACTIVITY

- Script de monitoring PostgreSQL pour Nagios
  - nombreuses sondes spécifiques à PostgreSQL
  - nombreuses métriques remontées
- Permet de faire la liste des points d'observation possibles
- https://github.com/OPMDG/check_pgactivity

Le script de monitoring `check_pgactivity` permet d'intégrer la supervision de bases de données PostgreSQL dans un système de supervision piloté par Nagios (entre autres).

La supervision d'un serveur PostgreSQL passe par la surveillance de sa disponibilité, des indicateurs sur son activité, l'identification des besoins de maintenance, et le suivi de la réplication le cas échéant. Voici l'ensemble des sondes `check_pgactivity` qu'il est possible de mettre en place sur ces différents aspects.

**Disponibilité** :

- `connection` : réalise un test de connexion pour vérifier que le serveur est accessible ;
- `backends` : compte le nombre de connexions au serveur comparé au paramètre `max_connections` ;
- `backend_status` : permet d'obtenir des statistiques plus précises sur l'état des connexions clientes et d'être alerté lorsqu'un certain nombre de connexions clientes sont dans un état donné (*waiting*, *idle in transaction*…) ;
- `uptime` : détecte un redémarrage du serveur ou du rechargement de la configuration.

**Vacuum** :

- `autovacuum` : suit le fonctionnement de l'autovacuum et des tâches en cours (`VACUUM`, `ANALYZE`, `FREEZE`…) ;
- `table_bloat` : vérifie le volume de données « mortes » et la fragmentation des tables ;
- `btree_bloat` : vérifie le volume de données « mortes » et la fragmentation des index - par rapport à `check_postgres`, le calcul est séparé entre tables et index ;

- `last_analyze` : vérifie si le dernier analyze (relevé des statistiques relatives aux calculs des plans d'exécution) est trop ancien ;
- `last_vacuum` : vérifie si le dernier vacuum (relevé des espaces réutilisables dans les tables) est trop ancien.

**Activité** :

- `locks` : permet d'obtenir des statistiques plus détaillées sur les verrous obtenus et tient notamment compte des spécificités des *predicate locks* du niveau d'isolation `SERIALIZABLE` ;
- `wal_files` : compte le nombre de segments du journal de transaction présents dans le répertoire `pg_wal` ;
- `longest_query` : permet d'être alerté si une requête est en cours d'exécution depuis plus d'un certain temps ;
- `oldest_xact` : permet d'être alerté si une transaction est ouverte depuis un certain temps sans être utilisée ;
- `oldest_2pc` : calcule l'âge de la plus ancienne transaction préparée (*two-phase commit transaction*) ;
- `bgwriter` : permet de collecter des données de performance des différents processus d'écritures de PostgreSQL ;
- `hit_ratio` : calcule le *hit ratio* (utilisation du cache de PostgreSQL) ;
- `commit_ratio` : calcule la proportion de `COMMIT` et `ROLLBACK` ;
- `checksum_errors` : détecte l'apparition d'erreurs de sommes de contrôle (à partir de PostgreSQL 12) ;
- `database_size` : suit la volumétrie des bases et leurs variations ;
- `max_freeze_age` : calcule l'âge des plus vieilles lignes stockées dans chaque base pour suivre le bon passage des `VACUUM FREEZE` ;
- `stat_snapshot_age` : calcule l'âge des statistiques d'activité pour repérer un blocage du collecteur ;
- `temp_files` : suivi des fichiers temporaires.

**Configuration** :

- `configuration` : permet de vérifier que les principaux paramètres mémoire n'ont pas leur valeur par défaut ;
- `minor_version` : détecte les instances n'ayant pas la dernière version mineure ;
- `settings` : repère un changement des paramètres ;
- `invalid_indexes` : repérer tout index invalide ;
- `pgdata_permission` : vérifie les droits sur `PGDATA` pour éviter un blocage au redémarrage ;
- `table_unlogged` : remonte le nombre de tables *unlogged*.

**Réplication & archivage** :

- `archiver` : compte le nombre de segments du journal de transaction en attente d'archivage ;
- `archive_folder` : vérifie qu'il n'y a pas de journal manquant dans les archives de sauvegarde PITR ;
- `hot_standby_delta` : calcule le délai de réplication entre un serveur primaire et un serveur secondaire ;
- `is_master` / `is_hot_standby` : vérifie que l'instance est bien démarrée en lecture/écriture, ou une instance secondaire ;
- `is_replay_paused` : vérifie si la réplication est en pause ;
- `replication_slots` : calcule la volumétrie conservée pour chaque slot de réplication.

**Sauvegarde physique et logique** :

- `backup_label_age` : calcule l'âge du fichier `backup_label` (sauvegardes PITR exclusives) ;
- `pg_dump_backup` : contrôle l'âge et la variation de taille des sauvegardes logiques.

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. La supervision
2. **L'audit**
3. La mise à jour applicative
4. Le contrôle de la fragmentation
5. La sauvegarde
6. L'optimisation SQL

---

## L'AUDIT

- Objectifs :
  - Approfondir la compréhension de notre instance
  - Repérer les points de contentions
  - Trouver les requêtes posant problème
- Quand ?
  - En test et validation
  - Lors de dysfonctionnements

---

## LES OUTILS D'AUDIT

- Les graphiques de métrologie
- Des outils dédiés :
  - pgCluu
  - pgBadger
  - temBoard

---

### PGCLUU

- Outils de collectes de métriques de performances
  - Dépôt github[4]
  - génère un rapport HTML complet
- Différents aspects mesurés :
  - informations sur le système
  - consommation des ressources CPU, RAM, I/O
  - utilisation de la base de données

---

[4] https://github.com/darold/pgcluu

---

## PGBADGER

- Script Perl
- site officiel : https://pgbadger.darold.net/
- Traite les journaux applicatifs de PostgreSQL
- Génére un rapport HTML très détaillé

pgBadger est un script Perl écrit par Gilles Darold. Il s'utilise en ligne de commande : il suffit de lui fournir le ou les fichiers de traces à analyser et il rend un rapport HTML sur les requêtes exécutées, sur les connexions, sur les bases, etc. Le rapport est très complet, il peut contenir des graphes zoomables.

C'est certainement le meilleur outil actuel de rétro-analyse d'un fichier de traces PostgreSQL.

Le site web de pgBadger se trouve sur https://pgbadger.darold.net/

---

## CONFIGURER POSTGRESQL POUR PGBADGER

- Utilisation des traces applicatives
- Où tracer ?
- Quel niveau de traces ?
- Tracer les requêtes
- Tracer certains comportements

Il est essentiel de bien configurer PostgreSQL pour que les traces ne soient pas en même temps trop nombreuses pour ne pas être submergées par les informations et trop peu pour ne pas savoir ce qu'il se passe. Un bon dosage du niveau des traces est important. Savoir où envoyer les traces est tout aussi important.

---

## CONFIGURATION MINIMALE

- traces en anglais
  - `lc_messages='C'`
- ajouter le plus d'information possible
  - `log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '`

Les traces sont enregistrées dans la locale par défaut du serveur. Avoir des traces en français peut présenter certains intérêts pour les débutants mais cela présente plusieurs gros inconvénients. Chercher sur un moteur de recherche avec des traces en français donnera beaucoup moins de résultats qu'avec des traces en anglais.

De plus, les outils d'analyse automatique des traces se basent principalement sur des traces en anglais. Donc, il est vraiment préférable d'avoir les traces en anglais. Cela peut se faire ainsi :

```
lc_messages = 'C'
```

Lorsque la destination des traces est `syslog` ou `eventlog`, elles se voient automatiquement ajouter quelques informations dont un horodatage, essentiel. Lorsque la destination est `stderr`, ce n'est pas le cas. Par défaut, l'utilisateur se retrouve avec des traces sans horodatage, autrement dit des traces inutilisables. PostgreSQL propose donc le paramètre `log_line_prefix` qui permet d'ajouter un préfixe à une trace.

Ce préfixe peut contenir un grand nombre d'informations, comme un horodatage, le PID du processus serveur, le nom de l'application cliente, le nom de l'utilisateur, le nom de la base. Un paramétrage possible est le suivant :

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
```

---

### TRACER CERTAINS COMPORTEMENTS

- `log_connections`, `log_disconnections`
- `log_autovacuum_min_duration`
- `log_checkpoints`
- `log_lock_waits`
- `log_temp_files`

En dehors des erreurs et des durées des requêtes, il est aussi possible de tracer certaines activités ou comportements.

Quand on souhaite avoir une trace de qui se connecte, il est intéressant de pouvoir tracer les connexions et, parfois aussi, les déconnexions. En activant les paramètres `log_connections` et `log_disconnections`, nous obtenons les heures de connexions, de déconnexions et la durée de la session.

`log_autovacuum_min_duration` correspond à `log_min_duration_statement`, mais pour l'autovacuum. Son but est de tracer l'activité de l'autovacuum si son exécution demande plus d'un certain temps.

`log_checkpoints` permet de tracer l'activité des checkpoints. Cela ajoute un message dans les traces pour indiquer qu'un checkpoint commence et une autre quand il termine. Cette deuxième trace est l'occasion d'ajouter des statistiques sur le travail du checkpoint :

```
2021-02-01 13:34:17 CEST LOG: checkpoint starting: xlog
2021-02-01 13:34:20 CEST LOG: checkpoint complete: wrote 13115 buffers (80.0%);
                              0 transaction log file(s) added, 0 removed,
                              0 recycled; write=3.007 s, sync=0.324 s,
                              total=3.400 s; sync files=16,
                              longest=0.285 s,
                              average=0.020 s; distance=404207 kB,
                              estimate=404207 kB
```

Le message indique donc en plus le nombre de blocs écrits sur disque, le nombre de journaux de transactions ajoutés, supprimés et recyclés. Il est rare que des journaux soient ajoutés, ils sont plutôt recyclés. Des journaux sont supprimés quand il y a eu une très grosse activité qui a généré plus de journaux que d'habitude. Les statistiques incluent aussi la durée des écritures, de la synchronisation sur disque, la durée totale, etc.

Le paramètre `log_lock_waits` permet de tracer une attente trop importante de verrous. En fait, quand un verrou est en attente, un chronomètre est déclenché. Lorsque l'attente dépasse la durée indiquée par le paramètre `deadlock_timeout`, un message est enregistré, comme dans cet exemple :

```
2021-02-01 13:38:40 CEST LOG:  process 15976 still waiting for
                               AccessExclusiveLock on relation 26160 of
                               database 16384 after 1000.123 ms
2021-02-01 13:38:40 CEST STATEMENT:  DROP TABLE t1;
```

Plus ce type de message apparaît dans les traces, plus des contentions ont lieu sur le serveur, ce qui peut diminuer fortement les performances.

Le paramètre `log_temp_files` permet de tracer toute création de fichiers temporaires, comme ici :

```
2021-02-01 13:41:11 CEST LOG:  temporary file: path
                               "base/pgsql_tmp/pgsql_tmp15617.1",
                               size 59645952
```

Tout fichier temporaire demande des écritures disques. Ces écritures peuvent poser problème pour les performances globales du système. Il est donc important de savoir si des fichiers temporaires sont créés ainsi que leur taille.

---

## TRACER LES REQUÊTES

- `log_min_duration_statement`
- en production, trace les requêtes longues
  - 10000 pour les requêtes de plus de 10 secondes
- pour un audit, à 0 : trace toutes les requêtes

Le paramètre `log_min_duration_statement`, trace toute requête dont la durée d'exécution dépasse la valeur du paramètre (l'unité est la milliseconde). Il trace aussi la durée d'exécution des requêtes tracées. Par exemple, avec une valeur de 500, toute requête dont la durée d'exécution dépasse 500 ms sera tracée. À 0, toutes les requêtes se voient tracées. Pour désactiver la trace, il suffit de mettre la valeur -1 (qui est la valeur par défaut).

Suivant la charge que le système va subir à cause des traces, il est possible de configurer finement la durée à partir de laquelle une requête est tracée. Cependant, il faut bien comprendre que plus la durée est importante, plus la vision des performances est partielle. Il est parfois plus intéressant de mettre 0 ou une très petite valeur sur une petite durée, qu'une grosse valeur sur une grosse durée. Cela étant dit, laisser 0 en permanence n'est pas recommandé. Il est préférable de configurer ce paramètre à une valeur plus importante en temps normal pour détecter seulement les requêtes les plus longues et, lorsqu'un audit de la plateforme est nécessaire, passer temporairement ce paramètre à une valeur très basse (0 étant le mieux).

La trace fournie par `log_min_duration_statement` ressemble à ceci :

```
2021-02-01 17:34:03 CEST LOG:  duration: 136.811 ms  statement: insert into t1
                               values (2000000,'test');
```

---

### CRÉATION D'UN RAPPORT PGBADGER

```
$ pgbadger postgresql-13-main.log
```

- Rapport dans le fichier *out.html*
- Très nombreuses options :
  - fichier de sortie : `--outfile`
  - filtrage par date : `--begin`, `--end`
  - autres filtrages : `--dbname`, `--dbuser`, `--appname`, ...

La façon la plus simple pour créer un rapport pgBadger est de simplement indiquer au script le fichier de traces de PostgreSQL à analyser.

Il existe énormément d'options. L'aide fournie sur le site web officiel[5] les cite intégralement. Il serait difficile de les citer ici, des options étant ajoutées très fréquemment.

A noter un mode de fonctionnement incrémental. Combiné au format binaire, il permet de parser régulièrement les fichiers de traces applicatives de PostgreSQL. Puis, de générer des rapports HTML à la demande.

On peut ainsi créer un fichier chaque heure :

```
pgbadger --last-parsed .pgbadger_last_state -o YY_MM_DD_HH.bin postgresql.log
```

On pourra créer un rapport en précisant les fichiers binaires voulus :

```
pgbadger -o rapport_2021_02_01.html 2021_02_01_**.bin
```

---

[5] http://pgbadger.darold.net/documentation.html#SYNOPSIS

## TEMBOARD - POSTGRESQL REMOTE CONTROL

- Multi-instances
- Surveillance OS / PostgreSQL
- Suivi de l'activité
- Configuration de chaque instance
- Historisation des requêtes `pg_stat_statements`

temBoard est un outil permettant à un DBA de mener à bien la plupart de ses tâches courantes.

Le serveur web est installé de façon centralisée et un agent est déployé pour chaque instance.

- Adresse: https://github.com/dalibo/temboard
- Version: 7.6
- Licence: PostgreSQL
- Notes: Serveur sur Linux, client web

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. La supervision
2. L'audit
3. **La mise à jour applicative**
4. Le contrôle de la fragmentation
5. La sauvegarde
6. L'optimisation SQL

---

## LA MISE À JOUR APPLICATIVE

- Des nouvelles versions logicielles régulières
- Apportent :
  - Des innovations
  - Des améliorations
  - Des corrections de dysfonctionnement
  - Des corrections de failles de sécurités

La communauté est très dynamique. Des modifications sont apportées tous les jours. Elles apportent de nouvelles fonctionnalités, des corrections de bugs ou encore corrigent des problèmes de sécurité. Elles sont fournies au travers de livraisons régulières de nouvelles versions logicielles.

---

## UNE NUMÉROTATION PAS SIMPLE...

- ... qui se soigne
  - Avant la version 10
    - X.Y : version majeure (8.4, 9.6)
    - X.Y.Z : version mineure (9.6.19)
  - Après la version 10
    - X : version majeure (10, 11, 12, 13)
    - X.Y : version mineure (12.4)

---

## VERSIONS MAJEURES

- Apporte des nouvelles fonctionnalités
- Nécessite une migration des données
- Une sortie tous les 12 à 15 mois
- Maintenue environ 5 ans par la communauté
- Les versions majeures supportées : 9.6, 10, 11, 12 et 13

La philosophie générale des développeurs de PostgreSQL peut se résumer ainsi :

« Notre politique se base sur la qualité, pas sur les dates de sortie. »

Toutefois, même si cette philosophie reste très présente parmi les développeurs, depuis quelques années, les choses évoluent et on constante la livraison d'une version stable majeure tous les 12 à 15 mois, tout en conservant la qualité des versions. De ce fait, toute fonctionnalité supposée pas suffisamment stable est repoussée à la version suivante.

La tendance actuelle est de garantir un support pour chaque version courante pendant une durée minimale de 5 ans. Ainsi n'est plus supportée, la 9.3 depuis novembre 2018, la 9.4 depuis février 2020, la 9.5 depuis février 2021. La prochaine version qui subira ce sort est la 9.6 en novembre 2021. Le support de la version 13 devrait durer jusqu'en 2025.

Pour plus de détails :

- Politique de versionnement[6] ;
- Roadmap des versions mineures[7] .

---

## MISE À JOUR MAJEURE

- Bien lire les *Release Notes*
- Bien tester l'application avec la nouvelle version
  - rechercher les régressions en terme de fonctionnalités et de performances
  - penser aux extensions et aux outils
- Pour mettre à jour
  - mise à jour des binaires
  - et mise à jour/traitement des fichiers de données

Faire une mise à jour majeure est une opération complexe à préparer prudemment.

La première action là-aussi est de lire les *Release Notes* pour bien prendre en compte les régressions potentielles en terme de fonctionnalités et/ou de performances. Cela n'arrive presque jamais mais c'est possible malgré toutes les précautions mises en place.

La deuxième action est de mettre en place un serveur de tests où se trouve la nouvelle version de PostgreSQL avec les données de production. Ce serveur sert à tester PostgreSQL mais aussi, et même surtout, l'application. Le but est de vérifier encore une fois les régressions possibles.

N'oubliez pas de tester les extensions non officielles, voire développées en interne, que vous avez installées. Elles sont souvent moins bien testées.

N'oubliez pas non plus de tester les outils d'administration, de monitoring, de modélisation. Ils nécessitent souvent une mise à jour pour être compatibles avec la nouvelle version installée.

Une fois que les tests sont concluants, arrive le moment de la mise en production. C'est une étape qui peut être longue car les fichiers de données doivent être traités. Il existe

[6] https://www.postgresql.org/support/versioning/

[7] https://www.postgresql.org/developer/roadmap/



DALIBO

plusieurs méthodes pour la mettre en oeuvre :

- dump / restore logique
- `pg_upgrade`
- réplication logique interne
- réplication logique externe (Slony)

---

### VERSIONS NON SUPPORTÉES

- Les versions 9.5 et inférieures : à migrer au plus vite !
- Les versions 9.6 et 10 : projet de migration à lancer

---

### VERSIONS MINEURES

- Corrections de bugs ou de failles de sécurité
- Sortie au moins trimestrielle dans toutes les versions majeures supportées
- Dernières releases (11 février 2021) :
  - 13.2, 12.6, 11.11, 10.16 et 9.6.21

Une version mineure ne comporte que des corrections de bugs ou de failles de sécurité. Les publications de versions mineures sont plus fréquentes que celles de versions majeures, avec un rythme de sortie trimestriel, sauf bug majeur ou faille de sécurité. Chaque bug est corrigé dans toutes les versions stables actuellement maintenues par le projet.

---

## MISE À JOUR MINEURE

- Méthode :
  - arrêter PostgreSQL
  - mettre à jour les binaires
  - redémarrer PostgreSQL
- Pas besoin de s'occuper des données, sauf cas exceptionnel
  - bien lire les *Release Notes* pour s'en assurer

Faire une mise à jour mineure est simple et rapide.

La première action est de lire les *Release Notes* pour s'assurer qu'il n'y a pas à se préoccuper des données. C'est généralement le cas mais il est préférable de s'en assurer avant qu'il ne soit trop tard.

La deuxième action est de faire la mise à jour. Tout dépend de la façon dont PostgreSQL a été installé :

- par compilation, il suffit de remplacer les anciens binaires par les nouveaux ;
- par paquets précompilés, il suffit d'utiliser le système de paquets `apt` sur Debian et affiliés, `yum` ou `dnf` sur Red Hat et affiliés) ;
- par l'installeur graphique, en le ré-exécutant.

Ceci fait, un redémarrage du serveur est nécessaire. Il est intéressant de noter que les paquets Debian s'occupent directement de cette opération. Il n'est donc pas nécessaire de le refaire.

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. La supervision
2. L'audit
3. La mise à jour applicative
4. **Le contrôle de la fragmentation**
5. La sauvegarde
6. L'optimisation SQL

---

## LE CONTRÔLE DE LA FRAGMENTATION

- Symptôme : baisse régulière des performances
- Raison : fragmentation des tables et des index
- Cause : implémentation du `MVCC`

---

## MULTIVERSION CONCURRENCY CONTROL (MVCC)

- Le « noyau » de PostgreSQL
- Garantit ACID
- Permet les écritures concurrentes sur la même table

MVCC (Multi Version Concurrency Control) est le mécanisme interne de PostgreSQL utilisé pour garantir la cohérence des données lorsque plusieurs processus accèdent simultanément à la même table.

C'est notamment MVCC qui permet de sauvegarder facilement une base *à chaud* et d'obtenir une sauvegarde cohérente alors même que plusieurs utilisateurs sont

potentiellement en train de modifier des données dans la base.

C'est la qualité de l'implémentation de ce système qui fait de PostgreSQL un des meilleurs SGBD au monde : chaque transaction travaille dans son image de la base, cohérent du début à la fin de ses opérations. Par ailleurs les écrivains ne bloquent pas les lecteurs et les lecteurs ne bloquent pas les écrivains, contrairement aux SGBD s'appuyant sur des verrous de lignes. Cela assure de meilleures performances, un fonctionnement plus fluide des outils s'appuyant sur PostgreSQL.

---

## L'IMPLÉMENTATION MVCC DE POSTGRESQL

- Colonnes supplémentaires masquées par défaut :
  - `ctid`
  - `xmin` et `xmax`

---

### `CTID`

- Codée sur 6 octets
  - 4 octets pour la page
  - 2 octets pour la ligne
- Fournit une adresse physique dans une table

La localisation physique de la version de ligne au sein de sa table. Bien que le `ctid` puisse être utilisé pour trouver la version de ligne très rapidement, le `ctid` d'une ligne change si la ligne est actualisée ou déplacée par un `VACUUM FULL`. Le `ctid` est donc inutilisable comme identifiant de ligne sur le long terme.

---

### XMIN ET XMAX (1/4)

Table initiale :

| xmin | xmax | Nom | Solde |
|---|---|---|---|
| 100 | | M. Durand | 1500 |
| 100 | | Mme Martin | 2200 |

PostgreSQL stocke des informations de visibilité dans chaque version d'enregistrement.

- `xmin` : l'identifiant de la transaction créant cette version.
- `xmax` : l'identifiant de la transaction invalidant cette version.

Ici, les deux enregistrements ont été créés par la transaction 100. Il s'agit peut-être, par exemple, de la transaction ayant importé tous les soldes à l'initialisation de la base.

---

### XMIN ET XMAX (2/4)

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

| xmin | xmax | Nom | Solde |
|---|---|---|---|
| 100 | **150** | M. Durand | 1500 |
| 100 | | Mme Martin | 2200 |
| **150** | | M. Durand | 1300 |

On décide d'enregistrer un virement de 200 € du compte de M. Durand vers celui de Mme Martin. Ceci doit être effectué dans une seule transaction : l'opération doit être atomique, sans quoi de l'argent pourrait apparaître ou disparaître de la table.

Nous allons donc tout d'abord démarrer une transaction (ordre `SQL BEGIN`). PostgreSQL fournit donc à notre session un nouveau numéro de transaction (150 dans notre exemple). Puis nous effectuerons :

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

---

### XMIN ET XMAX (3/4)

```
UPDATE soldes SET solde=solde+200 WHERE nom = 'Mme Martin';
```

| xmin | xmax | Nom | Solde |
|---|---|---|---|
| 100 | 150 | M. Durand | 1500 |
| 100 | **150** | Mme Martin | 2200 |
| 150 | | M. Durand | 1300 |
| **150** | | Mme Martin | 2400 |

Puis nous effectuerons :

```
UPDATE soldes SET solde=solde+200 WHERE nom = 'Mme Martin';
```

Nous avons maintenant deux versions de chaque enregistrement.

Notre session ne voit bien sûr plus que les nouvelles versions de ces enregistrements, sauf si elle décidait d'annuler la transaction, auquel cas elle reverrait les anciennes données.

Pour une autre session, la version visible de ces enregistrements dépend de plusieurs critères :

- La transaction 150 a-t-elle été validée ? Sinon elle est invisible
- La transaction 150 est-elle *postérieure* à la nôtre (numéro supérieur au notre), et sommes-nous dans un niveau d'isolation (*serializable*) qui nous interdit de voir les modifications faites depuis le début de notre transaction ?
- La transaction 150 a-t-elle été validée après le démarrage de la requête en cours ? Une requête, sous PostgreSQL, voit un instantané cohérent de la base, ce qui implique que toute transaction validée après le démarrage de la requête doit être ignorée.

Dans le cas le plus simple, 150 ayant été validée, une transaction 160 ne verra pas les premières versions : `xmax` valant 150, ces enregistrements ne sont pas visibles. Elle verra les secondes versions, puisque `xmin`=150, et pas de `xmax`.

---

## XMIN ET XMAX (4/4)

| xmin | xmax | Nom | Solde |
|---|---|---|---|
| 100 | 150 | M. Durand | 1500 |
| 100 | **150** | Mme Martin | 2200 |
| 150 | | M. Durand | 1300 |
| **150** | | Mme Martin | 2400 |

- Comment est effectuée la suppression d'un enregistrement ?
- Comment est effectuée l'annulation de la transaction 150 ?

- La suppression d'un enregistrement s'effectue simplement par l'écriture d'un `xmax` dans la version courante.
- Il n'y a rien à écrire dans les tables pour annuler une transaction. Il suffit de marquer la transaction comme étant annulée dans la `CLOG`.

---

## AVANTAGES DU MVCC POSTGRESQL

- Avantages :
  - avantages classiques de MVCC (concurrence d'accès)
  - implémentation simple et performante
  - peu de sources de contention
  - verrouillage simple d'enregistrement
  - rollback instantané
  - données conservées aussi longtemps que nécessaire

- Les lecteurs ne bloquent pas les écrivains, ni les écrivains les lecteurs.
- Le code gérant les instantanés est simple, ce qui est excellent pour la fiabilité, la maintenabilité et les performances.
- Les différentes sessions ne se gênent pas pour l'accès à une ressource commune (l'`UNDO`).
- Un enregistrement est facilement identifiable comme étant verrouillé en écriture : il suffit qu'il ait une version ayant un `xmax` correspondant à une transaction en cours.
- L'annulation est instantanée : il suffit d'écrire le nouvel état de la transaction dans la `clog`. Pas besoin de restaurer les valeurs précédentes, elles redeviennent automatiquement visibles.
- Les anciennes versions restent en ligne aussi longtemps que nécessaire. Elles ne pourront être effacées de la base qu'une fois qu'aucune transaction ne les considérera comme visibles.

---

## INCONVÉNIENTS DU MVCC POSTGRESQL

- Inconvénients :
  - Tables plus volumineuses
  - Pas de visibilité dans les index
  - Nettoyage des enregistrements (`VACUUM`)

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

- Il faut nettoyer les tables de leurs enregistrements morts. C'est le travail de la commande `VACUUM`. On peut aussi voir ce point comme un avantage : contrairement à la solution `UNDO`, ce travail de nettoyage n'est pas effectué par le client faisant des mises à jour (et créant donc des enregistrements morts). Le ressenti est donc meilleur.

- Les tables sont forcément plus volumineuses que dans l'implémentation par `UNDO`, pour deux raisons :
  - Les informations de visibilité qui y sont stockées. Il y a un surcoût d'une douzaine d'octets par enregistrement.
  - Il y a toujours des enregistrements morts dans une table, une sorte de *fond de roulement*, qui se stabilise quand l'application est en régime stationnaire. Ces enregistrements sont recyclés à chaque passage de `VACUUM`.
- Les index n'ont pas d'information de visibilité. Il est donc nécessaire d'aller vérifier dans la table associée que l'enregistrement trouvé dans l'index est bien visible. Cela a un impact sur le temps d'exécution de requêtes comme `SELECT count(*)` sur une table : dans le cas le plus défavorable, il est nécessaire d'aller visiter tous les enregistrements pour s'assurer qu'ils sont bien visibles. La *visibility map* permet de limiter cette vérification aux données les plus récentes.
- Le `VACUUM` ne s'occupe pas de l'espace libéré par des colonnes supprimées (fragmentation verticale).

---

### FONCTIONNEMENT DE VACUUM (1/3)

Le traitement `VACUUM` se déroule en trois passes. Cette première passe parcourt la table à nettoyer, à la recherche d'enregistrements morts. Un enregistrement est mort s'il possède un `xmax` qui correspond à une transaction validée, et que cet enregistrement n'est plus visible dans l'instantané d'aucune transaction en cours sur la base.

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

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

---

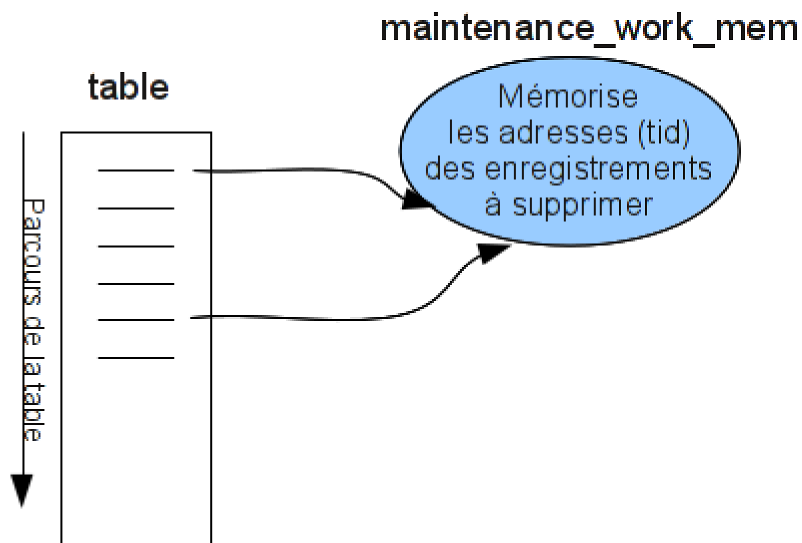



Figure 3: Algorithme du vacuum 1/3

### FONCTIONNEMENT DE VACUUM (2/3)

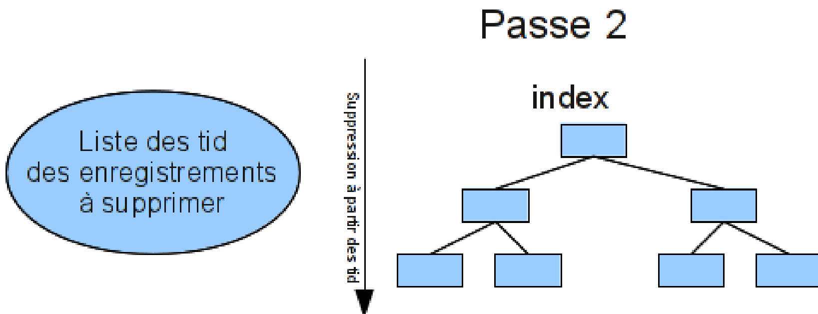



Figure 4: Algorithme du vacuum 2/3

La seconde passe se charge de nettoyer les entrées d'index. Vacuum possède une liste de `tid` à invalider. Il parcourt donc tous les index de la table à la recherche de ces tid et les supprime. En effet, les index sont triés afin de mettre en correspondance une valeur de clé (la colonne indexée par exemple) avec un `tid`. Il n'est par contre pas possible de trouver un `tid` directement. Les pages entièrement vides sont supprimées de l'arbre et stockées dans la liste des pages réutilisables, la **Free Space Map** (`FSM`).

---

### FONCTIONNEMENT DE VACUUM (3/3)

Maintenant qu'il n'y a plus d'entrée d'index pointant sur les enregistrements identifiés, nous pouvons supprimer les enregistrements de la table elle-même. C'est le rôle de cette passe, qui quant à elle, peut accéder directement aux enregistrements. Quand un enregistrement est supprimé d'un bloc, ce bloc est réorganisé afin de consolider l'espace libre, et cet espace libre est consolidé dans la **Free Space Map** (`FSM`).

Une fois cette passe terminée, si le parcours de la table n'a pas été terminé lors de la passe 1 (la `maintenance_work_mem` était pleine), le travail reprend où il en était du parcours de la table.

---

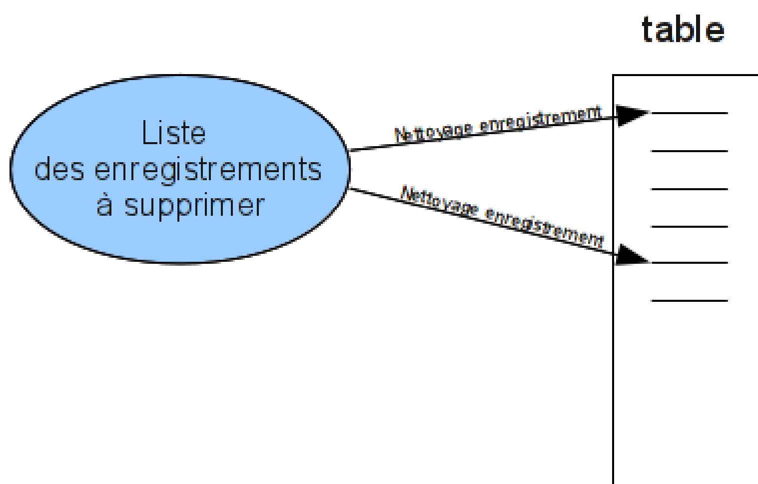



Figure 5: Algorithme du vacuum 3/3

## LUTTER CONTRE LA FRAGMENTATION

- Supervision de la fragmentation
- Réglage de l'autovacuum
- Reconstruction régulière des index

---

## RÉGLAGE DE L'AUTOVACUUM

- Déclenchement d'un vacuum si :

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

- Paramètres :
  - `autovacuum_vacuum_threshold` : 50
  - `autovacuum_vacuum_scale_factor` : 20 %

Depuis la version 13 de PostgreSQL, deux nouveaux paramètres on fait leur apparition :

- `autovacuum_vacuum_insert_threshold` : 1000
- `autovacuum_vacuum_insert_scale_factor` : 20 %

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

- `VACUUM` :
  - `autovacuum_vacuum_threshold` : nombre minimum d'enregistrements devant re morts (mis à jour ou supprimés) dans la table avant de déclencher un VACUUM' (50 par défaut).
  - `autovacuum_vacuum_scale_factor` : fraction du nombre d'enregistrements de table avant de déclencher un `VACUUM` (0,2 par défaut, soit 20 %).
  - Un `VACUUM` est donc déclenché si :
  - `autovacuum_vacuum_insert_threshold` : nombre minimum d'enregistrements vant être insérés dans la table avant de déclencher un `VACUUM` (1000 par faut).
  - `autovacuum_vacuum_insert_scale_factor` : fraction du nombre d'enregistrements de table avant de déclencher un `VACUUM` (0,2 par défaut, soit 20 %).
  - Un `VACUUM` est donc déclenché si :

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

+ ou si :

nb_enregistrements_insérés (n_ins_since_vacuum) >=
    autovacuum_vacuum_insert_threshold + nb_enregs × autovacuum_vacuum_insert_scale_fa
```

## ADAPTATION DE L'AUTOVACUUM

- Déconseillé de modifier le paramétrage de façon globale
- Adaptation conseillée :
  - table par table
  - suivant le nombre d'enregistrements

```
ALTER TABLE table_name SET (autovacuum_vacuum_scale_factor = 0.05);
```

Le paramétrage par défaut, s'il est pertinent pour des tables de taille moyenne, induisent un volume de fragmentation très important quand une table comporte plusieurs millions de lignes.

Pour une table de 100 mille lignes, un VACUUM se déclenchera au bout de 20 mille modifications. Pour une table de 200 millions de lignes, l'opération se déclenchera suite à 40 millions de modifications.

Il n'est pas conseillé de modifier ce paramétrage globalement. Cependant une configuration spécifique pour chaque table pourra être réalisée. Nous pourrions utiliser par exemple un abaque tel que donné dans le tableau suivant :

| *autovacuum* | NbL < 1 million | NbL >= 1 million | NbL >= 5 millions | NbL >= 10 millions |
|---|---|---|---|---|
| _vacuum_scale_factor | 0.2 | 0.1 | 0.05 | 0.0 |
| _vacuum_threshold | (défaut) | (défaut) | (défaut) | 500000 |

## RECONSTRUCTION DES INDEX

- Lancer REINDEX régulièrement permet :
  - de gagner de l'espace disque
  - d'améliorer les performances
  - de réparer un index corrompu
- VACUUM ne provoque pas de réindexation
- VACUUM FULL réindexe
- Option CONCURRENTLY à partir de la version 12

REINDEX reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. La même commande peut réindexer tous les index d'une table :

```
REINDEX INDEX nomindex;
REINDEX (VERBOSE) TABLE nomtable;
```

Les pages d'index qui sont devenues complètement vides sont récupérées pour être réutilisées. Il existe toujours la possibilité d'une utilisation inefficace de l'espace : même s'il ne reste qu'une clé d'index dans une page, la page reste allouée. La possibilité d'inflation n'est pas indéfinie mais il serait toujours utile de planifier une réindexation périodique pour les index fréquemment modifiés.

De plus, pour les index B-tree, un index tout juste construit est plus rapide qu'un index qui a été mis à jour plusieurs fois parce que les pages adjacentes logiquement sont habituellement aussi physiquement adjacentes dans un index nouvellement créé. Cette considération ne s'applique qu'aux index B-tree. Il pourrait être intéressant de ré-indexer périodiquement, simplement pour améliorer la vitesse d'accès.

La réindexation est aussi utile dans le cas d'un index corrompu ne contenant plus de données valides. Deux raisons peuvent amener un index à être invalide. Avant la version 10, les index de type hash n'étaient pas journalisés. Donc un système reconstruit à partir des journaux de transactions aura tous ses index hash invalides. Il faudra les reconstruire pour qu'ils soient utilisables. L'autre raison vient de la clause `CONCURRENTLY` des ordres `CREATE INDEX` / `REINDEX`. Cette clause permet de créer/réindexer un index sans bloquer les écritures dans la table. Si, au bout de deux passes, l'index n'est toujours pas complet, il est considéré comme invalide et doit être soit détruit puis construit, soit reconstruit avec la commande `REINDEX`.

Il est à savoir que l'opération `VACUUM` sans le mode `FULL` ne provoque pas de réindexation. Une réindexation est effectuée dans le cas de l'utilisation du mode `FULL`.

La commande système `reindexdb` peut être utilisée pour réindexer une table, une base ou une instance entière.

---

## MAINTENANCE : SCRIPT DE RÉINDEXATION

- Script plannifié régulièrement
- Recherche des N index les plus fragmentés
- Version 12+ : `REINDEX INDEX index CONCURRENTLY`
- Sinon (hors contrainte unique) :

```
CREATE INDEX CONCURRENTLY index_bis (...);
BEGIN;
DROP INDEX index CONCURRENTLY;
ALTER INDEX index_bis RENAME TO index;
COMMIT;
```

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. La supervision
2. L'audit
3. La mise à jour applicative
4. Le contrôle de la fragmentation
5. **La sauvegarde**
6. L'optimisation SQL

---

## LA SAUVEGARDE

- Opération essentielle pour la sécurisation des données
- PostgreSQL propose différentes solutions
  - de sauvegarde à froid ou à chaud, mais cohérentes
  - des méthodes de restauration partielle ou complète

La mise en place d'une solution de sauvegarde est une des opérations les plus importantes après avoir installé un serveur PostgreSQL. En effet, nul n'est à l'abri d'un bug logiciel, d'une panne matérielle, voire d'une erreur humaine.

Cette opération est néanmoins plus complexe qu'une sauvegarde standard car elle doit pouvoir s'adapter aux besoins des utilisateurs. Quand le serveur ne peut jamais être arrêté, la sauvegarde à froid des fichiers ne peut convenir. Il faudra passer dans ce cas par un outil qui pourra sauvegarder les données alors que les utilisateurs travaillent et qui devra respecter les contraintes ACID pour fournir une sauvegarde cohérente des données.

PostgreSQL va donc proposer des méthodes de sauvegardes à froid (autrement dit serveur arrêté) comme à chaud, mais de toute façon cohérente. Les sauvegardes pourront être partielles ou complètes, suivant le besoin des utilisateurs.

La méthode de sauvegarde dictera l'outil de restauration. Suivant l'outil, il fonctionnera à froid ou à chaud, et permettra même dans certains cas de faire une restauration partielle.

---

## DÉFINIR UNE POLITIQUE DE SAUVEGARDE

- Pourquoi établir une politique ?
- Que sauvegarder ?
- À quelle fréquence sauvegarder les données ?
- Quels supports ?
- Quels outils ?
- Vérifier la restauration des sauvegardes

Afin d'assurer la sécurité des données, il est nécessaire de faire des sauvegardes régulières.

Ces sauvegardes vont servir, en cas de problème, à restaurer les bases de données dans un état le plus proche possible du moment où le problème est survenu.

Cependant, le jour où une restauration sera nécessaire, il est possible que la personne qui a mis en place les sauvegardes ne soit pas présente. C'est pour cela qu'il est essentiel d'écrire et de maintenir un document qui indique la mise en place de la sauvegarde et qui détaille comment restaurer une sauvegarde.

En effet, suivant les besoins, les outils pour sauvegarder, le contenu de la sauvegarde, sa fréquence ne seront pas les mêmes.

Par exemple, il n'est pas toujours nécessaire de tout sauvegarder. Une base de données peut contenir des données de travail, temporaires et/ou faciles à reconstruire, stockées dans des tables standards. Il est également possible d'avoir une base dédiée pour stocker ce genre d'objets. Pour diminuer le temps de sauvegarde (et du coup de restauration), il est possible de sauvegarder partiellement son serveur pour ne conserver que les données importantes.

La fréquence peut aussi varier. Un utilisateur peut disposer d'un serveur PostgreSQL pour un entrepôt de données, serveur qu'il n'alimente qu'une fois par semaine. Dans ce cas, il est inutile de sauvegarder tous les jours. Une sauvegarde après chaque alimentation (donc chaque semaine) est suffisante. En fait, il faut déterminer la fréquence de sauvegarde des données selon :

- le volume de données à sauvegarder ;
- la criticité des données ;
- la quantité de données qu'il est « acceptable » de perdre en cas de problème.

Le support de sauvegarde est lui aussi très important. Il est possible de sauvegarder les données sur un disque réseau (à travers Netbios ou NFS), sur des disques locaux dédiés, sur des bandes ou tout autre support adapté. Dans tous les cas, il est fortement déconseillé de stocker les sauvegardes sur les disques utilisés par la base de données.

Ce document doit aussi indiquer comment effectuer la restauration. Si la sauvegarde est composée de plusieurs fichiers, l'ordre de restauration des fichiers peut être essentiel. De plus, savoir où se trouvent les sauvegardes permet de gagner un temps important, qui évitera une immobilisation trop longue.

De même, vérifier la restauration des sauvegardes de façon régulière est une précaution très utile.

---

## OBJECTIFS

- Sécuriser les données
- Mettre à jour le moteur de données
- Dupliquer une base de données de production
- Archiver les données

L'objectif essentiel de la sauvegarde est la sécurisation des données. Autrement dit, l'utilisateur cherche à se protéger d'une panne matérielle ou d'une erreur humaine (un

utilisateur qui supprimerait des données essentielles). La sauvegarde permet de restaurer les données perdues. Mais ce n 'est pas le seul objectif d'une sauvegarde.

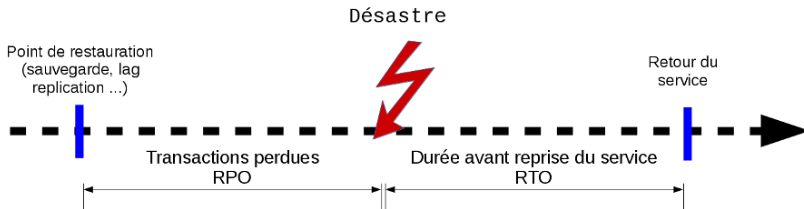
Une sauvegarde peut aussi servir à dupliquer une base de données sur un serveur de test ou de préproduction. Elle permet aussi d'archiver des tables. Cela se voit surtout dans le cadre des tables partitionnées où l'archivage de la table la plus ancienne permet ensuite sa suppression de la base pour gagner en espace disque.

Un autre cas d'utilisation de la sauvegarde est la mise à jour majeure de versions PostgreSQL. Il s'agit de la solution historique de mise à jour (export /import). Historique, mais pas obsolète.

---

### RTO / RPO

- RPO (*Recovery Point Objective*) : Perte de Données Maximale Admissible
- RTO (*Recovery Time Objective*) : Durée Maximale d'Interruption Admissible
- => Permettent de définir la politique de sauvegarde/restauration




Le RPO et RTO sont deux concepts déterminants dans le choix des politiques de sauvegardes.

- RPO faible : Le perte de données admissible est très faible voire nulle, il faudra s'orienter vers des solutions de type :
  - Sauvegarde à chaud
  - PITR
  - Réplication (asynchrone/synchrone).
- RPO important : On s'autorise une perte de données importante, on peut utiliser des solutions de type :
  - Sauvegarde logique (dump)
  - Sauvegarde fichier à froid
- RTO court : Durée d'interruption courte, le service doit vite remonter. Nécessite des procédures avec le moins de manipulations possible et réduisant le nombre

d'acteurs :
  - Réplication
  - Solutions Haut Disponibilité
- RTO long : La durée de reprise du service n'est pas critique on peut utiliser des solutions simple comme :
  - Restauration fichier
  - Restauration sauvegarde logique (dump).

Plus le besoin en RTO/RPO sera court plus les solutions seront complexes à mettre en œuvre. Inversement, pour des données non critiques, un RTO/RPO long permet d'utiliser des solutions simples.

---

## LES SAUVEGARDES / RESTAURATIONS AVEC TINA

- Sauvegarde à chaud avec Time Navigator
- Permet des restaurations à n'importe quel point dans le temps
- Solution puissante… mais complexe

---

## MIEUX COMPRENDRE LA SAUVEGARDE PITR

- *Point In Time Recovery*
- À chaud
- En continu
- Cohérente

PITR est l'acronyme de *Point In Time Recovery*, autrement dit restauration à un point dans le temps.

C'est une sauvegarde à chaud et surtout en continu. Là où une sauvegarde logique du type `pg_dump` se fait au mieux une fois toutes les 24 h, la sauvegarde PITR se fait en continue grâce à l'archivage des journaux de transactions. De ce fait, ce type de sauvegarde diminue très fortement la fenêtre de perte de données.

Bien qu'elle se fasse à chaud, la sauvegarde est cohérente.

---



DALIBO

### WRITE AHEAD LOGS, AKA WAL

- Chaque donnée est écrite **2 fois** sur le disque !
- Sécurité quasiment infaillible
- Comparable à la journalisation des systèmes de fichiers

Les journaux de transactions (appelés parfois WAL ou XLOG) sont une garantie contre les pertes de données.

Il s'agit d'une technique standard de journalisation appliquée à toutes les transactions.

Ainsi lors d'une modification de donnée, l'écriture au niveau du disque se fait en deux temps :

- écriture immédiate dans le journal de transactions ;
- écriture à l'emplacement final lors du prochain `CHECKPOINT`.

Ainsi en cas de crash :

- PostgreSQL redémarre ;
- PostgreSQL vérifie s'il reste des données non intégrées aux fichiers de données dans les journaux (mode *recovery*) ;
- si c'est le cas, ces données sont recopiées dans les fichiers de données afin de retrouver un état stable et cohérent.

**Plus d'information** : http://www.dalibo.org/glmf108_postgresql_et_ses_journaux_de_transactions

---

### AVANTAGES DES WAL

- Un seul *sync* sur le fichier de transactions
- Le fichier de transactions est écrit de manière séquentielle
- Les fichiers de données sont écrits de façon asynchrone
- Point In Time Recovery
- Réplication (*WAL shipping*)

Les écritures se font de façon séquentielle, donc sans grand déplacement de la tête d'écriture. Généralement, le déplacement des têtes d'un disque est l'opération la plus coûteuse. L'éviter est un énorme avantage.

De plus, comme on n'écrit que dans un seul fichier de transactions, la synchronisation sur disque peut se faire sur ce seul fichier, à condition que le système de fichiers le supporte.

L'écriture asynchrone dans les fichiers de données permet là aussi de gagner du temps.

Mais les performances ne sont pas la seule raison des journaux de transactions. Ces journaux ont aussi permis l'apparition de nouvelles fonctionnalités très intéressantes, comme le PITR et la réplication physique.

---

## PRINCIPES DE LA SAUVEGARDE PITR

- Les journaux de transactions (WAL) contiennent toutes les modifications
- Il faut les archiver
- … et avoir une image des fichiers des données à un instant t
- La restauration se fait en restaurant cette image
- … et en rejouant les journaux dans l'ordre

Quand une transaction est validée, les données à écrire dans les fichiers de données sont d'abord écrites dans un journal de transactions. Ces journaux décrivent donc toutes les modifications survenant sur les fichiers de données, que ce soit les objets utilisateurs comme les objets systèmes. Pour reconstruire un système, il suffit donc d'avoir ces journaux et d'avoir un état des fichiers du répertoire des données à un instant t. Toutes les actions effectuées après cet instant t pourront être rejouées en demandant à PostgreSQL d'appliquer les actions contenues dans les journaux. Les opérations stockées dans les journaux correspondent à des modifications physiques de fichiers, il faut donc partir d'une sauvegarde au niveau du système de fichier, un export avec `pg_dump` n'est pas utilisable.

Il est donc nécessaire de conserver ces journaux de transactions. Or PostgreSQL les recycle dès qu'il n'en a plus besoin. La solution est de demander au moteur de les archiver ailleurs avant ce recyclage. On doit aussi disposer de l'ensemble des fichiers qui composent le répertoire des données (incluant les tablespaces si ces derniers sont utilisés).

La restauration a besoin des journaux de transactions archivés. Il ne sera pas possible de restaurer et éventuellement revenir à un point donné avec la sauvegarde seule. En revanche, une fois la sauvegarde des fichiers restaurée et la configuration réalisée pour rejouer les journaux archivés, il sera possible de les rejouer tous ou seulement une partie d'entre eux (en s'arrêtant à un certain moment). Ils doivent impérativement être rejoués dans l'ordre de leur écriture (et donc de leur nom).

---

### POINTS D'ATTENTION

- Sauvegarde de l'instance complète
- Nécessite un grand espace de stockage (données + journaux)
- Risque d'accumulation des journaux en cas d'échec d'archivage
  - …avec arrêt de l'instance si `pg_wal` plein !
- Restauration de l'instance complète

Certains inconvénients viennent directement du fait qu'on copie les fichiers : sauvegarde et restauration complète (impossible de ne restaurer qu'une seule base ou que quelques tables), restauration sur la même architecture (32/64 bits, *little/big endian*). Il est même fortement conseillé de restaurer dans la même version du même système d'exploitation, sous peine de devoir réindexer l'instance (différence de définition des locales notamment).

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

En cas de problème dans l'archivage et selon la méthode choisie, l'instance ne voudra pas effacer les journaux non archivés. Il y a donc un risque d'accumulation de ceux-ci. Il faudra donc surveiller la taille du `pg_wal`. En cas de saturation, PostgreSQL s'arrête !

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

---

## PRÉVENTION DES INCIDENTS LIÉS AU PITR

- Superviser l'espace disques :
  - du FS de l'archivage
  - du FS des données de PostgreSQL
- Savoir où se trouve la procédure de restauration…
  - … et la tester régulièrement !

---

## AU PROGRAMME : QUELQUES BONNES PRATIQUES

1. La supervision
2. L'audit
3. La mise à jour applicative
4. Le contrôle de la fragmentation
5. La sauvegarde
6. **L'optimisation SQL**

---

## L'OPTIMISATION SQL

- Le vrai problème de l'informatique :
  - ne serait-ce pas les utilisateurs ???
- Un aperçu :
  - Schéma de données mal conçu
  - Pas d'indexation
  - Vues imbriquées
  - Requêtes baroques
  - etc.

---

## UN SUJET TRÈS VASTE

- **La** solution : la formation **PERF1** de Dalibo
  - https://www.dalibo.com/formation-postgresql-performance-1
- Au programme :
  - Configuration du système et de l'instance
  - Techniques d'indexation
  - Comprendre `EXPLAIN`
  - Analyses et diagnostics

---

## QUELQUES OUTILS POUR L'OPTIMISATION SQL

- `EXPLAIN` : récupère le plan d'exécution optimal
- `EXPLAIN ANALYSE` : plan et réalisé de la requête
- Difficile à lire, utiliser les outils :
  - https://explain.dalibo.com/
  - https://explain.depesz.com/

---

## QUESTIONS ? REMARQUES !

- Support Dalibo :
  - sur la plate-forme : https://support.dalibo.com
  - par mail à l'adresse : support@dalibo.com[8]
  - par téléphone au : +33 (0) 970 444 750

[8] mailto:support@dalibo.com