DALIBO

*L'Expertise* PostgreSQL

# PostgreSQL : Tâches courantes

**Formateur** : Jean-Paul Argudo
**Contact** : jean-paul.argudo@dalibo.com
**Date** : mars 2010

# Table des matières

1 Introduction 5

2 Licence Creative Commmons CC-BY-NC-SA 6

3 Gestion des bases 7

4 Création d'une base 8

5 Suppression d'une base 9

6 Configuration 10

7 Modèle de base (Template) 11

8 Utilisateurs & Groupes 12

9 Versions 13

10 Utilisateurs 14

11 Groupes 15

12 Rôles 16

13 Options de `CREATE ROLE` 17

14 `SET ROLE` 18

15 `REASSIGN OWNED` et `DROP OWNED` 19

16 Attributs 20

17 Droits 21

18 Maintenance 22

**19 Principes** **23**

**20 Sauvegarde** **24**

**21 Espace disque - VACUUM** **26**

**22 Espace disque - VACUUM FULL** **27**

**23 Espace disque - stratégie** **28**

**24 Statistiques (1/2)** **29**

**25 Statistiques (2/2)** **30**

**26 Gel des `XID`** **31**

**27 Le démon Autovacuum** **32**

**28 Autovacuum : paramètres (1/2)** **33**

**29 Autovacuum : paramètres (2/2)** **34**

**30 Indexation** **35**

**31 `Cluster` ou `vacuum` ?** **36**

**32 Journaux applicatifs** **37**

**33 Pour aller plus loin** **38**

**34 Conclusion** **39**

**35 Questions** **40**

# Introduction

- Création/suppression d'une base
- Templates
- Rôles et utilisateurs
- Maintenance
- Indexation

# Licence Creative Commmons CC-BY-NC-SA

# Gestion des bases

- Création
- Suppression
- Configuration
- Modèle (*Template*)

# Création d'une base

- *SQL* : `CREATE DATABASE`

  Quand une nouvelle base de données est créée à l'intérieur du groupe, `template1` est généralement clonée. Cela signifie que tous les changements effectués sur `template1` sont propagés à toutes les bases de données créées ultérieurement. Du coup, il est déconseillé d'utiliser `template1` pour votre travail quotidien mais cette propriété, utilisée judicieusement, peut être utile.

*ou*

- *Outil système* : `createdb`

  `createdb` se connecte à la base de données `postgres` et exécute la commande `CREATE DATABASE`, exactement comme ci-dessus. Appelée sans aucun argument, `createdb` crée une base de donnée portant le nom de l'utilisateur courant.

  À partir de la version 8.4, il est possible d'indiquer les options LC_COLLATE (ordre de tri) et LC_CTYPE (jeu de caractères) pour chaque base de données créée.

# Suppression d'une base

- *SQL* : `DROP DATABASE`
  Supprimer une base de données supprime tous les objets qui étaient contenus dans la base. La destruction d'une base de données ne peut pas être annulée. Vous ne pouvez pas exécuter la commande `DROP DATABASE` en étant connecté à la base de données cible. Néanmoins, vous pouvez être connecté à une autre base de données, ceci incluant la base `template1`. `template1` pourrait être la seule option pour supprimer la dernière base utilisateur d'un groupe donné.

*ou*

- *Outil système* : `dropdb`
  Contrairement à `createdb`, l'action par défaut n'est pas de supprimer la base possédant le nom de l'utilisateur en cours.

# Configuration

Il est possible de modifier *à chaud* un grand nombre de paramètres de la base, en utilisant la syntaxe suivante :

- ALTER DATABASE base SET param TO val ;

Les paramètres par base de données surchargent tous ceux reçus de la ligne de commande de postmaster ou du fichier de configuration, et sont aussi surchargés par ceux de l'utilisateur ; les deux sont surchargés par les options par session.

Cela sauvegarde le réglage (mais ne l'applique pas immédiatement). Lors des connexions ultérieures à cette base de données, tout se passe comme si la commande était exécutée juste avant de commencer la session. Notez que les utilisateurs peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut. Pour annuler un tel réglage par défaut, utilisez :

```
ALTER DATABASE nom_base RESET nom_variable ;
```

- Attention, paramétrage spécifique non copié.

Si vous copiez une base de données dont certains paramètres ont été configurés spécifiquement pour cette base de données, ces paramètres ne sont pas appliqués à la nouvelle base de données.

# Modèle de base (Template)

- Pour créer une base de données à partir de *template2*, utilisez :

```
CREATE DATABASE nom_base TEMPLATE template2 ;
```

  Il est essentiel que la base de données source soit inactive (pas de transaction en écriture en cours) pendant toute la durée de l'opération de copie.
  On peut par exemple décider que la base `test` doit être figée pour devenir un modèle :

```
test=# UPDATE pg_database
       SET datistemplate=true
       WHERE datname='test' ;
```

  Ainsi, tout utilisateur doté du droite CREATEDB pourra la cloner. Sans cette option, seul un super-utilisateur ou le propriétaire de la base clonée peut en faire une copie.
  La commande shell correspondante pour créer une base de données à partir d'un modèle est :

```
createdb -T test nom_base
```

- par défaut, *template1* est utilisée
  `CREATE DATABASE` fonctionne en copiant une base de données préexistante. Par défaut, cette commande copie la base de données système standard *template1*. Ainsi, cette base de données est le « modèle » à partir duquel de nouvelles bases de données sont créées. Si vous ajoutez des objets à *template1*, ces objets seront copiés dans les bases de données utilisateur créées ultérieurement. Par exemple, si vous installez le langage de procédures PL/pgSQL dans la base *template1*, celui-ci sera automatiquement disponible dans les bases de données utilisateur créées à partir de ce modèle.
- Lancer un `VACUUM FREEZE`, après la création du template
  Après avoir préparé ou modifié une base de données modèle, il est recommandé d'utiliser la commande `VACUUM FREEZE` dans cette base de données. Cela évite les problèmes de réutilisation d'ID de transaction déjà attribués (particulièrement important si, en plus, `datallowconn` vaut false car aucune connection de maintenance n'y est autorisée dans ce cas).

# Utilisateurs & Groupes

- Versions
- Utilisateurs
- Groupes
- Attributs
- Droits

# Versions

- *Version < 8.1.* : USER & GROUP
  Avec les versions antérieures à la 8.1.x de PostgreSQL, les droits sont gérés comme dans `Unix` : on peut créer des utilisateurs et des groupes. Les groupes étant une manière logique de grouper les utilisateurs pour faciliter la gestion des privilèges, les droits peuvent être accordés ou révoqués à un groupe entier.
- *Version >= 8.1.* : ROLE
  Un rôle peut être vu soit comme un utilisateur de la base de données, soit comme un groupe d'utilisateurs de la base de données, suivant la façon dont le rôle est configuré. Les rôles peuvent posséder des objets de la base de données (par exemple des tables) et peuvent affecter des droits sur ces objets à d'autres rôles pour contrôler qui a accès à ces objets. De plus, il est possible de donner l'appartenance d'un rôle à un autre rôle, l'autorisant du coup à utiliser les droits affectés au rôle dont il est membre.
  Les concepts des « utilisateurs » et des « groupes » restent disponibles dans les versions 8.1 et ultérieures. Il n'y a donc pas de problèmes de compatibilité ascendante.

# Utilisateurs

- *SQL* : `CREATE USER` : ajoute un nouvel utilisateur dans le groupe de bases de données PostgreSQL. Exemple : Créer un utilisateur valide jusqu'en 2010

```
CREATE USER tom WITH PASSWORD 'fud77val'
VALID UNTIL '2010-01-01';
```

  À partir de la version 8.1.x, la commande

```
CREATE USER nom;
```

  est équivalente à

```
CREATE ROLE nom LOGIN;
```

- *SQL* : `ALTER USER` : change les attributs d'un utilisateur.
- *SQL* : `DROP USER` : supprime un utilisateur.
- *SQL* : `ALTER GROUP` : gère l'appartenance d'un utilisateur à un groupe.
- *Outils système* : `createuser` / `dropuser` : commandes utilisées pour ajouter ou supprimer un utilisateur de base de données.
- *SQL* : `REASSIGN` et `DROP OWNED` : réaffectation/suppression des objets appartenant à un rôle.

# Groupes

- *SQL* : `CREATE GROUP` : crée un groupe d'utilisateurs de base de données.
  À partir des versions 8.1.x, `CREATE GROUP` est un alias de `CREATE ROLE` avec l'option `NOLOGIN`.
- *SQL* : `DROP GROUP` : supprime un groupe d'utilisateurs de base de données.
- *SQL* : `ALTER GROUP` : commande utilisée pour gérer un groupe de base de données.
  Pour ajouter ou supprimer un utilisateur à un groupe :

```
ALTER GROUP nom_group ADD USER utilisateur_1, ... ;
ALTER GROUP nom_group DROP USER utilisateur_1, ... ;
```

# Rôles

- Rôles = Utilisateur + Groupe
  Un rôle est une entité qui peut posséder des objets de la base de données et avoir des droits sur la base. Il peut être considéré comme un « utilisateur », un « groupe » ou les deux suivant la façon dont il est utilisé.
  Conceptuellement, les rôles de base de données sont totalement séparés des utilisateurs du système d'exploitation. En pratique, il peut être commode de maintenir une correspondance mais cela n'est pas requis. Les rôles sont globaux à toute une installation de groupe de bases de données (et non individuelle pour chaque base).
- *SQL* : `CREATE ROLE` : ajoute un nouveau rôle dans une grappe (cluster) de bases de données PostgreSQL.
- *SQL* : `DROP ROLE` : supprime un rôle d'un cluster de bases de données.
- *Outils système* : `createuser` / `dropuser`
- Table de rôles : `pg_roles`
  Les rôles existants sont stockés dans le catalogue système `pg_roles`. Il est possible d'afficher la liste des rôles créés en interrogeant cette table :

```
SELECT rolname FROM pg_roles ;
```

# Options de CREATE ROLE

- INHERIT, NOINHERIT
  Ces clauses précisent si un rôle « hérite » des droits d'un rôle dont il est membre.
  Un rôle qui possède l'attribut INHERIT peut automatiquement utiliser tout privilège détenu par un rôle dont il est membre. Sans INHERIT, l'appartenance à un autre rôle lui confère uniquement la possibilité d'utiliser SET ROLE pour acquérir les droits de l'autre rôle. Ils ne sont disponibles qu'après cela. Si un rôle herite d'un autre rôle lui aussi INHERIT, il hérite aussi des rôles dont ce dernier est membre.
  INHERIT est la valeur par défaut.
- LOGIN, NOLOGIN
  Ces clauses précisent si un rôle est autorisé à se connecter, c'est-à-dire si le rôle peut être donné comme nom pour l'autorisation initiale de session à la connexion du client.
  Un rôle ayant l'attribut LOGIN peut être vu comme un utilisateur. Un rôle ayant l'attribut NOLOGIN peut être vu comme un groupe.
  NOLOGIN est la valeur par défaut.
- CONNECTION LIMIT
  Cette clause permet de limiter le nombre de connexions en parallèle d'un utilisateur.
- [ ENCRYPTED | UNENCRYPTED ]PASSWORD
  Ces clauses indiquent si le mot de passe doit être chiffré (ENCRYPTED) ou non (UNENCRYPTED). Sans cette clause, le comportement par défaut dépend du paramètre password_encryption.

# SET ROLE

Syntaxe :

```
SET [ SESSION | LOCAL ] ROLE nom
```

Cette commande permet de changer l'utilisateur courant à l'intérieur d'une session SQL en cours.

`LOCAL` limite la portée de ce changement à la transaction en cours. `SESSION` (valeur par défaut) permet à ce changement d'être toujours valable une fois que la transaction est terminée si cette transaction est bien validée.

L'utilisateur de la session courante doit être membre du rôle nommé.

Si le rôle de l'utilisateur de la session comprend l'attribut `INHERITS`, alors il acquiert automatiquement les droits de chaque rôle qu'il peut prendre par la commande `SET ROLE`.

À l'opposé, si le rôle de l'utilisateur de la session dispose de l'attribut `NOINHERITS`, `SET ROLE` supprime les droits affectés directement à l'utilisateur de la session et les remplace par les droits du rôle nommé.

## REASSIGN OWNED et DROP OWNED

Syntaxe :

```
REASSIGN OWNED BY ancien_role [, ...] TO nouveau_role

DROP OWNED BY role [, ...] [ CASCADE | RESTRICT ]
```

Cette commande permet de remplacer le propriétaire d'objets par un autre propriétaire. La deuxième commande permet de supprimer les objets possédés par un rôle particulier.

Avant l'apparition de ces instructions (version 8.2), il était nécessaire de supprimer chaque objet avant de pouvoir supprimer le rôle propriétaire.

L'option `CASCADE` permet de supprimer automatiquement les objets qui dépendent des objets supprimés.

L'option `RESTRICT`, valeur par défaut, permet de stopper l'opération s'il existe des objets non supprimés dépendants des objets supprimés.

# Attributs

Les attributs d'un rôle peuvent être modifiés après sa création avec la commande `ALTER ROLE`.

Exemple

```
ALTER ROLE john WITH PASSWORD 'eurtg83b' ;
```

On peut aussi modifier certains paramètres de configuration pour les sessions des utilisateurs.

Syntaxe :

```
ALTER ROLE utilisateur SET attribut TO valeur ;
```

Exemple de désactivation des parcours d'index :

```
ALTER ROLE john SET enable_indexscan TO off ;
```

# Droits

- `GRANT` *privilèges* `ON` *base* `TO` *utilisateur* ;
- `REVOKE` *privileges* `ON` *base* `FROM` *utilisateur* ;
- `GRANT` sur les objets de la base de données
  Cette variante de la commande `GRANT` donne des droits spécifiques sur un objet de la base de données à un ou plusieurs rôles. Ces droits sont ajoutés à ceux déjà possédés.
  La liste des droits possibles est

  ```
  * SELECT, INSERT, UPDATE, DELETE,
  * RULE, REFERENCES, TRIGGER, CREATE,
  * TEMPORARY, EXECUTE, USAGE
  * ALL
  ```

  Il est possible de donner des droits sur les colonnes à partir de la version 8.4.
- `GRANT` sur les rôles
  Cette variante de la commande `GRANT` définit l'appartenance d'un (ou plusieurs) rôle(s) à un autre. L'appartenance à un rôle est importante car elle offre tous les droits accordés à un rôle à l'ensemble de ses membres.
  Si `WITH ADMIN OPTION` est spécifié, le membre peut à la fois octroyer l'appartenance à d'autres rôles, et la révoquer. Sans cette option, les utilisateurs ordinaires ne peuvent pas le faire. Toutefois, les superutilisateurs peuvent donner ou enlever à tout rôle l'appartenance à un rôle. Les rôles qui possèdent le droit `CREATEROLE` peuvent agir ainsi sur tout rôle qui n'est pas superutilisateur.
  Par exemple, rendre l'utilisateur `jean` membre de `admins` :

  ```
  GRANT admins TO jean ;
  ```

# Maintenance

- Principes et intérêts des opérations de maintenance
- Sauvegarde de l'instance, sauvegarde des bases de données
- Gestion de l'espace disque utilisée par les tables et index
- Mise à jour des statistiques du planificateur
- Intégration de l'autovacuum dans les opérations de maintenance
- Indexation et ré-indexation pour conserver des index propres
- Gestion des journaux applicatifs

# Principes

- Opérations régulières et nécessaires
  Pour fonctionner de façon optimale, un serveur PostgreSQL nécessite quelques opérations de maintenance régulières. Les tâches de maintenance récurrentes ont plusieurs objectifs :
  - ⇒ « Nettoyer » les données,
  - ⇒ Reconstruire les index,
  - ⇒ Garder des journaux applicatifs compacts et pertinents.
- Maintenance plus simple qu'avec d'autres SGBD
  PostgreSQL demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à conserver un système performant et agréable à utiliser.
- Tâches automatisables
  Par définition, les tâches récurrentes peuvent facilement être automatisées grâce aux outils standards d'`UNIX`, et notamment les scripts `cron`. La responsabilité de la mise en place de ces scripts et du contrôle de leur bon fonctionnement relève de l'administrateur de la base.

# Sauvegarde

- `pg_dump` & `pg_dumpall`
  Le principe est de générer un fichier texte (ou binaire en mode `-F c`) de commandes SQL (appelé « fichier dump »), qui, s'il est renvoyé au serveur, recrée une base de données identique à celle sauvegardée. L'usage basique est :

  ```
  pg_dump base_de_donnees > fichier_de_sortie
  ```

  Le fichier dump peut être restauré avec la commande :

  ```
  psql base_de_donnees < fichier_d_entree
  ```

  `pg_dumpall` simplifie la tâche de l'administrateur en sauvegardant toutes les bases de données d'un groupe de bases de données (`cluster`) et préserve les données communes au groupe de bases (les rôles par exemple) :

  ```
  pg_dumpall > fichier_de_sortie
  ```

  Le fichier de sauvegarde résultant peut être restauré avec `psql` :

  ```
  psql -f fichier_d_entree postgres
  ```

- Sauvegarde disque en ligne
  Le principe de cette méthode est de sauvegarder les fichiers de la base de données alors que celle-ci est encore en activité, en ignorant les erreurs de modification de fichiers détectées par l'outil de sauvegarde (erreurs normales, puisque la base continue de fonctionner). À la restauration, la base de données doit être capable de rejouer l'ensemble des journaux de transaction ayant été générés pendant la période de sauvegarde, afin de restaurer l'intégrité des fichiers de la base.
  Cette méthode impose donc avant tout que le cluster archive ses journaux de transaction, mais aussi qu'on notifie au cluster qu'il est en cours de sauvegarde, afin qu'il trace la liste des fichiers dont il aura besoin pour restaurer son intégrite en cas de restauration.
  Cette procédure est plus compliquée que pg_dump, car la procédure de restauration contient davantage d'étapes et impose des manipulations de fichiers. Son autre limitation est que la restauration se fait pour l'ensemble du cluster (plusieurs bases potentiellement, puisqu'on ne peut pas restaurer les fichiers de chaque base de façon individuelle).
  Elle présente par contre de nombreux avantages : elle est beaucoup plus performante, particulièrement dans le cas de grosse volumétrie (à partir de quelques dizaines de gigaoctets), et permet une sauvegarde dite 'en continu' : on peut restaurer la base à n'importe quel point dans le temps, pourvu qu'on dispose d'une sauvegarde antérieure à ce point, et des journaux de transaction archivés jusqu'à ce point.
  Pour davantage de détails sur cette procédure, consultez la page de KB suivante : https ://support.dalibo.com
- Définir une politique de sauvegarde
  Pour bien automatiser les procédures de sauvegardes, on peut utiliser les 5 axes suivants :
  - ⇒ Pourquoi sauvegarder ? Dans un but d'archivage ? Pour assurer la haute disponibilité des données ?
  - ⇒ Quels sont les ensembles de données essentiels et ceux qui ne le sont pas ?
  - ⇒ Quelle fréquence de sauvegarde est la plus adaptée à vos besoins ?
  - ⇒ Quels supports ? DVD ? Enregistrements sur bande ? Réplication sur une machine distante ?
  - ⇒ Quels outils ? Commandes internes de PostgreSQL ou applications externes ?

- Vérifier la restauration des sauvegardes

# Espace disque - VACUUM

La commande `VACUUM` doit être exécutée régulièrement pour que PostgreSQL connaisse les espaces libres disponibles dans les tables et index.

Elle peut être lancée en concurrence avec les autres opérations.

PostgreSQL ne supprime pas les versions périmées des lignes après un `UPDATE` ou un `DELETE`. La commande `VACUUM` permet de « libérer » l'espace utilisé par ces lignes afin d'éviter un accroissement continuel du volume occupé sur le disque.

Une table qui subit beaucoup de mises à jour et suppressions nécessitera des nettoyages plus fréquents que les tables rarement modifiées.

Le `VACUUM` « simple » marque les données expirées dans les tables et les index pour une utilisation future. Il ne tente pas de récupérer l'espace utilisé par les données obsolètes, sauf si l'espace est à la fin de la table et qu'un verrou exclusif de table puisse être facilement obtenu. L'espace inutilisé au début ou au milieu du fichier ne provoque pas un raccourcissement du fichier et ne redonne pas d'espace mémoire au système d'exploitation.

La version 8.4 améliore les performances du `VACUUM` en lui permettant de ne parcourir que la partie de la table qui a été modifiée. Cela a un gros impact pour les tables les moins modifiées.

# Espace disque - VACUUM FULL

`VACUUM FULL` :

- nettoyage plus efficace mais plus lent
- méthode « agressive »
- impact non négligeable sur les performances

La commande `VACUUM FULL` libère l'espace consommé par les lignes périmées et le rend au système d'exploitation.

Cette variante de la commande `VACUUM` acquiert un verrou exclusif sur chaque table. Elle peut donc avoir un effet extrêmement négatif sur les performances de la base de données.

# Espace disque - stratégie

**Quand faut-il utiliser « VACUUM » ?**

- Nettoyages réguliers (1 fois/jour)
- Maintenance de base

Des `VACUUM` standards et une fréquence modérée sont une meilleure approche que des `VACUUM FULL`, même non fréquents, pour maintenir des tables mises à jour fréquemment.

**Quand faut-il utiliser « VACUUM FULL » ?**

- Après des suppressions massives de données
- Lorsque la base n'est pas en production

`VACUUM FULL` est recommandé dans les cas où vous savez que vous avez supprimé ou modifié une grande partie des lignes d'une table, de façon à ce que la taille de la table soit réduite de façon conséquente.

# Statistiques (1/2)

- `ANALYZE`
- Met à jour les statistiques de la base
- Utile pour l'optimiseur de requêtes

**Conseil** : 1 fois/jour, en même temps que `VACUUM`

L'optimiseur de requêtes de PostgreSQL s'appuie sur des informations statistiques du contenu des tables. Ces statistiques sont collectées par la commande `ANALYZE`, qui peut être invoquée seule ou comme une option de `VACUUM`. Il est important d'avoir des statistiques relativement à jour sans quoi des mauvais choix dans les plans d'exécution pourraient pénaliser la performance de la base.

En général, une bonne stratégie est de programmer `ANALYZE` une fois par jour. Ceci peut être couplé à un `VACUUM` (la nuit par exemple) pour gagner en performances.

# Statistiques (2/2)

- `default_statistics_target`
  Initialise la cible par défaut des statistiques pour les colonnes de table qui n'ont pas une cible spécifique de colonne configurée via `ALTER TABLE SET STATISTICS`. Des valeurs plus importantes accroissent le temps nécessaire à exécuter `ANALYZE` mais pourraient améliorer les estimations du planificateur. La valeur par défaut est de 10, jusqu'en version 8.3, et 100 à partir de la version 8.4.
  C'est-à-dire que, pour chaque colonne, les 10 valeurs les plus fréquentes, et 10 histogrammes sont stockés dans `pg_stats` en guise d'échantillon représentatif des donnnées... Beaucoup de personnes pensent que la valeur par défaut devrait être de l'ordre de 100
  Si vous voulez des statistiques plus fines, vous pouvez passer ce paramètre à 300. Des valeurs supérieures sont possibles, sans dépasser la limite de 1000, mais provoquent un ralentissement d'`ANALYZE`, un accroissement de la table `pg_stats`, et un temps de calcul des plans d'exécution plus long (non mesurable jusqu'à 100).
- `ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 200 ;`
  Commande à utiliser si l'on veut définir une valeur colonne par colonne. La valeur ainsi spécifiée prévaut sur la valeur de `default_statistics_target`

# Gel des XID

- L'identifiant de transaction est un nombre croissant et limité
  Le mécanisme de contrôle de concurrence multi-version (MVCC) de PostgreSQL s'appuie sur la possibilité de comparer des identifiants de transactions (XID). La version d'une ligne dont le XID d'insertion est supérieur au XID de la transaction en cours est « dans le futur » et ne doit pas être visible de la transaction courante.
- Si le XID revient à zéro => Risque de perte des données !
  Comme les identifiants ont une taille limitée (32 bits à ce jour), une base en activité depuis longtemps (plus de 4 milliards de transactions) pourrait connaître un cycle des identifiants de transaction : le XID reviendra à 0 et soudainement les transactions du passé sembleront appartenir au futur - ce qui signifie qu'elles deviennent invisibles. Ceci peut conduire à une perte de données totale.
- 2 solutions : initdb ou VACUUM
  Avant PostgreSQL 7.2, la seule parade contre ces cycles de XID était de ré-exécuter initdb au minimum tous les 4 milliards de transaction.
  VACUUM a une approche plus fine : toute table dans la base doit être nettoyée au moins une fois tous les milliards de transactions. Lorsqu'il reste moins de 10 millions de transactions avant le renouveau du cycle, PostgreSQL émet des messages d'alertes (voir ci-dessous) pour chaque transaction exécutée.

```
WARNING : database "z" must be vacuumed within 177006525  transactions
HINT : To avoid a database shutdown, execute a full-database VACUUM in "z".
```

- Autovacuum à partir de la version 8.1
  À partir de la version 8.1, et même s'il est désactivé, l'autovacuum va se lancer automatiquement pour chaque table ayant besoin d'un rafraichissement du XID.

# Le démon Autovacuum

- Versions >= 8.1.x
- Automatiser `VACUUM` **et** `ANALYZE`
- Le démon détermine si le `vacuum` et/ou l'`analyze` sont nécessaires

À partir de PostgreSQL 8.1, il existe un processus serveur optionnel et séparé appelé le démon `autovacuum`, dont le but est d'automatiser l'exécution des commandes `VACUUM` et `ANALYZE`. Une fois activé, le démon `autovacuum` s'exécute périodiquement et vérifie les tables ayant un grand nombre de lignes insérées, mises à jour ou supprimées.

Deux conditions sont utilisées pour déterminer quelle opération appliquer :

```
limite du vacuum =
    autovacuum_vacuum_threshold
  + autovacuum_vacuum_scale_factor * N

limite du analyze =
    autovacuum_analyze_threshold
  + autovacuum_analyze_scale_factor * N
```

où *N* est le nombre de lignes dans la table.

Le démon `autovacuum` peut être paramétré à partir du fichier `postgresql.conf` de façon globale. Il est possible d'avoir une configuration plus fine en modifiant le contenu de la table système **pg_autovacuum** pour les versions antérieures à la 8.4, et en modifiant les informations de stockage des tables pour les versions ultérieures.

# Autovacuum : paramètres (1/2)

- `autovacuum`

  Contrôle si le serveur doit lancer le sous-processus `autovacuum`. Désactivé par défaut pour 8.1 et 8.2, activé pour 8.3 et supérieures.

  `stats_start_collector` et `stats_row_level` (respectivement `track_activities` et `track_counts` en 8.3) doivent aussi être actifs pour que ce démon soit exécuté.

- `log_autovacuum_min_duration`

  Trace l'activité du sous-processus `autovacuum` si ce dernier dure plus que ce nombre de secondes.

  -1 désactive les traces, 0 active la trace de toute exécution d'`autovacuum`.

  Paramètre disponible à partir de la version 8.3.

- `autovacuum_naptime`

  En 8.2, spécifie le délai entre les tours d'activité pour le sous-processus `autovacuum`. À chaque tour, le sous-processus examine une base de données et lance autant de commandes `VACUUM` et `ANALYZE` que nécessaire pour les tables de la base de données. Le délai est mesuré en secondes et vaut par défaut 60 secondes.

  En 8.3, ce nombre divisé par le nombre de bases de données spécifie le délai pour l'exécution d'un nouveau processus `autovacuum worker`.

- `autovacuum_max_workers`

  Spécifie le nombre maximum de sous-processus.

  Paramètre disponible à partir de la version 8.3.

- `autovacuum_freeze_max_age`

  Spécifie l'âge maximum (en transactions) que le champ `pg_class.relfrozenxid` puisse atteindre avant de forcer l'exécution d'un `VACUUM`.

  Paramètre disponible à partir de la version 8.2.

# Autovacuum : paramètres (2/2)

- `autovacuum_vacuum_threshold`

  Spécifie le nombre minimum de lignes mises à jour ou supprimées nécessaires pour déclencher un `VACUUM` sur une table.

- `autovacuum_analyze_threshold`

  Spécifie le nombre minimum de lignes insérées, mises à jour ou supprimées pour déclencher une commande `ANALYZE` sur une table.

- `autovacuum_vacuum_scale_factor`

  Spécifie une fraction de la taille de la table à ajouter à `autovacuum_vacuum_threshold` pour décider du moment pour déclencher un `VACUUM`.

- `autovacuum_analyze_scale_factor`

  Spécifie une fraction de la taille de la table à ajouter à `autovacuum_analyze_threshold` pour décider de déclencher une commande `ANALYZE`.

- `autovacuum_vacuum_cost_delay`

  Spécifie la valeur du coût du délai utilisée dans les opérations de `VACUUM`. Si -1 est spécifié (la valeur par défaut), la valeur habituelle de `vacuum_cost_delay` sera utilisée.
  En 8.3, ce délai est réparti entre tous les processus « autovacuum workers ».

- `autovacuum_vacuum_cost_limit`

  Spécifie la valeur limite du coût utilisée dans les opérations de `VACUUM` automatiques. Si -1 est spécifié (la valeur par défaut), la valeur courante de vacuum_cost_limit sera utilisée.

# Indexation

`REINDEX` reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index.

Lancer *REINDEX* régulièrement permet :

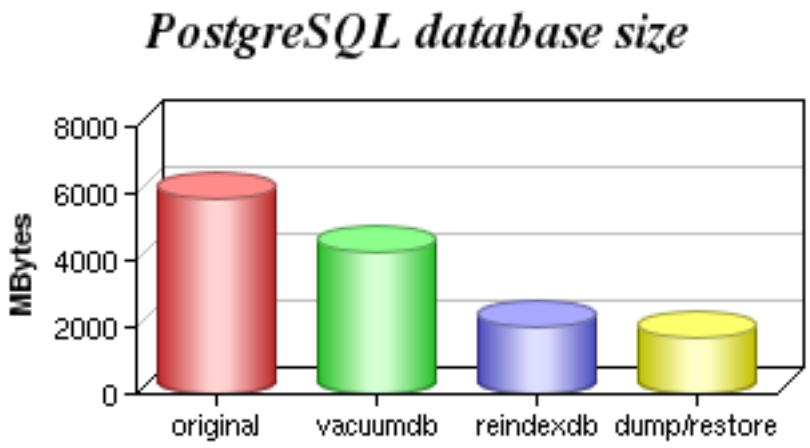
- de gagner de l'espace disque
  Dans les versions PostgreSQL antérieures à la 7.4, la réindexation périodique était fréquemment nécessaire pour éviter l'« inflation des index ».
  Dans les versions 7.4 et ultérieures, les pages d'index qui sont devenues complètement vides sont récupérées pour être réutilisées. Il existe toujours la possibilité d'une utilisation inefficace de l'espace : si pratiquement toutes les clés d'index d'une page ont été supprimées, 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 ayant un tel usage.
- d'améliorer les performances (pour les index B-Tree)
  Pour les index B-tree, un index tout juste construit est quelque peu 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 pas aux index non B-tree). Il pourrait être intéressant de ré-indexer périodiquement, simplement pour améliorer la vitesse d'accès.
- de réparer un index corrompu
  Un index a été corrompu et ne contient plus de données valides. Bien qu'en théorie, ceci ne devrait jamais arriver, en pratique, les index peuvent se corrompre à cause de bogues dans le logiciel ou d'échecs matériels.
  L'index à réparer peut être *utilisateur* ou *système* (utiliser `REINDEX SYSTEM` pour cela).
- `VACUUM` en `FULL` ou pas ne provoque pas de réindexation...

Il a même tendance à les fragmenter.

La commande `reindexdb` peut être utilisée en ligne de commande pour réindexer une table, une base ou un cluster de base de données.

# Cluster ou vacuum ?

- CLUSTER est une alternative à VACUUM FULL
  Parce que la commande CLUSTER provoque une réorganisation des données de la table, le résultat obtenu est équivalent à un VACUUM FULL : les données sont organisées comme l'index et les pages de données sont adjacentes dans les fichiers.
  Comme CLUSTER déplace les blocs de données, il provoque une réindexation de tous les index liés à la table.
  Au final, CLUSTER est équivalent à un VACUUM FULL suivi d'un REINDEX, à l'organisation physique des blocs de données près.
- Plus rapide que VACUUM FULL suivi de REINDEX
- Attention, CLUSTER nécessite près du double de l'espace disque utilisé pour stocker la table et ses index
- Variation de la table et des index associés :



# Journaux applicatifs

- Utiles en cas de problèmes
  Les journaux applicatifs jouent un rôle essentiel lorsque des problèmes surviennent ou pendant des phases de tests.
- Volumineux
  Ces fichiers ont tendance à être volumineux, en particulier si le niveau de débogage est important.
- Utiliser un système de gestion des journaux applicatifs tels que `logrotate` ou `sysklog` ou `rsyslog`
  Attention pour ceux qui veulent utiliser `sysklog` :
  - ⇒ Sur beaucoup de systèmes, `sysklog` n'est pas très fiable, particulièrement avec les messages très gros ; il pourrait tronquer ou supprimer des messages au moment où vous en aurez le plus besoin.
  - ⇒ `sysklog` force la synchronisation de tous les messages sur le disque, ce qui n'est pas très bon sur le plan des performances.
  - ⇒ À partir de Lenny, `rsyslog` devient le gestionnaire de syslog par défaut sur les serveurs debian. Il est intéressant de constater que `rsyslog` propose un plug-in pour PostgreSQL, c'est-à-dire que `rsyslog` est capable de stocker les messages syslog dans une base PostgreSQL.

    http ://www.rsyslog.com/

# Pour aller plus loin

- Documentation officielle :

  Chapitre 22. Planifier les tâches de maintenance
- « Opérations de maintenance sous PostgreSQL »

  http ://dalibo.org/glmf109_operations_de_maintenance_sous_postgresql

# Conclusion

PostgreSQL demande peu de travail au quotidien.

À l'installation, certaines tâches doivent être automatisées, par exemple la sauvegarde, les `VACUUM`.

Pour le reste, il s'agit surtout de surveiller la bonne exécution des scripts automatisés et le contenu des journaux applicatifs.

# Questions

N’hésitez pas, c’est le moment !