Configuration, mise en oeuvre de PostGreSQL et performances

Agnès BOCCHINO - Alexandra DANTE

PostgreSQL

Sommaire

1. Installation et architecture

Installation et architecture

Définition de l'utilitaire RPM

- RPM (RedHat Package Manager) = utilitaire de gestion de packages RedHat (installation / désinstallation de packages)
- Intégré aux distributions RedHat, Mandrake, Caldera et SuSE de Linux, utilisable avec d'autres distributions ou systèmes Unix, comme par exemple FreeBSD

- 'rpmbuild' installé avec le package « *rpm-build* »
  - Permet de prendre le code source de logiciels et de l'empaqueter :

soit sous forme de sources qui pourront être compilés
soit sous forme de binaires qui pourront être installés

- RPM maintient une base de données de tous les packages installés sur une machine

Référence : `http://www.rpm.org/max-rpm/index.html`

Définition d'un package RPM source

- Packages dont le nom se termine par « *.src.rpm* »
- Via le gestionnaire de fichier « *mc* », on voit que le RPM source contient :
  - un fichier tar.gz (fichiers source)
  - éventuellement des fichiers de patch
  - un fichier de spécifications (.spec), qui contient en fait toutes les informations sur le package (nom, version, url, licence, fichiers contenus ...)
- Fabrication d'un RPM à partir d'un RPM source, deux méthodes
  1. `rpmbuild --rebuild source_rpm`
  1. En deux temps :

`rpm -i source_rpm : éclate le fichier src.rpm sur l'arborescence /usr/src/.../`

```
cd /usr/src/.../SPECS/
rpmbuild -bb specfile
```

Définition d'un package tar.gz

- Contient les sources du produit à installer
- Nécessite un fichier « *specfile* » contenant les informations sur le package que l'on veut faire (nom, version ...) et des paramètres de compilation
- Même méthode de fabrication du rpm que pour un package RPM source :

```
rpmbuild -bb specfile
```

Avantages / Inconvénients des packages RPM

- Avantages :
  - faciliter l'administration des logiciels en terme :
    - installation
    - mise à jour facile des produits
    - désinstallation
    - gestion des conflits
    - gestion des dépendances
    - gestion de la sécurité
- Inconvénients :
  - Les rpms disponibles correspondent rarement à la dernière version du produit, mais à la dernière version certifiée

Installation et architecture

Présentation de PostGreSQL v8.1.2

- Date de sortie : 09/01/2006
- Corrections de bugs de la version 8.1.1 :

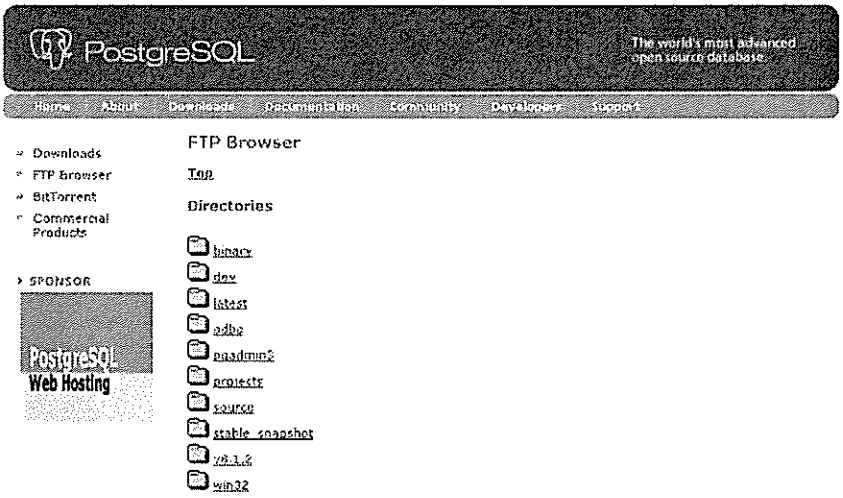
  [illegible]

- Changements majeurs faits depuis la v8.0 :
  - Commit à deux-phases
  - Savepoints
  - Point-In-Time Recovery
  - Tablespaces
  - Possibilité de modifier le type d'une colonne
  - ... ...
- Documentation complète v8.1.2 :

  [illegible]

Recherche de package pour PostGreSQL

- http://www.postgresql.org/ftp



Installation et architecture

Compilation et installation à partir d'un package tar.gz

PostgreSQL

- Téléchargement de l'archive tar.gz
- tar –xzvf postgresql-8.x.x.tar.gz
- création d'un super-utilisateur « postgres »
- chown –R postgres:postgres postgresqlxxxx
- su - postgres
- cd postgresql-8.x.x
- ./configure --prefix=/var/lib/psql
- make
- make check
- make install

Manipulations des RPM

PostgreSQL

- Lister le contenu d'un package qu'on se propose d'installer

```
rpm -qpli nom_du_package
```

- Installer : `rpm -ivh nom_du_package`
- Mettre à jour un logiciel installé :

```
rpm -Uvh nom_du_package.rpm
```

- Interroger la base de données des logiciels installés : `rpm -qa|grep post` renvoie les packages de nom *post* installés
- Désinstaller : `rpm -e nom_du_package`

Installation d'un package RPM

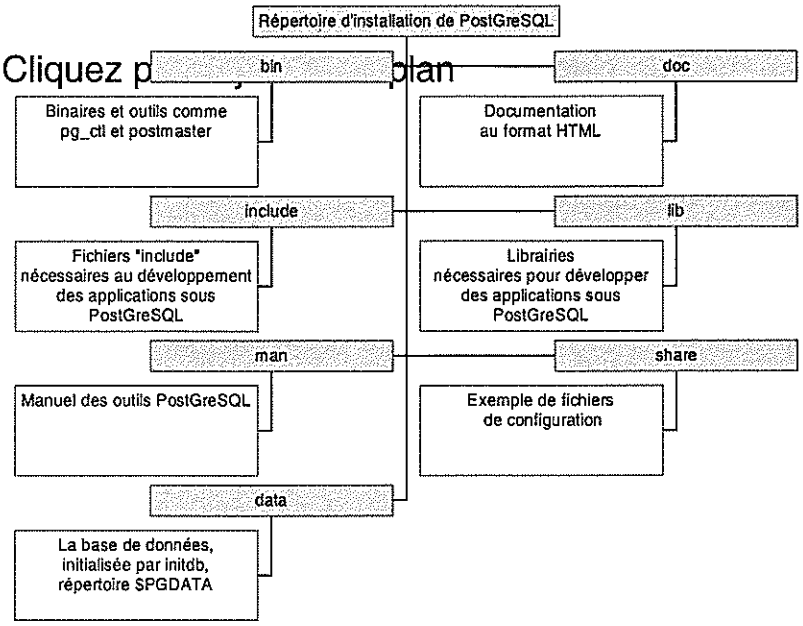
- RPM généré sous LINUX RHEL4-AS avec compilateur Intel icc V9
- RPM runtime des librairies partagées du compilateur Intel icc V9
- Pré-installation, Post-installation
- SRB d'installation

Espace disque

- Comparé à d 'autres SGBD, l'installation de PostGreSQL nécessite peu d 'espace disque

- Il faut approximativement :
  - **65 Mo** pour dépaqueter et compiler les sources
  - **30 Mo** pour le répertoire d 'installation du cluster de base de données
  - si vous désirez exécuter les tests de régression, vous aurez besoin temporairement de **135 Mo** supplémentaires.

Anatomie de l'installation PostGreSQL



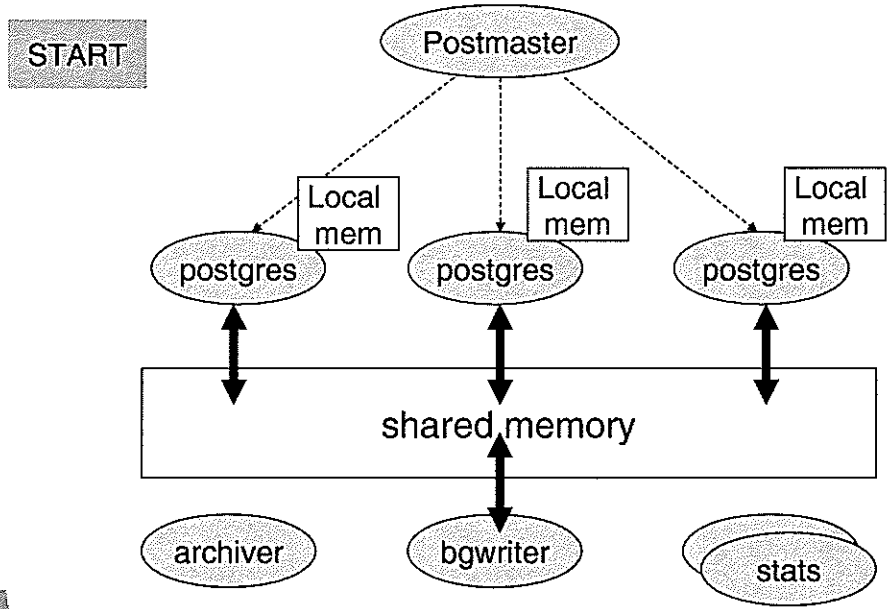
- Cliquez p[illegible]plan

Installation et architecture

Architecture système de PostGreSQL

- Utilisation d'un modèle client / serveur :
  - le processus serveur « ***postmaster*** »
  - l'application cliente des utilisateurs qui veulent effectuer des opérations sur la base de données
- Gestion de multiples connexions simultanées depuis les clients en dupliquant le processus « postgres »
- Process writer utilisé en background depuis la V8

Architecture de PostGreSQL



Process bgwriter

- Depuis V8.0, process d'écriture séparé des process serveur
- Processus d'écriture en tâche de fond
- Les processus serveur ne sont plus en attente
- Diminution des durées d'écriture lors des checkpoints
- Configuration fine et ajustement en fonction de l'application
- Augmentation de la charge en entrées/sorties

Cours PostGreSQL : Chapitre 2

2. Configuration de l'environnement et du serveur

Configuration de l'environnement et du serveur

Configuration initiale, environnement du système d'exploitation

- Création du super-utilisateur du moteur (souvent nommé « postgres »)
- Création du cluster ou groupe de bases de données
  - Créer éventuellement un emplacement de stockage pour le catalogue du cluster
  - En terme de système de fichier, c'est un simple répertoire (variable d'environnement $PGDATA)
- Donner les droits d'accès à ce répertoire pour le super utilisateur

Configuration initiale, environnement du système d'exploitation

PostgreSQL

- Mise à jour de l'environnement du super-utilisateur de la base de données en ajoutant dans le fichier d'initialisation « *.bash_profile* » :
  - PGDIR=répertoire d'installation de PostGreSQL
  - PGDATA=répertoire des bases de données
  - LD_LIBRARY_PATH=répertoire des librairies de PostGreSQL
  - PATH=répertoire des binaires
  - MANPATH=répertoire du manuel

BULL

Configuration de l'environnement et du serveur

PostgreSQL

BULL

Initialisation du serveur

- Pour initialiser un groupe de bases de données (cluster) :

  `initdb [option...] --pgdata | -D répertoire`

- Comme alternative à l'option –D , la variable d'environnement $PGDATA peut être initialisée

- L'option « --noclean » désactive le « nettoyage » de l'arborescence $PGDATA en cas d'erreur au lancement d'initdb et permet d'analyser les erreurs d'initialisation du serveur

Initialisation du serveur

- « initdb » crée l'arborescence de $PGDATA, avec les fichiers de configurations et plusieurs répertoires

- Sous le répertoire base, 3 bases sont créées :

→ Base template0

→ Base template1

→ Base postgres

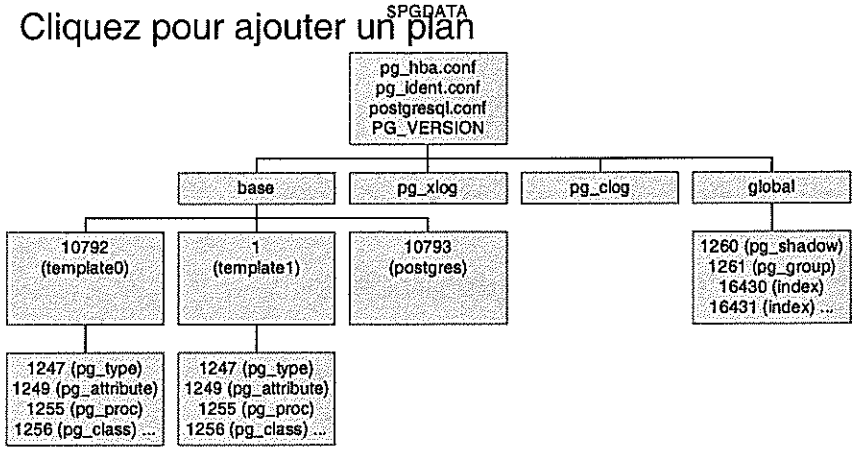
Initialisation du serveur

- template0 est une base de référence, elle contient la définition de toutes les tables systèmes, les définitions des objets standards
- template1, à l'initialisation du serveur est une base identique à template 0

  template1 est un modèle qui peut évoluer, en lui ajoutant des types de données, des fonctions, des tables...
- postgres est une base de données par défaut utilisée par les outils, les applications tiers

Initialisation du serveur : description de l'arborescence créée

- Cliquez pour ajouter un plan



Configuration de l'environnement et du serveur

Cinq fichiers de configuration

- postgresql.conf : fichier principal de configuration du serveur
- pg_hba.conf : fichier de configuration pour l'authentification basée sur l'hôte
- pg_ident.conf : le fichier de configuration pour l'authentification ident
- postmaster.pid : contient le PID du process postmaster
- postmaster.opts : options d'exécution du programme postmaster

Par défaut ces fichiers sont stockés dans le répertoire $PGDATA du groupe des bases de données

Configuration de l'environnement et du serveur

Paramètres initiaux de configuration

- Paramètres d'emplacement des fichiers de configuration
- Paramètres de sécurité
- Paramètres de connexion
- Paramètres d'authentification

Ces paramètres sont à positionner dans le fichier « **postgresql.conf** »

Paramètres d'emplacement

- Tous ces paramètres sont pris en compte au « démarrage du serveur »

- **DATA_DIRECTORY**
  Valeur par défaut $PGDATA
  Réécriture : `postmaster -D data-directory`

- **CONFIG_FILE**
  Valeur par défaut $PGDATA/postgresql.conf
  Réécriture : `postmaster -c config_file='nom de fichier'`

Paramètres d'emplacement

- **HBA_FILE**
  Valeur par défaut $PGDATA/pg_hba.conf
  Réécriture : `postmaster -c hba_file='nom de fichier'`

- **IDENT_FILE**
  Valeur par défaut $PGDATA/pg_ident.conf
  Réécriture : `postmaster -c ident_file='nom de fichier'`

Paramètres de sécurité

PostgreSQL

- **PASSWORD_ENCRYPTION**
  Valeur par défaut true
  Prise en compte : SET
  Réécriture :
  `SET PASSWORD_ENCRYTION TO true/false`

  `CREATE ROLE WITH ENCRYPTED/UNENCRYPTED PASSWORD`

  `ALTER ROLE WITH ENCRYPTED/UNENCRYPTED PASSWORD`

Paramètres de connexion

PostgreSQL

- **LISTEN_ADDRESSES :** autorisation de connexion réseaux,
  Spécifie les adresses TCP/IP sur lesquelles le serveur écoute les connexions des applications client.

  Valeur par défaut localhost
  Prise en compte : postmaster startup
  Réécriture : `postmaster -h address [,address, .....]`

- « * »signifie l'autorisation de toutes les connections réseaux

Ne pas oublier de configurer le fichier d'authentification pour les connections

Paramètres de connexion

- **PORT** : 5432 par défaut
  Le port TCP sur lequel le serveur écoute
- **MAX_CONNECTIONS**
  Détermine le nombre maximum de connexions concurrentes au serveur de la base de données

Paramètres d'authentification (fichier pg_hba.conf)

- Fichier d'authentification des utilisateurs
- Fichier pris en compte au lancement du moteur
- Indique les accès locaux (local) et distants (host)
- Indique au serveur les utilisateurs, clients, bases de données, et méthodes d'authentification à prendre en compte

Exemple :

```
# "local" is for Unix domain socket connections only
local   all     all                             trust
# IPv6 local connections:
host    all     all     129.183.0.0/16          trust
```

Paramètres d'authentification

- Format des lignes du fichier d'authentification

TYPE DATABASE USER CIDR-ADDRESS METHOD

TYPE	local ou host
DATABASE	liste des Databases (ou all)
USER	liste des utilisateurs ou groupes (rôles)
CIDR-ADDRESS	(Classless Inter-DomainRouting), liste @IP+ bit mask
METHOD	indique la méthode d'authentification

Paramètres d'authentification

- **METHOD** : méthode d'authentification

trust	pas d'authentification, pas de mot de passe exigé
reject	rejet systématique
md5	chiffrage md5 , mécanisme de chiffrage

L'utilisateur doit fournir un mot de passe crypté avec md5

crypt	chiffrage par crypt (similaire à md5 avant 7.2)
password	mot de passe en clair
krb4 ou **krb5**	kerberos version 4 ou 5
ident	sameuser ou ident map*

* L'utilisateur est authentifié avec le nom du client (OS du host) et les correspondances indiquées dans le fichier pg_ident.conf

pam	par les mécanismes Linux (Pluggable Authentication Modules)

Configuration de l'environnement et du serveur

Lancement du serveur « postmaster »

```
postmaster -D /usr/local/pgsql/data >fic_log 2>&1 &
```

- Principales options en mode multi utilisateur
  - **-B#** initialise le nombre de buffers 8ko utilisés
  - **-D** répertoire_de_données
  - **-I** active les connexions sécurisées utilisant SSL
  - **-N** nombre_max_connexions
  - **-p port** spécifie le port TCP/IP

Démarrage / arrêt du serveur : commande pg_ctl

pg_ctl start [-w] [-s] [-D datadir] [-p path] [-o options]
pg_ctl stop [-w] [-D datadir] [-m[s(mart)] [f(ast)] [i[mmediate]]]

- `pg_ctl start -l fic_log` → lancement du serveur
- `pg_ctl stop` → arrêt du serveur
- `pg_ctl reload` → re-lecture des fichiers de configuration et prise en compte des paramètres ne nécessitant pas l'arrêt du moteur
- `pg_ctl status` → information sur l'état du serveur

Démarrage automatique du serveur

- Se connecter 'root'
- Copier le script de démarrage du répertoire contrib/start-scripts nommé « **linux** » sous le répertoire **/etc/rc.d/init.d/postgresql**
- Vérifier les variables et path définis dans ce script
- Exécuter la commande **chkconfig** pour démarrer/arrêter automatiquement PostgreSQL au boot/arrêt du système:

```
/sbin/chkconfig -add postgresql
```

⇒ Au boot du système, PostGreSQL sera lancé en mode multi-utilisateur

⇒ A l'arrêt du système, le serveur sera arrêté

3. Base de données PostGreSQL

Base de données PostGreSQL

Création d'une base de données

- Commande SQL **CREATE DATABASE**

```
CREATE DATABASE nom [ [ WITH ] [ OWNER [=]
propriétaire ] [ TEMPLATE [=] modèle ] [ ENCODING
[=] codage ] [ TABLESPACE [=] espacelogique ]
[ CONNECTION LIMIT [=] limite_connexion ] ];
```

- Utilitaire **createdb**

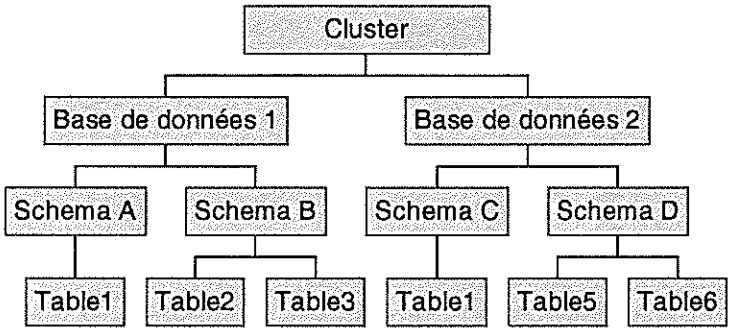
```
createdb [option...] [nombase] [description]
```

- Par défaut, toute base de données sera créée en clonant la base système standard **template1**. Pour spécifier un modèle différent :
  - `CREATE DATABASE db TEMPLATE template0;`
  - `createdb -T template0 db`

Base de données PostGreSQL

Notion de schémas

- Cliquez pour ajouter un plan



Définition de schémas

- Schéma = collection de tables, pouvant aussi contenir des vues, index, séquences, types de données, opérateurs et fonctions
- A chaque connexion, un « schema search_path » est utilisé. Par défaut : $user, public
- Syntaxe :

```
CREATE SCHEMA nom_schema;
```

Base de données PostGreSQL

Accès aux bases de données via psql

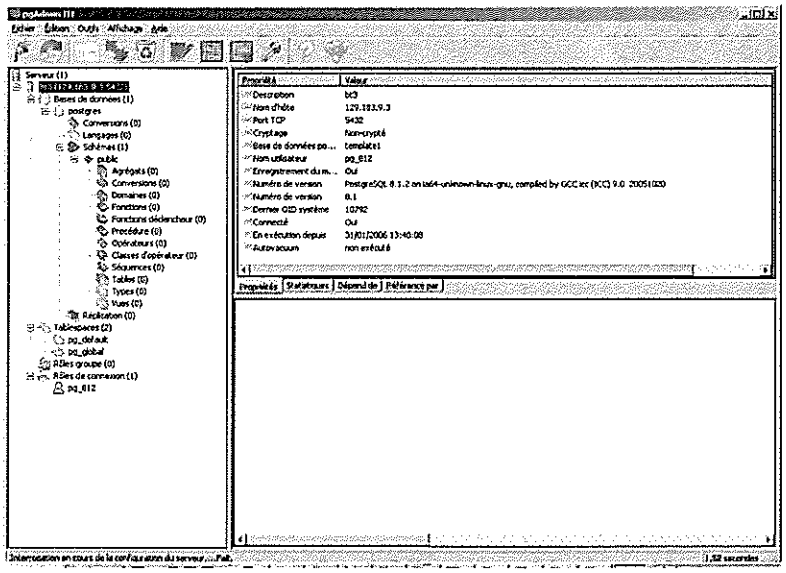
- Utilitaire **psql**
  - connexion aux bases de données
  - exécution de requêtes
  - administration de la base de données
  - possibilité d'exécuter des requêtes à partir d'un fichier
- Connexion via la ligne de commande :

  `psql [option...] [nombase [nomutilisateur]]`
- Détail de psql : `psql --help`
- Sans aucun paramètre précisé, utilisation du nom d'utilisateur système avec lequel on est connecté et connexion à la base de même nom

Accès aux bases de données via pgAdminIII

- PgAdminIII
  - logiciel libre d'administration (rôles, tablespaces, tables, ...)
  - créé pour les versions 7.3 et plus de PostGreSQL
  - interface graphique
  - outil de requêtes SQL
- Utilisable sur les plate-formes Linux, FreeBSD Solaris, Mac OSX et Windows
- Package téléchargeable à partir du site http://www.pgadmin.org/index.php

Accès aux bases de données via pgAdminIII



Base de données PostGreSQL

Création de tables

- Syntaxe SQL :

  CREATE TABLE *nom_table*

  (voir la page

  http://docs.postgresql.fr/pg/8.1.3/sql_cr

)

- Depuis la version 8.0, une table peut être créée dans un autre espace que le répertoire $PGDATA, dans un tablespace

Suppression de tables

- Syntaxe SQL

  `DROP TABLE` *nom* `[, ...] [ CASCADE | RESTRICT ]`
- Seul le propriétaire d'une table peut la détruire
- Pour vider une table de ses lignes, sans détruire la table, utiliser DELETE
- DROP TABLE supprime tout index, règle, déclencheur et contrainte existant sur la table cible
- Pour supprimer automatiquement les objets dépendants de la table (comme les vues), l'option CASCADE doit être ajoutée

Modification de tables

- Ajout d'une colonne :

  ```
  ALTER TABLE nom_table ADD nom_colonne type_colonne
  ```
- Suppression d'une colonne :

  ```
  ALTER TABLE nom_table DROP nom_colonne
  ```
- **Depuis la v8.0**, modification du type d'une colonne :

  `ALTER TABLE nom_table`
  - `ALTER [ COLUMN ]` *colonne* `TYPE` *type* `[ USING` *expression* `]`

Base de données PostGreSQL

Gestion des tablespaces

- Introduits avec la version 8.0
- Espace logique correspondant à un répertoire dans lequel sont stockés des fichiers contenant les objets de la base de données
- Pour créer un nouveau tablespace :

```
CREATE TABLESPACE nom-espace-logique
[ OWNER nom_utilisateur ] LOCATION
'répertoire'
```

- Suite à la création d'un nouveau tablespace :
  - PostGreSQL change les droits du « répertoire »
  - crée un fichier PG_VERSION sous le « répertoire », contenant la version de PostGreSQL au moment de la création du tablespace

Gestion des tablespaces

- ajoute une ligne à la table système *pg_tablespace* et assigne un nouvel OID (object-id) à cette ligne
- le serveur utilise cet OID pour créer un lien symbolique vers le « répertoire » du tablespace

- Réalisés par des liens symboliques, stockés sous le répertoire **$PGDATA/pg_tblspc**

- Par défaut, 2 tablespaces sont créés à l'initialisation du cluster de base de données, localisation $PGDATA :
  - pg_default
  - pg_global

Gestion des tablespaces

- Lister les tablespaces existants sous **psql**
  « \db+ »
  Exemple :

```
template1=# \db+
                    List of tablespaces
    Name     | Owner   | Location | Access
  privileges
------------+--------+----------
  +-------------
 pg_default | pg_812 |          |
 pg_global  | pg_812 |          |
(2 rows)
```

```
  select spcname from pg_tablespace
```

Tablespaces vs schémas

- Ne pas confondre les deux même s'ils permettent d'organiser les tables à l'intérieur d'un cluster
- Tablespace : concerne l'organisation physique des données, définit où sont stockées les données
- Schéma :
  - concerne l'organisation logique des données à l'intérieur d'une base de données
  - impacte le nommage d'accès aux tables, sans se préoccuper de leur localisation physique

Base de données PostGreSQL

Contrôle d'accès simultané

- Standard SQL : 4 niveaux d'isolation
- PostgreSQL maintient la cohérence des données à l'aide d'un modèle multi-versions (Multiversion Concurrency Control, MVCC)
- Deux niveaux d'isolation dans PostgreSQL:
  - *Read Commited* est le niveau d'isolation par défaut de PostgreSQL
  - Serializable
- Read uncommitted => *Read Commited*
- Repeatable read => Serializable

Niveau d'isolation standards SQL

- Standard SQL : 4 niveaux d'isolation

Read uncommitted

Read Commited

Read repetable

Serializable

- Trois comportements « indésirables »

Dirty read

Nonrepeatable read

*p*hantoms

Niveaux d'isolation : 3 comportements indésirables

- Une donnée est lue alors qu'une écriture concurrente est en cours. C'est un cas de ***lecture inconsistante ou sale*** *(Dirty read)*

  Si rollback, perte des mises à jour

Transaction 1	Transaction 2
	update Enr1
select Enr1	
	rollback

Niveaux d'isolation : 3 comportements indésirables

- Une transaction lit deux fois la même donnée et obtient des résultats différents car une modification est intervenue entre temps. C'est un cas de ***lecture non répétitive*** (Nonrepeatable read)

Transaction 1	Transaction 2
select Enr1	update Enr1
actions	update Enr1; commit;
select Enr1	

Niveaux d'isolation : 3 comportements indésirables

PostgreSQL

- Une transaction calcule deux fois un ensemble d'enregistrements satisfaisant la même condition et les résultats diffèrent. C'est un cas de ***lignes fantômes*** (*p*hantoms)

Transaction 1	Transaction 2
select Enr1	
select Enr2	
	update Enr1
	delete Enr2
	insert Enr5
select Enr3	
select Enr4	

BULL

Quatre Niveaux d'isolation Standard SQL

PostgreSQL

Niveau d'isolation	Dirty read	Nonrepeatable read	*p*hantoms
Read Uncommited	Possible	Possible	Possible
Read Commited	Impossible	Possible	Possible
Read Repeatable	Impossible	Impossible	Possible
Serializable	Impossible	Impossible	Impossible

BULL

Contrôle d'accès simultané

- Standard SQL : 4 niveaux d'isolation
- PostgreSQL maintient la cohérence des données à l'aide d'un modèle multi-versions (Multiversion Concurrency Control, MVCC)
- Deux niveaux d'isolation dans PostgreSQL:
  - *Read Commited* est le niveau d'isolation par défaut de PostgreSQL
  - Serializable
- Read uncommitted => *Read Commited*
- Repeatable read => Serializable

Contrôle d'accès simultané MVCC

- Versions concurrentes des lignes
- Version de la base au démarrage de la transaction
- Lire ne bloque jamais l'écriture
- Écrire ne bloque jamais la lecture
- L'écriture bloque tout autre écriture sur la ligne
- SET TRANSACTION mode_transaction (valide pour la transaction seulement)
- SET SESSION CHARACTERISTICS AS TRANSACTION *mode_transaction* (mode de transaction configurable au niveau d'une session)

Contrôle d'accès Verrous

- **Verrou exclusif** :un verrou est dit exclusif s'il interdit toute autre obtention d'un verrou, de quelque type que ce soit, sur l'élément verrouillé par une autre transaction
- **Verrou partagé** : si une transaction obtient un verrou partagé sur un élément, une autre transaction peut obtenir aussi un verrou partagé sur cet élément ; mais aucune transaction ne peut obtenir de verrou exclusif sur cet élément tant que tous les verrous partagés ne sont pas libérés
- Positionnement explicite des verrous par PostGreSQL pour la plupart des commandes
- PostGreSQL fournit des modes de verrous pour contrôler le verrouillage par l'application dans des situations où MVCC n'a pas le comportement désiré
- Verrous au niveau table et lignes. Consultation des verrous dans la vue *pg_locks*

Base de données PostGreSQL

Création d'index

- 4 types d'index :
  - **B-tree** : index par défaut, utilisé dans des égalités et des recherches sur des tranches de valeurs de l'index
  - **R-tree** : requêtes sur des données spatiales

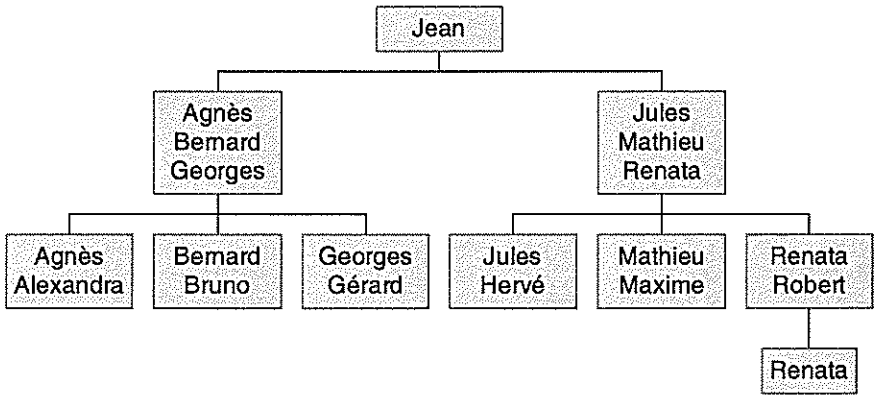
```
CREATE INDEX nom ON table USING RTREE
(colonne);
```

  - **Hash** : simples comparaisons d'égalité

```
CREATE INDEX nom ON table USING HASH
(colonne);
```

  - **GiST** : index pouvant être étendu à des objets complexes (images, ...)

Création d'index : B-tree



Création d'index

- **Index uniques** : garantir l'unicité des valeurs d'une colonne, ou l'unicité des valeurs combinées de plusieurs colonnes
- **Index multi-colonnes** : porte sur plus d'une colonne d'une table
- **Index sur les expressions** : fonction ou une expression calculée à partir d'une ou plusieurs colonnes de la table
- **Index partiel** : index construit sur un sous-ensemble d'une table

Base de données PostGreSQL

Chargement des tables

- Commande **INSERT** :

```
INSERT INTO nom_table VALUES (…);
```

- Commande **COPY** : plus performant pour charger beaucoup de données dans une table
  - `COPY nom_table TO nom_fichier` : écrit le contenu d'une table dans un fichier
  - `COPY nom_table FROM nom_fichier` : insère les données lues à partir d'un fichier dans la table

Base de données PostGreSQL

Analyse des tables

- Commande ANALYZE :
  - collecte des statistiques sur le contenu des tables de la base de données
  - stocke les résultats dans la table système « *pg_statistic* »
- Échantillons de valeurs les plus communes dans chaque colonne pour effectuer une distribution
- Seules quelques lignes de la table sont analysées au lieu d'examiner la table complète
- VACUUM ANALYZE...

Cours PostGreSQL : Chapitre 4

4. Gestion des rôles et accès à la base de données

Gestion des rôles et accès à la base de données

Gestion des rôles

- Intégrés avec la version 8.1
- Comprend les concepts des « utilisateurs » et des « groupes »
- Peuvent posséder des objets de la base de données et affecter des droits sur ces objets à d'autres rôles
- Séparés des utilisateurs du système d'exploitation
- Un utilisateur est le propriétaire des objets qu'il crée
- Un rôle peut être vu soit comme un utilisateur, soit comme un groupe d'utilisateurs

Gestion des rôles (suite)

- Création d'un rôle utilisateur :
  - `CREATE ROLE nom_utilisateur LOGIN;`
- Création d'un rôle groupe :
  - `CREATE ROLE nom_utilisateur;`
  - équivalent à NOLOGIN
- Paramètres principaux :
  - CREATEDB / NOCREATEDB
  - SUPERUSER / NOSUPERUSER : l'utilisateur créé avec SUPERUSER possède tous les privilèges
  - CREATEROLE / NOCREATEROLE : le nouveau rôle pourra créer d'autres rôles
  - INHERIT / NOINHERIT : le nouveau rôle hérite des privilèges des rôles dont il sera membre

Gestion des rôles (suite)

  - IN ROLE *rolename* : liste un ou plusieurs rôles existants auxquels le nouveau rôle sera immédiatement ajouté en tant que nouveau membre
  - ROLE *rolename* : liste un ou plusieurs rôles existants qui sont automatiquement ajoutés comme membre du nouveau rôle

- Par défaut, il existe un rôle utilisateur prédéfini qui a le même nom que l'utilisateur du système d'exploitation ayant initialisé le groupe de bases de données

- La commande `CREATE USER` est équivalent à `CREATE ROLE LOGIN`

Gestion des rôles

PostgreSQL

- Pour déterminer l'ensemble des rôles existants
  - `SELECT * FROM pg_roles;`
  - ou **\du** sous psql
- Pour déterminer l'ensemble des rôles utilisateurs
  - `SELECT * FROM pg_user;`
  - `SELECT * FROM pg_roles;`

Gestion des rôles et accès à la base de données

PostgreSQL

Types de privilèges

- **SELECT (symbole r)**
  - appliqué à des tables, vues et séquences
  - contrôle le droit de faire des SELECT sur les colonnes d'une table, vue ou séquence
- **DELETE (symbole d)**
  - appliqué à des tables, vues et séquences
  - contrôle le droit de supprimer des données d'une table, vue ou séquence
- **INSERT (symbole a)**
  - appliqué à des tables, vues et séquences
  - contrôle le droit d'insérer de nouvelles valeurs dans une table, vue ou séquence

Types de privilèges

- **UPDATE (symbole w)**
  - appliqué à des tables, vues et séquences
  - contrôle le droit de mettre à jour des valeurs dans une table, vue ou séquence
- **RULE (symbole R)**
  - appliqué à des tables et des vues
  - contrôle le droit de créer de nouvelles règles sur une table ou vue
- **REFERENCES (symbole x)**
  - appliqué à des tables
  - contrôle le droit de lier deux tables avec une contrainte de clé étrangère

Types de privilèges

- **TRIGGER (symbole t)**
  - appliqué à des tables
  - contrôle le droit de créer des triggers sur une table
- **CREATE (symbole C)**
  - appliqué à des bases de données, tablespaces et schémas
  - contrôle le droit de créer de nouveaux schémas à l'intérieur d'une base de données, de nouveaux objets dans un schéma ou de nouveaux index (ou tables) dans un tablespace
- **TEMPORARY ou TEMP (symbole T)**
  - appliqué à des bases de données
  - contrôle le droit de créer des tables temporaires dans une base de données

Types de privilèges

- **EXECUTE (symbole X)**
  - appliqué à des fonctions
  - contrôle le droit d'exécuter une fonction
- **USAGE (symbole U)**
  - appliqué à des schémas, langages
  - autorise l'accès aux objets contenus dans le schéma spécifié, ou la création de nouvelles fonctions avec un langage donné
- **ALL PRIVILEGES (symboles arwdRxt)**
  - donne tous les droits disponibles en une fois

Gestion des rôles et accès à la base de données

Gratifier / révoquer des privilèges

- **Gratifier**, deux variantes :
  1. donner des droits sur un objet de la base de données pour un ou plusieurs rôles
  2. rendre un ou plusieurs rôles membre(s) d'un autre rôle. L'appartenance à un rôle est significatif car il peut apporter tous les droits accordés à ce rôle à tous ses membres

```
GRANT type_privilège ON nom_table TO rôle
utilisateur | rôle groupe
```

- **Révoquer** : retirer des droits précédemment attribués à un ou plusieurs rôles

```
REVOKE type_privilège ON nom_table FROM rôle
utilisateur | rôle groupe
```

- Pour visualiser les privilèges sous **psql**, « `\z` » ou « `\dp` »

5. Les journaux

Les journaux

Journal des erreurs du moteur

- Plusieurs méthodes pour la journalisation des messages du serveur, dont stderr et syslog
- Possibilité de sélectionner le niveau de traces des clients, du serveur et des requêtes SQL
- Possibilité de faire une rotation des journaux

Plusieurs méthodes pour journaliser

- Paramètre **log_destination** de « postgresql.conf »
- Options à définir avant le démarrage du postmaster
  - `log_destination=stderr` que l'on peut rediriger avec le paramètre `redirect_stderr=true` puis
    - `log_directory`
    - `log_filename`
    - `log_truncate_on_rotations`
    - `log_rotation_age`
    - `log_rotation_size`
  - `log_destination=syslog` (Unix)
  - `log_destination=eventlog` (Windows)

Journalisation via syslog

- Par défaut, les messages système sont journalisés dans le fichier « /var/log/messages »
- Deux configurations à faire pour utiliser syslog :
  - Ajouter dans /etc/syslog.conf le service PostGreSQL
  - Positionner dans postgresql.conf :
    - `log_destination=syslog`
    - `syslog_ident='postgres'`
    - `syslog_facility='LOCALx'`
- Relancer syslog et postgresql via
  - service syslog restart
  - service postgresql restart (ou relancer le moteur)

Niveaux de traces

- Les évènements tracés sont définis dans « postgresql.conf »
- Niveaux en priorité croissante : DEBUG5, ..., DEBUG1, INFO, NOTICE, WARNING, ERROR, FATAL, PANIC
- Messages des clients : `client_min_messages=niveau`
- Messages du moteur : `log_min_messages=niveau`
- Ordres SQL en erreur : `log_min_error_statement=niveau`

Que journaliser ?

PostgreSQL

- Différents évènements journalisables :
  - Les connections
    - nom de l'utilisateur
    - adresse IP du serveur
  - Les déconnections
  - Les ordres SQL :
    - modification de données
    - modification de structures des tables
    - tous les ordres SQL
    - la durée de l'ordre SQL

BULL

Les journaux

PostgreSQL

BULL

Rotation des journaux Log

- Les journaux d'erreurs peuvent être assez volumineux en fonction du niveau de traces
  ⇒nécessité de faire une rotation
- Si stderr redirigé dans un fichier, la seule façon de le tronquer est d'arrêter et de redémarrer le postmaster
- Si `redirect_stderr=true`, il est possible de configurer un programme interne de rotation à l'aide des paramètres de contrôle :
  - log_rotation_age : durée de vie maximum d'un journal individuel
  - log_rotation_size : la taille maximum d'un journal individuel
  - log_truncate_on_rotation : possibilité de tronquer le journal

Les journaux

Journal des transactions (WAL)

- Objectifs des WAL(Write-Ahead Logging) :
  - Cohérence des données en cas de panne
  - Fiabilité
  - Limitation du nombre d'écritures disque

- Concept des WAL *:*
  - s'assurer que l'historique du déroulement des transactions sur disque a été sauvegardé vers le stockage permanent avant d'écrire les « data » sur disque

Journal des transactions (WAL)

- *Write-Ahead Logging* (*WAL*) : écriture d'un journal de transactions

- Écriture dans les fichiers de données uniquement après écriture dans un journal

- La description du travail effectué par les transactions est stockée dans un premier temps dans les WAL buffers, puis est écrite sur disque sous le répertoire *pg_xlog*

Les journaux

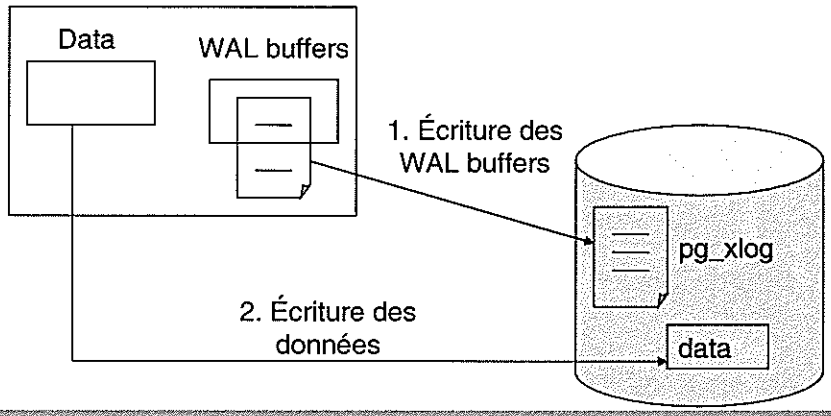
Checkpoints

- Les checkpoints assurent la synchronisation entre la version de la base de données en mémoire et la version de la base de données sur disque

- Quand se produisent les checkpoints ?
  - lors d'un ordre SQL « CHECKPOINT »
  - dès que « checkpoint_time » est écoulé
  - à l'arrêt normal de l'instance
  - tous les « checkpoint_segment » .....

Checkpoints

- Les « checkpoints » garantissent la mise à jour des fichiers de données avec les informations enregistrées dans le journal depuis le « checkpoint » précédent



Les journaux

Mécanisme interne des WAL

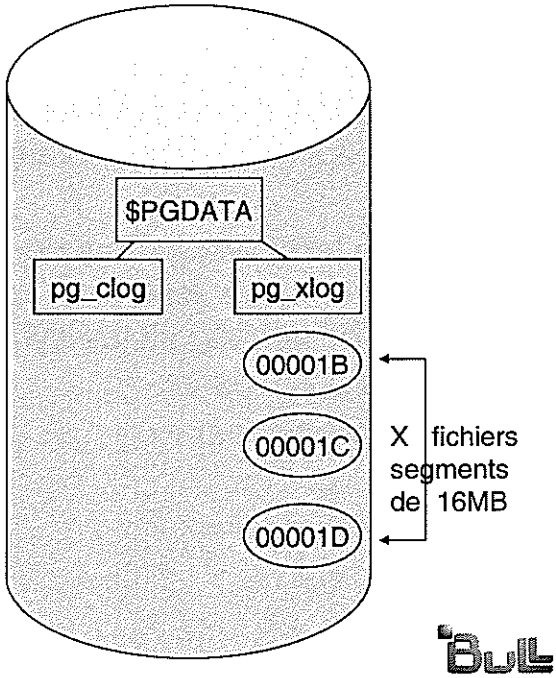
Nombre de segments WAL :

Min :1

Max : 2*checkpoint_segment + 1

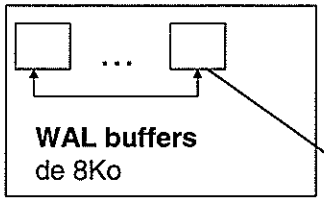
Noms des segments WAL :

000000010000000000000000



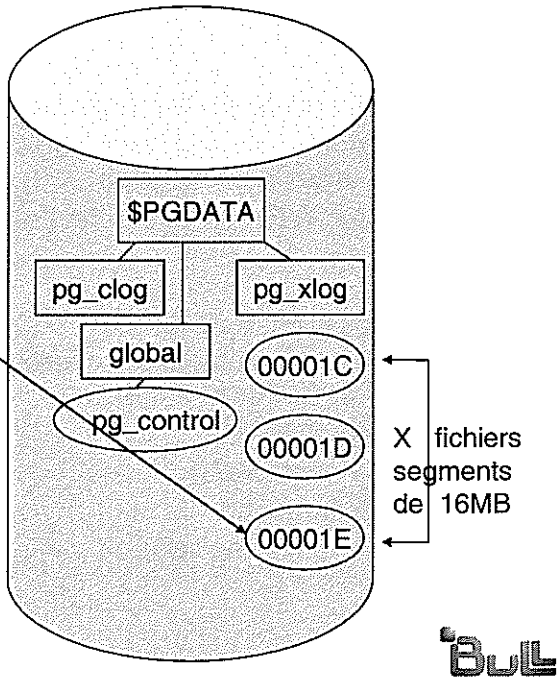
Mécanisme interne des WAL

Mémoire partagée



Checkpoint_timeout
WAL buffers pleins
CHECKPOINT
COMMIT

X segments =
2*checkpoint_segment + 1



Écriture sur le disque des WAL

PostgreSQL

- La copie sur disque d'Unix peut rendre la main même si tous ses buffers ne sont pas totalement écrits sur disque
- Pour éviter une incohérence des données dûe au décalage entre la réelle écriture et le retour de fin de copie, le choix de la méthode de synchronisation peut être indiqué au postmaster
- Paramètres à positionner :
  - `fsync=true`
  - `wal_sync_method=fsync`

WAL & Point-In-Time-Recovery (PITR)

PostgreSQL

- Depuis la v8.0, il est possible d'archiver les WAL
- Lorsque le système n'utilise pas l'archivage des WAL, les fichiers WAL sont créés puis recyclés
- Avec l'archivage des WAL, on peut effectuer une sauvegarde incrémentale :
  - faire un backup complet
  - archiver périodiquement les WAL
  - si crash, restaurer le backup complet et rejouer tous les WAL en séquence jusqu'à un instant t (PITR)
- Méthode d'archivage flexible
- Nouveau process « archiver »
- Paramètre de configuration :
  - `archive_command` (cp, tar, cpio, ...)

6. Sauvegarde et restauration

Sauvegarde et restauration

Sauvegarde d'une base données

PostgreSQL

- Deux méthodes :
  1. Créer une archive du file system, comprenant la base de données (tar, cpio, ou backup)
  2. Créer un fichier dump via l'utilitaire pg_dump, décrivant comment ré-écrire les données de la base de données

BULL

Sauvegarde d'une base données avec pg_dump

PostgreSQL

- Utilitaire pour sauvegarder une base de données
- Sauvegardes à « chaud »
- Ne bloque pas l'accès des autres utilisateurs (ni en lecture ni en écriture)
- Plusieurs formats de fichiers d'archive possibles :
  - un fichier de scripts SQL en texte simple (par défaut)
  - une archive tar utilisable par « pg_restore », option « -Ft »
  - une archive personnalisée convenable pour « pg_restore », option « -Fc »
- Commande : `pg_dump [option...] [nom_base]`

BULL

Sauvegarde d'une base données avec pg_dump (suite)

- Choix de COPY ou INSERT pour le rechargement ultérieur des tables (par défaut COPY)
  - -d (--inserts) : sauvegarde les données avec des commandes INSERT
  - -D (--columns –inserts) : sauvegarde les données avec des commandes INSERT et des noms de colonnes explicites
- Possibilité de sauvegarder uniquement une table avec l'option « `-t nom_table` »
- Sauvegarde les privilèges mais les rôles sont à re-créer après restauration

Sauvegarde d'une base données avec pg_dump (suite)

- En fonction des OS, taille maximum des fichiers de 2GB à 4GB
- Pour contourner ce problème : compression ou découpage
  - `pg_dump nom_base | gzip -9 > nom_base.gz`
  - `pg_dump nom_base | bzip2 -9 > nom_base.bz2`
  - `pg_dump --format c nom_base > nom_base.bak`
  - `pg_dump nom_base | split --bytes=100m - nom_base`

(génère un ensemble de fichiers de 100 mégas avec les noms « nom_baseaa » à « nom_basezz »)

Sauvegarde et restauration

Restauration d'une base données

- Si l'archive générée avec pg_dump est un script SQL, restauration via psql sinon utilisation de « pg_restore »

- Restauration via psql :

```
psql -d base -f base.out
```

- Restauration via pg_restore à partir d'un fichier d'archive créé par pg_dump :

```
pg_restore -d nouvellebase base.tar
```

Sauvegarde et restauration

Sauvegarde d'un cluster de bases données avec pg_dumpall

- Outil pour sauvegarder toutes les bases de données d'un cluster dans un script SQL
- Invocation de pg_dump pour chaque base de données du cluster
- Sauvegarde des rôles de la base de données et des privilèges
- Mêmes options disponibles que pour pg_dump mais produit une archive en format texte seulement
- Commande :

```
pg_dumpall > allDBs.out
```

Sauvegarde et restauration

Restauration d'un cluster

- Le script généré par « pg_dumpall » contient les ordres SQL de création de toutes les bases de données

- Commande :

```
psql -f db.out postgres
```

7. Optimisation et performances

Optimisation et performances

VACUUM

- Les versions périmées des lignes après un UPDATE ou un DELETE ne sont pas supprimées immédiatement
- VACUUM permet de réutiliser les lignes associées à des données « flaggées » périmées
- VACUUM FULL récupère l'espace disque et réorganise les tables sur disque

VACUUM (suite)

- Avec VACUUM FULL, un verrou exclusif est placé sur chaque table avant qu'elle ne soit traitée
- VACUUM FULL demande également beaucoup d'activités sur disque pour réorganiser les fichiers de données

⇒ Les performances de requêtes concurrentes sur la base de données peuvent être diminuées

Optimisation et performances

VACUUM et ANALYZE

- Les statistiques utilisées par l'optimiseur sont collectées par la commande ANALYZE, qui peut être invoquée seule ou comme une option de VACUUM
- VACUUM ANALYZE collecte des statistiques sur toute la base de données
- VACUUM et ANALYZE mettent à jour le catalogue système pg_class :
  - mise à jour du champ reltuples, nombre de lignes de la table
  - mise à jour du champ relpages, taille du fichier disque, exprimée en pages

VACUUM et ANALYZE (suite)

- ANALYZE et VACUUM ANALYZE mettent à jour le catalogue système pg_statistic
  - stocke des données statistiques sur les tables et les valeurs des expressions d'index
  - une entrée pour chaque colonne de table qui a été analysée
  - ne doit pas être lisible par le public car même les données statistiques peuvent être considérées comme sensibles
- La vue pg_stats fournit un accès aux informations stockées dans la table système pg_statistic
- Cette vue est aussi conçue pour afficher l'information dans un format plus lisible

Optimisation et performances

Démon autovacuum

- A partir de la v8.1, le démon autovacuum automatise l'exécution des commandes VACUUM et ANALYZE
- Pour activer autovacuum, configurer :
  - `stats_start_collector=true`
  - `stats_row_level=true`
- autovacuum s'exécute toutes les `autovacuum_naptime` secondes et détermine quelles bases de données traiter (bases dont les tables ont un grand nombre de lignes insérées, mises à jour ou supprimées)

Optimisation et performances

Exécution d'une requête

Client

Ordre SQL

Serveur PostGreSQL

Texte de la requête

« Parser » de requête

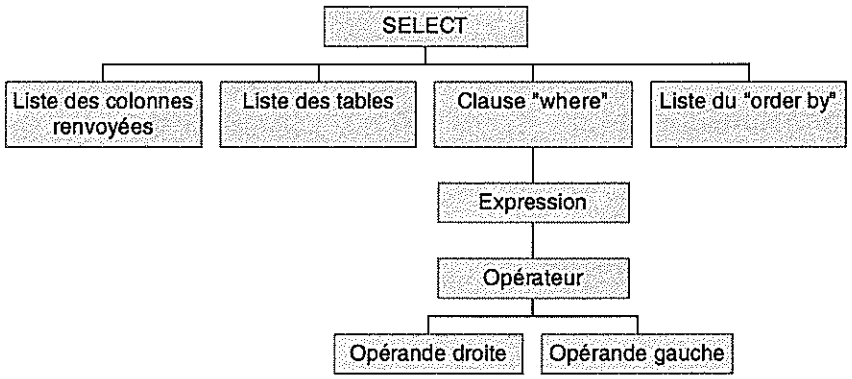
Vérification de la syntaxe de la requête

Si syntaxe correcte

Création d'un « **parse tree** »

Exécution d'une requête (suite)

Exemple de "parse tree"



Exécution d'une requête (suite)

- Le « parse tree » est confié à l'optimiseur de requêtes afin qu'il trouve tous les plans d'exécution possibles de cette requête
- Lorsque tous les plans d'exécution possibles ont été générés, l'optimiseur cherche le moins coûteux
- A chaque plan est assigné un coût d'exécution, mesuré en unité de disque I/O

- Une fois le meilleur plan d'exécution sélectionné, la requête est exécutée et le résultat est retourné au client

Optimisation et performances

Lecture d'un plan d'exécution

- Le plan d'exécution affiche comment la (les) table(s) référencée(s) par l'instruction sera(ont) parcourue(s) et quels algorithmes de jointure sera(ont) utilisés
- EXPLAIN permet d'avoir le plan d'exécution choisi par l'optimiseur
- EXPLAIN peut être lancé sur toute requête SELECT, INSERT, UPDATE, DELETE, EXECUTE ou DECLARE
- Coûts mesurés en unités de récupération de page disque : un coût de 1,0 équivaut à une lecture de page disque

Lecture d'un plan d'exécution (suite)

- Exemple :

```
entrepot=# EXPLAIN SELECT * FROM client;
                          QUERY PLAN
-----------------------------------------------------
 Seq Scan on client  (cost=0.00..10.90 rows=90
  width=864)
(1 row)
```

- Les nombres donnés par EXPLAIN sont :
  - le coût d'exécution avant que la première ligne ne soit renvoyée
  - le coût total estimé pour renvoyer toutes les lignes
  - le nombre de lignes estimées en sortie par ce nœud de plan
  - la largeur moyenne estimée (en octets) des lignes en sortie par ce nœud de plan

Lecture d'un plan d'exécution (suite)

PostgreSQL

- EXPLAIN ANALYZE ne planifie pas seulement l'instruction mais l'exécute
- Renvoie le temps « réel » passé (en millisecondes) et le nombre total de lignes renvoyées
- Permet de vérifier si les estimations du planificateur sont proches de la réalité

```
entrepot=# EXPLAIN ANALYZE SELECT * FROM client;

   QUERY PLAN

-----------------------------------------------------
 Seq Scan on client  (cost=0.00..10.90 rows=90
  width=864) (actual time=0.016..0.030 rows=13)
 Total runtime: 0.089 ms
(2 rows)
```


Lecture d'un plan d'exécution (suite)

PostgreSQL

- Si ANALYZE n'est pas exécutée au préalable pour enregistrer les statistiques sur la distribution des données à l'intérieur de la table :
  - les coûts estimés ne sont pas cohérents avec les données réelles
  - un plan de requête médiocre pourrait être choisi

- Exemple : la table « client » contient réellement 13 lignes

```
entrepot=# select  count(*) from client;
 count
-------
    13
(1 row)

entrepot=# ANALYZE client;
ANALYZE
```

Lecture d'un plan d'exécution (suite)

```
entrepot=# EXPLAIN SELECT * FROM client;
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on client  (cost=0.00..1.13 rows=13
  width=246)
(1 row)


entrepot=# EXPLAIN ANALYZE SELECT * FROM client;
                                                QUERY PLAN
-----------------------------------------------------------
 Seq Scan on client  (cost=0.00..1.13 rows=13
  width=246) (actual time=0.014..0.028 rows=13)
 Total runtime: 0.080 ms
(2 rows)
```


Opération « Seq scan »

- Parcours séquentiel de la table
- Une table peut contenir des lignes supprimées et des lignes non visibles car non « committées »
- Seq Scan ne les inclut pas dans les résultats mais les parcourt ce qui peut être coûteux en terme de temps d'exécution

- Seq Scan est choisi par le planificateur :
  - s'il n'existe pas d'index pouvant être utilisé pour exécuter la requête
  - s'il considère que le parcours entier de la table est moins coûteux

Opération « Index Scan »

- Parcours la structure d'un index
- Avantages par rapport à un Seq Scan :
  - Index Scan ne lira pas toutes les lignes de la table
  - Index Scan retournera les données ordonnées selon l'index
- Le planificateur utilise un Index Scan lorsqu'il peut :
  - réduire le nombre de données parcourues
  - éviter un tri des données à cause du tri implicite offert par un index

Opération « Bitmap Index »

- Avant la v8.1, un seul index pouvait être utilisé à la fois pour le parcours d'une table
- Les Bitmap Index permettent de combiner plusieurs index pour exécuter une requête
- Le plus souvent utilisé dans les applications décisionnelles
- Pour chaque valeur distincte de la colonne indexée, l'index contient un tableau de bits indiquant si chaque ligne de la colonne a cette valeur ou non
- Les différents tableaux de bits sont ensuite combinés pour répondre aux sélections de type « col1=valeur1 AND col2=valeur2 » où col1 et col2 sont indexées

Opération « Bitmap Index » (suite)

Table produit :

ident	couleur	taille
1	brun	medium
2	rouge	medium
3	rouge	small
4	bleu	large
5	rouge	medium
6	brun	Small

Couleur = 'bleu'	0001001010100010
Couleur = 'rouge'	0110100100001001
Couleur = 'brun'	1000010001010100

AND

Taille = 'small'	0010010101000101
Taille = 'medium'	1100101000010100
Taille = 'large'	0001000010101010

Requête :

```
SELECT count(*)
FROM PRODUIT
WHERE taille = 'medium' AND couleur = 'rouge';
```


Opération « Sort »

- PostGreSQL a deux stratégies pour trier les données
  - un tri en mémoire, en fonction de la taille définie dans le paramètre `work_mem`
  - un tri sur disque
- Si la taille nécessaire dépasse `work_mem`, Sort travaillera en mémoire puis dans des fichiers temporaires, sinon le tri sera fait entièrement en mémoire
- Sort est utilisé pour :
  - des opérations de tri (ORDER BY)
  - des jointures
  - des intersections de données, ...

Opération « Unique »

PostgreSQL

- Élimine les données dupliquées dans les résultats
- Unique compare chaque ligne avec la précédente trouvée. Si les valeurs sont identiques, la ligne en doublon est supprimée des résultats
- Unique est utilisé pour :
  - satisfaire un DISTINCT
  - pour éliminer des doublons dans un UNION

BULL

Opérations «Limit », « Aggregate », « Append »

PostgreSQL

- Limit limite le nombre de lignes renvoyées par un requête et est appliqué pour satisfaire un LIMIT et/ou OFFSET dans une requête
- Aggregate est appliqué pour satisfaire une fonction AVG(), COUNT(), MAX(), MIN(), SUM(), ...
- Append est utilisé pour implémenter un UNION entre une requête A et une requête B
  - Les coûts renvoyés par Append correspondent à la somme des coûts des requêtes A et B

BULL

Opération « Nested Loop »

PostgreSQL

- Utilisé pour réaliser une jointure entre deux tables
- Exemple :

```
SELECT * FROM client, commande
WHERE client.client_id = commande.client_id;
```

  - La table commande est la « outer table »
  - La table client est la « inner table »
  - La outer table est toujours parcourue en premier
  - Pour chaque ligne de la outer table, Nested Loop lit la ligne correspondante dans la inner table en utilisant, s'il existe, l'index défini sur la colonne de jointure
- Nested Loop est utilisé pour améliorer les performances de jointures internes, jointures externes, et unions

BULL

Opération « Merge Join »

PostgreSQL

- Merge Join est également utilisé pour améliorer les performances de jointures internes, jointures externes, et unions

- Différence avec Nested Loop : Merge Join effectue les jointures entre deux tables triées (au préalable ou triées par Merge Join lui-même)

BULL

Opérations « Hash » et « Hash Join »

PostgreSQL

- Ces deux opérations fonctionnent ensemble
- Hash Join n'a pas besoin que les tables soient triées sur la colonne de jointure
- Hash Join commence par créer une « inner table » en utilisant l'opération Hash
- L'opération Hash crée un index Hash temporaire qui couvre la colonne de jointure dans la « inner table »
- Hash Join lit ensuite chaque ligne dans la « outer table », parcourt la colonne de jointure et cherche dans l'index temporaire les valeurs correspondantes
- Hash Join est utilisé dans des jointures internes, jointures externes, et unions

Opérations « Group », « Subquery Scan » et « Subplan »

PostgreSQL

- Group est utilisé pour satisfaire un GROUP BY
- Subquery Plan est utilisé pour exécuter un UNION
- Subplan est utilisé pour exécuter des sous-selects

Optimisation et performances

Interaction entre jointures, index et choix de l'optimiseur

- Un index :
  - permet d'augmenter les performances d'une base de données
  - permet de retrouver une ligne spécifique bien plus rapidement que sans index
- Le planificateur réalisera une jointure entre « n » tables dans n'importe quel ordre
- Le planificateur va explorer toutes les possibilités de jointure pour trouver le plan de requête le plus efficace
  - ⇒ *Le nombre d'ordres de jointures possibles grandit de façon exponentielle au fur et à mesure que le nombre de tables augmente*

Optimisation et performances

Conseils d'indexation

- Indexer les colonnes de jointure
- Indexer les colonnes de filtre
- Indexer les colonnes fréquemment utilisées pour le tri
- Créer des index multi-colonnes

8. Genetic Query Optimizer

Genetic Query Optimizer (geqo)

Définition du Genetic Query Optimizer

- Le nombre de plans croît de façon exponentielle avec le nombre de jointures

⇒ Difficultés à exécuter et à optimiser la jointure entre tables

- L'implémentation de l'optimiseur produit un ordre de jointure presque optimal mais peut prendre beaucoup de temps et d'espace mémoire en fonction du nombre de jointures dans une requête

⇒ L'optimiseur ordinaire de requêtes de PostgreSQL est peu efficace pour de telles requêtes

Définition du Genetic Query Optimizer (suite)

- Genetic Query Optimizer est basé sur un algorithme génétique, c'est-à-dire un algorithme tentant de faire de la planification de requêtes sans recherche exhaustive
- Dans un algorithme génétique, l'ensemble des solutions possibles pour le problème d'optimisation est considéré comme une population d'individus
- Le degré d'adaptation d'un individu dans son environnement est spécifié par sa forme physique
- Les coordonnées d'un individu dans l'espace de recherche sont représentées par des chromosomes, un ensemble de chaînes de caractères

Définition du Genetic Query Optimizer (suite)

PostgreSQL

- Depuis la v6.5, le module geqo est la solution du problème d'optimisation des requêtes
- Les plans de requêtes possibles sont codés comme des chaînes d'entiers
- Chaque chaîne représente l'ordre de jointure d'une relation de la requête à une autre
- L'implémentation d'un algorithme génétique permet une convergence rapide vers des plans de requêtes améliorés

BULL

Genetic Query Optimizer (geqo)

PostgreSQL

BULL

Paramètres liés à geqo, fichier postgresql.conf

- geqo
  - Active ou désactive l'optimisation génétique des requêtes – activé par défaut
- geqo_threshold
  - Indique que geqo sera utilisé si le nombre d'éléments impliqués dans la clause FROM est supérieur ou égale à ce seuil
  - Valeur par défaut : 12
- geqo_effort
  - Contrôle l'équité entre le temps de planification et l'efficacité du plan de requête dans GEQO
  - Valeur entre 1 et 10, et par défaut : 5

Paramètres liés à geqo, fichier postgresql.conf (suite)

- geqo_pool_size
  - Contrôle la taille de la queue, nombre d'individus dans une population génétique, utilisée par GEQO
- geqo_generations
  - Contrôle le nombre d'itérations de l'algorithme utilisé par GEQO
  - Valeur : au moins 1, si 0 alors valeur choisie en fonction de geqo_pool_size
- geqo_selection_bias
  - Contrôle le biais de sélection utilisé par GEQO, pression sélective à l'intérieur de la population
  - Valeur par défaut : 2,00

9. Revue des paramètres d'optimisation

Revue des paramètres d'optimisation

Paramètres d'optimisation de postgresql.conf

- Mémoire
  - postmaster utilise des ressources système dont les IPC (mémoire partagée et sémaphores)
  - La mémoire partagée utilisée par le postmaster fait partie des segments de mémoire partagée du système
  - Pour visualiser le nombre de segments utilisés par PostGreSQL :

```
ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
     status
0x00000000 32768      gdm        600        393216     2
     dest
0x00000000 65537      ora101     640        1343488    14

0x00000000 294920     ora101     640        1241513984 25

0x0396f42c 327689     ora101     640        16384      25

0x0052e2c1 2785290    pg_812     600        11083776   2
```


Paramètres d'optimisation de postgresql.conf (suite)

- La mémoire allouée pour la mémoire partagée de PostGreSQL ne doit pas excéder SHMMAX

Nom	Multiplicateur approximatif (octets par incrément)
max_connections	400 + 220 * max_locks_per_transaction
max_prepared_transactions	600 + 220 * max_locks_per_transaction
shared_buffers	8300 (assuming 8K BLCKSZ)
wal_buffers	8200 (assuming 8K BLCKSZ)
max_fsm_relations	70
max_fsm_pages	6

Paramètres d'optimisation de postgresql.conf (suite)

- Approximativement, PostGreSQL consomme :
  250 Ko + 8,2KB * shared_buffers
  + 14,2KB * max_connections

- **Attention :** cette valeur qui ne doit pas excéder SHMMAX (taille maximum d'un segment de mémoire partagée)

- Pour connaître la valeur actuelle de SHMMAX :
  `cat /proc/sys/kernel/shmmax`

Paramètres d'optimisation de postgresql.conf , mémoire

- **shared_buffers**
  Initialise le nombre de buffers en mémoire partagée utilisés par le serveur de bases de données
  Min : 16 ou 2* max_connections
  Valeur par défaut 1000

- **temp_buffers**
  Configure le nombre maximum de buffers temporaires utilisés au niveau de la session seulement pour accéder aux tables temporaires
  Valeur par défaut 1000

- **max_prepared_transactions**
  Configure le nombre maximum de transactions simultanées dans l'état « prépare »
  Si 0 => désactive la fonctionnalité
  Valeur par défaut 5

Paramètres d'optimisation de postgresql.conf , mémoire

- **maintenance_work_mem**
  Spécifie la mémoire maximum utilisée dans les opérations de maintenance telles que VACUUM, CREATE INDEX et ALTER TABLE ADD FOREIGN KEY (anciennement vacuum_mem)
  Valeur par défaut 16384 (soit 16 Mo)

- **max_stack_depth**
  Spécifie la profondeur maximum de la pile d'exécution du serveur
  Valeur par défaut 2048 (soit 2 Mo)

- **work_mem**
  Spécifie la mémoire à utiliser pour les opérations de tri interne et pour les opérations de « hachage » avant de basculer sur des fichiers temporaires sur disque(anciennement sort_mem)
  Valeur par défaut 1024 (soit 1 Mo).

Paramètres d'optimisation de postgresql.conf , mémoire

Free Space Map

« Mapping » des emplacements libres dans la base de données.
Si la taille est insuffisante, PostGreSQL demandera plus d'espace disque au système d'exploitation lorsqu'il aura besoin de stocker de nouvelles données

- **max_fsm_pages**
  Initialise le nombre maximum de pages disque tracées
  Valeur par défaut 20000 (> à 16 * max_fsm_relations)

- **max_fsm_relations**
  Initialise le nombre maximum de relations (tables et index) tracées
  Valeur par défaut 1000

Revue des paramètres d'optimisation

Statistiques et surveillance du serveur

Collecteur de statistiques sur l'activité du serveur

- stats_start_collector= true pour activer le collecteur

- **stats_command_string**
  Active la récupération de statistiques sur les commandes en cours d'exécution par session, avec le temps d'exécution de la commande.
  Les données produites sont accessibles via la vue système « pg_stat_activity »

- **stats_block_level**
  Active la récupération des statistiques au niveau du bloc disque sur l'activité de la base de données.
  Les données produites sont accessibles via la famille de vues système « pg_stat_* » et « pg_statio_* »

Statistiques et surveillance du serveur (suite)

- **stats_row_level**

  Active la récupération de statistiques au niveau ligne sur l'activité de la base de données.

  Les données produites sont accessibles via la famille de vues système « pg_stat_* » et « pg_statio_* »

- **stats_reset_on_server_start**

  Si ce paramètre est activé, les statistiques récupérées sont réinitialisées à chaque fois que le serveur est redémarré

Cours PostGreSQL : Chapitre 10

10. Bilan sur l'administration

Bilan sur l'administration

Tâches d'administration courantes : VACUUM

- VACUUM et ANALYZE incontournables pour :
  - récupérer l'espace disque engendré par les mises à jour et suppression de données
  - mettre à jour les statistiques pour un choix judicieux d'exécution des requêtes
  - éviter de perdre les anciennes données de transactions (XID, transaction ID wraparound)
- Quand lancer VACUUM ?
  - dépend de l'application et de la disponibilité de la base de données pour des tâches de maintenance ou de batch
  - choisir, par exemple, les tables fréquemment modifiées (insertion / suppression de données)
  - si la durée d'une transaction augmente régulièrement
  - le lancer quotidiennement ou lancer autovacuum

Tâches d'administration courantes : VACUUM (suite)

- Bilan :
  - lancer quotidiennement VACUUM sur les tables fréquemment modifiées (insertion / suppression de données)
  - lancer VACUUM ANALYZE quotidiennement ou lancer ANALYZE à un instant t en fonction de l'impact sur la base de données
  - lancer VACUUM FULL s'il existe (ou pour prévenir) des problèmes d'espace disque
  - lancer VACUUM FULL ANALYZE régulièrement

Tâches d'administration courantes : gestion des index

- Depuis la v7.4, les pages d'index vides sont réutilisées
- Utilisation inefficace de l'espace en cas de pages d'index presque vides (pour les index B-tree)
- Une reconstruction des index permet de rééquilibrer l'index B-tree
  - parfois accès aux données plus rapide avec une réindexation pour des index souvent modifiés
  - performances meilleures avec une réindexation périodique des index

Tâches d'administration courantes : taille de la base de données

- **oid2name** fait partie des outils de contribution de PostGreSQL
- oid2name extrait les OID des bases de données, leurs noms ainsi que les tablespaces auxquelles elles sont rattachées

Exemple :

```
-bash-3.00$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  24594        entrepot  pg_default
  10793        postgres  pg_default
  10792       template0  pg_default
  1           template1  pg_default
```

Tâches d'administration courantes : taille de la base de données (suite)

- **dbsize** faisait partie des outils de contribution de PostGreSQL
- Les fonctions de ce module sont maintenant incluses dans PostgreSQL
- Pour avoir une liste des fonctions disponibles avec ce module, taper la commande suivante qui renvoie :

```
psql -d template1 -AtF " " -c "\df pg_*_size" |
awk '{print $2}'
```

pg_column_size
pg_database_size
pg_database_size
pg_relation_size
pg_relation_size
pg_tablespace_size
pg_tablespace_size
pg_total_relation_size
pg_total_relation_size

Tâches d'administration courantes : taille de la base de données (suite)

- pg_column_size affiche l'espace utilisé pour stocker toute valeur individuelle
- pg_tablespace_size et pg_database_size acceptent l'OID ou le nom d'un tablespace ou d'une base de données, et renvoient l'espace disque total utilisé
- pg_relation_size accepte l'OID ou le nom d'une table, d'un index ou d'une table toast, et renvoie la taille en octet
- pg_total_relation_size accepte l'OID ou le nom d'une table ou d'une table toast, et renvoie la taille en octets des données et de tous les index et tables toast associés
- pg_size_pretty peut être utilisé pour formater le résultat d'une des autres fonctions en utilisant KB, MB, GB ou TB lorsque cela est approprié

Tâches d'administration courantes : taille de la base de données (suite)

Exemple :

```
entrepot=# select pg_database_size('entrepot');
 pg_database_size
------------------
        121368444
(1 row)

entrepot=# select
  pg_size_pretty(pg_database_size('entrepot'));
 pg_size_pretty
----------------
 116 MB
(1 row)
```

Bilan sur l'administration

Conseils d'administration

- Au lieu de « commiter » chaque modification individuellement, englober les actions dans une transaction (un seul « commit » à la fin)
- Utiliser un « COPY » pour insérer de nombreuses lignes plutôt qu'une suite d'INSERT
  - COPY est optimisé
  - COPY est plus rapide qu'un INSERT (même si PREPARE est utilisé) et les insertions sont englobées dans une même transaction
- Lors de la modification d'un grand nombre de données d'une table, supprimer puis recréer les index plutôt que de les mettre à jour à chaque modification (plus rapide)

Conseils d'administration (suite)

- Même comportement avec les clés étrangères (suppression avant une mise à jour importante)
- Modifier temporairement certains paramètres de mémoire pour :
  - CREATE INDEX et ALTER TABLE ADD FOREIGN KEY : paramètre maintenance_work_mem
  - chargement de nombreuses données : paramètre checkpoint_segments (réduction du nombre de checkpoints)
- Exécuter ANALYZE après un chargement important de données
- Ajuster les paramètres définis pour le process writer, si la durée des checkpoints est trop pénalisante

Conseils d'administration (suite)

- ANALYZE et VACUUM ANALYZE permettent d'obtenir des statistiques en prenant aléatoirement plusieurs lignes de la table
- Par défaut, l'échantillon obtenu contiendra les 10 valeurs (valeur de la variable « *default_statistics_target* ») les plus communes dans chaque colonne
- L'ampleur de collecte des statistiques par colonne pour les opérations d'analyse peut prendre une valeur entre 10 et 1000
- La commande à exécuter pour la modifier est :
  ALTER TABLE nom_table ALTER COLUMN nom_colonne **SET STATISTICS** valeur;

Conseils d'administration (suite)

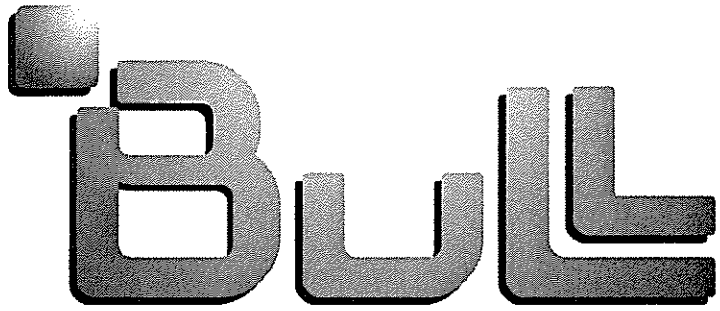
- Exemple :
  ALTER TABLE dummy3 ALTER COLUMN relname SET STATISTICS 1000;

⇒ l'échantillon obtenu contiendra les 1000 valeurs les plus communes dans chaque colonne

⇒ la totalité des lignes de la table dummy3 aura été utilisée pour récupérer ces 1000 valeurs

- Optimiser les paramètres de configuration de VACUUM et autovacuum pour limiter leurs impacts sur la base de données

Questions...



Architect of an Open World™