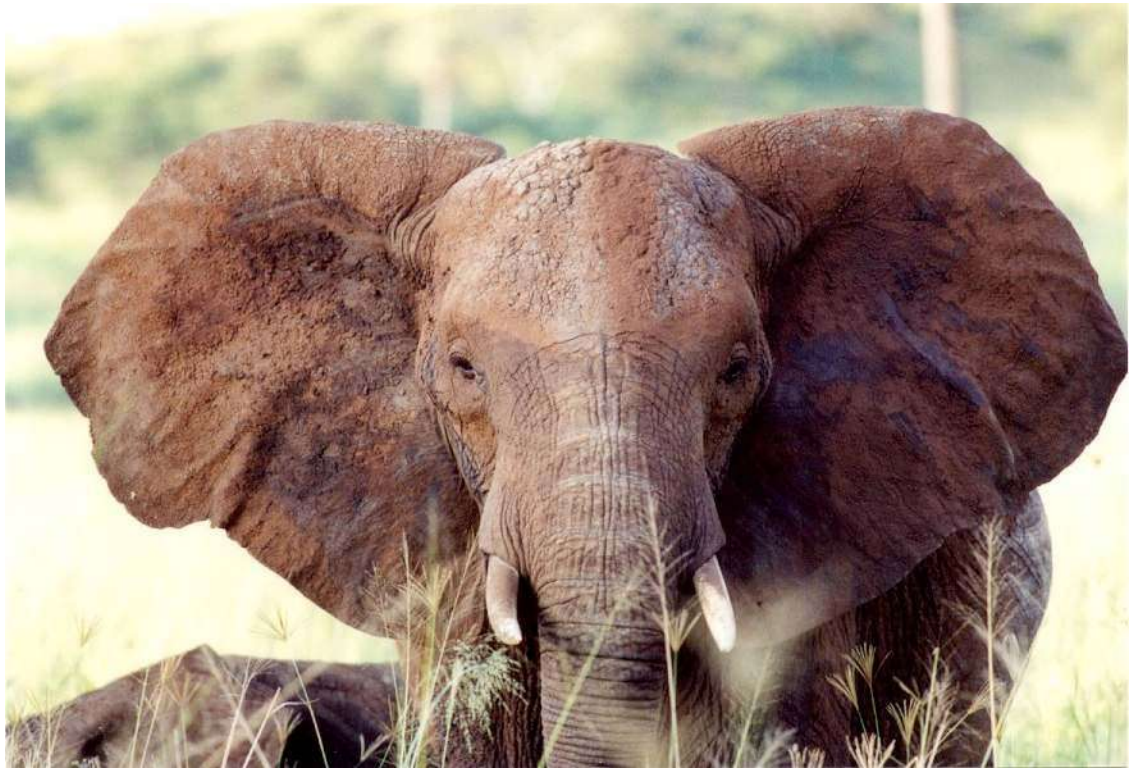
Support de cours

PostgreSQL 8

installation

configuration

exploitation



v_20081002

Licence pour la diffusion de ce document

Plan

La formation suivra ce plan :

- introduction ;
- partie 1 : installation et configuration ;
- partie 2 : structure et organisation des objets au sein d'un serveur PostgreSQL ;
- partie 3 : administration des bases de données.

Il s'agit d'une formation interactive, il est donc tout à fait indiqué d'interrompre le formateur pour lui poser des questions, lui faire préciser certains points, demander l'étude d'un cas particulier, etc.

Ce support fait référence à la version 8.3 de PostgreSQL.

Pré-requis

Cette formation est une introduction à l'administration du serveur de bases de données PostgreSQL. Elle se focalisera sur les tâches d'administration et les spécificités de ce logiciel par rapport à d'autres gestionnaires de bases de données.

En particulier, on attend de chaque stagiaire :

- des connaissances générales en administration des systèmes informatiques ;
- des notions sur les réseaux IP ;
- la maîtrise du langage SQL (LDD et LMD) ;
- la maîtrise d'un éditeur de texte.

Introduction

PostgreSQL est un gestionnaire de bases de données relationnelles (SGBDR) supportant le langage SQL. Il est a été développé à partir du projet Postgres 4.2 initié par l'Université de Californie à Berkeley (UCB), département informatique, dès 1986. Postgres est lui-même dérivé de Ingres.

PostgreSQL est développé selon le mode « *Open Source* », sous licence BSD.

Plusieurs dizaines de développeurs et des nombreuses entreprises participent au développement. L'équipe référente sur le projet reste, elle, indépendante.

PostgreSQL dispose notamment des fonctionnalités suivantes :

- respect de la norme SQL92 ;
- clés étrangères ;
- plusieurs langages procéduraux ;
- déclencheurs ;
- vues ;
- conforme au modèle transactionnel ACID.

La licence BSD

Contenu du fichier COPYRIGHT du répertoire du code source de PostreSQL :

```
PostgreSQL Database Management System
(formerly known as Postgres, then as Postgres95)

Portions Copyright (c) 1996-2008, PostgreSQL Global Development Group

Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose, without fee, and without a written agreement
is hereby granted, provided that the above copyright notice and this
paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING
LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
```

Versions de PostgreSQL

Les versions marquantes :

1995-05-01	Postgres95 0.01	première version
1995-09-05	**PostgreSQL 1.0**	**le code devient modifiable (évolution de la licence)**
1997-01-29	PostgreSQL 6.0	changement dans la numérotation, industrialisation du développement
1997-06-08	PostgreSQL 6.1	nouvel optimiseur
1998-03-01	PostgreSQL 6.3	*subselect*, variables d'environnement côté client
1998-10-30	PostgreSQL 6.4	vues, règles (*rules*), UTF réellement utilisable
1999-06-09	PostgreSQL 6.5	MVCC, sauvegardes à chaud, tables temporaires
2000-05-08	**PostgreSQL 7.0**	**clés étrangères, amélioration de l'optimiseur, jointures SQL92**
2001-04-13	**PostgreSQL 7.1**	**WAL, TOAST, jointures ouvertes, requêtes complexes**
2002-02-04	**PostgreSQL 7.2**	**améliore la gestion des bases conséquentes, internationalisation**
2002-11-27	**PostgreSQL 7.3**	**schémas, requêtes préparées**
2008-01-07	*PostgreSQL 7.3.21*	
2003-11-17	PostgreSQL 7.4	dictionnaire, *tsearch2*, autovacuum, nombreuses optimisations
2005-01-19	**PostgreSQL 8.0**	**natif Windows, *savepoints*, PITR, *tablespaces*, prêt pour l'entreprise**
2005-11-08	PostgreSQL 8.1	validation en deux phases, rôles
2006-12-05	PostgreSQL 8.2	nombreuses améliorations fonctionnelles et optimisations
2008-02-04	PostgreSQL 8.3	encore plus d'améliorations fonctionnelles et d'optimisations
2008-09-22	*PostgreSQL 7.4.22, 8.0.18, 8.1.14, 8.2.10, 8.3.4*	

Communauté

Le site *pgFoundry.org* héberge de nombreux projets développés par des équipes indépendantes :

- connecteurs pour les différents langages ;
- langages procéduraux ;
- outils d'aide à l'administration ;
- logiciels pour la haute disponibilité (réplication, gestion des connexions, *etc.*).

Citons :

- *pgFouine*	création de rapports à partir du journal d'activité de PostgreSQL
- *PGCluster*	réplication synchrone multi-maîtres
- *pgpool*	gestion des connexions (limitation, réplication, répartition, parallélisation)
- *pg-toolbox*	un ensemble de scripts pour l'aide à l'administration

D'autres projets existent :

- *Slony*	*http://slony.info*	réplication maître vers plusieurs esclaves
- *phpPgAdmin*	*http://phppgadmin.sf.net*	interface Web d'administration
- *pgAdmin*	*http://pgadmin.org*	client d'administration

Le site français *PostgreSQLFr.fr* propose des nouvelles et des traductions de la documentation officielle et héberge un forum de discussion.

Documentation

La documentation de PostgreSQL est consultable en ligne :

http://www.postgresql.org/docs/8.3/interactive/index.html

Ou disponible au format PDF :

http://www.postgresql.org/files/documentation/pdf/8.3/postgresql-8.3-A4.pdf

Ou installée avec PostgreSQL, par exemple dans ce répertoire :

$PREFIX/doc/html/index.html

L'association PostgreSQLFr maintient une traduction en ligne :

http://docs.postgresql.fr/8.3/

Téléchargeable :

http://docs.postgresqlfr.org/8.3/pg833.tar.gz

Ou sous forme PDF ou au format d'aide Windows :

http://docs.postgresqlfr.org/pg83.pdf
http://docs.postgresqlfr.org/8.3/pg833.chm

Un fois PostgreSQL installé, des pages de manuels Unix sont également consultables avec la commande *man* (commandes Unix en section 1, commandes SQL en section 7).

Comparatif avec les autres BDD

De telles comparaisons sont souvent périlleuses.
Malgré tout :

	PostgreSQL 8	MySQL 5	Oracle 10g
espaces de tables	oui	non	oui
tables partitionnées	émulées	oui	oui
conforme ACID	oui	avec InnoDB	oui
intégrité référetielle	oui	avec InnoDB	oui
langage procédural	plusieurs	oui	PL/SQL, Java
déclencheurs	oui	limités	oui
curseurs	oui	limités	oui
authentifications externes	LDAP, kerberos, PAM, GSSAPI, *etc.*	non	oui

Concernant les performances, il est souvent admis que MySQL est plus rapide lorsqu'il est utilisé avec le moteur MyISAM. Cependant, cet environnement pose plusieurs problèmes (notamment corruption possible des données et contention des verrous).

Installation et configuration

Les étapes de l'installation

PostgreSQL a été portés sur de très nombreux systèmes d'exploitation : Linux, Unix, Windows et Mac OS X.

Quelque soit l'environnement, ces étapes seront nécessaires :

1) Installation des binaires.
2) Préparation du système d'exploitation.
3) Initialisation des fichiers de l'instance.
4) Configuration.
5) Installation des scripts de démarrage sous Unix ou services sous Windows.

1 – Installation des binaires

En fonction du système d'exploitation et des contraintes de développement et d'exploitation, plusieurs solutions existent pour installer les binaires :

- sous Linux, à partir des paquets livrés avec la distribution ;
- pour les systèmes BSD, en utilisant les ports ;
- en récupérant un paquet binaire sur le site officiel :
  (pour Solaris, Windows ou les systèmes Fedora et RHEL)
  http://www.postgresql.org/ftp/binary
- à partir d'un paquet binaire compilé par la société EnterpriseDB :
  http://www.enterprisedb.com/products/download.do
- avec Yum à partir du dépôt PGDG, *pgsqlrpms.org* (pour Fedora, CentOS et RHEL) ;
- en compilant le code source :
  http://www.postgresql.org/ftp/source

Installer PostgreSQL sous Debian

La version stable de Debian (*etch* à ce jour) propose une version 8.1 de PostgreSQL. Pour l'installer, il suffit d'utiliser APT :

```
# apt-get install postgresql-8.1 postgresql-client-8.1
```

Les paquets suivants sont recommandés :

```
# apt-get install postgresql-contrib-8.1 postgresql-doc-8.1
```

Les paquets des langages procéduraux sont :

```
# apt-get install postgresql-8.1-pljava-gcj postgresql-8.1-plr  \
                  postgresql-8.1-plruby postgresql-8.1-plperl   \
                  postgresql-plpython-8.1 postgresql-pltcl-8.1
```

Le dépôt *backports.org* propose des versions plus récentes de PostgreSQL (8.2 et 8.3) sous forme de paquets Debian.

Installer PostgreSQL sous CentOS

CentOS 5.1 propose égalements des paquets pour la version 8.1 de PostgreSQL. D'autres versions des paquets RPM sont disponibles sur le site de PostgreSQL ici :

http://www.postgresql.org/ftp/binary/v8.*x*.*x*/linux/rpms/

Pour CentOS 5.1, utiliser les paquets disponibles dans les répertoires *redhat/rhel-5-i386* ou *redhat/rhel-5-x86_64*, les paquets suivants, au moins, sont nécessaires :

- postgresql-libs
- postgresql
- postgresql-server

Ceux-ci sont recommandés : *postgresql-contrib*, *postgresql-docs* et *postgresql-devel*
Langages procéduraux : *postgresql-plperl*, *postgresql-plpython* et *postgresql-pltcl*

Installation et démarrage :

```
# rpm -ivh postgresql-*
# service postgresql initdb
# service postgresql start
```

Installer PostgreSQL à partir du dépôt PGDG

Le site *pgsqlrpms.org* fournit des paquets binaires pour les systèmes suivants :

- Fedora 7 à 9 ;
- RedHat Enterprise Linux 4 et 5 ;
- CentOS 4 et 5.

Les versions de PostgreSQL disponibles sont 7.3 et 7.4 ainsi que 8.0 à 8.4.

Les étapes sont :

- choisir un fichier RPM pour la distribution et la version de PostgreSQL à partir de :
  http://yum.pgsqlrpms.org/reporpms/repoview/letter_p.group.html
- installer ce paquet, par exemple :
```
# rpm -ivh pgdg-fedora-8.4-1.noarch.rpm
```
- supprimer les paquets PostgreSQL éventuellement installés :
```
# rpm -qa |grep -i postgres |xargs rpm -e
```
- installer les paquets avec Yum :
```
# yum install postgresql postgresql-server
```

Compiler PostgreSQL sous Unix

PostgreSQL utilise l'outil `configure`. Sa compilation est donc relativement simple. Le code source est disponible à partir de :

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

Les outils de compilation et bibliothèques nécessaires sont :

- GNU make ;
- compilateur C ANSI (tel que *gcc*) ;
- *tar* et *gzip* ou *bzip2* ;
- la bibliothèque GNU *readline* (optionnelle) ;
- la bibliothèque *zlib* (optionnelle) ;
- une implémentation de l'API *gettext* pour activer NLS (*Native Language Support*) ;
- les bibliothèques Kerberos, OpenSSL, OpenLDAP et/ou PAM pour activer des types d'authentification et de chiffrements spécifiques.

D'autres composants logiciels sont nécessaires pour certains langages procéduraux :

- Perl et la bibliothèque *libperl* pour PL/Perl ;
- Python et le module *distutils* pour PL/Python ;
- Tcl pour PL/Tcl.

Compiler PostgreSQL sous Unix

Le processus général à suivre pour compiler PostgreSQL est le suivant :

```
# wget ftp://ftp.postgresql.org/pub/source/v8.x.x/postgresql-8.x.x.tar.gz
# zcat postgresql-8.x.x.tar.gz | tar xf -
# cd postgresql-8.x.x
# ./configure --prefix=/usr/local/postgresql8xx --with-openssl
# make
# make install
# echo 'PATH=/usr/local/postgresql8xx/bin:$PATH' >> /etc/profile
# echo 'MANPATH=/usr/local/postgresql8xx/man:$MANPATH' >> /etc/profile
# echo 'export PATH MANPATH' >> /etc/profile
```

D'autres options de `configure` sont :

```
  --without-readline
  --without-zlib
  --enable-nls='fr de'
  --with-ldap
  --with-pam
  --with-perl
  --with-python
  --with-tcl
```

Pour obtenir toutes les options, exécuter :

```
# ./configure --help
```

Installer PostgreSQL sous Windows

Une version de PostgreSQL compilée pour Windows est disponible sous forme de paquet installable à cet endroit :

http://www.postgresql.org/ftp/binary/v8.*x*.*x*/win32/

Télécharger le fichier *postgresql-8*.x.x-x.*zip*, en extraire le fichier *postgresql-8*.x.*msi* et l'exécuter. Dérouler les écrans d'installation. La procédure prend en charge la création d'un service Windows ainsi que la création d'un premier groupe de bases de données pour le serveur PostgreSQL.

L'installation des différents langages procéduraux est possible si ceux-ci sont déjà installés sur le système d'exploitation.

Parmi les contributions utiles à installer figure notamment *adminpack* qui permet d'étendre les fonctions de l'interface graphique pgAdminIII.

2 – Préparation du système d'exploitation

Trois tâches essentielles :

1) Création d'un utilisateur système.
2) Choix des répertoires pour les fichiers de l'instance.
3) Paramétrage de la mémoire partagée.

Le serveur PostgreSQL n'est jamais exécuté en tant que *root*. L'utilisateur système utilisé est traditionnellement *postgres* et n'a pas besoin de mot de passe.

Le choix du répertoire de l'instance est important :

- prendre en compte la capacité et la vitesse du disque ;
- prévoir la possibilité de créer plusieurs instances ;
- les fichiers de données, les index et les journaux des transactions pourront être répartis sur plusieurs disques grâce aux espaces de tables (*tablespaces*) ;
- ce répertoire doit être accessible en lecture et écriture pour l'utilisateur *postgres* (et uniquement cet utilisateur).

Par exemple : */var/postgres/u01/inst1*

Paramètres de la mémoire partagée sous Unix

PostgreSQL utilise des segments de mémoire partagée entre les différents processus du serveur. Cette fonction est propre aux systèmes Unix.

Les valeurs de ces paramètres sont souvent trop faibles :

```
[root@pga ~]# sysctl -a |grep kernel.shm |sort
kernel.shmall = 2097152                    (soit 8 Go, car exprimé en nombre de blocs)
kernel.shmmax = 33554432                   (soit 32 Mo)
kernel.shmmni = 4096
```

```
[root@pga ~]# ipcs -m -l
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1
```

En général, mais cela dépend des systèmes Unix, les paramètres du noyau sont configurés par l'intermédiaire du fichier */etc/sysctl.conf*. Par exemple, sous Linux :

```
kernel.shmall = 2097152                    (soit 8 Go, car exprimé en nombre de blocs)
kernel.shmmax = 134217728                  (soit 128 Mo)
kernel.shmmni = 256
```

Attention, le paramètre *shmall* peut s'exprimer en octets ou nombre de blocs.

3 – Initialiser une instance

Une instance PostgreSQL est composée :

- d'un ensemble de processus dont un écoutant sur un port TCP (5432 par défaut) ;
- d'un répertoire (appelé « groupe de bases de données ») contenant notamment :
  * les bases de données (dans le répertoire *base*) ;
  * un fichier de configuration *postgresql.conf* ;
  * un fichier pour la gestion de l'authentification *pg_hba.conf*.

Il est possible de configurer plusieurs instances actives simultanément sur un même serveur à condition d'allouer un répertoire et un port TCP distincts pour chacune. Ces instances peuvent même exécuter des versions différentes de PostgreSQL.

En fonction de la manière dont PostgreSQL a été installé, il peut être nécessaire de créer ou non le premier groupe de bases de données et de l'initialiser.

Initialiser une instance

La procédure est la suivante :

1) Créer l'utilisateur et le groupe *postgres* :

```
# groupadd postgres
# useradd -g postgres -d /var/pg -c "PostgreSQL Software Owner"  \
          -s /bin/sh postgres
```

2) Créer le répertoire et lui associer les droits adéquats :

```
# mkdir /var/pg
# chown postgres:postgres /var/pg
# chmod 700 /var/pg
```

3) Initialiser les fichiers de l'instance :

```
# su - postgres
$ initdb --encoding=UTF8 --locale=C --pwprompt /var/pg/u01/inst1
$ ls /var/pg/u01/inst1
base    pg_clog      pg_ident.conf  pg_subtrans  pg_twophase  pg_xlog
global  pg_hba.conf  pg_multixact   pg_tblspc    PG_VERSION   postgresql.conf
```

4 – Configurer l'instance

Le fichier *postgresql.conf* contient les différents paramètres de l'instance. Les valeurs par défaut permettent d'exécuter une instance de taille modeste sans problème particulier. Nous reviendrons sur différents paramètres de ce fichier. Pour l'instant, seuls ceux destinés à configurer les accès réseau nous intéressent.

Le paramètre *port* (dont la valeur par défaut est 5432) permet de choisir un port TCP différent, ce qui est nécessaire si plusieurs instances sont exécutées sur une même machine.

Le paramètre *listen_addresses* liste les adresses IP sur lesquelles PostgreSQL écoutera. Par défaut seule l'adresse locale 127.0.0.1 sera utilisée. Pour permettre la connexion de puis d'autres machines du réseau, il faut donc modifier ce paramètre. Par exemple :

```
listen_adresses = '*'
```

Configurer l'authentification

L'authentification est gérée par :

- le fichier *pg_hba.conf* ;
- les objets rôles.

Le fichier *pg_hba.conf* est utilisé pour configurer la manière d'authentifier les connexions en fonction de leur origine et de la base de données concernée.

> **Attention !**
> Le fichier *pg_hba.conf* par défaut autorise toutes les connexions locales de n'importe quel utilisateur vers n'importe quelle base de données sans authentification. Aucune connexion ne sera acceptée depuis une autre machine du réseau.

Il donc nécessaire de modifier ce fichier la plupart du temps.

Configurer l'authentification

Le format d'une ligne du fichier *pg_hba.conf* est le suivant :

```
type  base_de_données  utilisateur  addresse  méthode  option
```

Les types supportés sont :

- *local* : connexion par une socket du domaine Unix (la colonne adresse est vide) ;
- *host* : connexion chiffrée ou non par une socket TCP/IP ;
- *hostssl* : connexion chiffréepar une socket TCP/IP ;
- *hostnossl* : connexion non chiffrée par une socket TCP/IP.

La colonne *base_de_données* peut contenir le nom d'une base de données ou :

- *all* ;
- *samerole* ;
- une liste de ces éléments séparés par une virgule.

La colonne *utilisateur* peut contenir le nom d'un utilisateur ou :

- *all* ;
- le nom d'un groupe (rôle) précédé du caractère « + » ;
- une liste de ces éléments séparés par une virgule.

Configurer l'authentification

La colonne *adresse* contient l'adresse IP d'origine de la connexion au format CIDR. Elle doit être vide lorsque le type de connexion est *local*.

Enfin, la méthode d'authentification est choisie parmi :

- *trust* : autoriser sans même vérifier le mot de passe ;
- *reject* : rejeter la connexion ;
- *md5* : vérifier le mot de passe fourni (ne plus utiliser *crypt* ou *password*) ;
- *ident* : utiliser le protocole IDENT pour vérifier l'utilisateur ;
- *krb5*, *pam* ou *ldap* : authentifications spécifiques.

La colonne *option* n'est employée que lorsque la méthode d'authentification l'exige (par exemple, pour *ident* ou *pam*).

Configurer l'authentification

Voici un exemple sensé (lire « recommandé ») pour le fichier *pg_hba.conf* :

```
# TYPE    DATABASE    USER        CIDR-ADDRESS      METHOD
local     all         postgres                      ident sameuser
host      all         all         192.168.1.0/24    md5
host      all         all         0.0.0.0/0         reject
```

L'ordre des lignes dans ce fichier est important car elles sont évaluées les unes à la suite des autres. La première correspondant aux critères type, base de données, utilisateur et origine l'emporte.

L'exemple ci-dessus permet à l'utilisateur *postgres* de se connecter en local à condition que cette connexion soit réalisée par l'utilisateur du système qui porte le même nom.

Les connexions à partir du réseau 192.168.1.0/24 seront autorisées mais nécessiteront un mot de passe. Un utilisateur (rôle) devra exister pour cela dans l'instance.

Toutes les autres connexions seront refusées.

Démarrer et arrêter l'instance

La commande *pg_ctl* peut être utilisée pour contrôler une instance. Notamment pour la démarrer, l'arrêter, obtenir son état ou lui faire relire ses fichiers de configuration :

```
# su - postgres
$ pg_ctl -D /var/pg/u01/inst1 -l /var/pg/u01/inst1/stderr.log start
$ pg_ctl -D /var/pg/u01/inst1 status
$ pg_ctl -D /var/pg/u01/inst1 reload
$ pg_ctl -D /var/pg/u01/inst1 [ -m smart|fast|immediate ] stop
```

Si PostgreSQL a été installé avec un paquet d'une distribution, il est préférable d'utiliser les scripts fournis.

Sous Debian :

```
# /etc/init.d/postgresq-8.1 start|stop|status|reload
```

Sous RedHat ou CentOS :

```
# service postgresql start|stop|status|reload
```

Enfin, sous Windows :

```
> net start/stop pgsql
```

Processus de l'instance

Les processus associés à l'instance sont les suivantes :

```
# ps faux
postgres 30603  ?  S  /usr/local/postgresql834/bin/postmaster -D /var/pg/u01/inst1
postgres 30605  ?  Ss  \_ postgres: logger process
postgres 30606  ?  Ss  \_ postgres: writer process
postgres 30607  ?  Ss  \_ postgres: wal writer process
postgres 30608  ?  Ss  \_ postgres: autovacuum launcher process
postgres 30609  ?  Ss  \_ postgres: stats collector process
postgres 30617  ?  Ss  \_ postgres: postgres postgres 127.0.0.1(35342) idle
```

Le processus père de tous les autres est le p*ostmaster*. Certains processus fils ne sont présents que lorsque certains paramètres sont activés.

Le dernier processus de cette liste est un processus associé à un client. Un tel processus sera créé pour chaque nouvelle session.

5 – Configurer le démarrage

Sous Unix, lorsque PostgreSQL est compilé, les scripts de démarrage ne sont pas installés.

Pour Linux, un script de démarrage est disponible dans le répertoire *contrib/start-scripts* du code source.

Par exemple :

```
# cp contrib/start-scripts/linux /etc/init.d/postgresql_inst1
# chmod 755 /etc/init.d/postgresql_inst1
# /etc/init.d/postgresql_inst1
```

Modifier les variables suivantes :

```
prefix=/usr/local/postgresql834
PGDATA="/var/pg/u01/inst1"
```

```
# ln -s /etc/init.d/postgresql_inst1 /etc/rc2.d/S30postgresql_inst1
# ln -s /etc/init.d/postgresql_inst1 /etc/rc5.d/S30postgresql_inst1
# ln -s /etc/init.d/postgresql_inst1 /etc/rc0.d/K70postgresql_inst1
# ln -s /etc/init.d/postgresql_inst1 /etc/rc6.d/K70postgresql_inst1
```

Sessions

Les clients se connectent au serveur par l'intermédiaire d'une *socket* Unix ou d'une *socket* TCP (sur le port 5432 par défaut). Deux informations sont nécessaires pour ouvrir une session :

- le nom d'une base de données ;
- le nom d'un rôle.

Un mot de passe est requis selon la configuration de *pg_hba.conf*.

Le jeu de caractères utilisé par le client peut être différent de celui de la base de données. Le ré-encodage sera effectué par le serveur.

Instances supplémentaires

Pourquoi utiliser de nouvelles instances :

- pour exécuter des versions différentes de PostgreSQL ;
- pour exécuter des instances de PostgreSQL avec des paramètres différents ;
- pour des environnements différents (développement, pré-production, production) ;
- pour isoler des bases de données.

Pour créer une nouvelle instance sur le même serveur, il faut :

1) Choisir un port TCP et un répertoire différent pour l'instance.
2) Créer les fichiers de l'instance avec *initdb*.
3) Modifier la configuration de l'instance.
4) Installer le script de démarrage.

Instances supplémentaires

Par exemple :

```
# su - postgres
$ initdb --encoding=UTF8 --locale=C --pwprompt /var/pg/u01/inst2
$ vi /var/pg/u01/inst2/postgresql.conf
```

Modifier la variable suivante :

```
port = 5433
```

```
$ exit
# cp /etc/init.d/postgresql_inst1 /etc/init.d/postgresql_inst2
# vi /etc/init.d/postgresql_inst2
```

Modifier la variable suivante :

```
PGDATA="/var/pg/u01/inst2"
```

```
# ln -s /etc/init.d/postgresl_inst2 /etc/rc2.d/S30postresql_inst2
# ln -s /etc/init.d/postgresl_inst2 /etc/rc0.d/K70postresql_inst2
# ln -s /etc/init.d/postgresl_inst2 /etc/rc6.d/K70postresql_inst2
```

Les scripts installés par Debian

Les développeurs Debian des paquets PostgreSQL ont créé des scripts qui facilitent la gestion des versions de PostgreSQL et des instances sur un même serveur.

Les scripts pour la gestion des instances sont :

- *pg_createcluster* création d'une instance ;
- *pg_lsclusters* liste des instances existantes ;
- *pg_ctlclusters* arrêt/démarrage des instances ;
- *pg_dropcluster* suppression d'une instance ;
- *pg_upgradecluster* migration d'une instance vers une version plus récente ;
- *pg_maintenance* exécution des actions de maintenance sur toutes les instances.

Le démarrage des instances lors du démarrage du système est également pris en charge, il n'est donc pas nécessaire de créer et modifier des scripts dans */etc/init.d*.

Sessions

Pour utiliser des sessions chiffrées en SSL, trois étapes sont nécessaires :

1) Générer une clé privée et un certificat x509 pour le serveur.
2) Activer l'option *ssl* dans le fichier *postgresql.conf* de l'instance.
3) Redémarrer l'instance.

La clé privée et le certificat x509 doivent être enregistrés respectivement dans les fichiers *server.key* et *server.crt* du répertoire de l'instance.

Par exemple, pour générer un certificat auto-signé avec OpenSSL :

```
# cd /var/pg/u01/inst1
# openssl req -new -text -nodes -out server.req -keyout server.key
# openssl req -x509 -in server.req -text -key server.key -out server.crt
# chown postgres server.*
# chmod 600 server.key
```

Clients

Plusieurs outils sont installés avec PostgreSQL :

```
clusterdb   droplang  pg_config       pg_resetxlog  reindexdb
createdb    dropuser  pg_controldata  pg_restore    vacuumdb
createlang  ecpg      pg_ctl          postgres
createuser  initdb    pg_dump         postmaster
dropdb      ipcclean  pg_dumpall      psql
```

Beaucoup permettent d'exécuter des actions qui peuvent l'être en SQL. Nous nous concentrerons sur leur équivalent SQL. Pour cela, le client principal est l'interpréteur de commandes *psql*. Ses principaux paramères (tous optionnels) sont :

```
$ psql -h machine -p port nombdd utilisateur
```

Par exemple :

```
$ psql -h 192.168.1.42 postgres postgres
Password for user postgres:
Welcome to psql 8.2.7, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

postgres=#
```

Client psql

L'invite de commande de *psql* change en fonction du contexte, elle est composée de trois parties :

```
pagila=#
```

`pagila`	est le nom de la base de données active pour cette session
`=`	signifie que *psql* attend une nouvelle commande
`#`	indique que l'utilisateur connecté est un administrateur de l'instance

```
postgres->
```

`postgres`	est le nom de la base de données active pour cette session
`-`	signifie que *psql* attend la suite d'une instruction SQL
`>`	indique que l'utilisateur connecté n'est pas un administrateur de l'instance

Les signes = et - peuvent également être remplacés par :

`'`	une chaîne est ouverte (un caractère ' a été saisi sur une ligne précédente)
`(`	une parenthèse a été ouverte sur une ligne précédente

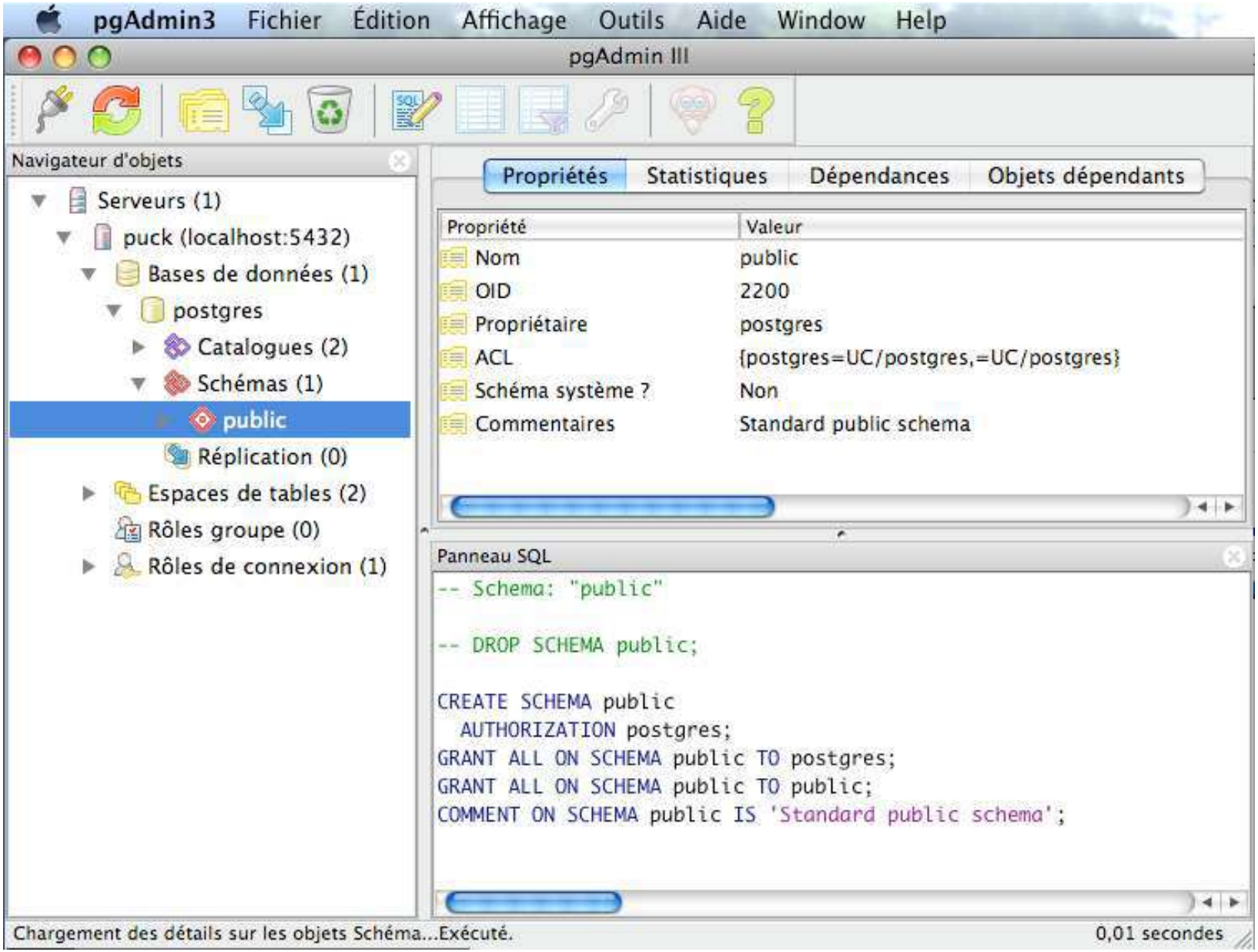
Client psql

En plus des instructions SQL LDD et LMD, l'interpréteur *psql* comprend certaines commandes qui lui sont propres. Ces commandes sont précédées du caractère « \ ». Voici celles qui sont le plus souvents utilisées :

`\c [nombdd|- [utilisateur|- [machine|- [port|-]]]]`	se connecter à une autre instance PostgreSQL
`\i fichier`	exécuter un fichier SQL
`\o fichier`	envoyer la sortie vers un fichier
`\d nom`	décrire une relation (table, index, séquence ou vue)
`\d{t|i|s|v} [modèle]`	lister les tables, index, séquences ou vues
`\db [modèle]`	lister les espaces de tables
`\du [modèle]`	lister les utilisateurs
`\dg [modèle]`	lister les groupes
`\dn [modèle]`	lister les espaces de noms (schémas et catalogues)
`\l`	lister les bases de données de l'instance
`\z nom`	afficher les privilèges associés à une relation
`\?`	lister les commandes de *psql*
`\h instruction`	obtenir de l'aide sur une instruction SQL
`\encoding [encodage]`	obtenir ou configurer l'encodage pour la session
`\q`	quitter *psql*

Client pgAdminIII

Le logiciel pgAdminIII est souvent utilisé comme alternative graphique à *psql*.



Client pgAdminIII

L'installation de pgAdminIII dépend du système d'exploitation.

Pour Debian :

```
# apt-get install pgadmin3
```

Des versions compilées sont disponibles pour FreeBSD, Mac OS X et certaines distributions de Linux à cet endroit :

http://www.postgresql.org/ftp/pgadmin3/release/

Pour RedHat, CentOS et Fedora, le dépôt de logiciels Dag en fournit une version à cette adresse :

http://dag.wieers.com/rpm/packages/pgadmin3/

Par exemple, sous CentOS ou RHEL 5 :

```
# wget http://dag.wieers.com/rpm/packages/pgadmin3/pgadmin3-1.4.3-1.el5.rf.i386.rpm
# rpm -ivh pgadmin3-1.4.3-1.el5.rf.i386.rpm
```

Pour Windows, pgAdminIII est compris dans le paquet d'installation présenté précédemment.

Client pgAdminIII

Afin de pouvoir utiliser toutes les fonctions de pgAdminIII, il est nécessaire de créer certaines fonctions dans chaque base de données. Le paquet *postgresql-contrib* fournit les fichiers nécessaires pour cela, notamment le fichier *adminpack.sql*.

Par exemple, sous CentOS 5, il suffit d'exécuter :

```
# su - postgres
$ psql base_de_données < /usr/share/pgsql/contrib/adminpack.sql
```

Ce fichier SQL se trouve également dans les contributions fournies avec le code source de PostgreSQL si celui-ci a été compilé. Par exemple :

```
# cd /root/postgresql-8.3.4/contrib/adminpack
# make install
# psql -U postgres < adminpack.sql
```

Client phpPgAdmin

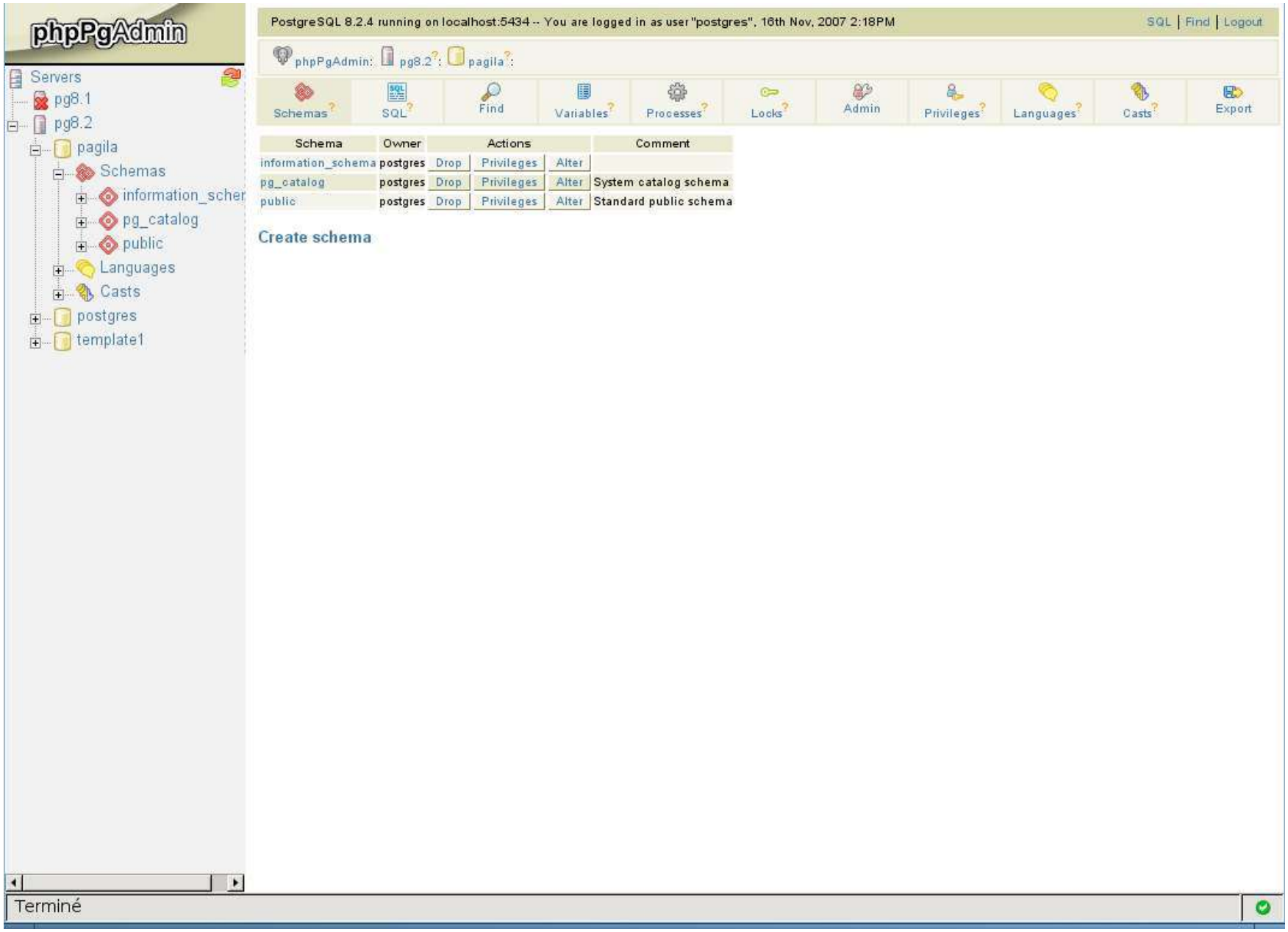
Le client d'administration pgAdmin III requiert un accès direct au serveur *via* le port TCP 5432. Cela n'est pas toujours possible, ni souhaitable. L'application Web phpPgAdmin peut être utilisée dans cette situation. Il s'installe sur le serveur.

Il est disponible à cette adresse :

http://phppgadmin.sourceforge.net

Son installation requiert un serveur Web et le langage PHP avec l'extension *pgsql*.

Client phpPgAdmin



Structure et organisation d'une instance PostgreSQL

LDD/LMD

Le langage SQL est constitué de deux sous-ensembles :

- le Langage de Description des Données (LDD) ;
- le Langage de Manipulation des Données (LMD).

Le langage de description des données est utilisé pour créer, modifier et détruire les objets de la base de données (tables, index, séquences, procédures, rôles, *etc.*). Ce sous-ensemble est représenté par les verbes :

```
create  alter  drop  grant  revoke
```

Souvent, un groupe d'instructions LDD qui décrivent un ensemble d'objets liés est appelé schéma de données.

Le langage de manipulation des données permet d'ajouter, modifier, supprimer et récupérer les données qui sont stockées dans les objets créés avec le langage de description des données. Ce sous-ensemble est représenté par les verbes :

```
insert  select  update  delete
```

Notion de OID

Les OID (*Object Identifiers*) sont utilisés par PostgreSQL pour la clé primaire d'un certain nombre de tables système.

Par exemple :

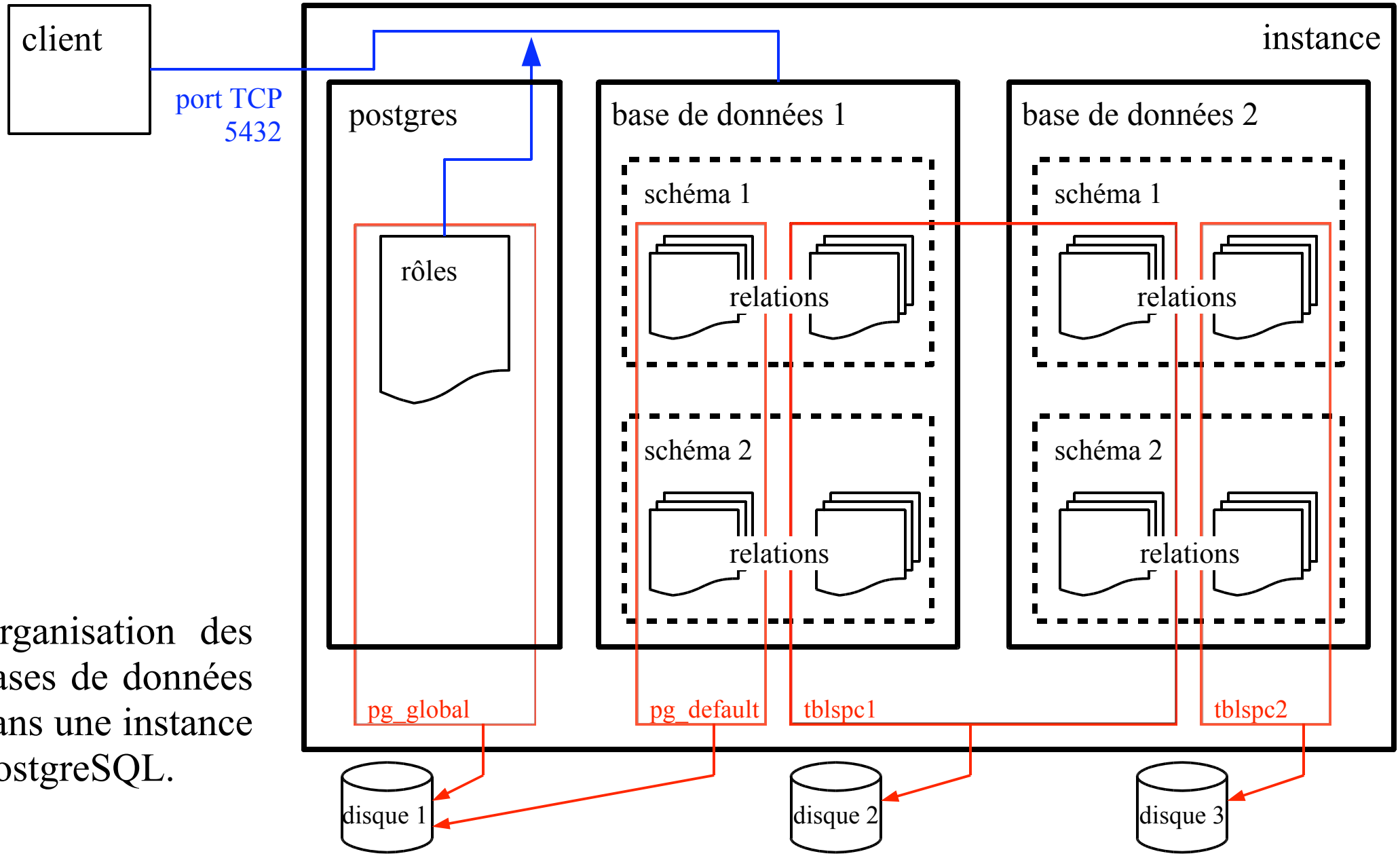
```
postgres=# \d pg_database
    Table "pg_catalog.pg_database"
    Column      |   Type    | Modifiers
----------------+-----------+-----------
 datname        | name      | not null
 datdba         | oid       | not null
 encoding       | integer   | not null
 datistemplate  | boolean   | not null
 datallowconn   | boolean   | not null
 datconnlimit   | integer   | not null
.../...

postgres=# select oid, datname from pg_database;
  oid  |  datname
-------+-----------
     1 | template1
 11510 | template0
 11511 | postgres
 16395 | bla
(4 rows)
```

Arborescence des répertoires et des fichiers

Le répertoire de chaque instance contient les fichiers et sous-répertoires suivants :

`$PGDATA`	
`PG_VERSION`	version de PostgreSQL associée à cette instance
`pg_hba.conf`	configuration de l'authentification
`pg_ident.conf`	configuration de l'authentification *ident*
`postgresql.conf`	configuration de l'instance
`postmaster.opts`	options de démarrage du processus *postmaster*
`postmaster.pid`	PID du *postmaster*
`$PGDATA/base`	répertoire associé à l'espace de tables *pg_default*
`$PGDATA/base/`*`oid`*	répertoire de la base de données associées à l'OID *oid*
`$PGDATA/base/pgsql_tmp`	fichiers temporaires pour certaines opérations
`$PGDATA/global`	tables système (espace de tables *pg_global*)
`$PGDATA/pg_clog`	données relatives au statut de validation des transactions
`$PGDATA/pg_log`	journaux d'activité
`$PGDATA/pg_multixact`	données relatives au statut des transactions multiples
`$PGDATA/pg_subtrans`	données relatives au statut des transactions imbriquées
`$PGDATA/pg_tblspc`	contient un lien symbolique pour chaque espace de tables
`$PGDATA/pg_twophase`	données pour les transactions préparées
`$PGDATA/pg_xlog`	journaux binaires (WAL)

Bases de données



Organisation des bases de données dans une instance PostgreSQL.

Instances, bases de données, schémas

Une instance est composée par :

- un ensemble de fichiers (groupe de bases de données, ou *database cluster*) ;
- un ensemble de processus.

Une instance gère plusieurs bases de données. Pour chaque instance, il existe au moins ces bases de données :

- *postgres* ;
- *template0* ;
- *template1*.

Les relations (tables, index, etc.) sont stockées dans les bases de données. Chaque base de données est compartimentée en espaces de noms : les schémas et les catalogues.

Pour chaque base de données il existe au moins le schéma *public* et les catalogues :

- *information_schema* ;
- *pg_catalog*.

Arborescence des répertoires et des fichiers

Toutes les relations (terme utilisé par PostgreSQL pour désigner un ensemble d'objets tels que les tables, index, séquences, *etc.*) appartiennent à une base de données.

Les données associées à chaque relation sont stockées dans un fichier dont le nom est l'OID de la relation dans le répertoire de la base de données :

`$PGDATA/base/`*`oid_bdd`*`/`*`filenode_relation`*

La base de données *postgres* fait exception à la règle, ses relations sont enregistrées dans le répertoire *$PGDATA/global*.

Ce répertoire contient les fichiers suivants :

`$PGDATA/global`

`pg_auth`	fichier texte, copie des identifiants et mots de passe des rôles utilisateurs
`pg_control`	positions et données de contrôle des journaux et transactions
`pg_database`	fichier texte, copie des bases de données présentes dans l'instance
`pgstat.stat`	statistiques collectées par PostgreSQL
`oid`	OID de la relation associée

Bases de données

La création d'une base de données se fait avec l'instruction :

```
CREATE DATABASE nom
    [ [ WITH ] [ OWNER [=] utilisateur ]
           [ TEMPLATE [=] modèle ]
           [ ENCODING [=] encodage ]
           [ TABLESPACE [=] espace_de_tables ]
```

Le propriétaire par défaut est l'utilisateur connecté.
La base de données modèle par défaut est *template1*.
L'encodage par défaut est celui de l'instance.
D'autres sont UTF8, WIN1252, LATIN1, LATIN9 et SQL_ASCII.
L'espace de tables par défaut est *pg_default*.

Seul un administrateur ou un utilisateur qui dispose l'attribut *createdb* peut créer une base de données (mais elle lui appartiendra obligatoirement).

Par exemple :

```
postgres=# create database test;
CREATE DATABASE
postgres=# \c test
You are now connected to database "test".
```

Bases de données

Pour supprimer une base de données :

```
DROP DATABASE [ IF EXISTS ] nom
```

Une base de données ne peut être supprimée tant que des utilisateurs y sont connectés. Seul l'administrateur ou le propriétaire d'une base de données peut la détruire.

> **Attention !**
> Lors de la suppression d'une base de données, tous les objets qu'elle contient sont détruits sans avertissement et sans possibilité de retour en arrière.

Par exemple :

```
test=# drop database test;
ERREUR:  n'a pas pu supprimer la base de données actuellement ouverte
test=# \c postgres
You are now connected to database "postgres".
postgres=# drop database test;
DROP DATABASE
postgres=#
```

Schémas

Les schémas sont simplement des espaces de noms. Ainsi, il est possible de créer dans une même base de données des objets qui portent le même nom tant qu'ils le sont dans des schémas différents.

Les objets d'un schéma sont accessibles en préfixant leur nom par le nom du schéma suivit d'un point : « schema.objet ».

La variable *search_path* contient une liste de noms de schéma qui seront considérés pour trouver des noms d'objet qui ne sont pas qualifiés. Cette variable contient par défaut :

```
"$user",public
```

La commande *\dn* de *psql* permet de lister les espaces de noms (dont les schémas) présents dans la base de données. Pour créer un schéma :

```
CREATE SCHEMA nom [ AUTHORIZATION utilisateur ]
```

Le propriétaire par défaut est l'utilisateur connecté. Seuls les administrateurs et les utilisateurs qui disposent du privilège *create* sur la base de données peuvent créer de nouveaux schémas.

Schémas

Pour détruire un schéma :

```
DROP SCHEMA [ IF EXISTS ] nom [ CASCADE ]
```

Seuls un administrateur et le propriétaire du schéma peuvent le détruire.
L'option CASCADE permet la suppression d'un schéma qui contient des objets.

Par exemple :

```
test=# create schema compta;
CREATE SCHEMA
test=# create table compta.clients (id int, nom varchar);
CREATE TABLE
test=# drop schema compta;
INFO:  table compta.clients dépend de schéma compta
ERREUR:  n'a pas pu supprimer schéma compta car d'autres objets en dépendent
HINT:  Utilisez DROP ... CASCADE pour supprimer aussi les objets dépendants.
test=# drop schema compta cascade;
INFO:  DROP cascade sur table compta.clients
DROP SCHEMA
```

Note : par défaut tous les droits sont octroyés à tous les utilisateurs sur le schéma *public*.

Catalogue système

Le catalogue *pg_catalog* contient notamment :

- l'ensemble des fonctions et types implémentées par le moteur de PostgreSQL ;
- un ensemble de tables dont certaines en écriture servent à contrôler certains aspects du fonctionnement du serveur ;
- un ensemble de vues permettant de comprendre le comportement de l'instance.

Par exemple :

La table *pg_autovacuum* permet de contrôler le nettoyage automatique des tables.
La table *pg_database* contient la liste des bases de données de l'instance.
La table *pg_tablespace* contient la liste espaces de table de l'instance.
La table *pg_namespace* contient la liste des schémas de la base de données.

La vue *pg_roles* liste les rôles présents dans l'instance.
La vue *pg_stat_activity* liste l'activité (les sessions) sur le serveur.
La vue *pg_stat_user_tables* donne des statistiques sur les tables de la base de données.
La vue *pg_stat_user_index* donne des statistiques sur les index de la base de données.

Types de données élémentaires

PostgreSQL supporte les types de données SQL standards :

`boolean`	
`smallint`	entier sur 2 octets
`integer`	entier sur 4 octets
`bigint`	entier sur 8 octets
`numeric(p, s)`	numérique exacte
`real`	virgule flotante sur 4 octets
`double precision`	virgule flotante sur 8 octets
`date`	date
`time`	heure
`timestamp [with time zone]`	date et heure
`interval`	intervalle de temps
`char(n)`	
`varchar(n)`	

Ainsi que des types élémentaires plus évolués :

`box`	`path`	`lseg`	`cidr`
`circle`	`point`	`bytea`	`inet`
`line`	`polygon`	`text`	`macaddr`

Relations, tables

Les tables sont créées ainsi :

```
CREATE [ TEMPORARY ] TABLE nom_table (
  nom_colonne  nom_type [ DEFAULT expression ] [ contrainte ] [, ...]
  [ contrainte_table [, ...] ] )
[ WITH ( parametre_stockage [= valeur] [, ... ] ) ] [ TABLESPACE tablespace ]
```

Les contraintes sur les colonnes et les tables s'expriment ainsi :

```
[ CONSTRAINT nom_contrainte ] { NOT NULL | UNIQUE parametres_index
                              | PRIMARY KEY parametres_index
                              | CHECK ( expression )
                              | REFERENCES autre_table [ (colonne) ]
                                [ ON DELETE action ] [ ON UPDATE action ] }

[ CONSTRAINT nom_contrainte ] { UNIQUE ( colonne [, ... ] ) parametres_index
                              | PRIMARY KEY ( colonne [, ... ] ) parametres_index
                              | CHECK ( expression )
                              | FOREIGN KEY ( colonne [, ... ] )
                                REFERENCES autre_table [ ( colonne [, ... ] ) ]
                                [ ON DELETE action ] [ ON UPDATE action ] }
```

Les paramètres des index s'expriment ainsi :

```
[ WITH ( parametre_stockage [= valeur] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace ]
```

Relations, index

Les indexes peuvent être ajoutés à une table déjà créée :

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] nom
ON nom_table
    ( { colonne | ( expression ) } [, ...] )
    [ WITH ( parametre_stockage = valeur [, ... ] ) ]
    [ TABLESPACE tablespace ]
```

Le seul paramètre de stockage supporté à ce jour est :

```
  FILLFACTOR = { 0-100 }
```

Relations, séquences

Les séquences sont des objets qui fournissent des suites d'entiers garantis uniques et ce à travers les transactions.

Pour créer une séquence :

```
CREATE [ TEMPORARY ] SEQUENCE nom [ INCREMENT [ BY ] increment ]
    [ MINVALUE valeur ] [ MAXVALUE valeur ]
    [ START [ WITH ] valeur ] [ [ NO ] CYCLE ]
    [ OWNED BY table.colonne } ]
```

Par exemple :

```
plop=# create sequence journal.seq_article_id;
CREATE SEQUENCE
plop=# insert into journal.articles
plop-# values ( nextval('journal.seq_article_id'), 'Bla...' );
INSERT 0 1
plop=# select currval('journal.seq_article_id');
 currval
---------
       1
(1 row)
```

Relations, vues

Pour créer une vue, la syntaxe est :

```
CREATE [ OR REPLACE ] [ TEMPORARY ] VIEW nom [ ( nom_colonne [, ...] ) ]
    AS requête
```

Langage de manipulation des données

Les verbes *insert*, *select*, *update* et *delete* du langage de manipulation de données se comportent conformément à la norme SQL.

L'instruction copy

L'instruction *copy* permet de transférer un nombre important de lignes d'une table vers un fichier ou inversement.

Pour copier les lignes d'un fichier vers une table :

```
COPY table [ (colonnes) ]
  FROM { 'nom_fichier' | STDIN }
  [ DELIMITER 'séparateur' ]
  [ NULL 'chaîne nulle' ]
  [ CSV [ HEADER ] [ QUOTE 'quote' ] [ ESCAPE 'escape' ] ]
```

Pour copier les lignes d'une table vers un fichier :

```
COPY { table [ (colonnes) ] | requête }
  TO { 'nom_fichier' | STDOUT }
  [ DELIMITER 'séparateur' ]
  [ NULL 'chaîne nulle' ]
  [ CSV [ HEADER ] [ QUOTE 'quote' ] [ ESCAPE 'escape' ] ]
```

Seul un utilisateur de la base de données disposant des privilèges administrateur peut utiliser un nom de fichier. De plus, ce fichier doit se trouver sur le serveur et est être accessible en lecture ou en écriture par le serveur PostgreSQL.

Transactions

PostgreSQL implémente les transactions conforment au standard ACID :

Atomicité	toutes les instructions de la transaction sont validées ou aucune ne l'est
Cohérence	chaque transaction validée garantit un état cohérent de la base de données
Isolation	les modifications d'une transaction sont invisibles aux autres transactions
Durabilité	si la transaction est validée, les modifications engendrées sont pérennes

Par défaut, chaque instruction est réalisée dans le cadre d'une transaction qui est automatiquement validée.

Pour commencer une nouvelle transaction, il faut utiliser explicitement l'instruction *begin*. Les instructions *commit* et *rollback* permettent respectivement de valider et d'annuler la transaction.

Les instructions LDD sont prises en charge par les transactions.
La fermeture d'une session provoque un *rollback* implicite.

Niveaux d'isolation

Le standard SQL définit quatre niveaux d'isolation entre les transactions :

	dirty read	*non-repeatable read*	*phantom read*
read uncommited	possible	possible	possible
read commited	impossible	possible	possible
repeatable read	impossible	impossible	possible
serializable	impossible	impossible	impossible

PostgreSQL accepte ces quatre niveaux d'isolation mais n'implémente que les niveaux *read commited* et *serializable*. Le niveau *read uncommited* correspond à *read commited* et *repeatable read* à *serializable*.

L'instruction est *set transaction* :

```
SET TRANSACTION ISOLATION LEVEL
    { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
```

L'isolation entre les transactions est implémenté par le mécanisme MVCC (*Multi-Version Concurrency Control*) qui génère bien moins de soucis qu'une implémentation basée sur les verrous.

Verrous

L'utilisation des verrous est devenue inutile grâce à MVCC.

Il est néanmoins possible d'utiliser des verrous explicites. De plus, certaines opérations requièrent un accès exclusifs aux objets de la base de données (notamment les instructions LDD), PostgreSQL est donc amené à poser des verrous automatiquement.

Les verrous peuvent être posés :

- avec l'instruction *lock* ;
- en utilisant les clauses *for update* et *for share* de l'instruction *select*.

Il existe pas moins de 8 niveaux de verrous sur les tables (et 2 sur les lignes) :

```
LOCK table [, ...]
   [ IN ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
     | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE MODE ]
   [ NOWAIT ]
```

Les verrous ne peuvent être posés que dans le cadre d'une transaction commencée avec *begin*. Ils sont libérés lorsque la transaction est terminée.

La liste des verrous actifs est consultable dans la vue système *pg_locks*.

Recherche de texte avec tsearch2

tsearch2 est une contribution qui a été intégrée à PostreSQL 8.3, elle permet d'effectuer des recherches intégrales de texte (*full text searches*).

Une recherche avec *tsearch2* nécessite deux objets :

- un vecteur *tsvector* construit à partir du jeux de données à parcourir ;
- une requête *tsquery* qui représente les critères de la recherche.

Les fonctions *to_tsvector* et *to_tsquery* permettent d'obtenir ces objets qu'il suffit ensuite de combiner avec l'opérateur @@. Par exemple :

```
bdd=> select to_tsvector('french', 'il était un petit navire qui n''avait jamais navigué hohé hohé');
                    to_tsvector
---------------------------------------------------
 'hoh':11,12 'jam':9 'pet':4 'navir':5 'navigu':10

postgres=> select to_tsquery('french', 'un & petit & naviguer');
    to_tsquery
------------------
 'pet' & 'navigu'

bdd=> select to_tsvector( 'french', 'il était un petit navire qui n''avait jamais navigué hohé hohé')
bdd->     @@ to_tsquery('french', 'un & petit & naviguer');
 ?column?
----------
 t

bdd=> select to_tsvector( 'french', 'il était un petit navire qui n''avait jamais navigué hohé hohé')
bdd->     @@ to_tsquery('french', 'un & petit & bateau');
 ?column?
----------
 f
```

Recherche de texte avec tsearch2

Pour recherche dans une table :

```
select art_id, art_titre from articles
where to_tsvector('french', art_texte)
   @@ to_tsquery('french', 'premier & homme & lune');
```

Pour améliorer la vitesse de recherche, il est possible de créer un index GIN :

```
create index idx_art_tsearch on articles
  using gin(to_tsvector('french', art_texte));
```

D'autres fonctions permettent de classer les résultats (*ts_rank* et *ts_rank_cd*) ou même de mettre en valeur dans le texte les termes recherchés (*ts_headline*).

Rôles

Pour gérer l'ensemble des droits d'accès, PostgreSQL utilise le concept des rôles.

Un rôle peut représenter un utilisateur ou un groupe d'utilisateurs en fonction de la manière dont il est configuré.

Un rôle peut :

- être propriétaire d'objets de l'instance (bases de données, tables, etc.) ;
- se voir octroyer des droits d'accès à des objets ou des droit système ;
- être membre d'un autre rôle ;
- se voir attribuer un mot de passe.

Les rôles remplacent donc les notions d'utilisateurs et de groupes qui étaient encore utilisés par PostgreSQL 8.1.

Les rôles ne sont pas liés aux utilisateurs et groupes du système d'exploitation.
Ils sont associés à une instance.

Rôles

Pour créer un rôle :

```
CREATE ROLE nom_role [ [ WITH ] [ SUPERUSER ] [ CREATEDB ]
                                [ CREATEROLE ] [ LOGIN ]
                                [ ENCRYPTED PASSWORD 'mot_de_passe' ]
                                [ IN ROLE nom_role [, ...] ]
                                [ ROLE nom_role [, ...] ] ]
```

Seul l'administrateur ou un utilisateur disposant de l'attribut *createrole* peut créer un rôle.

Par exemple, pour créer un rôle modélisant un utilisateur, on utilisera :

```
postgres=# create role sebastien login encrypted password 'blabla';
postgres=# \c postgres sebastien
Password for user "sebastien":
You are now connected to database "postgres" as user "sebastien".
postgres=>
```

Pour supprimer un rôle :

```
DROP ROLE [ IF EXISTS ] nom_role
```

Rôles, groupes

Les rôles sont également utilisés pour modéliser la notion de groupes d'utilisateurs. Il suffit pour cela de créer un rôle sans mot de passe et ne disposant pas de l'attribut *login*.

Par exemple, l'instruction suivante crée un rôle (groupe) *compta* et lui associe deux autres rôles. Ces derniers hériteront des droits octroyés au rôle *compta* :

```
create role compta role sebastien, dominique;
```

Cette instruction crée un autre rôle (utilisateur) et l'ajoute au rôle (groupe) *compta* :

```
create role daniel login encrypted password 'abcde' in role compta;
```

Droits

Le système des droits de PostgreSQL est basé sur les rôles et utilise les instructions SQL GRANT et REVOKE. Ces instructions sont utilisées pour :

- octroyer ou révoquer des droits d'accès sur des objets ;
- ajouter ou retirer des rôles (utilisateurs) à d'autres rôles (groupes).

Les rôles dispose également d'un certains nombres d'attributs :

- LOGIN / NOLOGIN ;
- SUPERUSER / NOSUPERUSER ;
- CREATEDB / NOCREATEDB ;
- CREATEROLE / NOCREATEROLE ;
- PASSWORD.

Ces attributs sont associés aux rôles lors de leur création (CREATE ROLE) ou avec l'instruction ALTER ROLE.

Les rôles qui disposent de l'attribut SUPERUSER contournent le système des privilèges (c'est-à-dire qu'ils ont tous les privilèges).

Droits, accès aux objets

Par défaut, seul le propriétaire d'un objet (souvent son créateur) a accès à cet objet.

Des privilèges supplémentaires peuvent être associés aux objets avec l'instruction GRANT comme ceci :

```
GRANT { privilège [,...] | ALL } ON [ type_objet ] nom_objet [, ...]
    TO { nom_role | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

Les privilèges octroyés dépendent du type d'objet :

- tables : SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER
- séquences : USAGE | SELECT | UPDATE
- base de données : CREATE | CONNECT | TEMPORARY
- fonctions : EXECUTE
- langage : USAGE
- schémas : CREATE | USAGE
- espaces de tables : CREATE

ALL représente tous les privilèges et PUBLIC tous les rôles.
L'option WITH GRANT OPTION autorise le ou les rôles cibles à octroyer ces privilèges.

Droits, accès aux objets

La révocation des droits utilise cette syntaxe :

```
REVOKE [ GRANT OPTION FOR ]
    { privilège [,...] | ALL } ON [ type_objet ] nom_objet [, ...]
    FROM { nom_role | PUBLIC } [, ...] [ CASCADE | RESTRICT ]
```

Par exemple, pour révoquer tous les droits sur le schéma public :

```
revoke all on schema public from public;
```

Droits, accès aux objets

Exemple :

```
plop=# revoke all on schema public from public;
REVOKE
plop=# create table public.test (id int);
CREATE TABLE
plop=# insert into test values (1);
INSERT 0 1

plop=# \c plop sebastien
Password for user "sebastien":
You are now connected to database "plop" as user "sebastien".
plop=> select * from public.test;
ERREUR:  droit refusé pour le schéma public

plop=> \c plop postgres
You are now connected to database "plop" as user "postgres".
plop=# grant select on table public.test to sebastien;
GRANT
plop=# grant usage on schema public to sebastien;
GRANT

plop=# \c plop sebastien
Password for user "sebastien":
You are now connected to database "plop" as user "sebastien".
plop=> select * from public.test;
 id
----
  1
(1 row)
```

Droits, notion de groupes

Les instructions SQL sont également utilisées pour ajouter ou retirer des rôles à d'autres rôles. Il est ainsi possible de simuler la notion de groupes.

Par exemple :

```
plop=# create schema journal;
CREATE SCHEMA
plop=# create table journal.articles (id int, contenu text);
CREATE TABLE
plop=# create role utilisateurs;
CREATE ROLE
plop=# grant usage on schema journal to utilisateurs;
GRANT
plop=# alter role utilisateurs set search_path = 'journal';
ALTER ROLE
plop=# create role lecteur in role utilisateurs;
CREATE ROLE
plop=# create role auteur in role utilisateurs;
CREATE ROLE
plop=# grant select on journal.articles to lecteur;
GRANT
plop=# grant insert, update on journal.articles to auteur;
GRANT
plop=# create role daniel login encrypted password 'blabla' in role lecteur;
CREATE ROLE
plop=# create role marc login encrypted password 'plopplop' in role auteur;
CREATE ROLE
```

Droits, bonnes pratiques

Plusieurs politiques peuvent être mises en œuvre :

1) Une base de données est associée à un rôle utilisateur. En général le nom de la base de données est identique à celui du propriétaire. Souvent, une base de données est créée pour une application.

2) Un schéma est associé à un utilisateur. Le nom du schéma est identique à celui de l'utilisateur (cf. variable *search_path* dont la valeur par défaut est *"$user",public*).

3) Un schéma est associé à un rôle utilisateur, tous les objets utilisés par une application sont créés dans ce schéma et appartiennent à cet utilisateur qui est l'administrateur de l'application. Un rôle groupe est créé, des droits lui sont affectés pour utiliser les objets du schéma. Les utilisateurs de l'application sont des rôles utilisateurs et sont membre du rôle groupe.

Espaces de tables

Un espace de tables (ou *tablespace*) est un répertoire du système de fichiers qui peut être utilisé pour stocker les données de relations (tables et indexes).

Ils sont utilisés pour deux raisons :

1) Pour étendre des bases de données contraintes par la taille d'un système de fichiers qui ne peut être agrandi.

2) Plus fréquemment, pour optimiser la vitesse d'exécution des requêtes en stockant, par exemple, les tables sur un disque et les indexes sur un autre.

Les espaces de tables sont associés à l'instance, ils sont dons disponibles pour l'ensemble des bases de données de l'instance.

Espaces de tables

Pour créer un espace de tables :

```
CREATE TABLESPACE nom [ OWNER utilisateur ] LOCATION 'répertoire'
```

Seul l'administrateur peut créer un espace de tables.
Le propriétaire par défaut est l'utilisateur connecté.
Le répertoire doit exister et l'utilisateur système qui exécute PostgreSQL doit y avoir accès en écriture.

Sous *psql*, *\db* permet de lister les espaces de tables existants. Ou :

```
SELECT spcname FROM pg_tablespace;
```

Seul un administrateur ou son propriétaire peut détruire un espace de tables.
Pour supprimer un espace de tables :

```
DROP TABLESPACE [ IF EXISTS ] nom
```

Il ne peut l'être s'il contient des objets.
Mais il est possible de déplacer une table d'un espace de tables à un autre :

```
ALTER TABLE nom_table SET TABLESPACE nom_espace
```

Espaces de tables

Par exemple :

```
# mkdir -m 700 /var/pg/u02/inst1
# chown postgres:postgres /var/pg/u02/inst1
# su - postgres -c psql
Bienvenue dans psql 8.2.7, l'interface interactive de PostgreSQL.

postgres=# create tablespace tbl_data location '/var/pg/u02/inst1';
CREATE TABLESPACE
postgres=# \c test
You are now connected to database "test".
test=# \db
              List of tablespaces
    Name    |  Owner   |     Location
------------+----------+------------------
 pg_default | postgres |
 pg_global  | postgres |
 tbl_data   | postgres | /var/pg/u02/inst1
(3 rows)

test=# create table big_data (id int, data text) tablespace tbl_data;
CREATE TABLE
postgres=# \!ls -l /var/pg/u02/inst1
total 8
drwx------ 2 postgres postgres 4096 jan 23 11:56 16390
-rw------- 1 postgres postgres    4 jan 23 11:50 PG_VERSION
```

Langages procéduraux

Pour écrire des procédures stockées exécutables sur le serveur de bases de données, PotgreSQL permet l'utilisation de plusieurs langages, notamment :

- PL/pgSQL, le langage procédural fournit avec PostgreSQL ;
- PL/Perl ;
- PL/Python ;
- PL/Tcl ;
- PL/Java.

Ces langages sont implémentés par des bibliothèques externes et ils doivent être créés avant d'être utilisés. À l'exception, du langage PL/pgSQL, ces bibliothèques sont souvent disponibles dans des paquets séparés des distributions Linux.

Pour obtenir la liste des langages créés dans une base de données :

```
postgres=# select lanname from pg_language where lanispl = true;
 lanname
---------
(0 rows)
```

Langages procéduraux

Pour obtenir la liste des langages disponibles pour une base de données :

```
postgres=# select tmplname, tmpltrusted from pg_pltemplate;
 tmplname   | tmpltrusted
------------+-------------
 plperl     | t
 plperlu    | f
 plpgsql    | t
 plpythonu  | f
 pltcl      | t
 pltclu     | f
(6 rows)
```

Un langage dit sûr (*trusted*) est un langage qui ne permet pas d'outrepasser le système des privilèges de PostgreSQL. Seul les rôles disposants de l'attribut administrateur auront l'autorisation de créer des fonctions écrites avec un langage non sûr (*untrusted*).

Pour créer un langage dont un modèle est disponible, par exemple PL/pgSQL :

```
postgres=# create language plpgsql;
CREATE LANGUAGE
```

Le privilège USAGE est octroyé à PUBLIC pour un langage sûr lors de sa création.

PL/pgSQL, fonctions

Par exemple, si le fichier `repete.sql` contient ce code :

```
create function repete(str varchar, nb int)
returns varchar as $$
declare
  res varchar := '';
begin
  for i in 1..nb loop
    res := res || str;
  end loop;
  return res;
end;
$$
language plpgsql;
```

Alors, la création et l'exécution de cette fonction se font ainsi :

```
plop=# \i repete.sql
CREATE FUNCTION

plop=# select repete('x', 20);
        repete
----------------------
 xxxxxxxxxxxxxxxxxxxx
(1 row)
```

PL/pgSQL, procédures

Les procédures PostgreSQL sont des fonctions qui retournent *void*. Par exemple :

```
drop table if exists scores;
create table scores ( equipe varchar(20), points int );

create or replace function maj_points( i_equipe varchar, i_score int )
returns void as $$
begin
  update scores set points = points + i_score where equipe = i_equipe;
  if not found then
    insert into scores ( equipe, points ) values ( i_equipe, i_score );
  end if;
end;
$$
language plpgsql;
```

Cette procédure stockée est créée et utilisée ainsi :

```
postgres=# select maj_points( 'Grand-Champ', 10 );
 maj_points
------------

(1 row)

plop=# select * from scores;
   equipe     | points
--------------+--------
 Grand-Champ |     10
(1 row)
```

Déclencheurs

Les déclencheurs (ou *triggers*) sont des fonctions qui sont exécutées automatiquement lorsqu'une action spécifique (insertion, mise-à-jour, suppression d'enregistrements) a lieu sur une table.

Les déclencheurs sont créés avec cette instruction SQL :

```
CREATE TRIGGER nom [ BEFORE | AFTER ] { évènement [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE fonction ( paramètres )
```

Les évènements autorisés sont INSERT, UPDATE ou DELETE.

Une image de l'enregistrement en cours de traitement est disponible dans les pseudo-enregistrements NEW (pour les insertions et mises-à-jour) et OLD (mises-à-jour et suppression).

Un déclencheur appelle une fonction, il est donc nécessaire de commencer par écrire cette fonction.

Déclencheurs, premier exemple

Une utilisation des déclencheurs est, parmi d'autres, l'historisation des modifications de valeurs sur une table.

Par exemple :

```
drop table if exists histo;
create table histo (equipe varchar(20), jour date, avant int, apres int);

create or replace function trig_histo_scores()
returns trigger as $$
begin
  if OLD.points <> NEW.points then
    insert into histo ( equipe, jour, avant, apres )
    values ( NEW.equipe, current_date, OLD.points, NEW.points );
  end if;
  return NEW;
end;
$$
language plpgsql;

drop trigger if exists trig_after_update_scores on scores;
create trigger trig_after_update_scores after update on scores
for each row execute procedure trig_histo_scores();
```

Déclencheurs, premier exemple

Suite de l'exemple :

```
plop=# \i histo.sql
DROP TABLE
CREATE TABLE
CREATE FUNCTION
DROP TRIGGER
CREATE TRIGGER

postgres=# select maj_points( 'Grand-Champ', 5 );
 maj_points
------------

(1 row)

postgres=# select * from scores;
   equipe     | points
--------------+--------
 Grand-Champ |     15
(1 row)

postgres=# select * from histo;
   equipe     |    jour    | avant | apres
--------------+------------+-------+-------
 Grand-Champ | 2008-05-09 |    10 |    15
(1 row)
```

Déclencheurs, second exemple

Autre exemple :

```
create or replace function trig_update_timestamp()
returns trigger as $$
begin
  NEW.when_modified = current_timestamp;
  NEW.who_modified = current_user;
  return NEW;
end;
$$ language plpgsql;

create or replace function trig_insert_timestamp()
returns trigger as $$
begin
  NEW.when_created = current_timestamp;
  NEW.who_created = current_user;
  return NEW;
end;
$$ language plpgsql;

drop trigger if exists trig_before_update_testtg on testtg;
create trigger trig_before_update_testtg before insert or update on testtg
for each row execute procedure trig_update_timestamp();

drop trigger if exists trig_before_insert_testtg on testtg;
create trigger trig_before_insert_testtg before insert or update on testtg
for each row execute procedure trig_insert_timestamp();
```

Les extensions en C

Le serveur PostgreSQL est extensible : il est possible d'écrire de nouvelles fonctions en C et de les compiler sous forme de bibliothèques dynamiques utilisables ensuite dans les instructions SQL.

Une fois la bibliothèque compilée et placée au bonne endroit, il suffit d'utiliser l'instruction *create function* ainsi :

```
CREATE [ OR REPLACE ] FUNCTION nom ( [ paramètres... ] )
  [ RETURNS type ]
  LANGUAGE c
  AS 'fichier', 'symbole'
```

Programmation côté client – PHP

La version 5 de PHP offre un pilote générique pour l'accès aux données, PDO (*PHP Data Objects*), compatible avec PostgreSQL.

Par exemple :

```
<?php
  header("Content-Type: text/plain; charset=ISO-8859-1");

  $user = 'sebastien';
  $pass = 'plopplop';
  $dsn  = 'pgsql:host=localhost;dbname=test';

  try {
    $dbh = new PDO($dsn, $user, $pass);
    foreach ( $dbh->query('select * from articles') as $row ){
      print_r($row);
    }
    $dbh = null;
  }
  catch (PDOException $e) {
    print "Erreur ! : " . $e->getMessage();
    die();
  }
?>
```

Programmation côté client – Java

Un pilote JDBC de type IV est disponible sur le site de PostgreSQL à cette adresse :

http://jdbc.postgresql.org/

Il est recommandé d'utiliser un gestionnaire de sessions JNDI dans les serveurs d'applications afin d'optimiser la gestion des connexions.

Par exemple :

```
import java.sql.*;

Class.forName("org.postgresql.Driver");

String url      = "jdbc:postgresql://localhost/test";
String username = "sebastien";
String password = "plopplop";

Connection db = DriverManager.getConnection(url, username, password);
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("select * FROM articles");
while ( rs.next() ){
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

Administration, exploitation

Les journaux binaires

Les journaux binaires sont utilisés pour garantir la durabilité (le « D » de ACID) des transactions validées en évitant de grever significativement les performances.

Plutôt que d'écrire directement dans les fichiers de données à chaque fois qu'une transaction est validée, les données modifiées sous écrites dans un journal linéaire. De nombreux déplacement des têtes de lecture des disques sont ainsi évités.

Ce journal linéaire est lui-même enregistrés séquentiellement dans des fichiers du répertoire *$PGBASE/pg_xlog* dont le nom est un entier qui est incrémenté. Sous PostgreSQL, ces journaux sont appelés WAL (*Write Ahead Logs*). Le processus *wal writer process* écrit dans ces journaux.

À intervalles réguliers un mécanisme appelé point de contrôle (*checkpoint*) permet de basculer les transactions présentes dans les journaux binaires vers les fichiers de données. Le processus *writer process* est chargé de cette tâche.

Si un arrêt intempestif survient, PostgreSQL utilise les journaux binaires pour appliquer aux fichiers de données les modifications générées par les dernières transactions validées avant l'arrêt.

Les journaux binaires

À chaque point de contrôle un enregistrement spécifique est écrit dans les journaux binaires. Toutes les données antérieures contenues dans ces journaux deviennent inutiles. Les fichiers (appelés segments) qui contiennent des journaux binaires dont toutes les données sont antérieures au dernier point de contrôle sont recyclés.

Les journaux binaires peuvent également être archivés. Ils permettent ainsi d'effectuer une sauvegarde au fil de l'eau et fournissent un mécanisme de retour en arrière (nommé PITR, pour *Point-In-Time Recovery*).

Un point de contrôle est exécuté lorsque l'une de ces situations survient :

- *checkpoint_segments* segments sont actifs ;
- *checkpoint_timeout* secondes ce sont écoulées depuis le dernier point de contrôle ;
- l'instruction *checkpoint* est exécutée.

La taille des segments est figée (16 Mo par défaut).

Si la base de données subit une forte activité transactionnelle, les performances seront largement améliorées si les fichiers de données sont stockés sur des disques différents des journaux binaires.

Gestion de la mémoire

Pour déterminer les valeurs des paramètres de PostgreSQL liés à l'utilisation de la mémoire, il est nécessaire de garder à l'esprit que ces zones de mémoires peuvent être partagées en deux grandes catégories :

1) les zones de mémoires locales allouées par processus ;
2) les zones de mémoires partagées entre tous les processus de l'instance.

Ces paramètres sont fixés dans le fichier *postgresql.conf*. Elles sont exprimées en Ko, mais il est possible d'utiliser des unités de taille dans le fichier de configuration.

Dans *psql*, il est possible de voir les valeurs avec l'instruction *show* et de les modifier avec l'instruction *set* pour celles qui peuvent l'être en fonctionnement.

Par exemple :

```
postgres=# show work_mem;
 work_mem
----------
 2MB
(1 row)
postgres=# set work_mem = 4096;
SET
```

Gestion de la mémoire

Il y a essentiellement six paramètres importants :

shared_buffers (32 Mo par défaut, mémoire partagée)
Tampons de mémoire partagée entre tous les processus (ne peut dépasser *shmmax*).
La valeur de ce paramètre a un impact important sur les performances.

temp_buffers (8 Mo par défaut, mémoire locale)
Taille maximum des tampons locaux pour l'accès aux tables temporaires.

work_mem (1Mo par défaut, mémoire locale)
Mémoire utilisée pour les opérations de tri et hash en avant utilisation du disque.

maintenance_work_mem (16Mo, mémoire locale)
Mémoire utilisée pour les opérations de maintenance (*vacuum*, *create index*, etc.)
Peu d'utilisations simultanées, possibilité d'augmenter significativement.

max_stack_depth (2Mo par défaut, mémoire locale)
Taille maximum de la pile d'exécution (utiliser *ulimit -s* moins 1 ou 2 Mo).
Peut gêner l'exécution de requêtes ou procédures complexes si trop faible.

wal_buffers (64 Ko par défaut, mémoire partagée)
Doit être assez grand pour stocker les données d'une transaction type.

Le collecteur de statistiques

Le collecteur de statistiques est un processus (*stats collector process*) qui maintient un certains nombres de compteurs qui aident l'optimiseur de requêtes à fournir des plans d'exécution optimaux. Ces indicateurs sont également utiles pour l'administration de la base de données.

La collecte des statistiques est activée avec le paramètre *track_counts*.

Les statistiques collectées sont accessibles par l'intermédiaire des vues *pg_stat_** et *pg_statio_** ainsi que des fonctions *pg_stat_get_** du catalogue *pg_catalog*.

La fonction *pg_stat_reset* permet de réinitialiser les statistiques de la base de données active.

Le journal d'activité

Le journal d'activité est l'endroit où PostgreSQL consigne l'ensemble des messages générés pendant son exécution, du plus insignifiant à l'erreur fatale selon la configuration des paramètres du fichier *postgresql.conf* :

log_destination	une combinaison de *stderr*, *csvlog*, *syslog* et *eventlog* (Windows)
logging_collector	active l'envoi des messages vers des fichiers
log_directory	répertoire dans lequel seront enregistrés ces fichiers
log_statement	écrire les requêtes exécutées (*none*, *ddl*, *mod* ou *all*)

Des dizaines d'autres paramètres sont disponibles pour sélectionner les informations qui doivent être envoyées dans le journal d'activité.

Sauvegardes

Trois stratégies de sauvegardes sont possibles :

1) Export à partir d'une base de données active en utilisant le programme *pg_dump*.
2) Sauvegarde des fichiers de l'instance lorsqu'elle est arrêtée.
3) Archivage des journaux des transactions.

L'outil *pg_dump* est intéressant car :

- il permet de transférer rapidement des données entre bases de données ;
- il support trois types de format, dont le SQL ;
- il est simple d'emploi.

Mais il ne peut sauvegarder des bases de taille importante.

Sauvegardes, pg_dump

La commande *pg_dump* exporte une base de données d'une instance. Elle peut être exécutée sur le serveur PostgreSQL (connexion par *socket* Unix) ou à partir d'une machine du réseau (avec les mêmes options *-h* et *-p* que *psql*).

L'outil *pg_dump* supporte trois types de format :

1) Format texte (par défaut), les données seront restaurées avec *psql*.
2) Format tar, les données seront restaurées avec *pg_restore*.
3) Format spécifique, restauration avec *pg_restore*.

Par exemple :

```
$ pg_dump -h 192.168.2.42 -p 5433 compta > compta.sql
```

Quelques options :

-C	ajouter les commandes pour créer la base de données dans l'export
-a	n'exporter que les données (LMD)
-s	n'exporter que le schéma (LDD)
-Ft	utiliser le format tar
-Fc	utiliser le format spécifique

Sauvegardes, pg_restore

Pour restaure un export réalisé au format texte :

```
$ pg_dump -h 192.168.2.42 -p 5433 -U postgres -C compta > compta.sql
$ psql -h 192.168.2.42 -p 5433 -U postgres postgres < compta.sql
```

Autres exemples :

```
$ pg_dump -h host1 db | psql -h host2 db
```

```
$ pg_dump db | gzip > db.sql.gz
$ zcat db.sql.gz | psql db
```

Pour les formats *tar* et spécifique, la restauration est faite avec *pg_restore* :

```
$ pg_dump -Ft db > db.tar
$ dropdb db
$ pg_restore -C -d postgres < db.tar

$ pg_dump -Fc db > db.dump
$ dropdb db
$ pg_restore -C -d postgres < db.dump

$ pg_dump -Fc db > db.dump
$ createdb copy
$ pg_restore -d copy < db.dump
```

Sauvegardes, système de fichiers

À l'exception de cas simples, la sauvegarde des fichiers d'une instance doit se faire lorsque celle-ci est arrêtée.

Si la taille des fichiers de l'instance est importante, l'utilisation d'un gestionnaire de volumes logiques avec possibilité de réaliser des clichés peut aider à réduire les temps d'interruption.

Le répertoire de l'instance doit être sauvegardé ainsi que les répertoires des espaces de tables supplémentaires.

Le principe est le suivant :

```
$ pg_ctl stop
$ tar czfC /backups/AAAA-MM-JJ.tar.gz / var/pg/u01/inst1 var/pg/u02/inst1
$ pg_ctl start
```

Sauvegarde en continu (journaux des transactions)

Les journaux des transactions (WAL) contiennent les données de toutes les transactions exécutées sur une instance.

L'idée de la sauvegarde en continu consiste à sauvegarder chacun de ces journaux dès qu'il est rempli ou lorsqu'un délai est dépassé. Ces journaux pourront ensuite être rejoués sur une base de données restaurées à partir d'une sauvegarde complète.

Deux étapes sont nécessaires :

1) Activer l'archivage des journaux des transactions.
2) Exécuter régulièrement une sauvegarde complète de l'instance.

L'archivage des journaux est contrôlé avec ces options de *postgresql.conf* :

```
archive_mode = on
archive_command = 'test ! -f /mnt/logarch/%f && cp %p /mnt/logarch/%f'
archive_timeout = 1h
```

%f est le nom du fichier qui contient le journal à archiver, *%p* contient son chemin complet (relativement à *$PGDATA*). La commande doit retourner le code sortie 0 si la copie a été réalisée correctement.

Sauvegarde en continu (journaux des transactions)

Les sauvegardes complètes de l'instance doivent être réalisées d'une manière particulière :

1) Exécution de la function SQL *pg_start_backup*.
2) Sauvegarde des fichiers de l'instances à l'exception du répertoire *pg_xlog*.
3) Exécution de la fonction *pg_stop_backup*.

Par exemple :

```
$ psql
postgres=# select pg_start_backup('Sauvegarde du AAAA-MM-JJ');
postgres=# \q
$ tar czfC /mnt/logarch/AAAA-MM-JJ.tar.gz / var/pg/*/inst1
$ psql
postgres=# select pg_stop_backup();
postgres=# \q
```

L'étiquette est conservée dans le fichier *$PGDATA/backup_label*.

Un fichier *$PGDATA/pg_xlog/000000010000123400005 5CD.007C9330.backup* est créé pour indiquer le dernier journal utilisé avant la sauvegarde.

Sauvegarde en continu, restauration

Le processus de restauration est assez complexe, il est recommandé de l'écrire dans une procédure et de la rejouer régulièrement afin de valider que l'environnement nécessaire à la restauration n'a pas changé.

Le schéma est le suivant :

1) Installer un nouveau serveur identique.
2) Si disponibles, mettre de côté les journaux du répertoire *$PGDATA/pg_xlog*.
3) Supprimer tous les fichiers et répertoires de l'instance et ses espaces de tables.
4) Restaurer la sauvegarde complète en prenant garde à rétablir les permissions.
5) Supprimer les fichiers dans *$PGDATA/pg_xlog* ou recréer ce répertoire.
6) S'assurer que le répertoire *$PGDATA/pg_xlog/archive_status* existe.
7) Recopier les journaux éventuellement mis de côté en 2) dans *$PGDATA/pg_xlog*.
8) Créer un fichier *recovery.conf* dans *$PGDATA* à partir de *recovery.conf.sample*.
9) Démarrer PostgreSQL en s'assurant que les utilisateurs ne pourront se connecter.

Les deux paramètres importants de *recovery.conf* sont :

```
restore_command = 'cp /mnt/logarch/%f %p'
#recovery_target_time = '2008-04-23 11:20:55 EST'
```

L'analyse des requêtes avec explain

Pour chaque requête, l'optimiseur élabore un plan d'exécution. De sa pertinence dépend la performance du serveur. L'instruction *explain* permet d'afficher le plan d'exécution prévu pour une requête :

```
EXPLAIN [ ANALYZE ] requête
```

La requête ne sera pas exécutée à moins que la clause *analyze* ne soit utilisée. Pour être certain de ne modifier aucune donnée, utiliser :

```
bdd=> begin;
bdd=> explain analyze requête;
bdd=> rollback;
```

Chaque ligne correspond à un nœud d'exécution, pour chaque nœud sont affichées les informations suivantes :

- le type de l'action (parcours séquentiel, parcours d'un index, jointure, *etc.*) ;
- le coût estimé pour le démarrage du parcours ;
- le coût total estimé ;
- le nombre estimé de lignes renvoyées par le nœud d'exécution ;
- la largeur moyenne estimée des lignes.

L'analyse des requêtes avec explain

Avec la clause *analyze*, les données estimées sont complétées par les donnée réelles.

Par exemple :

```
pagila=# explain select count(*) from film where language_id=1;
                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=69.00..69.01 rows=1 width=0)
   ->  Seq Scan on film  (cost=0.00..66.50 rows=1000 width=0)
         Filter: (language_id = 1)
(3 rows)

pagila=# explain analyze select count(*) from film where language_id=1;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=69.00..69.01 rows=1 width=0) (actual time=5.086..5.088 rows=1 loops=1)
   ->  Seq Scan on film  (cost=0.00..66.50 rows=1000 width=0) (actual time=0.014..2.558 rows=1000 loops=1)
         Filter: (language_id = 1)
 Total runtime: 5.135 ms
(4 rows)
```

L'analyse des requêtes avec explain

explain nous permet de comparer les plans d'exécution avec ou sans utilisation d'un index :

```
pagila=# explain select rental_id from rental where return_date is null;
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on rental  (cost=0.00..294.44 rows=166 width=4)
   Filter: (return_date IS NULL)
(2 rows)

pagila=# create index idx_rental_return on rental (return_date);
CREATE INDEX

pagila=# explain select rental_id from rental where return_date is null;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_rental_return on rental (cost=0.00..32.97 rows=166 width=4)
   Index Cond: (return_date IS NULL)
(2 rows)
```

La reconstruction d'index avec reindex

L'instruction *reindex* permet de reconstruire le contenu d'un index à partir des données de la table. Cela peut être nécessaire :

- si l'index est corrompu ;
- s'il est trop fragmenté ;
- si l'un de ses paramètres a changé (soit, *fillfactor*).

Plusieurs syntaxes sont possibles :

```
REINDEX INDEX nom_index
REINDEX TABLE nom_table
REINDEX DATABASE nom_bdd
```

Échantillonnage avec analyze

L'instruction *analyze* est utilisée pour échantillonner les données contenues dans une table. Ces informations statistiques sont stockées dans la table *pg_statistic* du catalogue *pg_catalog*.

L'optimiseur de requêtes les utilisera afin d'élaborer le meilleur plan d'exécution. Il est donc important de les mettre à jour régulièrement.

Sa syntaxe est :

```
ANALYZE [ VERBOSE ] nom_table
```

Par exemple :

```
pagila=# analyze verbose rental;
INFO: analyzing "public.rental"
INFO: "rental": scanned 134 of 134 pages, containing 16045 live rows
          and 7 dead rows; 3000 rows in sample, 16045 estimated total rows
ANALYZE
```

Les statistiques ainsi échantillonnées ne doivent pas être confondues avec celles qui sont maintenues par le collecteur de statistiques (processus *stats collector process*).

Vacuum

Le processus nommé *Vacuum* (littéralement « aspirateur ») doit être exécuté régulièrement sur une base de données PostgreSQL. Il permet de :

- récupérer l'espace libéré suite à la mise-à-jour ou la suppression d'enregistrements ;
- mettre à jour les statistiques utilisées par l'optimiseur de requêtes (*analyze*) ;
- prévenir la perte d'anciennes données par la rotation des identifiants de transaction.

Avant PostgreSQL 8.1, ce processus devait être planifié à intervalles réguliers. Depuis, un processus (*Autovacuum*) prend en charge son exécution.

La procédure peut être exécutée pendant que la base de données est utilisée. Mais :

- un ralentissement sera perceptible ;
- les instructions qui modifient le schéma (LDD) ne pouront pas être utilisées.

Pour réduire le temps nécessaire à l'exécution de *Vacuum*, certains paramètres peuvent être ajustés. Par exemple, les tables qui contiennent des enregistrements qui sont peu modifiés et rarement supprimés n'ont pas besoin d'être nettoyées aussi souvent que d'autres tables plus « volatiles ».

Autovacuum

Pour exécuter le processus vacuum manuellement, la syntaxe est :

```
VACUUM [ FULL ] [ ANALYZE ] table
```

L'option *full* permet de récupérer tout l'espace libre (défragmentation), mais le processus peut s'avérer bien plus long et nécessite la pose d'un verrou exclusif sur les tables.

L'option *analyze* déclenche, en plus, l'échantillonnage des statistiques.

À partir de la version 8.1 de PostgreSQL, un démon, *pg_autovocuum* (processus *autovacuum launcher process*), est chargé d'exécuter régulièrement les processus *vacuum* et *analyze* sur les tables de l'instance.

Pour le configurer, modifier ces paramètres de *postgresql.conf* :

```
autovacuum = on
superuser_reserved_connections = 4
autovacuum_naptime = 5min
```

Puis redémarrer PostgreSQL.

Optimisation

L'optimisation d'une base de données dépend de plusieurs paramètres :

- le schéma de la base de données (tables, index, vues, déclencheurs) ;
- sa volumétrie ;
- son usage (OLTP, *data warehouse*).

Les informations utiles sont obtenues :

- dans le journal d'activité ;
- dans les objets du catalogue *pg_catalog* ;
- en exécutant l'instruction *explain* ;
- à partir du résultat de l'instruction *vacuum*.

Les réglages sur lesquels il est possible d'intervenir sont :

- les paramètres du fichier *postgresql.conf* ;
- le schéma de la base de données (index, structure des tables, vues, *etc.*) ;
- le matériel (mémoire, disques durs, processeur) ;
- les paramètres du système d'exploitation.

Annexes

Le gestionnaire de connexions PgPool

PgPool est hébergé sur pgFoundry. Il permet de :

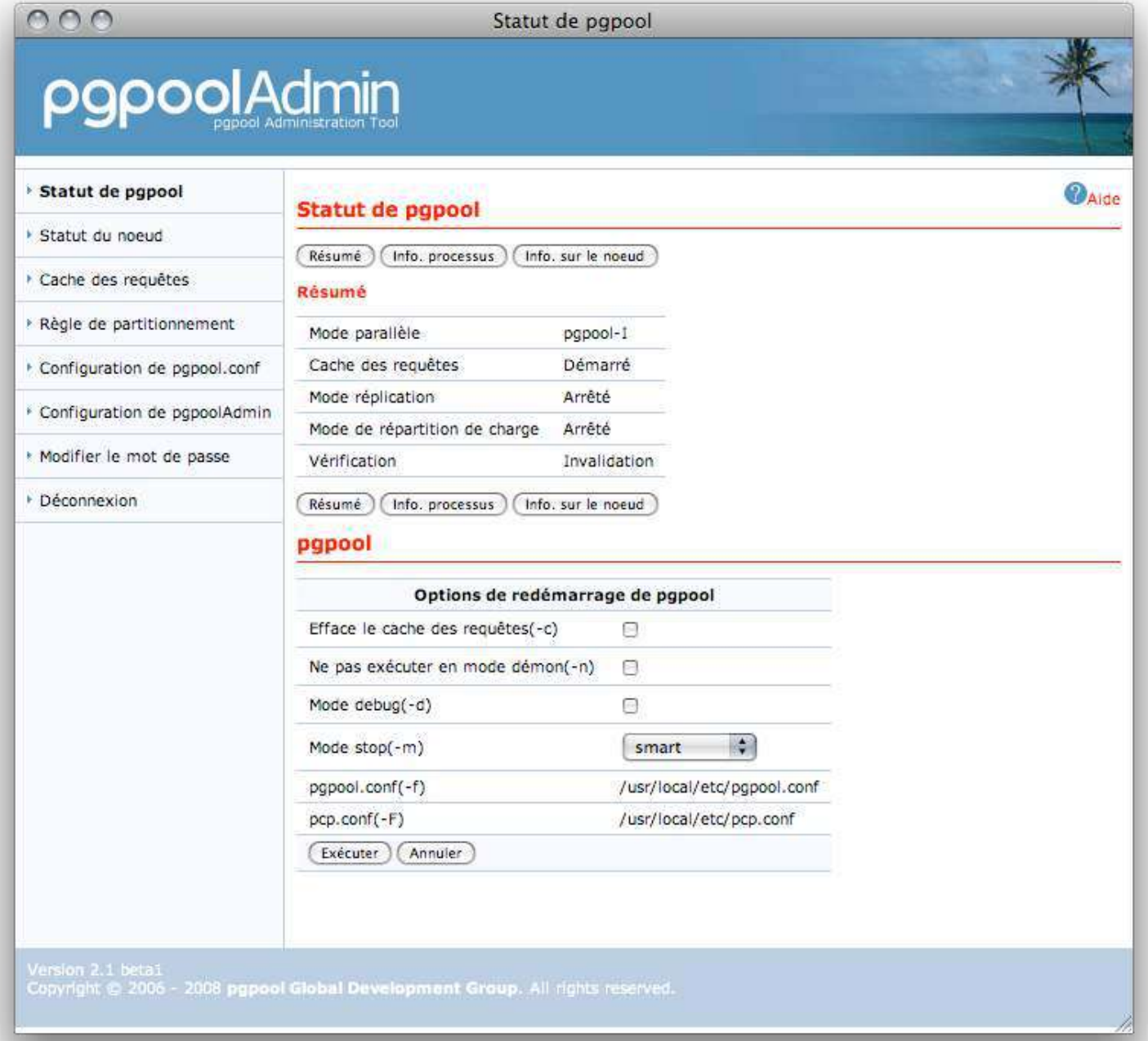
- limiter le nombre de connexions au serveur PostgreSQL ;
- réutiliser des sessions dont les propriétés sont identiques ;
- répliquer les instructions SQL vers plusieurs serveurs ;
- distribuer l'exécution des requêtes sur plusieurs serveurs ;
- répartir l'exécution d'une requête sur plusieurs serveurs (parallélisation).

Le site du projet est :

http://pgpool.projects.postgresql.org/

Le gestionnaire de connexions PgPool

Une interface Web, pgPoolAdmin, peut être utilisée pour la gestion de pgPool.



Notions avancées

PostgreSQL gère les types avancés suivants :

- domaines ;
- tableaux ;
- types composites.

Les relations (tables) peuvent héritées les unes des autres.

Le mécanisme *Toast* permet de stocker des lignes de taille importante.

Les espaces libres dans les fichiers de données sont gérés par l'intermédiaire d'un mécanisme connu sous le nom de FSM (*Free Space Map*).

Un système de règles (*rules*) permet de détourner l'exécution des instructions LMD sur les relations (tables et vues).

Références

Livres

« PostgreSQL » 2ème édition - Sébastien Lardière – ENI Éditions

Sites Web

Site officiel – http://www.postgresql.org/

Site de la communauté francophone – http://www.postgresqlfr.org/

Site du projet pgAdmin – http://www.pgadmin.org/

Pour en savoir plus...

Sébastien Namèche

Conseil
- Architecture des systèmes et réseaux
- Sécurité transverse

Formation
- Logiciels Libres
- Systèmes Unix
- Réseaux et protocoles IP
- Base de données (Oracle, PostgreSQL, MySQL)
- annuaires
- Messageries
- Supervision

Mob. +33 6 0373 1442
http://sebastien.nameche.fr
sebastien@nameche.fr